Reinforcement Learning: An Introduction

Richard S. Sutton
and Andrew G. Barto

MIT Press, Cambridge, MA, 1998
A Bradford Book

Endorsements Code Solutions Figures
Errata Course Slides

This introductory textbook on reinforcement learning is targeted toward engineers and scientists in artificial intelligence, operations research, neural networks, and control systems, and we hope it will also be of interest to psychologists and neuroscientists.

If you would like to order a copy of the book, or if you are qualified instructor and would like to see an examination copy, please see the MIT Press home page for this book. Or you might be interested in the reviews at amazon.com. There is also a Japanese translation available.

The table of contents of the book is given below, with associated HTML. The HTML version has a number of presentation problems, and its text is slightly different from the real book, but it may be useful for some purposes.

Endorsements for:

Reinforcement Learning: An Introduction
by Richard S. Sutton and Andrew G. Barto

"This is a highly intuitive and accessible introduction to the recent major developments in reinforcement learning, written by two of the field's pioneering contributors"

> *Dimitri P. Bertsekas and John N. Tsitsiklis, Professors, Department of Electrical Enginneering and Computer Science, Massachusetts Institute of Technology*

"This book not only provides an introduction to learning theory but also serves as a tremendous sourve of ideas for further development and applications in the real world"

> *Toshio Fukuda, Nagoya University, Japan; President, IEEE Robotics and Automation Society*

"Reinforcement learning has always been important in the understanding of the driving forces behind biological systems, but in the past two decades it has become increasingly important, owing to the development of mathematical algorithms. Barto and Sutton were the prime movers in leading the development of these algorithms and have described them with wonderful clarity in this new text. I predict it will be the standard text."

> *Dana Ballard, Professor of Computer Science, University of Rochester*

"The widely acclaimed work of Sutton and Barto on reinforcement learning applies some essentials of animal learning, in clever ways, to artificial learning systems. This is a very readable and comprehensive account of the background, algorithms, applications, and future directions of this pioneering and far-reaching work."

> *Wolfram Schultz, University of Fribourg, Switzerland*

Code for:

Reinforcement Learning: An Introduction
by Richard S. Sutton and Andrew G. Barto

Below are links to a variety of software related to examples and exercises in the book, organized by chapters (some files appear in multiple places). See particularly the Mountain Car code. Most of the rest of the code is written in Common Lisp and requires utility routines available here. For the graphics, you will need the the packages for G and in some cases my graphing tool. Even if you can not run this code, it still may clarify some of the details of the experiments. However, there is no guarantee that the examples in the book were run using exactly the software given. This code also has not been extensively tested or documented and is being made available "as is". If you have corrections, extensions, additions or improvements of any kind, please send them to me at rich@richsutton.com for inclusion here.

- Chapter 1: Introduction
  - Tic-Tac-Toe Example (Lisp). In C.
- Chapter 2: Evaluative Feedback
  - 10-armed Testbed Example, Figure 2.1 (Lisp)
  - Testbed with Softmax Action Selection, Exercise 2.2 (Lisp)
  - Bandits A and B, Figure 2.3 (Lisp)
  - Testbed with Constant Alpha, cf. Exercise 2.7 (Lisp)
  - Optimistic Initial Values Example, Figure 2.4 (Lisp)
  - Code Pertaining to Reinforcement Comparison: File1, File2, File3 (Lisp)
  - Pursuit Methods Example, Figure 2.6 (Lisp)
- Chapter 3: The Reinforcement Learning Problem
  - Pole-Balancing Example, Figure 3.2 (C)
  - Gridworld Example 3.8, Code for Figures 3.5 and 3.8 (Lisp)
- Chapter 4: Dynamic Programming
  - Policy Evaluation, Gridworld Example 4.1, Figure 4.2 (Lisp)
  - Policy Iteration, Jack's Car Rental Example, Figure 4.4 (Lisp)
  - Value Iteration, Gambler's Problem Example, Figure 4.6 (Lisp)
- Chapter 5: Monte Carlo Methods
  - Monte Carlo Policy Evaluation, Blackjack Example 5.1, Figure 5.2 (Lisp)
  - Monte Carlo ES, Blackjack Example 5.3, Figure 5.5 (Lisp)
- Chapter 6: Temporal-Difference Learning
  - TD Prediction in Random Walk, Example 6.2, Figures 6.5 and 6.6 (Lisp)

    - TD Prediction in Random Walk with Batch Training, Example 6.3, Figure 6.8 (Lisp)
    - TD Prediction in Random Walk (MatLab by Jim Stone)
    - R-learning on Access-Control Queuing Task, Example 6.7, Figure 6.17 (Lisp), (C version)
- Chapter 7: Eligibility Traces
    - N-step TD on the Random Walk, Example 7.1, Figure 7.2: online and offline (Lisp). In C.
    - lambda-return Algorithm on the Random Walk, Example 7.2, Figure 7.6 (Lisp)
    - Online TD(lambda) on the Random Walk, Example 7.3, Figure 7.9 (Lisp)
- Chapter 8: Generalization and Function Approximation
    - Coarseness of Coarse Coding, Example 8.1, Figure 8.4 (Lisp)
    - Tile Coding, a.k.a. CMACs
    - Linear Sarsa(lambda) on the Mountain-Car, a la Example 8.2
    - Baird's Counterexample, Example 8.3, Figures 8.12 and 8.13 (Lisp)
- Chapter 9: Planning and Learning
    - Trajectory Sampling Experiment, Figure 9.14 (Lisp)
- Chapter 10: Dimensions of Reinforcement Learning
- Chapter 11: Case Studies
    - Acrobot (Lisp, environment only)
    - Java Demo of RL Dynamic Channel Assignment

For other RL software see the Reinforcement Learning Repository at Michigan State University and here.

```
;-*- Mode: Lisp; Package: (rss-utilities :use (common-lisp ccl) :nicknames (:ut)) -*-

(defpackage :rss-utilities
  (:use :common-lisp :ccl)
  (:nicknames :ut))

(in-package :ut)

(defun center-view (view)
  "Centers the view in its container, or on the screen if it has no container;
   reduces view-size if needed to fit on screen."
  (let* ((container (view-container view))
         (max-v (if container
                  (point-v (view-size container))
                  (- *screen-height* *menubar-bottom*)))
         (max-h (if container
                  (point-h (view-size container))
                  *screen-width*))
         (v-size (min max-v (point-v (view-size view))))
         (h-size (min max-h (point-h (view-size view)))))
    (set-view-size view h-size v-size)
    (set-view-position view
                       (/ (- max-h h-size) 2)
                       (+ *menubar-bottom* (/ (- max-v v-size) 2)))))
(export 'center-view)

(defmacro square (x)
  `(if (> (abs ,x) 1e10) 1e20 (* ,x ,x)))
(export 'square)

(defun with-probability (p &optional (state *random-state*))
  (> p (random 1.0 state)))
(export 'with-probability)

(defun with-prob (p x y &optional (random-state *random-state*))
  (if (< (random 1.0 random-state) p)
      x
      y))
(export 'with-prob)

(defun random-exponential (tau &optional (state *random-state*))
  (- (* tau
        (log (- 1
                (random 1.0 state))))))
(export 'random-exponential)

(defun random-normal (&optional (random-state cl::*random-state*))
  (do ((u 0.0)
       (v 0.0))
      ((progn
         (setq u (random 1.0 random-state)      ; U is bounded (0 1)
               v (* 2.0 (sqrt 2.0) (exp -0.5)   ; V is bounded (-MAX MAX)
                    (- (random 1.0 random-state) 0.5)))
         (<= (* v v) (* -4.0 u u (log u))))     ; < should be <=
       (/ v u))
    (declare (float u v))))
(export 'random-normal)

;stats

(defun mean (l)
  (float
    (/ (loop for i in l sum i)
       (length l))))
(export 'mean)

(defun mse (target values)
      (mean (loop for v in values collect (square (- v target)))))
(export 'mse)

(defun rmse (target values)                     ;root mean square error
  (sqrt (mse target values))) (export 'rmse)
(export 'rmse)

(defun stdev (l)
  (rmse (mean l) l))
(export 'stdev)

(defun stats (list)
  (list (mean list) (stdev list)))
(export 'stats)

(defun multi-stats (list-of-lists)
  (loop for list in (reorder-list-of-lists list-of-lists)
        collect (stats list)))
(export 'multi-stats)

(defun multi-mean (list-of-lists)
  (loop for list in (reorder-list-of-lists list-of-lists)
        collect (mean list)))
(export 'multi-mean)

(defun logistic (s)
  (/ 1.0 (+ 1.0 (exp (max -20 (min 20 (- s)))))))
(export 'logistic)

(defun reorder-list-of-lists (list-of-lists)
  (loop for n from 0 below (length (first list-of-lists))
        collect (loop for list in list-of-lists collect (nth n list))))
(export 'reorder-list-of-lists)

(defun flatten (list)
  (if (null list)
      (list)
      (if (atom (car list))
          (cons (car list) (flatten (cdr list)))
          (flatten (append (car list) (cdr list))))))
(export 'flatten)


(defun interpolate (x fs xs)
  "Uses linear interpolation to estimate f(x), where fs and xs are lists of
corresponding
   values (f's) and inputs (x's).  The x's must be in increasing order."
  (if (< x (first xs))
      (first fs)
      (loop for last-x in xs
            for next-x in (rest xs)
            for last-f in fs
            for next-f in (rest fs)
            until (< x next-x)
            finally (return (if (< x next-x)
                                (+ last-f
                                   (* (- next-f last-f)
                                      (/ (- x last-x)
                                         (- next-x last-x))))
```

```
                                 next-f)))))
(export 'interpolate)


(defun normal-distribution-function (x mean standard-deviation)
  "Returns the probability with which a normally distributed random number with the
given
   mean and standard deviation will be less than x."
  (let ((fs '(.5 .5398 .5793 .6179 .6554 .6915 .7257 .7580 .7881 .8159 .8413 .8643
.8849
              .9032 .9192 .9332 .9452 .9554 .9641 .9713 .9772 .9938 .9987 .9998 1.0))
        (xs '(0 .1 .2 .3 .4 .5 .6 .7 .8 .9 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9
2.0
                2.5 3.0 3.6 100.0))
        (z (if (= 0 standard-deviation)
              1e10
              (/ (- x mean) standard-deviation))))
    (if (> z 0)
        (interpolate z fs xs)
        (- 1.0 (interpolate (- z) fs xs)))))
(export 'normal-distribution-function)

(defconstant +sqrt-2-PI (sqrt (* 2 3.1415926)) "Square root of 2 PI")

(defun normal-density (z)
  "Returns value of the normal density function at z; mean assumed 0, sd 1"
  (/ (exp (- (* .5 (square (max -20 (min 20 z))))))
     +sqrt-2-PI))
(export 'normal-density)

(defun poisson (n lambda)
  "The probability of n events according to the poisson distribution"
  (* (exp (- lambda))
     (/ (expt lambda n)
        (factorial n))))
(export 'poisson)

(defun factorial (n)
  (if (= n 0)
    1
    (* n (factorial (- n 1)))))
(export 'factorial)

(defun q (&rest ignore)
  (declare (ignore ignore))
  (values))                     ;evaluates it's arg and returns nothing
(export 'q)

(defmacro swap (x y)
  (let ((var (gensym)))
    `(let ((,var ,x))
       (setf ,x ,y)
       (setf ,y ,var))))
(export 'swap)

(defmacro setq-list (list-of-vars list-of-values-form)
  (append (list 'let (list (list 'list-of-values list-of-values-form)))
          (loop for var in list-of-vars
                for n from 0 by 1
                collect (list 'setf var (list 'nth n 'list-of-values)))))
(export 'setq-list)

(defmacro bound (x limit)
  `(setf ,x (max (- ,limit) (min ,limit ,x))))
(export 'bound)

(defmacro limit (x limit)
  `(max (- ,limit) (min ,limit ,x)))
(export 'limit)

(defvar *z-alphas* '((2.33 .01) (1.645 .05) (1.28 .1)))
(defmacro z-alpha (za) `(first ,za))
(defmacro z-level (za) `(second ,za))
(defun z-test (mean1 stdev1 size1 mean2 stdev2 size2)
  (let* ((stdev (sqrt (+ (/ (* stdev1 stdev1) size1)
                        (/ (* stdev2 stdev2) size2))))
         (z (/ (- mean1 mean2) stdev)))
        (dolist (za *z-alphas*)
                (when (> (abs z) (z-alpha za))
                      (return-from z-test (* (signum z) (z-level za)))))
        0.0))
(export 'z-test)

;; STRUCTURE OF A SAMPLE
(defmacro s-name  (sample) `(first ,sample))
(defmacro s-mean  (sample) `(second ,sample))
(defmacro s-stdev (sample) `(third ,sample))
(defmacro s-size  (sample) `(fourth ,sample))

(defun z-tests (samples)
  (mapcar #'(lambda (sample) (z-tests* sample samples)) samples))
(defun z-tests* (s1 samples)
  `(,(s-name s1)
    ,@(mapcar #'(lambda (s2)
                  (let ((z (z-test (s-mean s1) (s-stdev s1) (s-size s1)
                                   (s-mean s2) (s-stdev s2) (s-size s2))))
                       `(,(if (minusp z) '>
                              (if (plusp z) '< '=))
                         ,(s-name s2) ,(abs z))))
          samples)))
(export 'z-tests)


(export 'point-lineseg-distance)
(defun point-lineseg-distance (x y x1 y1 x2 y2)
 "Returns the euclidean distance between a point and a line segment"
 ; In the following, all variables labeled dist's are SQUARES of distances.
 ; The only tricky part here is figuring out whether to use the distance
 ; to the nearest point or the distance to the line defined by the line segment.
 ; This all depends on the angles (the ones touching the lineseg) of the triangle
 ; formed by the three points.  If the larger is obtuse we use nearest point,
 ; otherwise point-line.  We check for the angle being greater or less than
 ; 90 degrees with the famous right-triangle equality A^2 = B^2 + c^2.
 (let ((near-point-dist (point-point-distance-squared x y x1 y1))
       (far-point-dist (point-point-distance-squared x y x2 y2))
       (lineseg-dist (point-point-distance-squared x1 y1 x2 y2)))
   (if (< far-point-dist near-point-dist)
       (swap far-point-dist near-point-dist))
   (if (>= far-point-dist
           (+ near-point-dist lineseg-dist))
       (sqrt near-point-dist)
       (point-line-distance x y x1 y1 x2 y2))))

(export 'point-line-distance)
(defun point-line-distance (x y x1 y1 x2 y2)
  "Returns the euclidean distance between the first point and the line given by the
```

```
other two points"
  (if (= x1 x2)
      (abs (- x1 x))
      (let* ((slope (/ (- y2 y1)
                       (float (- x2 x1))))
             (intercept (- y1 (* slope
                                 x1))))
        (/ (abs (+ (* slope x)
                   (- y)
                   intercept))
           (sqrt (+ 1 (* slope slope)))))))

(export 'point-point-distance-squared)
(defun point-point-distance-squared (x1 y1 x2 y2)
  "Returns the square of the euclidean distance between two points"
  (+ (square (- x1 x2))
     (square (- y1 y2))))

(export 'point-point-distance)
(defun point-point-distance (x1 y1 x2 y2)
  "Returns the euclidean distance between two points"
  (sqrt (point-point-distance-squared x1 y1 x2 y2)))

(defun lv (vector) (loop for i below (length vector) collect (aref vector i)))

(defun l1 (vector)
  (lv vector))

(defun l2 (array)
  (loop for k below (array-dimension array 0) do
    (print (loop for j below (array-dimension array 1) collect (aref array k j))))
  (values))

(export 'l)
(defun l (array)
  (if (= 1 (array-rank array))
      (l1 array)
      (l2 array)))

(export 'subsample)
(defun subsample (bin-size l)
  "l is a list OR a list of lists"
  (if (listp (first l))
      (loop for list in l collect (subsample list bin-size))
      (loop while l
            for bin = (loop repeat bin-size while l collect (pop l))
            collect (mean bin))))

(export 'copy-of-standard-random-state)
(defun copy-of-standard-random-state ()
  (make-random-state #.(RANDOM-STATE 64497 9)))

(export 'permanent-data)
(export 'permanent-record-file)
(export 'record-fields)
(export 'record)
(export 'read-record-file)
(export 'record-value)
(export 'records)
(export 'my-time-stamp)
(export 'prepare-for-recording!)
(export 'prepare-for-recording)
```

```lisp
(defvar permanent-data nil)
(defvar permanent-record-file nil)
(defvar record-fields '(:day :hour :min :alpha :data))

(defun prepare-for-recording! (file-name &rest data-fields)
  (setq permanent-record-file file-name)
  (setq permanent-data nil)
  (setq record-fields (append '(:day :hour :min) data-fields))
  (with-open-file (file file-name
                        :direction :output
                        :if-exists :supersede
                        :if-does-not-exist :create)
    (format file "~A~%" (apply #'concatenate 'string "(:record-fields"
                               (append (loop for f in record-fields collect
                                             (concatenate 'string " :"
                                                          (format nil "~A" f)))
                                       (list ")"))))))

(defun record (&rest record-data)
  "Record data with time stamp in file and permanent-data"
  (let ((record (append (my-time-stamp) record-data)))
    (unless (= (length record) (length record-fields))
      (error "data does not match template "))
    (when permanent-record-file
      (with-open-file (file permanent-record-file
                            :direction :output
                            :if-exists :append
                            :if-does-not-exist :create)
        (format file "~A~%" record)))
    (push record permanent-data)
    record))

(defun read-record-file (&optional (file (choose-file-dialog)))
  "Load permanent-data from file"
  (with-open-file (file file :direction :input)
    (setq permanent-data
          (reverse (let ((first-read (read file nil nil))
                         (rest-read (loop for record = (read file nil nil)
                                          while record collect record)))
                     (cond ((null first-read))
                           ((eq (car first-read) :record-fields)
                            (setq record-fields (rest first-read))
                            rest-read)
                           (t (cons first-read rest-read))))))
    (setq permanent-record-file file)
    (cons (length permanent-data) record-fields)))

(defun record-value (record field)
  "extract the value of a particular field of a record"
  (unless (member field record-fields) (error "Bad field name"))
  (loop for f in record-fields
        for v in record
        until (eq f field)
        finally (return v)))

(defun records (&rest field-value-pairs)
  "extract all records from data that match the field-value pairs"
  (unless (evenp (length field-value-pairs)) (error "odd number of args to records"))
  (loop for f-v-list = field-value-pairs then (cddr f-v-list)
        while f-v-list
        for f = (first f-v-list)
        unless (member f record-fields) do (error "Bad field name"))
  (loop for record in (reverse permanent-data)
        when (loop for f-v-list = field-value-pairs then (cddr f-v-list)
                   while f-v-list
                   for f = (first f-v-list)
                   for v = (second f-v-list)
                   always (OR (equal v (record-value record f))
                              (ignore-errors (= v (record-value record f)))))
        collect record))

(defun my-time-stamp ()
  (multiple-value-bind (sec min hour day) (decode-universal-time (get-universal-
time))
    (declare (ignore sec))
    (list day hour min)))


;; For writing a list to a file for input to Cricket-Graph

(export 'write-for-graphing)

(defun write-for-graphing (data)
  (with-open-file (file "Macintosh HD:Desktop Folder:temp-graphing-data"
                        :direction :output
                        :if-exists :supersede
                        :if-does-not-exist :create)
    (if (atom (first data))
      (loop for d in data do (format file "~8,4F~%" d))
      (loop with num-rows = (length (first data))
            for row below num-rows
            do (loop for list in data do (format file "~8,4F    " (nth row list)))
            do (format file "~%")))))



(export 'standard-random-state)
(export 'standardize-random-state)
(export 'advance-random-state)

(defvar standard-random-state #.(RANDOM-STATE 64497 9))
#|
        #S(FUTURE-COMMON-LISP:RANDOM-STATE
            :ARRAY
            #(1323496585 1001191002 -587767537 -1071730568 -1147853915 -731089434
1865874377 -387582935
                         -1548911375 -52859678 1489907255 226907840 -1801820277
145270258 -1784780698 895203347
                         2101883890 756363165 -2047410492 1182268120 -1417582076 -
2101366199 -436910048 92474021
                         -850512131 -40946116 -723207257 429572592 -262857859
1972410780 -828461337 154333198
                         -2110101118 -1646877073 -1259707441 972398391 1375765096
240797851 -1042450772 -257783169
                         -1922575120 1037722597 -1774511059 1408209885 -1035031755
2143021556 785694559 1785244199
                         -586057545 216629327 -370552912 441425683 803899475 -
122403238 -2071490833 679238967
                         1666337352 984812380 501833545 1010617864 -1990258125 -
1465744262 869839181 -634081314
                         254104851 -129645892 -1542655512 1765669869 -1055430844 -
1069176569 -1400149912)
            :SIZE 71 :SEED 224772007 :POINTER-1 0 :POINTER-2 35))
|#
(defmacro standardize-random-state (&optional (random-state 'cl::*random-state*))
  `(setq ,random-state (make-random-state ut:standard-random-state)))

(defun advance-random-state (num-advances &optional (random-state *random-state*))
  (loop repeat num-advances do (random 2 random-state)))

(export 'firstn)
(defun firstn (n list)
  "Returns a list of the first n elements of list"
  (loop for e in list
        repeat n
        collect e))
```

```
; This is code to implement the Tic-Tac-Toe example in Chapter 1 of the
; book "Learning by Interacting". Read that chapter before trying to
; understand this code.


; States are lists of two lists and an index, e.g., ((1 2 3) (4 5 6) index),
; where the first list is the location of the X's and the second list is
; the location of the O's.   The index is into a large array holding the value
; of the states.  There is a one-to-one mapping from index to the lists.
; The locations refer not to the standard positions, but to the "magic square"
; positions:
;
;    2 9 4
;    7 5 3
;    6 1 8
;
; Labelling the locations of the Tic-Tac-Toe board in this way is useful because
; then we can just add up any three positions, and if the sum is 15, then we
; know they are three in a row.  The following function then tells us if a list
; of X or O positions contains any that are three in a row.

(defvar magic-square '(2 9 4 7 5 3 6 1 8))

(defun any-n-sum-to-k? (n k list)
  (cond ((= n 0)
         (= k 0))
        ((< k 0)
         nil)
        ((null list)
         nil)
        ((any-n-sum-to-k? (- n 1) (- k (first list)) (rest list))
         t)                             ; either the first element is included
        ((any-n-sum-to-k? n k (rest list))
         t)))                           ; or it's not

; This representation need not be confusing.  To see any state, print it with:

(defun show-state (state)
  (let ((X-moves (first state))
        (O-moves (second state)))
    (format t "~%")
    (loop for location in magic-square
          for i from 0
          do
          (format t (cond ((member location X-moves)
                           " X")
                          ((member location O-moves)
                           " O")
                          (t " -")))
          (when (= i 5) (format t "  ~,3F" (value state)))
          (when (= 2 (mod i 3)) (format t "~%"))))
  (values))


; The value function will be implemented as a big, mostly empty array.  Remember
; that a state is of the form (X-locations O-locations index), where the index
; is an index into the value array.  The index is computed from the locations.
; Basically, each side gets a bit for each position.  The bit is 1 is that side
; has played there.  The index is the integer with those bits on.  X gets the
; first (low-order) nine bits, O the second nine.  Here is the function that
; computes the indices:

(defvar powers-of-2
```

```lisp
  (make-array 10
              :initial-contents
              (cons nil (loop for i below 9 collect (expt 2 i)))))

(defun state-index (X-locations O-locations)
  (+ (loop for l in X-locations sum (aref powers-of-2 l))
     (* 512 (loop for l in O-locations sum (aref powers-of-2 l)))))

(defvar value-table)
(defvar initial-state)

(defun init ()
  (setq value-table (make-array (* 512 512) :initial-element nil))
  (setq initial-state '(nil nil 0))
  (set-value initial-state 0.5)
  (values))

(defun value (state)
  (aref value-table (third state)))

(defun set-value (state value)
  (setf (aref value-table (third state)) value))

(defun next-state (player state move)
  "returns new state after making the indicated move by the indicated player"
  (let ((X-moves (first state))
        (O-moves (second state)))
    (if (eq player :X)
      (push move X-moves)
      (push move O-moves))
    (setq state (list X-moves O-moves (state-index X-moves O-moves)))
    (when (null (value state))
      (set-value state (cond ((any-n-sum-to-k? 3 15 X-moves)
                              0)
                             ((any-n-sum-to-k? 3 15 O-moves)
                              1)
                             ((= 9 (+ (length X-moves) (length O-moves)))
                              0)
                             (t 0.5))))
    state))


(defun terminal-state-p (state)
  (integerp (value state)))

(defvar alpha 0.5)
(defvar epsilon 0.01)

(defun possible-moves (state)
  "Returns a list of unplayed locations"
  (loop for i from 1 to 9
        unless (or (member i (first state))
                   (member i (second state)))
        collect i))


(defun random-move (state)
  "Returns one of the unplayed locations, selected at random"
  (let ((possible-moves (possible-moves state)))
    (if (null possible-moves)
      nil
      (nth (random (length possible-moves))
           possible-moves))))
```

```lisp
(defun greedy-move (player state)
  "Returns the move that, when played, gives the highest valued position"
  (let ((possible-moves (possible-moves state)))
    (if (null possible-moves)
      nil
      (loop with best-value = -1
            with best-move
            for move in possible-moves
            for move-value = (value (next-state player state move))
            do (when (> move-value best-value)
                 (setf best-value move-value)
                 (setf best-move move))
            finally (return best-move)))))

; Now here is the main function

(defvar state)

(defun game (&optional quiet)
  "Plays 1 game against the random player. Also learns and prints.
   :X moves first and is random.  :O learns"
  (setq state initial-state)
  (unless quiet (show-state state))
  (loop for new-state = (next-state :X state (random-move state))
        for exploratory-move? = (< (random 1.0) epsilon)
        do
        (when (terminal-state-p new-state)
          (unless quiet (show-state new-state))
          (update state new-state quiet)
          (return (value new-state)))
        (setf new-state (next-state :O new-state
                                    (if exploratory-move?
                                      (random-move new-state)
                                      (greedy-move :O new-state))))
        (unless exploratory-move?
          (update state new-state quiet))
        (unless quiet (show-state new-state))
        (when (terminal-state-p new-state) (return (value new-state)))
        (setq state new-state)))

(defun update (state new-state &optional quiet)
  "This is the learning rule"
  (set-value state (+ (value state)
                      (* alpha
                         (- (value new-state)
                            (value state)))))
  (unless quiet (format t "                    ~,3F" (value state))))

(defun run ()
  (loop repeat 40 do (print (/ (loop repeat 100 sum (game t))
                                100.0))))

(defun runs (num-runs num-bins bin-size)   ; e.g., (runs 10 40 100)
  (loop with array = (make-array num-bins :initial-element 0.0)
        repeat num-runs do
        (init)
        (loop for i below num-bins do
              (incf (aref array i)
                    (loop repeat bin-size sum (game t))))
        finally (loop for i below num-bins
                      do (print (/ (aref array i)
                                   (* bin-size num-runs))))))
```

```
; To run, call (setup), (init), and then, e.g., (runs 2000 1000 .1)

(defvar n)
(defvar epsilon .1)
(defvar Q*)
(defvar Q)
(defvar n_a)
(defvar randomness)
(defvar max-num-tasks 2000)

(defun setup ()
  (setq n 10)
  (setq Q (make-array n))
  (setq n_a (make-array n))
  (setq Q* (make-array (list n max-num-tasks)))
  (setq randomness (make-array max-num-tasks))
  (standardize-random-state)
  (advance-random-state 0)
  (loop for task below max-num-tasks do
        (loop for a below n do
              (setf (aref Q* a task) (random-normal)))
        (setf (aref randomness task)
              (make-random-state))))

(defun init ()
  (loop for a below n do
        (setf (aref Q a) 0.0)
        (setf (aref n_a a) 0)))

(defun runs (&optional (num-runs 1000) (num-steps 100) (epsilon 0))
  (loop with average-reward = (make-list num-steps :initial-element 0.0)
        with prob-a* = (make-list num-steps :initial-element 0.0)
        for run-num below num-runs
        for a* = 0
        do (loop for a from 1 below n
                 when (> (aref Q* a run-num)
                         (aref Q* a* run-num))
                 do (setq a* a))
        do (init)
        do (setq *random-state* (aref randomness run-num))
        collect (loop for time-step below num-steps
                      for a = (epsilon-greedy epsilon)
                      for r = (reward a run-num)
                      do (learn a r)
                      do (incf (nth time-step average-reward) r)
                      do (when (= a a*) (incf (nth time-step prob-a*))))
        finally (return (loop for i below num-steps
                              do (setf (nth i average-reward)
                                       (/ (nth i average-reward)
                                          num-runs))
                              do (setf (nth i prob-a*)
                                       (/ (nth i prob-a*)
                                          (float num-runs)))
                              finally (return (values average-reward prob-a*))))))

(defun learn (a r)
  (incf (aref n_a a))
  (incf (aref Q a) (/ (- r (aref Q a))
                      (aref n_a a))))

(defun reward (a task-num)
  (+ (aref Q* a task-num)
     (random-normal)))
```

```
(defun epsilon-greedy (epsilon)
  (with-prob epsilon
    (random n)
    (arg-max-random-tiebreak Q)))

(defun greedy ()
  (arg-max-random-tiebreak Q))

(defun arg-max-random-tiebreak (array)
  "Returns index to first instance of the largest value in the array"
  (loop with best-args = (list 0)
        with best-value = (aref array 0)
        for i from 1 below (length array)
        for value = (aref array i)
        do (cond ((< value best-value))
                 ((> value best-value)
                  (setq best-value value)
                  (setq best-args (list i)))
                 ((= value best-value)
                  (push i best-args)))
        finally (return (values (nth (random (length best-args))
                                     best-args)
                                best-value))))

(defun max-Q* (num-tasks)
  (mean (loop for task below num-tasks
              collect (loop for a below n
                            maximize (aref Q* a task)))))
```

```
(defvar n)
(defvar epsilon .1)
(defvar Q*)
(defvar Q)
(defvar n_a)
(defvar randomness)
(defvar max-num-tasks 2000)

(defun setup ()
  (setq n 10)
  (setq Q (make-array n))
  (setq n_a (make-array n))
  (setq Q* (make-array (list n max-num-tasks)))
  (setq randomness (make-array max-num-tasks))
  (standardize-random-state)
  (advance-random-state 0)
  (loop for task below max-num-tasks do
        (loop for a below n do
              (setf (aref Q* a task) (random-normal)))
        (setf (aref randomness task)
              (make-random-state))))

(defun init ()
  (loop for a below n do
        (setf (aref Q a) 0.0)
        (setf (aref n_a a) 0)))

(defun runs (&optional (num-runs 1000) (num-steps 100) (temperature 1))
  (loop with average-reward = (make-list num-steps :initial-element 0.0)
        with prob-a* = (make-list num-steps :initial-element 0.0)
        for run-num below num-runs
        for a* = 0
        do (format t " ~A" run-num)
        do (loop for a from 1 below n
                 when (> (aref Q* a run-num)
                         (aref Q* a* run-num))
                 do (setq a* a))
        do (init)
        do (setq *random-state* (aref randomness run-num))
        collect (loop for time-step below num-steps
                      for a = (policy temperature)
                      for r = (reward a run-num)
                      do (learn a r)
                      do (incf (nth time-step average-reward) r)
                      do (when (= a a*) (incf (nth time-step prob-a*))))
        finally (return (loop for i below num-steps
                              do (setf (nth i average-reward)
                                       (/ (nth i average-reward)
                                          num-runs))
                              do (setf (nth i prob-a*)
                                       (/ (nth i prob-a*)
                                          (float num-runs)))
                              finally (record num-runs num-steps :av-soft temperature
                                              average-reward prob-a*)))))

(defun policy (temperature)
  "Returns soft-max action selection"
  (loop for a below n
        for value = (aref Q a)
        sum (exp (/ value temperature)) into total-sum
        collect total-sum into partial-sums
        finally (return
                 (loop with rand = (random (float total-sum))
                       for partial-sum in partial-sums
                       for a from 0
                       until (> partial-sum rand)
                       finally (return a)))))


(defun learn (a r)
  (incf (aref n_a a))
  (incf (aref Q a) (/ (- r (aref Q a))
                      (aref n_a a))))

(defun reward (a task-num)
  (+ (aref Q* a task-num)
     (random-normal)))

(defun epsilon-greedy (epsilon)
  (with-prob epsilon
    (random n)
    (arg-max-random-tiebreak Q)))

(defun greedy ()
  (arg-max-random-tiebreak Q))

(defun arg-max-random-tiebreak (array)
  "Returns index to first instance of the largest value in the array"
  (loop with best-args = (list 0)
        with best-value = (aref array 0)
        for i from 1 below (length array)
        for value = (aref array i)
        do (cond ((< value best-value))
                 ((> value best-value)
                  (setq best-value value)
                  (setq best-args (list i)))
                 ((= value best-value)
                  (push i best-args)))
        finally (return (values (nth (random (length best-args))
                                     best-args)
                                best-value))))

(defun max-Q* (num-tasks)
  (mean (loop for task below num-tasks
              collect (loop for a below n
                            maximize (aref Q* a task)))))
```

```lisp
;-*- Mode: Lisp; Package: (bandits :use (common-lisp ccl ut)) -*-

(defvar n)
(defvar epsilon .1)
(defvar alpha .1)
(defvar QQ*)
(defvar QQ)
(defvar n_a)
(defvar randomness)
(defvar max-num-tasks 2)
(defvar rbar)
(defvar timetime)

(defun setup ()
  (setq n 2)
  (setq QQ (make-array n))
  (setq n_a (make-array n))
  (setq QQ* (make-array (list n max-num-tasks)
                        :initial-contents '((.1 .8) (.2 .9)))))


(defun init (algorithm)
  (loop for a below n do
        (setf (aref QQ a) (ecase algorithm
                            ((:rc :action-values) 0.0)
                            (:sl 0)
                            ((:Lrp :Lri) 0.5)))
        (setf (aref n_a a) 0))
  (setq rbar 0.0)
  (setq timetime 0))

(defun runs (task algorithm &optional (num-runs 2000) (num-steps 1000))
  "algorithm is one of :sl :action-values :Lrp :Lrp :rc"
  (standardize-random-state)
  (loop with average-reward = (make-list num-steps :initial-element 0.0)
        with prob-a* = (make-list num-steps :initial-element 0.0)
        with a* = (if (> (aref QQ* 0 task) (aref QQ* 1 task)) 0 1)
        for run-num below num-runs
        do (init algorithm)
        collect (loop for timetime-step below num-steps
                      for a = (policy algorithm)
                      for r = (reward a task)
                      do (learn algorithm a r)
                      do (incf (nth timetime-step average-reward) r)
                      do (when (= a a*) (incf (nth timetime-step prob-a*))))
        finally (return
                 (loop for i below num-steps
                       do (setf (nth i average-reward)
                                (/ (nth i average-reward)
                                   num-runs))
                       do (setf (nth i prob-a*)
                                (/ (nth i prob-a*)
                                   (float num-runs)))
                       finally (return (values average-reward prob-a*))))))

(defun policy (algorithm)
  (ecase algorithm
    ((:rc :action-values)
     (epsilon-greedy epsilon))
    (:sl
     (greedy))
    ((:Lrp :Lri)
     (with-prob (aref QQ 0) 0 1))))
```

```
(defun learn (algorithm a r)
  (ecase algorithm
    (:rc
     (incf timetime)
     (incf rbar (/ (- r rbar)
                   timetime))
     (incf (aref QQ a) (- r rbar)))
    (:action-values
     (incf (aref n_a a))
     (incf (aref QQ a) (/ (- r (aref QQ a))
                         (aref n_a a))))
    (:sl
     (incf (aref QQ (if (= r 1) a (- 1 a)))))
    ((:Lrp :Lri)
     (unless (and (= r 0) (eq algorithm :Lri))
       (let* ((target-action (if (= r 1) a (- 1 a)))
              (other-action (- 1 target-action)))
         (incf (aref QQ target-action)
               (* alpha (- 1 (aref QQ target-action))))
         (setf (aref QQ other-action)
               (- 1 (aref QQ target-action))))))))

(defun reward (a task-num)
  (with-prob (aref QQ* a task-num)
    1 0))

(defun epsilon-greedy (epsilon)
  (with-prob epsilon
    (random n)
    (arg-max-random-tiebreak QQ)))

(defun greedy ()
  (arg-max-random-tiebreak QQ))

(defun arg-max-random-tiebreak (array)
  "Returns index to first instance of the largest value in the array"
  (loop with best-args = (list 0)
        with best-value = (aref array 0)
        for i from 1 below (length array)
        for value = (aref array i)
        do (cond ((< value best-value))
                 ((> value best-value)
                  (setq best-value value)
                  (setq best-args (list i)))
                 ((= value best-value)
                  (push i best-args)))
        finally (return (values (nth (random (length best-args))
                                     best-args)
                                best-value))))

(defun max-QQ* (num-tasks)
  (mean (loop for task below num-tasks
              collect (loop for a below n
                            maximize (aref QQ* a task)))))
```

```lisp
(defvar n)
(defvar epsilon .1)
(defvar Q*)
(defvar Q)
(defvar n_a)
(defvar randomness)
(defvar max-num-tasks 2000)
(defvar alpha 0.1)

(defun setup ()
  (setq n 10)
  (setq Q (make-array n))
  (setq n_a (make-array n))
  (setq Q* (make-array (list n max-num-tasks)))
  (setq randomness (make-array max-num-tasks))
  (standardize-random-state)
  (advance-random-state 0)
  (loop for task below max-num-tasks do
        (loop for a below n do
              (setf (aref Q* a task) (random-normal)))
        (setf (aref randomness task)
              (make-random-state))))

(defun init ()
  (loop for a below n do
        (setf (aref Q a) 0.0)
        (setf (aref n_a a) 0)))

(defun runs (&optional (num-runs 1000) (num-steps 100) (epsilon 0))
  (loop with average-reward = (make-list num-steps :initial-element 0.0)
        with prob-a* = (make-list num-steps :initial-element 0.0)
        for run-num below num-runs
        for a* = 0
        do (loop for a from 1 below n
                 when (> (aref Q* a run-num)
                         (aref Q* a* run-num))
                 do (setq a* a))
        do (format t "~A " run-num)
        do (init)
        do (setq *random-state* (aref randomness run-num))
        collect (loop for time-step below num-steps
                      for a = (epsilon-greedy epsilon)
                      for r = (reward a run-num)
                      do (learn a r)
                      do (incf (nth time-step average-reward) r)
                      do (when (= a a*) (incf (nth time-step prob-a*))))
        finally (return (loop for i below num-steps
                              do (setf (nth i average-reward)
                                       (/ (nth i average-reward)
                                          num-runs))
                              do (setf (nth i prob-a*)
                                       (/ (nth i prob-a*)
                                          (float num-runs)))
                              finally (record num-runs num-steps :avi epsilon
                                              average-reward prob-a*)))))

(defun learn (a r)
  (incf (aref Q a) (* alpha (- r (aref Q a)))))

(defun reward (a task-num)
  (+ (aref Q* a task-num)
     (random-normal)))
```

```
(defun epsilon-greedy (epsilon)
  (with-prob epsilon
    (random n)
    (arg-max-random-tiebreak Q)))

(defun greedy ()
  (arg-max-random-tiebreak Q))

(defun arg-max-random-tiebreak (array)
  "Returns index to first instance of the largest value in the array"
  (loop with best-args = (list 0)
        with best-value = (aref array 0)
        for i from 1 below (length array)
        for value = (aref array i)
        do (cond ((< value best-value))
                 ((> value best-value)
                  (setq best-value value)
                  (setq best-args (list i)))
                 ((= value best-value)
                  (push i best-args)))
        finally (return (values (nth (random (length best-args))
                                     best-args)
                                best-value))))

(defun max-Q* (num-tasks)
  (mean (loop for task below num-tasks
              collect (loop for a below n
                            maximize (aref Q* a task)))))
```

```
(defvar n)
(defvar epsilon .1)
(defvar Q*)
(defvar Q)
(defvar n_a)
(defvar randomness)
(defvar max-num-tasks 2000)
(defvar alpha 0.1)

(defun setup ()
  (setq n 10)
  (setq Q (make-array n))
  (setq n_a (make-array n))
  (setq Q* (make-array (list n max-num-tasks)))
  (setq randomness (make-array max-num-tasks))
  (standardize-random-state)
  (advance-random-state 0)
  (loop for task below max-num-tasks do
        (loop for a below n do
              (setf (aref Q* a task) (random-normal)))
        (setf (aref randomness task)
              (make-random-state))))

(defvar Q0)
(defun init ()
  (loop for a below n do
        (setf (aref Q a) Q0)
        (setf (aref n_a a) 0)))

(defun runs (&optional (num-runs 1000) (num-steps 100) (epsilon 0))
  (loop with average-reward = (make-list num-steps :initial-element 0.0)
        with prob-a* = (make-list num-steps :initial-element 0.0)
        for run-num below num-runs
        for a* = 0
        do (loop for a from 1 below n
                 when (> (aref Q* a run-num)
                         (aref Q* a* run-num))
                 do (setq a* a))
        do (format t "~A " run-num)
        do (init)
        do (setq *random-state* (aref randomness run-num))
        collect (loop for time-step below num-steps
                      for a = (epsilon-greedy epsilon)
                      for r = (reward a run-num)
                      do (learn a r)
                      do (incf (nth time-step average-reward) r)
                      do (when (= a a*) (incf (nth time-step prob-a*))))
        finally (return (loop for i below num-steps
                              do (setf (nth i average-reward)
                                       (/ (nth i average-reward)
                                          num-runs))
                              do (setf (nth i prob-a*)
                                       (/ (nth i prob-a*)
                                          (float num-runs)))
                              finally (record num-runs num-steps :avi-opt Q0
                                              average-reward prob-a*)))))

(defun learn (a r)
  (incf (aref Q a) (* alpha (- r (aref Q a)))))

(defun reward (a task-num)
  (+ (aref Q* a task-num)
     (random-normal)))

(defun epsilon-greedy (epsilon)
  (with-prob epsilon
    (random n)
    (arg-max-random-tiebreak Q)))

(defun greedy ()
  (arg-max-random-tiebreak Q))

(defun arg-max-random-tiebreak (array)
  "Returns index to first instance of the largest value in the array"
  (loop with best-args = (list 0)
        with best-value = (aref array 0)
        for i from 1 below (length array)
        for value = (aref array i)
        do (cond ((< value best-value))
                 ((> value best-value)
                  (setq best-value value)
                  (setq best-args (list i)))
                 ((= value best-value)
                  (push i best-args)))
        finally (return (values (nth (random (length best-args))
                                     best-args)
                                best-value))))

(defun max-Q* (num-tasks)
  (mean (loop for task below num-tasks
              collect (loop for a below n
                            maximize (aref Q* a task)))))
```

```
(defvar n)
(defvar epsilon .1)
(defvar Q*)
(defvar Q)
(defvar n_a)
(defvar randomness)
(defvar max-num-tasks 2000)
(defvar rbar)
(defvar time)

(defun setup ()
  (setq n 10)
  (setq Q (make-array n))
  (setq n_a (make-array n))
  (setq Q* (make-array (list n max-num-tasks)))
  (setq randomness (make-array max-num-tasks))
  (standardize-random-state)
  (advance-random-state 0)
  (loop for task below max-num-tasks do
        (loop for a below n do
              (setf (aref Q* a task) (random-normal)))
        (setf (aref randomness task)
              (make-random-state))))

(defun init ()
  (loop for a below n do
        (setf (aref Q a) 0.0)
        (setf (aref n_a a) 0))
  (setq rbar 0.0)
  (setq time 0))

(defun runs (&optional (num-runs 1000) (num-steps 100) (epsilon 0))
  (loop with average-reward = (make-list num-steps :initial-element 0.0)
        with prob-a* = (make-list num-steps :initial-element 0.0)
        for run-num below num-runs
        for a* = 0
        do (loop for a from 1 below n
                 when (> (aref Q* a run-num)
                         (aref Q* a* run-num))
                 do (setq a* a))
        do (format t " ~A" run-num)
;       do (print a*)
;       do (print (loop for a below n collect (aref Q* a run-num)))
        do (init)
        do (setq *random-state* (aref randomness run-num))
        collect (loop for time-step below num-steps
                      for a-greedy = (arg-max-random-tiebreak Q)
                      for a = (with-prob epsilon (random n) a-greedy)
                      for prob-a = (+ (* epsilon (/ n))
                                      (if (= a a-greedy) (- 1 epsilon) 0))
                      for r = (reward a run-num)
;                     do (format t "~%a:~A prob-a:~,3F r:~,3F rbar:~,3F Q:~,3F " a
prob-a r rbar (aref Q a))
                      do (learn a r prob-a)
;                     do (format t "Q:~,3F " (aref Q a))
                      do (incf (nth time-step average-reward) r)
                      do (when (= a a*) (incf (nth time-step prob-a*))))
        finally (return (loop for i below num-steps
                              do (setf (nth i average-reward)
                                       (/ (nth i average-reward)
                                          num-runs))
                              do (setf (nth i prob-a*)
                                       (/ (nth i prob-a*)
                                          (float num-runs)))
                              finally (record num-runs num-steps :rc epsilon
                                              average-reward prob-a*)))))

(defun learn (a r prob-a)
;  (incf (aref n_a a))
  (incf time)
  (incf rbar (* .1 (- r rbar)))
  (incf (aref Q a) (* (- r rbar)
                      (- 1 prob-a))))

(defun reward (a task-num)
  (+ (aref Q* a task-num)
     (random-normal)))

(defun epsilon-greedy (epsilon)
  (with-prob epsilon
    (random n)
    (arg-max-random-tiebreak Q)))

(defun greedy ()
  (arg-max-random-tiebreak Q))

(defun arg-max-random-tiebreak (array)
  "Returns index to first instance of the largest value in the array"
  (loop with best-args = (list 0)
        with best-value = (aref array 0)
        for i from 1 below (length array)
        for value = (aref array i)
        do (cond ((< value best-value))
                 ((> value best-value)
                  (setq best-value value)
                  (setq best-args (list i)))
                 ((= value best-value)
                  (push i best-args)))
        finally (return (values (nth (random (length best-args))
                                     best-args)
                                best-value))))

(defun max-Q* (num-tasks)
  (mean (loop for task below num-tasks
              collect (loop for a below n
                            maximize (aref Q* a task)))))

(defun prob-a* (&rest field-value-pairs)
    (loop for d in (apply #'records field-value-pairs)
          collect (record-value d :prob-a*)))
```

```lisp
(defvar n)
(defvar epsilon .1)
(defvar Q*)
(defvar Q)
(defvar n_a)
(defvar randomness)
(defvar max-num-tasks 2000)
(defvar rbar)
(defvar time)
(defvar abar)

(defun setup ()
  (setq n 10)
  (setq Q (make-array n))
  (setq n_a (make-array n))
  (setq abar (make-array n))
  (setq Q* (make-array (list n max-num-tasks)))
  (setq randomness (make-array max-num-tasks))
  (standardize-random-state)
  (advance-random-state 0)
  (loop for task below max-num-tasks do
        (loop for a below n do
              (setf (aref Q* a task) (random-normal)))
        (setf (aref randomness task)
              (make-random-state))))

(defun init ()
  (loop for a below n do
        (setf (aref Q a) 0.0)
        (setf (aref abar a) (/ 1.0 n))
        (setf (aref n_a a) 0))
  (setq rbar 0.0)
  (setq time 0))

(defun runs (&optional (num-runs 1000) (num-steps 100) (temperature 1))
  (loop with average-reward = (make-list num-steps :initial-element 0.0)
        with prob-a* = (make-list num-steps :initial-element 0.0)
        for run-num below num-runs
        for a* = 0
        do (loop for a from 1 below n
                 when (> (aref Q* a run-num)
                         (aref Q* a* run-num))
                 do (setq a* a))
        do (format t " ~A" run-num)
;       do (print a*)
;       do (print (loop for a below n collect (aref Q* a run-num)))
        do (init)
        do (setq *random-state* (aref randomness run-num))
        collect (loop for time-step below num-steps
                      with r
                      do (multiple-value-bind (a prob-a) (policy temperature)
                           (setq r (reward a run-num))
;                          (format t "~%a:~A prob-a:~,3F r:~,3F rbar:~,3F Q:~,3F " a
prob-a r rbar (aref Q a))
                           (learn a r prob-a)
;                          (format t "Q:~,3F " (aref Q a))
                           (incf (nth time-step average-reward) r)
                           (when (= a a*) (incf (nth time-step prob-a*)))))
        finally (return (loop for i below num-steps
                              do (setf (nth i average-reward)
                                       (/ (nth i average-reward)
                                          num-runs))
                              do (setf (nth i prob-a*)
                                       (/ (nth i prob-a*)
                                          (float num-runs)))
                              finally (record num-runs num-steps "rc-2soft"
temperature
                                              average-reward prob-a*)))))

(defun policy (temperature)
  "Returns action and is probabilitity of being selected"
  (loop for a below n
        for value = (aref Q a)
        sum (exp (/ value temperature)) into total-sum
        collect total-sum into partial-sums
        finally (return
                 (loop with rand = (random (float total-sum))
                       for last-partial = 0 then partial-sum
                       for partial-sum in partial-sums
                       for a from 0
                       until (> partial-sum rand)
                       finally (return (values a (/ (- partial-sum last-partial)
                                                    total-sum)))))))


(defun learn (a r prob-a)
  (incf (aref Q a) (* (- r rbar)
                      (- 1 (aref abar a))))
  (incf rbar (* .1 (- r rbar)))
  (loop for b below n do
        (incf (aref abar b) (* .1 (- (if (= a b) 1 0)
                                     (aref abar b))))))

(defun reward (a task-num)
  (+ (aref Q* a task-num)
     (random-normal)))

(defun epsilon-greedy (epsilon)
  (with-prob epsilon
    (random n)
    (arg-max-random-tiebreak Q)))

(defun greedy ()
  (arg-max-random-tiebreak Q))

(defun arg-max-random-tiebreak (array)
  "Returns index to first instance of the largest value in the array"
  (loop with best-args = (list 0)
        with best-value = (aref array 0)
        for i from 1 below (length array)
        for value = (aref array i)
        do (cond ((< value best-value))
                 ((> value best-value)
                  (setq best-value value)
                  (setq best-args (list i)))
                 ((= value best-value)
                  (push i best-args)))
        finally (return (values (nth (random (length best-args))
                                     best-args)
                                best-value))))

(defun max-Q* (num-tasks)
  (mean (loop for task below num-tasks
              collect (loop for a below n
                            maximize (aref Q* a task)))))
```

```lisp
(defvar n)
(defvar epsilon .1)
(defvar Q*)
(defvar Q)
(defvar n_a)
(defvar randomness)
(defvar max-num-tasks 2000)
(defvar rbar)
(defvar time)
(defvar abar)

(defun setup ()
  (setq n 10)
  (setq Q (make-array n))
  (setq n_a (make-array n))
  (setq abar (make-array n))
  (setq Q* (make-array (list n max-num-tasks)))
  (setq randomness (make-array max-num-tasks))
  (standardize-random-state)
  (advance-random-state 0)
  (loop for task below max-num-tasks do
        (loop for a below n do
              (setf (aref Q* a task) (random-normal)))
        (setf (aref randomness task)
              (make-random-state))))

(defun init ()
  (loop for a below n do
        (setf (aref Q a) 0.0)
        (setf (aref abar a) (/ 1.0 n))
        (setf (aref n_a a) 0))
  (setq rbar 0.0)
  (setq time 0))

(defun runs (&optional (num-runs 1000) (num-steps 100) (temperature 1))
  (loop with average-reward = (make-list num-steps :initial-element 0.0)
        with prob-a* = (make-list num-steps :initial-element 0.0)
        for run-num below num-runs
        for a* = 0
        do (loop for a from 1 below n
                 when (> (aref Q* a run-num)
                         (aref Q* a* run-num))
                 do (setq a* a))
        do (format t " ~A" run-num)
;       do (print a*)
;       do (print (loop for a below n collect (aref Q* a run-num)))
        do (init)
        do (setq *random-state* (aref randomness run-num))
        collect (loop for time-step below num-steps
                      with r
                      do (multiple-value-bind (a) (policy temperature)
                           (setq r (reward a run-num))
;                          (format t "~%a:~A prob-a:~,3F r:~,3F rbar:~,3F Q:~,3F " a
prob-a r rbar (aref Q a))
                           (learn a r)
;                          (format t "Q:~,3F " (aref Q a))
                           (incf (nth time-step average-reward) r)
                           (when (= a a*) (incf (nth time-step prob-a*)))))
        finally (return (loop for i below num-steps
                              do (setf (nth i average-reward)
                                       (/ (nth i average-reward)
                                          num-runs))
                              do (setf (nth i prob-a*)
                                       (/ (nth i prob-a*)
                                          (float num-runs)))
                              finally (record num-runs num-steps :rc-noelig
temperature
                                              average-reward prob-a*)))))

(defun policy (temperature)
  "Returns action and its probabilitity of being selected"
  (loop for a below n
        for value = (aref Q a)
        sum (exp (/ value temperature)) into total-sum
        collect total-sum into partial-sums
        finally (return
                 (loop with rand = (random (float total-sum))
                       for partial-sum in partial-sums
                       for a from 0
                       until (> partial-sum rand)
                       finally (return (values a))))))


(defun learn (a r)
;  (loop for b below n do
;        (incf (aref abar b) (* .1 (- (if (= a b) 1 0)
;                                      (aref abar b)))))
;  (incf (aref Q a) (* (- r rbar)
;                      (- 1 (aref abar a))))
  (incf (aref Q a) (- r rbar))
  (incf rbar (* .1 (- r rbar))))

(defun reward (a task-num)
  (+ (aref Q* a task-num)
     (random-normal)))

(defun epsilon-greedy (epsilon)
  (with-prob epsilon
    (random n)
    (arg-max-random-tiebreak Q)))

(defun greedy ()
  (arg-max-random-tiebreak Q))

(defun arg-max-random-tiebreak (array)
  "Returns index to first instance of the largest value in the array"
  (loop with best-args = (list 0)
        with best-value = (aref array 0)
        for i from 1 below (length array)
        for value = (aref array i)
        do (cond ((< value best-value))
                 ((> value best-value)
                  (setq best-value value)
                  (setq best-args (list i)))
                 ((= value best-value)
                  (push i best-args)))
        finally (return (values (nth (random (length best-args))
                                     best-args)
                                best-value))))

(defun max-Q* (num-tasks)
  (mean (loop for task below num-tasks
              collect (loop for a below n
                            maximize (aref Q* a task)))))
```

```
(defvar n)
(defvar epsilon .1)
(defvar Q*)
(defvar Q)
(defvar p)
(defvar n_a)
(defvar randomness)
(defvar max-num-tasks 2000)

(defun setup ()
  (setq n 10)
  (setq Q (make-array n))
  (setq p (make-array n))
  (setq n_a (make-array n))
  (setq Q* (make-array (list n max-num-tasks)))
  (setq randomness (make-array max-num-tasks))
  (standardize-random-state)
  (advance-random-state 0)
  (loop for task below max-num-tasks do
        (loop for a below n do
              (setf (aref Q* a task) (random-normal)))
        (setf (aref randomness task)
              (make-random-state))))

(defun init ()
  (loop for a below n do
        (setf (aref Q a) 0.0)
        (setf (aref P a) (/ 1.0 n))
        (setf (aref n_a a) 0)))

(defun runs (&optional (num-runs 1000) (num-steps 100) (beta 0))
  (loop with average-reward = (make-list num-steps :initial-element 0.0)
        with prob-a* = (make-list num-steps :initial-element 0.0)
        for run-num below num-runs
        for a* = 0
        do (format t " ~A" run-num)
        do (loop for a from 1 below n
                 when (> (aref Q* a run-num)
                         (aref Q* a* run-num))
                 do (setq a* a))
        do (init)
        do (setq *random-state* (aref randomness run-num))
        do (loop for time-step below num-steps
                 for a = (policy)
                 for r = (reward a run-num)
                 do (learn a r beta)
                 do (incf (nth time-step average-reward) r)
                 do (when (= a a*) (incf (nth time-step prob-a*))))
        finally (loop for i below num-steps
                      do (setf (nth i average-reward)
                               (/ (nth i average-reward)
                                  num-runs))
                      do (setf (nth i prob-a*)
                               (/ (nth i prob-a*)
                                  (float num-runs)))
                      finally (record num-runs num-steps "av-pursuit"
                                      beta average-reward prob-a*))))

(defun policy ()
  (loop with rand = (random 1.0)
        for a below n
        sum (aref p a) into partial-sum
        until (>= partial-sum rand)
        finally (return a)))

(defun learn (a r beta)
  (incf (aref n_a a))
  (incf (aref Q a) (/ (- r (aref Q a))
                      (aref n_a a)))
  (loop for a below n
        do (decf (aref p a) (* beta (aref p a))))
  (incf (aref p (arg-max-random-tiebreak Q)) beta))

(defun reward (a task-num)
  (+ (aref Q* a task-num)
     (random-normal)))

(defun epsilon-greedy (epsilon)
  (with-prob epsilon
    (random n)
    (arg-max-random-tiebreak Q)))

(defun greedy ()
  (arg-max-random-tiebreak Q))

(defun arg-max-random-tiebreak (array)
  "Returns index to first instance of the largest value in the array"
  (loop with best-args = (list 0)
        with best-value = (aref array 0)
        for i from 1 below (length array)
        for value = (aref array i)
        do (cond ((< value best-value))
                 ((> value best-value)
                  (setq best-value value)
                  (setq best-args (list i)))
                 ((= value best-value)
                  (push i best-args)))
        finally (return (values (nth (random (length best-args))
                                     best-args)
                                best-value))))

(defun max-Q* (num-tasks)
  (mean (loop for task below num-tasks
              collect (loop for a below n
                            maximize (aref Q* a task)))))
```

```
/*----------------------------------------------------------------------
    This file contains a simulation of the cart and pole dynamic system and
 a procedure for learning to balance the pole.  Both are described in
 Barto, Sutton, and Anderson, "Neuronlike Adaptive Elements That Can Solve
 Difficult Learning Control Problems," IEEE Trans. Syst., Man, Cybern.,
 Vol. SMC-13, pp. 834--846, Sept.--Oct. 1983, and in Sutton, "Temporal
 Aspects of Credit Assignment in Reinforcement Learning", PhD
 Dissertation, Department of Computer and Information Science, University
 of Massachusetts, Amherst, 1984.  The following routines are included:

       main:              controls simulation interations and implements
                          the learning system.

       cart_and_pole:     the cart and pole dynamics; given action and
                          current state, estimates next state

       get_box:           The cart-pole's state space is divided into 162
                          boxes.  get_box returns the index of the box into
                          which the current state appears.

 These routines were written by Rich Sutton and Chuck Anderson.  Claude Sammut
 translated parts from Fortran to C.  Please address correspondence to
 sutton@gte.com or anderson@cs.colostate.edu
---------------------------------------
Changes:
  1/93: A bug was found and fixed in the state -> box mapping which resulted
        in array addressing outside the range of the array.  It's amazing this
        program worked at all before this bug was fixed.  -RSS
----------------------------------------------------------------------*/

#include <math.h>

#define min(x, y)               ((x <= y) ? x : y)
#define max(x, y)               ((x >= y) ? x : y)
#define prob_push_right(s)      (1.0 / (1.0 + exp(-max(-50.0, min(s, 50.0)))))
#define random                  ((float) rand() / (float)((1 << 31) - 1))

#define N_BOXES         162         /* Number of disjoint boxes of state space. */
#define ALPHA           1000        /* Learning rate for action weights, w. */
#define BETA            0.5         /* Learning rate for critic weights, v. */
#define GAMMA           0.95        /* Discount factor for critic. */
#define LAMBDAw         0.9         /* Decay rate for w eligibility trace. */
#define LAMBDAv         0.8         /* Decay rate for v eligibility trace. */

#define MAX_FAILURES     100         /* Termination criterion. */
#define MAX_STEPS        100000

typedef float vector[N_BOXES];

main()
{
  float x,                      /* cart position, meters */
        x_dot,                  /* cart velocity */
        theta,                  /* pole angle, radians */
        theta_dot;              /* pole angular velocity */
  vector  w,                    /* vector of action weights */
          v,                    /* vector of critic weights */
          e,                    /* vector of action weight eligibilities */
          xbar;                 /* vector of critic weight eligibilities */
  float p, oldp, rhat, r;
  int box, i, y, steps = 0, failures=0, failed;

  printf("Seed? ");
```

```
scanf("%d",&i);
srand(i);

/*--- Initialize action and heuristic critic weights and traces. ---*/
for (i = 0; i < N_BOXES; i++)
  w[i] = v[i] = xbar[i] = e[i] = 0.0;

/*--- Starting state is (0 0 0 0) ---*/
x = x_dot = theta = theta_dot = 0.0;

/*--- Find box in state space containing start state ---*/
box = get_box(x, x_dot, theta, theta_dot);

/*--- Iterate through the action-learn loop. ---*/
while (steps++ < MAX_STEPS && failures < MAX_FAILURES)
  {
    /*--- Choose action randomly, biased by current weight. ---*/
    y = (random < prob_push_right(w[box]));

    /*--- Update traces. ---*/
    e[box] += (1.0 - LAMBDAw) * (y - 0.5);
    xbar[box] += (1.0 - LAMBDAv);

    /*--- Remember prediction of failure for current state ---*/
    oldp = v[box];

    /*--- Apply action to the simulated cart-pole ---*/
    cart_pole(y, &x, &x_dot, &theta, &theta_dot);

    /*--- Get box of state space containing the resulting state. ---*/
    box = get_box(x, x_dot, theta, theta_dot);

    if (box < 0)
      {
        /*--- Failure occurred. ---*/
        failed = 1;
        failures++;
        printf("Trial %d was %d steps.\n", failures, steps);
        steps = 0;

        /*--- Reset state to (0 0 0 0).  Find the box. ---*/
        x = x_dot = theta = theta_dot = 0.0;
        box = get_box(x, x_dot, theta, theta_dot);

        /*--- Reinforcement upon failure is -1. Prediction of failure is 0. ---*/
        r = -1.0;
        p = 0.;
      }
    else
      {
        /*--- Not a failure. ---*/
        failed = 0;

        /*--- Reinforcement is 0. Prediction of failure given by v weight. ---*/
        r = 0;
        p= v[box];
      }

    /*--- Heuristic reinforcement is:   current reinforcement
            + gamma * new failure prediction - previous failure prediction ---*/
    rhat = r + GAMMA * p - oldp;

    for (i = 0; i < N_BOXES; i++)
      {
          /*--- Update all weights. ---*/
          w[i] += ALPHA * rhat * e[i];
          v[i] += BETA * rhat * xbar[i];
          if (v[i] < -1.0)
            v[i] = v[i];

          if (failed)
            {
              /*--- If failure, zero all traces. ---*/
              e[i] = 0.;
              xbar[i] = 0.;
            }
          else
            {
              /*--- Otherwise, update (decay) the traces. ---*/
              e[i] *= LAMBDAw;
              xbar[i] *= LAMBDAv;
            }
        }

    }
  if (failures == MAX_FAILURES)
    printf("Pole not balanced. Stopping after %d failures.",failures);
  else
    printf("Pole balanced successfully for at least %d steps\n", steps);
}
```

```c
/*----------------------------------------------------------------------
   cart_pole:  Takes an action (0 or 1) and the current values of the
 four state variables and updates their values by estimating the state
 TAU seconds later.
----------------------------------------------------------------------*/

/*** Parameters for simulation ***/

#define GRAVITY 9.8
#define MASSCART 1.0
#define MASSPOLE 0.1
#define TOTAL_MASS (MASSPOLE + MASSCART)
#define LENGTH 0.5                /* actually half the pole's length */
#define POLEMASS_LENGTH (MASSPOLE * LENGTH)
#define FORCE_MAG 10.0
#define TAU 0.02                  /* seconds between state updates */
#define FOURTHIRDS 1.3333333333333


cart_pole(action, x, x_dot, theta, theta_dot)
int action;
float *x, *x_dot, *theta, *theta_dot;
{
    float xacc,thetaacc,force,costheta,sintheta,temp;

    force = (action>0)? FORCE_MAG : -FORCE_MAG;
    costheta = cos(*theta);
    sintheta = sin(*theta);

    temp = (force + POLEMASS_LENGTH * *theta_dot * *theta_dot * sintheta)
                         / TOTAL_MASS;

    thetaacc = (GRAVITY * sintheta - costheta* temp)
               / (LENGTH * (FOURTHIRDS - MASSPOLE * costheta * costheta
                                                / TOTAL_MASS));

    xacc  = temp - POLEMASS_LENGTH * thetaacc* costheta / TOTAL_MASS;

/*** Update the four state variables, using Euler's method. ***/

    *x  += TAU * *x_dot;
    *x_dot += TAU * xacc;
    *theta += TAU * *theta_dot;
    *theta_dot += TAU * thetaacc;
}
```

```
/*----------------------------------------------------------------------
   get_box:  Given the current state, returns a number from 1 to 162
  designating the region of the state space encompassing the current state.
  Returns a value of -1 if a failure state is encountered.
----------------------------------------------------------------------*/

#define one_degree 0.0174532    /* 2pi/360 */
#define six_degrees 0.1047192
#define twelve_degrees 0.2094384
#define fifty_degrees 0.87266

get_box(x,x_dot,theta,theta_dot)
float x,x_dot,theta,theta_dot;
{
  int box=0;

  if (x < -2.4 ||
      x > 2.4  ||
      theta < -twelve_degrees ||
      theta > twelve_degrees)          return(-1); /* to signal failure */

  if (x < -0.8)                        box = 0;
  else if (x < 0.8)                    box = 1;
  else                                 box = 2;

  if (x_dot < -0.5)                    ;
  else if (x_dot < 0.5)                box += 3;
  else                                 box += 6;

  if (theta < -six_degrees)            ;
  else if (theta < -one_degree)        box += 9;
  else if (theta < 0)                  box += 18;
  else if (theta < one_degree)         box += 27;
  else if (theta < six_degrees)        box += 36;
  else                                 box += 45;

  if (theta_dot < -fifty_degrees)       ;
  else if (theta_dot < fifty_degrees)  box += 54;
  else                                 box += 108;

  return(box);
}
```

```
/*----------------------------------------------------------------------
  Result of:  cc -o pole pole.c -lm          (assuming this file is pole.c)
              pole
----------------------------------------------------------------------*/
/*
Trial 1 was 21 steps.
Trial 2 was 12 steps.
Trial 3 was 28 steps.
Trial 4 was 44 steps.
Trial 5 was 15 steps.
Trial 6 was 9 steps.
Trial 7 was 10 steps.
Trial 8 was 16 steps.
Trial 9 was 59 steps.
Trial 10 was 25 steps.
Trial 11 was 86 steps.
Trial 12 was 118 steps.
Trial 13 was 218 steps.
Trial 14 was 290 steps.
Trial 15 was 19 steps.
Trial 16 was 180 steps.
Trial 17 was 109 steps.
Trial 18 was 38 steps.
Trial 19 was 13 steps.
Trial 20 was 144 steps.
Trial 21 was 41 steps.
Trial 22 was 323 steps.
Trial 23 was 172 steps.
Trial 24 was 33 steps.
Trial 25 was 1166 steps.
Trial 26 was 905 steps.
Trial 27 was 874 steps.
Trial 28 was 758 steps.
Trial 29 was 758 steps.
Trial 30 was 756 steps.
Trial 31 was 165 steps.
Trial 32 was 176 steps.
Trial 33 was 216 steps.
Trial 34 was 176 steps.
Trial 35 was 185 steps.
Trial 36 was 368 steps.
Trial 37 was 274 steps.
Trial 38 was 323 steps.
Trial 39 was 244 steps.
Trial 40 was 352 steps.
Trial 41 was 366 steps.
Trial 42 was 622 steps.
Trial 43 was 236 steps.
Trial 44 was 241 steps.
Trial 45 was 245 steps.
Trial 46 was 250 steps.
Trial 47 was 346 steps.
Trial 48 was 384 steps.
Trial 49 was 961 steps.
Trial 50 was 526 steps.
Trial 51 was 500 steps.
Trial 52 was 321 steps.
Trial 53 was 455 steps.
Trial 54 was 646 steps.
Trial 55 was 1579 steps.
Trial 56 was 1131 steps.
Trial 57 was 1055 steps.
Trial 58 was 967 steps.
Trial 59 was 1061 steps.
Trial 60 was 1009 steps.
Trial 61 was 1050 steps.
Trial 62 was 4815 steps.
Trial 63 was 863 steps.
Trial 64 was 9748 steps.
Trial 65 was 14073 steps.
Trial 66 was 9697 steps.
Trial 67 was 16815 steps.
Trial 68 was 21896 steps.
Trial 69 was 11566 steps.
Trial 70 was 22968 steps.
Trial 71 was 17811 steps.
Trial 72 was 11580 steps.
Trial 73 was 16805 steps.
Trial 74 was 16825 steps.
Trial 75 was 16872 steps.
Trial 76 was 16827 steps.
Trial 77 was 9777 steps.
Trial 78 was 19185 steps.
Trial 79 was 98799 steps.
Pole balanced successfully for at least 100001 steps
*/
```

```
(defvar V)
(defvar VV)
(defvar rows)
(defvar columns)
(defvar states)
(defvar AA)
(defvar BB)
(defvar AAprime)
(defvar BBprime)
(defvar gamma 0.9)


(defun setup ()
  (setq rows 5)
  (setq columns 5)
  (setq states 25)
  (setq AA (state-from-xy 1 0))
  (setq BB (state-from-xy 3 0))
  (setq AAprime (state-from-xy 1 4))
  (setq BBprime (state-from-xy 3 2))
  (setq V (make-array states :initial-element 0.0))
  (setq VV (make-array (list rows columns)))
)

(defun compute-V ()
  (loop for delta = (loop for x below states
                          for old-V = (aref V x)
                          do (setf (aref V x)
                                   (mean (loop for a below 4 collect
                                               (full-backup x a))))
                          sum (abs (- old-V (aref V x))))
        until (< delta 0.000001))
  (loop for state below states do
        (multiple-value-bind (x y) (xy-from-state state)
          (setf (aref VV y x) (aref V state))))
  (sfa VV))

(defun compute-V* ()
  (loop for delta = (loop for x below states
                          for old-V = (aref V x)
                          do (setf (aref V x)
                                   (loop for a below 4 maximize
                                               (full-backup x a)))
                          sum (abs (- old-V (aref V x))))
        until (< delta 0.000001))
  (loop for state below states do
        (multiple-value-bind (x y) (xy-from-state state)
          (setf (aref VV y x) (aref V state))))
  (sfa VV))

(defun sfa (array)
  "Show Floating-Point Array"
  (cond ((= 1 (array-rank array))
         (loop for e across array do (format t "~5,1F" e)))
        (t (loop for i below (array-dimension array 0) do
                 (format t "~%")
                 (loop for j below (array-dimension array 1) do
                       (format t "~5,1F" (aref array i j)))))))

(defun full-backup (x a)
  (let (r y)
    (cond ((= x AA)
           (setq r +10)
           (setq y AAprime))
          ((= x BB)
           (setq r +5)
           (setq y BBprime))
          ((off-grid x a)
           (setq r -1)
           (setq y x))
          (t
           (setq r 0)
           (setq y (next-state x a))))
    (+ r (* gamma (aref V y)))))

(defun off-grid (state a)
  (multiple-value-bind (x y) (xy-from-state state)
    (case a
      (0 (incf y) (>= y rows))
      (1 (incf x) (>= x columns))
      (2 (decf y) (< y 0))
      (3 (decf x) (< x 0)))))

(defun next-state (state a)
  (multiple-value-bind (x y) (xy-from-state state)
    (case a
      (0 (incf y))
      (1 (incf x))
      (2 (decf y))
      (3 (decf x)))
    (state-from-xy x y)))

(defun state-from-xy (x y)
      (+ y (* x columns)))

(defun xy-from-state (state)
  (truncate state columns))
```

```
(defvar c)
(defvar V)
(defvar V-)
(defvar VV)
(defvar rows)
(defvar columns)
(defvar states)
(defvar gamma 1.0)
(defvar terminals)
(defvar Vk)

#|
;(setq c (g-init-context))
;(set-view-size c 500 700)

;(defun scrap-it ()
;  (start-picture c)
;  (truncate-last-values)
;  (vgrids 0 1 2 3 10 999)
;  (put-scrap :pict (get-picture c)))

(defun gd-draw-grid (context xbase ybase xinc yinc numx numy color)
;  (gd-fill-rect context xbase ybase (+ xbase (* xinc numx))
;                 (+ ybase (* yinc numy)) (g-color-bw context 0))
  (let ((white (g-color-bw context 0.2)))
    (gd-draw-rect context xbase (+ ybase (* yinc (- numy 1))) xinc yinc white)
    (gd-draw-rect context (+ xbase (* xinc (- numx 1))) ybase xinc yinc white))
  (loop for i from 0 to numx
        for x from xbase by xinc
        do (gd-draw-vector context x ybase 0 (* numy yinc) color))
  (loop for i from 0 to numy
        for y from ybase by yinc
        do (gd-draw-vector context xbase y (* numx xinc) 0 color)))

(defun gd-draw-text-in-grid (context text x y xbase ybase xinc yinc
                                       &optional (font-spec '("times" 12)))
  (gd-draw-text context
                text
                font-spec
                (+ xbase 3 (* x xinc))
                (+ ybase 4 (* (- 3 y) yinc))
                nil))

(defun Vgrid (context pos k)
  (let* ((xinc 25)
         (yinc 20)
         (numx columns)
         (numy rows)
         (yspace 25)
         (xbase 50)
         (ybase (- 700 (* (+ yspace (* yinc numy)) (+ pos 1)))))
    (gd-draw-grid context xbase ybase xinc yinc numx numy (g-color-bw context 1))
    (loop for r below rows do
          (loop for c below columns do
                (gd-draw-text-in-grid context (format-number (aref (aref Vk k) c r))
                                      c r xbase ybase xinc yinc)))
    (incf xbase (+ xbase (* xinc numx)))
    (gd-draw-grid context xbase ybase xinc yinc numx numy (g-color-bw context 1))
    (loop for state from 1 below (- states 1)
          do (multiple-value-bind (x y) (xy-from-state state)
               (setf (aref V state) (aref (aref Vk k) x y))))
    (loop for r below rows do
          (loop for c below columns do
                (gd-draw-policy context (greedy-policy (state-from-xy c r))
                                c r xbase ybase xinc yinc)))))

(defun gd-draw-policy (context actions x y xbase ybase xinc yinc)
  (let ((centerx (+ xbase (* x xinc) (truncate xinc 2)))
        (centery (+ ybase (* (- 3 y) yinc) (truncate yinc 2)))
        (xsize (truncate (* xinc 0.4)))
        (ysize (truncate (* yinc 0.4)))
        (bl (g-color-bw context 1)))
    (loop for a in actions do
          (case a
            (0 (gd-draw-arrow context centerx centery centerx (- centery ysize) bl))
            (1 (gd-draw-arrow context centerx centery (+ centerx xsize) centery bl))
            (2 (gd-draw-arrow context centerx centery centerx (+ centery ysize) bl))
            (3 (gd-draw-arrow context centerx centery (- centerx xsize) centery
bl))))))
|#

(defun greedy-policy (state)
  (if (member state terminals)
    nil
    (loop with bestQ = -10000.0 and bestas = nil
          for a below 4
          for Q = (full-backup state a)
          do (cond ((> Q bestQ)
                    (setq bestQ Q)
                    (setq bestas (list a)))
                   ((= Q bestQ)
                    (push a bestas)))
          finally (return bestas))))


(defun format-number (num)
  (cond ((null num)
         "  T")
        ((<= (abs num) 9.95)
         (format nil "~4,1F" num))
        (t
         (format nil "~4,0F" num))))

(defun Vgrids (&rest Ks)
  (loop for pos from 0
        for k in Ks
        do (vgrid c pos k)))


(defun setup ()
  (setq terminals '(0 15))
  (setq rows 4)
  (setq columns 4)
  (setq states 16)
  (setq V (make-array states :initial-element 0.0))
  (setq V- (make-array states :initial-element 0.0))
  (setq VV (make-array (list rows columns)))
  (setq Vk (make-array 1000 :initial-contents
                       (loop repeat 1000 collect (make-array (list rows columns)))))
  nil
)

(defun compute-V ()
  (loop for i below states do (setf (aref V i) 0.0))
  (loop for k below 1000
        do (loop for state from 1 below (- states 1)
                 do (setf (aref V- state)
                          (mean (loop for a below 4 collect
                                      (full-backup state a))))
                 do (multiple-value-bind (x y) (xy-from-state state)
                      (setf (aref (aref Vk k) x y) (aref V state))))
        do (ut::swap V V-))
  (loop for state below states do
        (multiple-value-bind (x y) (xy-from-state state)
          (setf (aref VV y x) (aref V state))))
  (sfa VV))

(defun compute-V* ()
  (loop for i below states do (setf (aref V i) 0.0))
  (loop for k below 1000
        do (loop for x from 1 below (- states 1)
                 do (setf (aref V- x)
                          (loop for a below 4 maximize
                                      (full-backup x a)))
                 do (multiple-value-bind (x y) (xy-from-state x)
                      (setf (aref (aref Vk k) x y) (aref V x))))
        do (ut::swap V V-))
  (loop for state below states do
        (multiple-value-bind (x y) (xy-from-state state)
          (setf (aref VV y x) (aref V state))))
  (sfa VV))



(defun sfa (array)
  "Show Floating-Point Array"
  (cond ((= 1 (array-rank array))
         (loop for e across array do (format t "~8,3F" e)))
        (t (loop for i below (array-dimension array 0) do
                 (format t "~%")
                 (loop for j below (array-dimension array 1) do
                       (format t "~8,3F" (aref array i j)))))))

(defun full-backup (x a)
  (let (r y)
    (cond ((off-grid x a)
           (setq r -1)
           (setq y x))
          (t
           (setq r -1)
           (setq y (next-state x a))))
    (+ r (* gamma (aref V y)))))

(defun off-grid (state a)
  (multiple-value-bind (x y) (xy-from-state state)
    (case a
      (0 (incf y) (>= y rows))
      (1 (incf x) (>= x columns))
      (2 (decf y) (< y 0))
      (3 (decf x) (< x 0)))))

(defun next-state (state a)
  (multiple-value-bind (x y) (xy-from-state state)
    (case a
      (0 (incf y))
      (1 (incf x))
      (2 (decf y))
      (3 (decf x)))
    (state-from-xy x y)))

(defun state-from-xy (x y)
      (+ y (* x columns)))

(defun xy-from-state (state)
  (truncate state columns))

(defun truncate-last-values ()
  (loop for state from 1 below (- states 1)
        do (multiple-value-bind (x y) (xy-from-state state)
             (setf (aref (aref Vk 999) x y)
                   (round (aref (aref Vk 999) x y))))))
```

```lisp
;;; Jack's car rental problem.  The state is n1 and n2, the number of cars
;;; at each location a the end of the day, at most 20.  Actions are numbers of cars
;;; to switch from location 1 to location 2, a number between -5 and +5.

;;; P1(n1,new-n1) is a 26x21 array giving the probability that the number of cars at
;;; location 1 is new-n1, given that it starts the day at n1.  Similarly for P2

;;; R1(n1) is a 26 array giving the expected reward due to satisfied requests at
;;; location, given that the day starts with n1 cars at location 1.  SImilarly for
R2.


;;; Here we implement policy iteration.


(defvar V)
(defvar lambda-requests1 3)
(defvar lambda-requests2 4)
(defvar lambda-dropoffs1 3)
(defvar lambda-dropoffs2 2)
(defvar policy)
(defvar P1)
(defvar P2)
(defvar R1)
(defvar R2)
(defvar gamma .9)
(defvar theta .0000001)

(defun setup ()
  (setq V (make-array '(21 21) :initial-element 0))
  (setq policy (make-array '(21 21) :initial-element 0))
  (setq P1 (make-array '(26 21) :initial-element 0))
  (setq P2 (make-array '(26 21) :initial-element 0))
  (setq R1 (make-array 26 :initial-element 0))
  (setq R2 (make-array 26 :initial-element 0))
  (load-P-and-R P1 R1 lambda-requests1 lambda-dropoffs1)
  (load-P-and-R P2 R2 lambda-requests2 lambda-dropoffs2))

(defun load-P-and-R (P R lambda-requests lambda-dropoffs)
  (loop for requests from 0
        for request-prob = (poisson requests lambda-requests)
        until (< request-prob .000001)
        do (loop for n upto 25 do
                 (incf (aref R n) (* 10 request-prob (min requests n))))
        do (loop for dropoffs from 0
                 for drop-prob = (poisson dropoffs lambda-dropoffs)
                 until (< drop-prob .000001)
                 do (loop for n upto 25
                          for satisfied-requests = (min requests n)
                          for new-n = (max 0 (min 20 (- (+ n dropoffs)
                                                        satisfied-requests)))
                          do (incf (aref P n new-n) (* request-prob drop-prob))))))

(defun poisson (n lambda)
  "The probability of n events according to the poisson distribution"
  (* (exp (- lambda))
     (/ (expt lambda n)
        (factorial n))))

(defun factorial (n)
  (if (= n 0)
    1
    (* n (factorial (- n 1)))))
```

```
(defun backup-action (n1 n2 a)
  (setq a (max (- n2) (min a n1)))
  (setq a (max -5 (min 5 a)))
  (+ (* -2 (abs a))
     (loop with morning-n1 = (- n1 a)
           with morning-n2 = (+ n2 a)
           for new-n1 upto 20 sum
           (loop for new-n2 upto 20 sum
                 (* (aref P1 morning-n1 new-n1)
                    (aref P2 morning-n2 new-n2)
                    (+ (aref R1 morning-n1)
                       (aref R2 morning-n2)
                       (* gamma (aref V new-n1 new-n2))))))))

(defun policy-eval ()
  (loop while (< theta
                 (loop for n1 upto 20 maximize
                       (loop for n2 upto 20
                             for old-v = (aref V n1 n2)
                             for a = (aref policy n1 n2)
                             do (setf (aref V n1 n2) (backup-action n1 n2 a))
                             maximize (abs (- old-v (aref V n1 n2))))))))

(defun policy (n1 n2 &optional (epsilon .0000000001))
  (loop with best-value = -1
        with best-action
        for a from (max -5 (- n2)) upto (min 5 n1)
        for this-value = (backup-action n1 n2 a)
        do (when (> this-value (+ best-value epsilon))
             (setq best-value this-value)
             (setq best-action a))
        finally (return best-action)))

(defun show-greedy-policy ()
  (loop for n1 from 0 upto 20 do
        (format t "~%")
        (loop for n2 from 0 upto 20 do
              (format t "~3A" (policy n1 n2)))))

(defun greedify ()
  (loop with policy-improved = nil
        for n1 from 0 upto 20 do
        (loop for n2 from 0 upto 20
              for b = (aref policy n1 n2) do
              (setf (aref policy n1 n2) (policy n1 n2))
              (unless (= b (aref policy n1 n2))
                (setq policy-improved t)))
        finally (progn (show-policy) (return policy-improved))))

(defun show-policy ()
  (loop for n1 from 0 upto 20 do
        (format t "~%")
        (loop for n2 from 0 upto 20 do
              (format t "~3A" (aref policy n1 n2)))))

(defun policy-iteration ()
  (loop for count from 0
        do (policy-eval)
        do (print count)
        while (greedify)))
```

```
;;; Gambler's problem.  The gambler has a stake s between 0 and 100.  At each
;;; play he wagers an integer <= s.  He wins that much with prob p, else he
;;; loses that much.  If he builds his stake to 100 he wins (thus he never
;;; wagers more than (- 100 s)); if his stake falls to 0 he loses.

;;; Thus, the stake s is the state the actions are the size of the bid.

;;; Here we implement value iteration.


(defvar V (make-array 101 :initial-element 0))
(setf (aref V 100) 1)
(defvar p .45)

(defun backup-action (s a)
  (+ (* p (aref V (+ s a)))
     (* (- 1 p) (aref V (- s a)))))

(defun vi (&optional (epsilon .00000001))
  "Value iteration to the criterion epsilon"
  (loop collect (loop for i from 1 to 99 collect (list i (aref v i)))
        while (< epsilon
                 (loop for s from 1 below 100
                       for old-V = (aref V s)
                       do (setf (aref V s)
                                (loop for a from 1 upto (min s (- 100 s))
                                      maximize (backup-action s a)))
                       maximize (abs (- old-V (aref V s)))))
        ))


(defun policy (s &optional (epsilon .0000000001))
  (loop with best-value = -1
        with best-action
        for a from 1 upto (min s (- 100 s))
        for this-value = (backup-action s a)
        do (when (> this-value (+ best-value epsilon))
             (setq best-value this-value)
             (setq best-action a))
        finally (return best-action)))
```

```
;;; Monte Carlo and DP solution of simple blackjack.

;;; The state is (dc,pc,ace01), i.e., (dealer-card, player-count, usable-ace?),
;;; in the ranges ([12-21],[12-21],[0-1]).

;;; The actions are hit or stick, t or nil

(defvar V)
(defvar policy)
(defvar N)                              ; Number of returns seen for this state
(defvar dc)                             ; count of dealer's showing card
(defvar pc)                             ; total count of player's hand
(defvar ace)                            ; does play have a usable ace?
(defvar episode)

(defun card ()
  (min 10 (+ 1 (random 13))))

(defun setup ()
  (setq V (make-array '(11 22 2) :initial-element 0.0))
  (setq N (make-array '(11 22 2) :initial-element 0))
  (setq policy (make-array '(11 22 2) :initial-element 1))
  (loop for dc from 1 to 10 do
        (loop for pc from 20 to 21 do
              (loop for ace from 0 to 1 do
                    (setf (aref policy dc pc ace) 0)))))

(defun episode ()
  (let (dc-hidden pcard1 pcard2 outcome)
    (setq episode nil)
    (setq dc-hidden (card))
    (setq dc (card))
    (setq pcard1 (card))
    (setq pcard2 (card))
    (setq ace (OR (= 1 pcard1) (= 1 pcard2)))
    (setq pc (+ pcard1 pcard2))
    (if ace (incf pc 10))
    (unless (= pc 21)                   ; natural blackjack ends all
      (loop do (push (list dc pc ace) episode)
            while (= 1 (aref policy dc pc (if ace 1 0)))
            do (draw-card)
            until (bust?)))
    (setq outcome (outcome dc dc-hidden))
    (learn episode outcome)
    (cons outcome episode)))

(defun learn (episode outcome)
  (loop for (dc pc ace-boolean) in episode
        for ace = (if ace-boolean 1 0) do
        (when (> pc 11)
          (incf (aref N dc pc ace))
          (incf (aref V dc pc ace) (/ (- outcome (aref V dc pc ace))
                                       (aref N dc pc ace))))))

(defun outcome (dc dc-hidden)
  (let (dcount dace dnatural pnatural)
    (setq dace (OR (= 1 dc) (= 1 dc-hidden)))
    (setq dcount (+ dc dc-hidden))
    (if dace (incf dcount 10))
    (setq dnatural (= dcount 21))
    (setq pnatural (not episode))
    (cond
     ((AND pnatural dnatural) 0)
     (pnatural 1)
     (dnatural -1)
     ((bust?) -1)
     (t (loop while (< dcount 17)
              for card = (card) do
              (incf dcount card)
              (when (AND (not dace) (= card 1))
                (incf dcount 10)
                (setf dace t))
              (when (AND dace (> dcount 21))
                (decf dcount 10)
                (setq dace nil))
              finally (return (cond ((> dcount 21) 1)
                                    ((> dcount pc) -1)
                                    ((= dcount pc) 0)
                                    (t 1))))))))

(defun draw-card ()
  (let (card)
    (setq card (card))
    (incf pc card)
    (when (AND (not ace) (= card 1))
      (incf pc 10)
      (setf ace t))
    (when (AND ace (> pc 21))
      (decf pc 10)
      (setq ace nil))))

(defun bust? ()
  (> pc 21))

(defvar w)
(defvar array (make-array '(10 10)))
(defun gr (source ace &optional (arr array))
  (loop with ace = (if ace 1 0)
        for i below 10 do
        (loop for j below 10 do
              (setf (aref arr i j) (aref source (+ i 1) (+ j 12) ace))))
  (g::graph-surface w arr))

(defun experiment ()
  (setup)
  (loop for count below 500
        for ar0 = (make-array '(10 10))
        for ar1 = (make-array '(10 10))
        do
        (print count)
        (gr V nil ar0)
        (gr V t ar1)
        collect ar0
        collect ar1
        do (loop repeat 1000 do (episode))))
```

```
;;; Monte Carlo and DP solution of simple blackjack.

;;; The state is (dc,pc,ace01), i.e., (dealer-card, player-count, usable-ace?),
;;; in the ranges ([12-21],[12-21],[0-1]).

;;; The actions are hit or stick, t or nil

(defvar Q)
(defvar policy)
(defvar N)                              ; Number of returns seen for this state
(defvar dc)                             ; count of dealer's showing card
(defvar pc)                             ; total count of player's hand
(defvar ace)                            ; does play have a usable ace?
(defvar episode)

(defun card ()
  (min 10 (+ 1 (random 13))))

(defun setup ()
  (setq Q (make-array '(11 22 2 2) :initial-element 0.0))
  (setq N (make-array '(11 22 2 2) :initial-element 0))
  (setq policy (make-array '(11 22 2) :initial-element 1))
  (loop for dc from 1 to 10 do
        (loop for pc from 20 to 21 do
              (loop for ace from 0 to 1 do
                    (setf (aref policy dc pc ace) 0)))))

(defun exploring-episode ()
  (let (dc-hidden outcome action)
    (setq episode nil)
    (setq dc-hidden (card))
    (setq dc (+ 1 (random 10)))
    (setq ace (if (= 0 (random 2)) t nil))
    (setq pc (+ 12 (random 10)))
    (setq action (random 2))
;    (print (list pc ace action))
    (loop do (push (list dc pc ace action) episode)
          while (= action 1)
          do (draw-card)
          until (bust?)
          do (setq action (aref policy dc pc (if ace 1 0))))
    (setq outcome (outcome dc dc-hidden))
    (learn episode outcome)
    (cons outcome episode)))

(defun episode ()
  (let (dc-hidden pcard1 pcard2 outcome)
    (setq episode nil)
    (setq dc-hidden (card))
    (setq dc (card))
    (setq pcard1 (card))
    (setq pcard2 (card))
    (setq ace (OR (= 1 pcard1) (= 1 pcard2)))
    (setq pc (+ pcard1 pcard2))
    (if ace (incf pc 10))
    (unless (= pc 21)                   ; natural blackjack ends all
      (loop do (push (list dc pc ace) episode)
            while (= 1 (aref policy dc pc (if ace 1 0)))
            do (draw-card)
            until (bust?)))
    (setq outcome (outcome dc dc-hidden))
    (learn episode outcome)
    (cons outcome episode)))
```

```
(defun learn (episode outcome)
  (loop for (dc pc ace-boolean action) in episode
        for ace = (if ace-boolean 1 0) do
        (when (> pc 11)
          (incf (aref N dc pc ace action))
          (incf (aref Q dc pc ace action) (/ (- outcome (aref Q dc pc ace action))
                                             (aref N dc pc ace action)))
          (let (policy-action other-action)
            (setq policy-action (aref policy dc pc ace))
            (setq other-action (- 1 policy-action))
            (when (> (aref Q dc pc ace other-action)
                     (aref Q dc pc ace policy-action))
                (setf (aref policy dc pc ace) other-action))))))

(defun outcome (dc dc-hidden)
  (let (dcount dace dnatural pnatural)
    (setq dace (OR (= 1 dc) (= 1 dc-hidden)))
    (setq dcount (+ dc dc-hidden))
    (if dace (incf dcount 10))
    (setq dnatural (= dcount 21))
    (setq pnatural (not episode))
    (cond
     ((AND pnatural dnatural) 0)
     (pnatural 1)
     (dnatural -1)
     ((bust?) -1)
     (t (loop while (< dcount 17)
              for card = (card) do
              (incf dcount card)
              (when (AND (not dace) (= card 1))
                (incf dcount 10)
                (setf dace t))
              (when (AND dace (> dcount 21))
                (decf dcount 10)
                (setq dace nil))
              finally (return (cond ((> dcount 21) 1)
                                    ((> dcount pc) -1)
                                    ((= dcount pc) 0)
                                    (t 1))))))))

(defun draw-card ()
  (let (card)
    (setq card (card))
    (incf pc card)
    (when (AND (not ace) (= card 1))
      (incf pc 10)
      (setf ace t))
    (when (AND ace (> pc 21))
      (decf pc 10)
      (setq ace nil))))

(defun bust? ()
  (> pc 21))

(defvar w)
(defvar array (make-array '(10 10)))
(defun gr (source ace action &optional (arr array))
  (loop with ace = (if ace 1 0)
        for i below 10 do
        (loop for j below 10 do
              (setf (aref arr i j) (aref source (+ i 1) (+ j 12) ace action)))))
  (g::graph-surface w arr))

(defun grp (ace &optional (arr array))
  (loop with ace = (if ace 1 0)
        for i below 10 do
        (loop for j below 10 do
              (setf (aref arr i j) (aref policy (+ i 1) (+ j 12) ace))))
  (g::graph-surface w arr))

(defun experiment ()
  (setup)
  (loop for count below 500
        for ar0 = (make-array '(10 10))
        for ar1 = (make-array '(10 10))
        do
        (print count)
        (gr Q nil ar0)
        (gr Q t ar1)
        collect ar0
        collect ar1
        do (loop repeat 1000 do (episode))))
```

```
;-*- Package: (discrete-walk) -*-

;;; A simulation of a TD(lambda) learning system to predict the expected outcome
;;; of a discrete-state random walk like that in the original 1988 TD paper.


(defpackage :discrete-walk
  (:use :common-lisp :g :ut :graph)
  (:nicknames :dwalk))

(in-package :dwalk)


(defvar n 5)                            ; the number of nonterminal states
(defvar w)                              ; the vector of weights = predictions
(defvar e)                              ; the eligibility trace
(defvar lambda .9)                      ; trace decay parameter
(defvar alpha 0.1)                      ; learning-rate parameter
(defvar initial-w 0.5)
(defvar standard-walks nil)             ; list of standard walks
(defvar trace-type :none)               ; :replace, :accumulate, :average, :1/t or
:none
(defvar alpha-type :fixed)              ; :fixed, :1/t, or :1/t-max
(defvar alpha-array)                    ; used when each state has a different alpha
(defvar u)                              ; usage count = number of times updated

(defun setup (num-runs num-walks)
  (setq w (make-array n))
  (setq e (make-array n))
  (setq u (make-array n))
  (setq alpha-array (make-array n))
  (setq standard-walks (standard-walks num-runs num-walks))
  (length standard-walks))

(defun init ()
  (loop for i below n do (setf (aref w i) initial-w))
  (loop for i below n do (setf (aref alpha-array i) alpha))
  (loop for i below n do (setf (aref u i) 0)))

(defun init-traces ()
  (loop for i below n do (setf (aref e i) 0)))

(defun learn (x target)
  (ecase alpha-type
    (:1/t (incf (aref u x))
          (setf (aref alpha-array x) (/ 1.0 (aref u x))))
    (:fixed)
    (:1/t-max (when (<= (aref u x) (/ 1 alpha))
                (incf (aref u x))
                (setf (aref alpha-array x) (/ 1.0 (aref u x))))))
  (ecase trace-type
    (:none)
    (:replace (loop for i below n do (setf (aref e i) (* lambda (aref e i))))
              (decf (aref u x) (aref e x))
              (setf (aref e x) 1))
    (:accumulate (loop for i below n do (setf (aref e i) (* lambda (aref e i))))
                 (incf (aref e x) 1))
    (:average (loop for i below n do (setf (aref e i) (* lambda (aref e i))))
              (setf (aref e x) (+ 1 (* (aref e x) (- 1 (aref alpha-array x))))))
    (:1/t (incf (aref u x))
          (incf (aref e x) 1)
          (loop for i below n
                for lambda = (float (/ (aref u x)))
                do (setf (aref e i) (* lambda (aref e i))))))
  (if (eq trace-type :none)
    (incf (aref w x) (* (aref alpha-array x) (- target (aref w x))))
    (loop for i below n
          with error = (- target (aref w x))
          do (incf (aref w i) (* (aref alpha-array i) error (aref e i))))))

(defun process-walk (walk)
  (destructuring-bind (outcome states) walk
    (unless (eq trace-type :none) (init-traces))
    (loop for s1 in states
          for s2 in (rest states)
          do (learn s1 (aref w s2)))
    (learn (first (last states)) outcome)))

(defun process-walk-backwards (walk)
  (destructuring-bind (outcome states) walk
    (unless (eq trace-type :none) (init-traces))
    (learn (first (last states)) outcome)
    (loop for s1 in (reverse (butlast states))
          for s2 in (reverse (rest states))
          do (learn s1 (aref w s2)))))

(defun process-walk-MC (walk)
  (destructuring-bind (outcome states) walk
    (loop for s in (reverse states)
          do (learn s outcome))))

(defun standard-walks (num-sets-of-walks num-walks)
  (loop repeat num-sets-of-walks
        with random-state = (ut::copy-of-standard-random-state)
        collect (loop repeat num-walks
                      collect (random-walk n random-state))))

(defun random-walk (n &optional (random-state *random-state*))
  (loop with start-state = (round (/ n 2))
        for x = start-state then (with-prob .5 (+ x 1) (- x 1) random-state)
        while (AND (>= x 0) (< x n))
        collect x into xs
        finally (return (list (if (< x 0) 0 1) xs))))

(defun residual-error ()
  "Returns the residual RMSE between the current and correct predictions"
  (rmse 0 (loop for i below n
                when (>= (aref w i) -.1)
                collect (- (aref w i)
                           (/ (+ i 1) (+ n 1) )))))

(defun explore (alpha-type-arg alpha-arg lambda-arg trace-type-arg forward?
                           &optional (number-type 'float))
  (setq alpha-type alpha-type-arg)
  (setq alpha alpha-arg)
  (setq lambda lambda-arg)
  (setq lambda (coerce lambda number-type))
  (setq alpha (coerce alpha number-type))
  (setq trace-type trace-type-arg)
  (record (stats (loop for walk-set in standard-walks
               do (init)
               do (loop repeat 100 do (loop for walk in walk-set do (if forward?
                                                   (process-walk walk)
                                                   (process-walk-backwards walk))))
               collect (residual-error)))))
```

```
(defun learning-curve (alpha-type-arg alpha-arg lambda-arg trace-type-arg
                                 &optional (processing :forward) (initial-w-arg 0.5)
                                 (number-type 'float))
  (setq alpha-type alpha-type-arg)
  (setq alpha alpha-arg)
  (setq lambda lambda-arg)
  (setq lambda (coerce lambda number-type))
  (setq alpha (coerce alpha number-type))
  (setq trace-type trace-type-arg)
  (setq initial-w initial-w-arg)
  (multi-mean
   (loop for walk-set in standard-walks
         do (init)
         collect (cons (residual-error)
                       (loop for walk in walk-set
                             do (ecase processing
                                  (:forward (process-walk walk))
                                  (:backward (process-walk-backwards walk))
                                  (:MC (process-walk-MC walk)))
                             collect (residual-error))))))

(defun batch-learning-curve-TD ()
  (setq alpha 0.01)
  (setq lambda 0.0)
  (setq trace-type :none)
  (setq initial-w -1)
  (multi-mean
   (loop with last-w = (make-array n)
         for walk-set in standard-walks
         do (init)
         collect (loop for num-walks from 1 to (length walk-set)
                       for walk-subset = (firstn num-walks walk-set) do
                       (loop do (loop for i below n do (setf (aref last-w i) (aref w
i)))
                             do (loop for walk in walk-subset
                                      do (process-walk walk))
                             until (> .0000001 (loop for i below n
                                                     sum (abs (- (aref w i) (aref last-w
i))))))
                       collect (residual-error)))))
```

```
;-*- Package: (discrete-walk) -*-

;;; A simulation of a TD(lambda) learning system to predict the expected outcome
;;; of a discrete-state random walk like that in the original 1988 TD paper.


(defpackage :discrete-walk
  (:use :common-lisp :g :ut :graph)
  (:nicknames :dwalk))

(in-package :dwalk)


(defvar n 5)                            ; the number of nonterminal states
(defvar w)                              ; the vector of weights = predictions
(defvar e)                              ; the eligibility trace
(defvar lambda .9)                      ; trace decay parameter
(defvar alpha 0.1)                      ; learning-rate parameter
(defvar initial-w 0.5)
(defvar standard-walks nil)             ; list of standard walks
(defvar trace-type :none)               ; :replace, :accumulate, :average, :1/t or
:none
(defvar alpha-type :fixed)              ; :fixed, :1/t, or :1/t-max
(defvar alpha-array)                    ; used when each state has a different alpha
(defvar u)                              ; usage count = number of times updated
(defvar delta-w)

(defun setup (num-runs num-walks)
  (setq w (make-array n))
  (setq delta-w (make-array n))
  (setq e (make-array n))
  (setq u (make-array n))
  (setq alpha-array (make-array n))
  (setq standard-walks (standard-walks num-runs num-walks))
  (length standard-walks))

(defun init ()
  (loop for i below n do (setf (aref w i) initial-w))
  (loop for i below n do (setf (aref alpha-array i) alpha))
  (loop for i below n do (setf (aref u i) 0)))

(defun init-traces ()
  (loop for i below n do (setf (aref e i) 0)))

(defun learn (x target)
  (ecase alpha-type
    (:1/t (incf (aref u x))
          (setf (aref alpha-array x) (/ 1.0 (aref u x))))
    (:fixed)
    (:1/t-max (when (<= (aref u x) (/ 1 alpha))
                (incf (aref u x))
                (setf (aref alpha-array x) (/ 1.0 (aref u x))))))
  (ecase trace-type
    (:none)
    (:replace (loop for i below n do (setf (aref e i) (* lambda (aref e i))))
              (decf (aref u x) (aref e x))
              (setf (aref e x) 1))
    (:accumulate (loop for i below n do (setf (aref e i) (* lambda (aref e i))))
                 (incf (aref e x) 1))
    (:average (loop for i below n do (setf (aref e i) (* lambda (aref e i))))
              (setf (aref e x) (+ 1 (* (aref e x) (- 1 (aref alpha-array x))))))
    (:1/t (incf (aref u x))
          (incf (aref e x) 1)
          (loop for i below n
                for lambda = (float (/ (aref u x)))
                do (setf (aref e i) (* lambda (aref e i))))))
  (if (eq trace-type :none)
    (incf (aref delta-w x) (* alpha (- target (aref w x))))
    (loop for i below n
          with error = (- target (aref w x))
          do (incf (aref delta-w i) (* (aref alpha-array i) error (aref e i))))))

(defun process-walk (walk)
  (destructuring-bind (outcome states) walk
    (unless (eq trace-type :none) (init-traces))
    (loop for s1 in states
          for s2 in (rest states)
          do (learn s1 (aref w s2)))
    (learn (first (last states)) outcome)))

(defun process-walk-backwards (walk)
  (destructuring-bind (outcome states) walk
    (unless (eq trace-type :none) (init-traces))
    (learn (first (last states)) outcome)
    (loop for s1 in (reverse (butlast states))
          for s2 in (reverse (rest states))
          do (learn s1 (aref w s2)))))

(defun process-walk-MC (walk)
  (destructuring-bind (outcome states) walk
    (loop for s in (reverse states)
          do (learn s outcome))))

(defun standard-walks (num-sets-of-walks num-walks)
  (loop repeat num-sets-of-walks
        with random-state = (ut::copy-of-standard-random-state)
        collect (loop repeat num-walks
                      collect (random-walk n random-state))))

(defun random-walk (n &optional (random-state *random-state*))
  (loop with start-state = (round (/ n 2))
        for x = start-state then (with-prob .5 (+ x 1) (- x 1) random-state)
        while (AND (>= x 0) (< x n))
        collect x into xs
        finally (return (list (if (< x 0) 0 1) xs))))

(defun residual-error ()
  "Returns the residual RMSE between the current and correct predictions"
  (rmse 0 (loop for i below n
                when (>= (aref w i) -.1)
                collect (- (aref w i)
                           (/ (+ i 1) (+ n 1) )))))

(defun batch-exp ()
  (setq lambda 0.0)
  (setq trace-type :none)
  (setq initial-w -1)
  (loop for walk-set in standard-walks
        for run-num from 0
        do (loop for l in '(0 1) do
                 (init)
                 (record l run-num
                         (loop for num-walks from 1 to (length walk-set)
                               for walk-subset = (firstn num-walks walk-set) do
                               (setf alpha (/ 1.0 n num-walks 3))
                               (loop do (loop for i below n do (setf (aref delta-w i)
0))
                                     do (loop for walk in walk-subset
                                              do (ecase l
                                                   (0 (process-walk walk))
                                                   (1 (process-walk-mc walk))))
                                     do (loop for i below n do (incf (aref w i) (aref
delta-w i)))
                                     until (> .0000001 (loop for i below n
                                                             sum (abs (aref delta-w
i)))))
                               collect (residual-error))))))
```

```
;;; Code for access-control queuing problem from chapter 6.
;;; N is the number of servers, M is the number of priorities

;;; Using R-learning

(defvar N 10)
(defvar N+1)
(defvar num-states)
(defvar M 2)
;(defvar h)
(defvar p .05)
(defvar alpha .1)
(defvar beta .01)
(defvar epsilon .1)
(defvar Q)
(defvar count)
(defvar rho)
(defvar num-free-servers)               ; these two are
(defvar priority)                       ; the state variables
(defvar reward)

(defun setup ()
  (setq N+1 (+ N 1))
  (setq num-states (* M N+1))
  (setq Q (make-array (list num-states 2) :initial-element 0))
  (setq count (make-array (list num-states 2) :initial-element 0))
;  (loop for s below num-states do
;        (setf (aref Q s 0) -.1)
;        (setf (aref Q s 1) +.1))
  (setq reward (make-array M :initial-contents '(1 2 4 8)))
;  (setq h (make-array M :initial-contents '((/ 1 3) (/ 1 3) (/ 1 3))))
  (setq rho 0)
  (setq num-free-servers N)
  (new-priority))

(defun new-priority ()
  (setq priority (random M)))

(defun R-learning (steps)
  (loop repeat steps
        for s = (+ num-free-servers (* priority N+1)) then s-prime
        for a = (with-prob epsilon (random 2)
                  (if (> (aref Q s 0) (aref Q s 1)) 0 1))
        for r = (if (AND (= a 1) (> num-free-servers 0))
                  (aref reward priority)
                  0)
        for new-priority = (new-priority)
        for s-prime = (progn (unless (= r 0) (decf num-free-servers))
                             (loop repeat (- N num-free-servers)
                                   do (when (with-probability p)
                                        (incf num-free-servers)))
                             (+ num-free-servers (* new-priority N+1)))
;        do (print (list s a r s-prime rho (max (aref Q s-prime 0) (aref Q s-prime
1))))
        do (incf (aref Q s a) (* alpha (+ r (- rho)
                                          (max (aref Q s-prime 0)
                                               (aref Q s-prime 1))
                                          (- (aref Q s a)))))
        do (incf (aref count s a))
        do (when (= (aref Q s a) (max (aref Q s 0) (aref Q s 1)))
             (incf rho (* beta (+ r (- rho)
                                  (max (aref Q s-prime 0) (aref Q s-prime 1))
```

```
                              (- (max (aref Q s 0) (aref Q s 1)))))))
        do (setq priority new-priority)))

(defun policy ()
  (loop for pri below M do
        (format t "~%")
        (loop for free upto N
              for s = (+ free (* pri N+1))
              do (format t (if (> (aref Q s 0) (aref Q s 1)) " 0" " 1"))))
  (values))

(defun num ()
  (loop for pri below M do
        (format t "~%")
        (loop for free upto N
              for s = (+ free (* pri N+1))
              do (format t "~A/~A " (aref count s 0) (aref count s 1))))
  (values))

(defun Q ()
  (loop for pri below M do
        (format t "~%")
        (loop for free upto N
              for s = (+ free (* pri N+1))
              do (format t "~6,3F/~6,3F " (aref Q s 0) (aref Q s 1))))
  (values))

(defun gr ()
  (graph (cons (list '(1 0) (list N+1 0))
               (loop for pri below M
                     collect (loop for free upto N
                                   collect (aref Q (+ free (* pri N+1)) 0))
                     collect (loop for free upto N
                                   collect (aref Q (+ free (* pri N+1)) 1))))))

(defun grV* ()
  (graph (cons (list '(1 0) (list N+1 0))
               (loop for pri below M
                     collect (loop for free upto N
                                   collect (max (aref Q (+ free (* pri N+1)) 0)
                                                (aref Q (+ free (* pri N+1)) 1)))))))
```

```
/* This code was written by Abhinav Garg, abhinav@cs.umass.edu, August, 1998 */

# include <stdio.h>
# include <math.h>
# include <stdlib.h>

# define n       10
# define h        0.5
# define p        6
# define alpha    0.01
# define beta     0.01
# define epsilon  10
# define iterations  2000000

# define max(a,b) (((a)>(b)) ? (a) : (b));

double q[n+1][4][2] = {0.0}; /* 0 - reject, 1 - accept */

/* To generate a priority request */
int req_priority(void)
{
  int req = 0;
  /*  req = rand() % 4;
  return req; */ /* CHECK OUT LATER */
  req = rand() %100;
  if (req<40)
   return 3;
  else if (req >= 40 && req <60)
   return 2;
  else if (req >=60 && req <80)
   return 1;
  else if (req >=80 && req <100)
  return 0;
}

/* Find out whether server is to be freed or not */
int free_server(void)
{
  int free = 0;
  free  = (rand() % 100);
  if ( free < p )
     return 1;      /* free the server */
  return 0;      /* server is still busy */
}

int  choose_action(int r, int c)
     /* r,c are the state's  subscripts and q is the array */
{
  int number;
  int row, column ;

  if (r==0)
    return 0;

  number = (rand() % 100) ;


  if (number > epsilon)
     {
       if ( q[r][c][0] > q[r][c][1] )
          return 0;
         /* {printf("\n1"); fflush(stdout); return 0;} */
       else if  ( q[r][c][0] < q[r][c][1] )
```

```
          return 1;
         /* {printf("\n2"); ffluqsh(stdout); return 1;} */
       else if  ( q[r][c][0] == q[r][c][1] )
          return (rand() % 2 );
         /* {printf("\n3");fflush(stdout);  return (rand() % 2 );} */
     }
  else
          return (rand() % 2 );
         /* {printf("\n4");fflush(stdout); return (rand() % 2 );} */
 }

  int states_optimal_action(int r ,int c)
 {
  if (r == 0)
    return 0;
  if ( q[r][c][0] > q[r][c][1] )
    return 0;
  else
    return 1;
 }

  int request_reward(int a)
  {
    if (a==0)
      return 1;
    else if (a==1)
      return 2;
    else if (a==2)
      return 4;
    else if (a==3)
      return 8;
  }


 main (int argc, char ** argv)
{
  double rho = 0.0;
  int reward;
  int i, j, k;
  int server_free = n ; /* Current no. of free servers */
  int server_state[n];
  int action , step ;
  int cur_state[2], next_state[2];
  double next_optimal_q_value, best_q;

  /* Making all servers initially free */
  for (i = 0; i< n; i++)
    server_state[i] = 1;

   /* Initialization */
   srand(123456);

   cur_state[0] = n;
   cur_state[1] = req_priority();
   server_free = n;

   for(step = 0; step <= iterations ; step++)
     {
       /* Choose an action */
       action = choose_action(cur_state[0], cur_state[1]);

       /* Find reward */
       reward = request_reward(cur_state[1]);
```

```
        if (action==0)
          reward = 0;

        /* Whether to free a busy server or not */
        for(i = 0; i<n ; i++)
          {
            if (server_state[i] == 0)
              server_state[i] = free_server(); /* free the busy server */
          }

        /*Find out how many servers are free */
        server_free = 0;
        for(i = 0; i<n ; i++)
          {
            if (server_state[i] == 1)
              server_free++;
          }

        if (action == 1)
          server_free--;

        if (server_free<0 || server_free>=n+1)
          {
            printf("error!\n");
            getchar();
          }


        for (i=0; i<server_free; i++)
          server_state[i] = 1;

        for (i=server_free; i<n; i++)
          server_state[i] = 0;

        next_state[0] = server_free;
        next_state[1] = req_priority();

        /*printf("%d %d\n",  next_state[0],   next_state[1]);
          getchar();*/

        if (next_state[0]==0)
          next_optimal_q_value = q[next_state[0]][next_state[1]][0];
        else
          next_optimal_q_value = max(q[next_state[0]][next_state[1]][0],
                                     q[next_state[0]][next_state[1]][1]);

        q[cur_state[0]][cur_state[1]][action]
          += alpha*(  reward - rho
                      + next_optimal_q_value
                      - q[cur_state[0]][cur_state[1]][action] );


        if (cur_state[0]==0)
          best_q = q[cur_state[0]][cur_state[1]][0];
        else
          best_q = max(q[cur_state[0]][cur_state[1]][0],
q[cur_state[0]][cur_state[1]][1]);



        if (fabs(q[cur_state[0]][cur_state[1]][action]-best_q)<=0.00000001)
          rho += beta*(reward  - rho
                       + next_optimal_q_value
```

```
                     - best_q);

      cur_state[0] = next_state[0];
      cur_state[1] = next_state[1];

      if (step % 500000 == 0)
        {
          printf("\n Step: %d ", step);
          for( i = 0; i <= n; i++)
            for( j = 0; j <= 3 ; j++)
              for( k = 0; k <= 1 ; k++)
                printf("\n q[%d][%d][%d] = %lf",i,j,k,q[i][j][k]);
          getchar();
        }


    }
  printf("rho=%lf\n", rho);
}
```

```
;-*- Package: (nstep) -*-

;;; A simulation of a TD(lambda) learning system to predict the expected outcome
;;; of a discrete-state random walk like that in the original 1988 TD paper.

;;; This version for n-step methods.  That n is the variable NN.
(defvar NN 1)


(defpackage :nstep
  (:use :common-lisp :g :ut :graph)
  (:nicknames :nstep))

(in-package :nstep)


(defvar n 5)                            ; the number of nonterminal states
(defvar w)                              ; the vector of weights = predictions
(defvar e)                              ; the eligibility trace
(defvar lambda .9)                      ; trace decay parameter
(defvar alpha 0.1)                      ; learning-rate parameter
(defvar standard-walks nil)             ; list of standard walks
(defvar targets)                        ; the correct predictions
(defvar right-outcome 1)
(defvar left-outcome -1)
(defvar initial-w 0.0)

(defun setup (num-runs num-walks)
  (setq w (make-array n))
  (setq e (make-array n))
  (setq standard-walks (standard-walks num-runs num-walks))
  (length standard-walks))

(defun init ()
  (loop for i below n do (setf (aref w i) initial-w))
  (setq targets
        (loop for i below n collect
              (+ (* (- right-outcome left-outcome)
                    (/ (+ i 1) (+ n 1)))
                 left-outcome))))

(defun init-traces ()
  (loop for i below n do (setf (aref e i) 0)))

(defun learn (x target)
  (if (= lambda 0)
    (incf (aref w x) (* alpha (- target (aref w x))))
    (progn
      (loop for i below n do (setf (aref e i) (* lambda (aref e i))))
      (incf (aref e x) 1)
      (loop for i below n
            with error = (- target (aref w x))
            do (incf (aref w i) (* alpha error (aref e i)))))))

(defun process-walk (walk)
  (destructuring-bind (outcome states) walk
    (init-traces)
    (loop for s1 in states
          for s2 in (rest states)
          do (learn s1 (aref w s2)))
    (learn (first (last states)) outcome)))

(defun process-walk-nstep (walk)
  (destructuring-bind (outcome states) walk
```

```lisp
   (loop for s1 in states
          for rest on states
          do (learn s1 (if (>= NN (length rest))
                         outcome
                         (aref w (nth NN rest)))))))

(defun standard-walks (num-sets-of-walks num-walks)
  (loop repeat num-sets-of-walks
        with random-state = (ut::copy-of-standard-random-state)
        collect (loop repeat num-walks
                      collect (random-walk n random-state))))

(defun random-walk (n &optional (random-state *random-state*))
  (loop with start-state = (truncate (/ n 2))
        for x = start-state then (with-prob .5 (+ x 1) (- x 1) random-state)
        while (AND (>= x 0) (< x n))
        collect x into xs
        finally (return (list (if (< x 0) left-outcome right-outcome) xs))))

(defun residual-error ()
  "Returns the residual RMSE between the current and correct predictions"
  (rmse 0 (loop for w-i across w
                for target-i in targets
                collect (- w-i target-i))))

(defun learning-curve (alpha-arg lambda-arg)
  (setq alpha alpha-arg)
  (setq lambda lambda-arg)
  (multi-mean
   (loop for walk-set in standard-walks
         do (init)
         collect (cons (residual-error)
                       (loop for walk in walk-set
                             do (process-walk walk)
                             collect (residual-error))))))

(defun learning-curve-nstep (alpha-arg NN-arg)
  (setq alpha alpha-arg)
  (setq NN NN-arg)
  (setq lambda 0)
  (multi-mean
   (loop for walk-set in standard-walks
         do (init)
         collect (cons (residual-error)
                       (loop for walk in walk-set
                             do (process-walk-nstep walk)
                             collect (residual-error))))))
```

```lisp
;-*- Package: (discrete-walk) -*-

;;; A simulation of a TD(lambda) learning system to predict the expected outcome
;;; of a discrete-state random walk like that in the original 1988 TD paper.

;;; This version for n-step methods.  That n is the variable NN.
(defvar NN 1)


(defpackage :discrete-walk
  (:use :common-lisp :g :ut :graph)
  (:nicknames :dwalk))

(in-package :dwalk)


(defvar n 5)                            ; the number of nonterminal states
(defvar w)                              ; the vector of weights = predictions
(defvar delta-w)
(defvar e)                              ; the eligibility trace
(defvar lambda .9)                      ; trace decay parameter
(defvar alpha 0.1)                      ; learning-rate parameter
(defvar standard-walks nil)             ; list of standard walks
(defvar targets)                        ; the correct predictions
(defvar right-outcome 1)
(defvar left-outcome -1)
(defvar initial-w 0.0)

(defun setup (num-runs num-walks)
  (setq w (make-array n))
  (setq delta-w (make-array n))
  (setq e (make-array n))
  (setq standard-walks (standard-walks num-runs num-walks))
  (length standard-walks))

(defun init ()
  (loop for i below n do (setf (aref w i) initial-w))
  (setq targets
        (loop for i below n collect
              (+ (* (- right-outcome left-outcome)
                    (/ (+ i 1) (+ n 1)))
                 left-outcome))))

(defun init-traces ()
  (loop for i below n do (setf (aref e i) 0)))

(defun learn (x target)
  (if (= lambda 0)
    (incf (aref delta-w x) (* alpha (- target (aref w x))))
    (progn
      (loop for i below n do (setf (aref e i) (* lambda (aref e i))))
      (incf (aref e x) 1)
      (loop for i below n
            with error = (- target (aref w x))
            do (incf (aref delta-w i) (* alpha error (aref e i)))))))

(defun process-walk (walk)
  (destructuring-bind (outcome states) walk
    (loop for i below n do (setf (aref delta-w i) 0))
    (unless (eq lambda 0) (init-traces))
    (loop for s1 in states
          for s2 in (rest states)
          do (learn s1 (aref w s2)))
    (learn (first (last states)) outcome)
    (loop for i below n do (incf (aref w i) (aref delta-w i)))))

(defun process-walk-nstep (walk)
  (destructuring-bind (outcome states) walk
    (loop for i below n do (setf (aref delta-w i) 0))
    (unless (eq lambda 0) (init-traces))
    (loop for s1 in states
          for rest on states
          do (learn s1 (if (>= NN (length rest))
                         outcome
                         (aref w (nth NN rest)))))
    (loop for i below n do (incf (aref w i) (aref delta-w i)))))

(defun standard-walks (num-sets-of-walks num-walks)
  (loop repeat num-sets-of-walks
        with random-state = (ut::copy-of-standard-random-state)
        collect (loop repeat num-walks
                      collect (random-walk n random-state))))

(defun random-walk (n &optional (random-state *random-state*))
  (loop with start-state = (truncate (/ n 2))
        for x = start-state then (with-prob .5 (+ x 1) (- x 1) random-state)
        while (AND (>= x 0) (< x n))
        collect x into xs
        finally (return (list (if (< x 0) -1 1) xs))))

(defun residual-error ()
  "Returns the residual RMSE between the current and correct predictions"
  (rmse 0 (loop for w-i across w
                for target-i in targets
                collect (- w-i target-i))))

(defun learning-curve (alpha-arg lambda-arg)
  (setq alpha alpha-arg)
  (setq lambda lambda-arg)
  (multi-mean
   (loop for walk-set in standard-walks
         do (init)
         collect (cons (residual-error)
                       (loop for walk in walk-set
                             do (process-walk walk)
                             collect (residual-error))))))

(defun learning-curve-nstep (alpha-arg NN-arg)
  (setq alpha alpha-arg)
  (setq NN NN-arg)
  (setq lambda 0)
  (multi-mean
   (loop for walk-set in standard-walks
         do (init)
         collect (cons (residual-error)
                       (loop for walk in walk-set
                             do (process-walk-nstep walk)
                             collect (residual-error))))))
```

```
; This code illustrates linear generalization in a 1D case.  The features
; correspond to patches of the 0-1 interval.  We vary the width of the patches
; to see the effect on generalization after various amounts of training.

; In particular, we image a case with 100 patches, each centered .01 apart,
; and with widths of either 3, 10 or 30 (or whatever).  For each we train on
; 100 input points also distributed uniformly over the interval.  The target
; function is a step function from 0 to 1 at the halfway point across the
; interval.  We plot the resultant estimated function after various numbers
; of passes through the training set.

(defvar w)                              ; the weights
(defvar deltaw)                         ; the change in the weights
(defvar n)                              ; number of patches (100)
(defvar widths)                         ; the list of patch widths
(defvar width)                          ; the current width
(defvar alpha)                          ; step size

(defvar random-state (make-random-state))

(defun setup ()
  (setq alpha 0.2)
  (setq n 100)
  (setq w (make-array n))
  (setq deltaw (make-array n))
  (setq widths '(3 9 27)))

(defun init ()
  (loop for i below n do
        (setf (aref w i) 0.0)))

(defun f (x)
  (loop for i below n
        when (within-patch x i) sum (aref w i)))

(defun targetf (x)
  (if (and (>= x .4) (< x .6)) 1.0 0.0))

(defun within-patch (x i &optional (width width))
  (<= (abs (- x (/ i (float n)))) (/ width 2.0 n)))

(defun train ()
  (loop for j below n
        with x = (+ .25 (random 0.5 random-state))
        with target  = (targetf x)
        with f = (f x)
        with alpha-error = (* (/ alpha width) (- target f))
        when (within-patch x j) do (incf (aref w j) alpha-error)))

(defun show-f (context xbase ybase xscale yscale)
  (loop for i from (/ n 4) below (- n (/ n 4))
        for lastix = nil then ix
        for x = (/ i (float n))
        for ix = (round (+ xbase (* x xscale)))
        for lastiy = nil then iy
        for y = (f x)
        for iy = (round (+ ybase (* y yscale)))
        with color = (g-color-name context :black)
        when lastix do
        ;(print (list i x ix lastix y iy lastiy))
        (gd-draw-lineseg context lastix lastiy ix iy color)))

(defun show-target (context xbase ybase xscale yscale)
  (loop for i from (/ n 4) below (- n (/ n 4))
        for lastix = nil then ix
        for x = (/ i (float n))
        for ix = (round (+ xbase (* x xscale)))
        for lastiy = nil then iy
        for y = (targetf x)
        for iy = (round (+ ybase (* y yscale)))
        with color = (g-color-name context :black)
        when lastix do
        ;(print (list i x ix lastix y iy lastiy))
        (gd-draw-lineseg context lastix lastiy ix iy color)))

(defun sequence (context xbase ybase offset
                         width-arg alpha-arg num-examples-list)
  (setq width width-arg)
  (setq alpha alpha-arg)
  (init)
  (gd-fill-rect context (+ xbase 100) (+ ybase 200)
                (+ xbase 300) (- ybase (* (length num-examples-list) offset) 200)
                (g-color-name context :white))
  (show-target context xbase ybase 400 80)
  (loop for num-examples in num-examples-list
        for i from 1
        do
        (loop repeat num-examples do (train))
        (show-f context xbase (- ybase (* i offset)) 400 80)))

(defun figure (context)
  (let ((examples '(10 30 120 480 1920 7680 30720));50 150 600 2400 9600));40 40 80
160 320 640 1280 2560)); 5120 10240))
        (ybase 640)
        (offset 80)
        (alph .2)
        (advances 0))
    (loop for n in examples
          for y from (- ybase offset) by (- offset)
          sum n into sum
          do (gd-draw-text context (format nil "~A" sum)
                           '("Helvetica" 24) 0 y (g-color-name context :black)))
    (standardize-random-state random-state) (advance-random-state advances random-
state)
    (sequence context 0 ybase offset 3 alph examples)
    (standardize-random-state random-state) (advance-random-state advances random-
state)
    (sequence context 300 ybase offset 9 alph examples)
    (standardize-random-state random-state) (advance-random-state advances random-
state)
    (sequence context 600 ybase offset 27 alph examples)))


(defun scrap-figure (c)
    (start-picture c)
    (figure c)
    (put-scrap :pict (get-picture c)))
```

```
(defvar v (make-array 7))
(defvar w (make-array 7))
(defvar delta-w (make-array 7))
(defvar alpha .1)
(defvar epsilon 0.01)
(defvar gamma 0.99)
(defvar x (make-array '(7 7) :initial-contents
                      '((0 0 0 0 0 0 0)
                        (1 2 0 0 0 0 0)
                        (1 0 2 0 0 0 0)
                        (1 0 0 2 0 0 0)
                        (1 0 0 0 2 0 0)
                        (1 0 0 0 0 2 0)
                        (2 0 0 0 0 0 1))))

(defun init ()
  (loop for i from 0 to 5 do (setf (aref w i) 1))
  (setf (aref w 6) 10)
  (loop for i from 0 to 6 do (setf (aref delta-w i) 0.0))
)

(defun update ()
  (loop for i from 0 to 6 do (setf (aref delta-w i) 0.0))
  (loop for i from 1 to 6 do
        (setf (aref v i)
              (loop for j from 0 to 6 sum (* (aref w j) (aref x i j)))))
  (loop for i from 1 to 5
        for alpha-error = (* alpha (- (* gamma (aref v 6)) (aref v i))) do
        (loop for j from 0 to 6 do
              (incf (aref delta-w j) (* alpha-error (aref x i j)))))
  (loop for j from 0 to 6
        with alpha-error1 = (* alpha epsilon (- 0 (aref v 6)))
        with alpha-error2 = (* alpha (- 1 epsilon) (- (* gamma (aref v 6)) (aref v
6)))
        do (incf (aref delta-w j) (* alpha-error1 (aref x 6 j)))
        do (incf (aref delta-w j) (* alpha-error2 (aref x 6 j))))
  (loop for i from 0 to 6 do (incf (aref w i) (aref delta-w i))))


#|
? (init)
NIL
? (setq alpha 0.01)
0.01
? (q (setq data (loop repeat 10000 collect (list (aref w 0) (aref w 1) (aref w 6)) do
(update))))
? (q (setq data (reorder-list-of-lists data)))
? (q (setq log-data (loop for list in data collect
                          (loop for d in list collect
                                (if (> d 0)
                                  (max 0 (log d 10))
                                  (min 0 (- (log (- d) 10))))))))
? (gn 5000 log-data)
NIL
?
? (q (setq reduced-data
           (loop for list in log-data collect
                 (loop while list
                       collect (first list)
                       do (setq list (nthcdr 10 list))))))
;reduced data:
(q (setq reduced data
         ((0 0.743590923721095 1.01766219063027 1.193813999706169 1.3257484974510294
1.432236226973832 1.5220708975099817 1.6000845586478505 1.669217769523681
1.7313923030092346 1.7879320004848647 1.839786333524481 1.8876581880450238
1.93208122913197 1.9734689643558088 2.0121470679589546 2.048375363718498
2.0823631781162586 2.114280304915592 2.144264981284994 2.1724297761532547
2.198865984806183 2.223646929710347 2.2468304413467015 2.2684607085155686
2.2885696296458096 2.307177755569855 2.3242948839456736 2.339920341996758
2.3540429746818132 2.3666408376419525 2.37768057644211 2.3871164538517626
2.394888963007749 2.4009229333814903 2.405124994397533 2.4073802020551125
2.4075475471176615 2.4054539332847433 2.400886013267928 2.3935789531228244
2.383200676787571 2.3693292676296163 2.3514196688886018 2.3287530113143213
2.300356464346572 2.2648703498326 2.2203145402592104 2.1636459704522326
2.08983335306259 1.9896356396127306 1.8430213381021932 1.59080319936432
0.7407356142438424 -1.4886794929522706 -1.8451921290129314 -2.0498099956953095 -
2.196589279845079 -2.3124312110945304 -2.408858207389486 -2.491878097677662 -
2.565022220752951 -2.630542243042043 -2.6899611190990593 -2.744355962903856 -
2.794515560618568 -2.8410337804264905 -2.884367826716676 -2.9248760945954877 -
2.9628436222608334 -2.998499701286641 -3.032030358167704 -3.0635873813442824 -
3.0932949594381407 -3.121254627499104 -3.1475489873533204 -3.1722445197713807 -
3.1953937082130404 -3.2170366275348976 -3.237202104837507 -3.2559085264283953 -
3.2731643400657178 -3.2889682821115684 -3.3033093427422617 -3.316166467204251 -
3.3275079757445165 -3.337290667727791 -3.3454585547483777 -3.351941140836871 -
3.356651131707685 -3.3594814042148395 -3.3603009936954913 -3.3589497476540284 -
3.3552311276514617 -3.3489023803360753 -3.3396608779537984 -3.327124728769123 -
3.3108045503912025 -3.2900611278898406 -3.264039581231984 -3.2315624773538767 -
3.1909467791360266 -3.1396685736005625 -3.073692863575478 -2.9859642405934785 -
2.862361955381825 -2.667270627376725 -2.247547474940148 2.1338610541983196
2.6761884286402653 2.923550954946462 3.089485407015026 3.2162376306472362
3.319733831284352 3.407723451281857 3.4845653028059154 3.5529547239884613
3.614674007709349 3.670962045487803 3.722713814925774 3.7705958138093174
3.8151166403367633 3.8566721181219306 3.895575228753132 3.932076587090917
3.9663788144223346 3.998646849860823 4.029015482705157 4.05759493547027
4.0844750475380325 4.109728431489631 4.13341285798713 4.155573047180666
4.176241990996307 4.19544189263588 4.213184781665453 4.229472841525539
4.24429846861552 4.25764406637356 4.269481562385851 4.279771619963276
4.288462496078413 4.295488472844871 4.300767756756516 4.30419969411508
4.305661085401154 4.305001284515281 4.302035622294472 4.296536466081843
4.288220863451978 4.2767331189722535 4.261619630709511 4.242291498949984
4.217967047554955 4.18757978033449 4.149623428211775 4.101874258083447
4.040851688371752 3.96065104805589 3.8499971446518213 3.6827677305218107
3.3693165795135567 -2.5499293769700393 -3.514996745139801 -3.8074633416560624 -
3.9909673360027442 -4.127131646090098 -4.236528909235187 -4.328589147584981 -
4.408427029161088 -4.479128275509111 -4.5426972431513395 -4.600508520999946 -
4.653544739299014 -4.702531587720051 -4.748019126651269 -4.790433133142877 -
4.830108811903571 -4.867313657564834 -4.90226338757086 -4.935133303336088 -
4.966066548014144 -4.995180203055304 -5.022569843778609 -5.0483129711887065 -
5.0724716057113435 -5.095094241011623 -5.116217296305433 -5.135866163598655 -
5.154055915791404 -5.1707917184478305 -5.186068969437787 -5.199873174543177 -
5.212179551822583 -5.222952341503789 -5.232143779772866 -5.2396926720197365 -
5.2455224711216655 -5.249538725174227 -5.251625700572926 -5.251641900797947 -
5.2494140728342344 -5.24472909530361 -5.237322829100649 -5.226864500246938 -
5.212934322649606 -5.194990558175308 -5.1723194466046145 -5.143956108444713 -
5.108553597012589 -5.0641531265192 -5.007749864660341 -4.934387849553118 -
4.834996889728672 -4.690031872179976 -4.442455122453738 -3.652707038232502
4.315346289539543 4.679707226873661 4.886565597343564 5.034372723739953
5.150784322470198 5.247559445382907 5.33080385256656 5.404096695064507
5.469715717723319 5.529199277175606 5.583634143549693 5.633815442269407
5.680341356367166 5.723672111848381 5.764168267005693 5.802116434235923
5.8377470631294495 5.871247036052415 5.902768772202625 5.932436918830747
5.960353334381957 5.986600834657938 6.0112460229322515 6.034341425863786
6.055927089950372 6.076031746559887 6.09467362002383 6.1118609281970935
6.127592105124749 6.141855758759333 6.154630361282352 6.165883653948304
```

```
6.175571730897117 6.183637745190159 6.190010152926611 6.194600374130886
6.197299696803534 6.197975174701973 6.196464156504752 6.192566911462276
6.186036545784904 6.176564966377091 6.163762918277493 6.147130858468315
6.126015148241738 6.099539724260189 6.066494726126497 6.025144842942116
5.972876112640977 5.9054840598517115 5.815551997349065 5.688027646057652
5.483934810830967 5.021543338348691 -5.048410816006276 -5.53933848019666 -
5.77700737591667 -5.938827712100194 -6.063295713764958 -6.165332754762839 -
6.252305169584405 -6.32839393140638 -6.396200555974099 -6.457453144623656 -
6.513357091003548 -6.564785541719162 -6.612390148464143 -6.65666905230293 -
6.698010489213224 -6.736721792359023 -6.773049275732815 -6.807192218493238 -
6.839312913413947 -6.869544016802517 -6.897994001956981 -6.924751248741484 -
6.949887130130266 -6.97345834416032 -6.995508664223877 -7.016070228536522 -
7.03516445258172 -7.052802621018474 -7.068986194400691 -7.083706848613963 -
7.096946249311341 -7.108675548234611 -7.118854571587634 -7.127430650821707 -
7.1343370210399515 -7.139490678562863 -7.142789542318351 -7.144108696345915 -
7.1432953911879435 -7.140162331083304 -7.134478539079963 -7.125956716299082 -
7.114235390860164 -7.098853090444145 -7.079209882854189 -7.054508104754528 -
7.023657153911895 -6.985112596718306 -6.936586434146526 -6.874480741190235 -
6.7926512830527 -6.679245778634511 -6.506309280797947 -6.173613778771936
5.576041266914911 6.380040436547238 6.660668778633563 6.8398040743468345
6.9736829799272915 7.081666956801822 7.172761218637834 7.2518928944759935
7.322052061606485 7.385189168944012 7.442646094885387 7.495384458415738
7.544115825195638 7.589380400321739 7.631596864896959 7.671095174643723
7.708138848137196 7.742940525936823 7.775673080989805 7.806477703626378
7.83546987614625 7.862743840295438 7.888375964018224 7.912427286023002
7.934945431509074 7.955966034123311 7.975513758190673 7.993602985378317
8.010238207203235 8.025414146425977 8.039115614362789 8.051317095845643
8.061982037424318 8.071061795767358 8.078494180010729 8.084201491193866
8.08808791977472 8.090036102194775 8.089902549536154 8.08751152908583
8.082646775525273 8.075040084655798 8.064355313165736 8.050165412788765
8.031918554417805 8.008886508887732 7.9800828605958305 7.944127114095245
7.899005151914184 7.841613873244281 7.766804368602924 7.665068497381794
7.515608190955049 7.255770717478437 6.286813724629085 -7.190417327901858 -
7.53552721764736 -7.7366231299504475 -7.881643819302885 -7.996403924805664 -
8.092080322060907 -8.17453653258181 -8.247232877412253 -8.312381954962397 -
8.371483406661362 -8.42559953666462 -8.47550923951956 -8.521799496218065 -
8.56492254198327 -8.60523307204226 -8.643013289703648 -8.678490255203974 -
8.711848192552093 -8.743237396235767 -8.772780784157233 -8.8005787816084 -
8.826712994716228 -8.851248986024231 -8.874238368519134 -8.895720369039395 -
8.915722966409406 -8.934263676809385 -8.951350034268629 -8.966979794685368 -
8.981140875223602 -8.993811025579951 -9.004957211904324 -9.014534676493659 -
9.022485614846351 -9.028737383719704 -9.033200115816514 -9.035763563111914 -
9.03629291294688 -9.03462320482752 -9.030551797991796 -9.023828060053459 -
9.014138994052846 -9.001088763604146 -8.984168761164979 -8.9627124845152 -
8.935824959194248 -8.902267309353626 -8.860257280040186 -8.807099613955922 -
8.738435671529686 -8.646516409457696 -8.51541671613074 -8.30289497506852 -
7.795458667144319 7.943822581079266 8.398415887579361 8.628345143209811
8.786779836329005 8.909338671907353 9.010143309977071 9.09624948919797
9.171692337166023 9.238995811543798 9.299842950642354 9.355411163156527
9.406555287461558 9.453914483472984 9.497978066977462 9.539127852088482
9.577666377753827 9.613836296650582 9.647834033574341 9.679819612926941
9.709923854958337 9.738253719790675 9.76489631729797 9.789921934300414
9.813386321275987 9.835332407267881 9.855791560841952 9.874784478734338
9.892321757042886 9.908404179005267 9.923022736145738 9.936158384013861
9.94778151830257 9.957851140259642 9.966313660209666 9.97310126237553
9.978129719781009 9.981295499987748 9.982471933256479 9.981504112375472
9.978202037971975 9.972331280755709 9.963600043347293 9.95164086082806
9.935984075766761 9.916018253456514 9.890929015989661 9.859600477287282
9.820448018485386 9.771115637467481 9.707880424327826 9.624340594676946
9.508014424609849 9.32889471232094 8.974026327228026 -8.547703642940181 -
9.244072653581846 -9.513670961240262 -9.688586483004016 -9.82022487593611 -
```

```
9.926812019801458 -10.016946692043135 -10.095374433249042 -10.16499196727388 -
10.227696822857743 -10.284798627037524 -10.337238211755981 -10.385713131011205 -
10.430753790939788 -10.472771803951456 -10.512091888587792 -10.548973590878745 -
10.583626473576363 -10.616220977926298 -10.646896337065732 -10.675766429287512 -
10.70292415777004 -10.728444752438268 -10.752388265430671 -10.774801448742432 -
10.795719145794909 -10.815165288596688 -10.833153562883643 -10.84968778125795 -
10.864761986201612 -10.878360288929652 -10.890456434737299 -10.901013069243456 -
10.909980661051465 -10.917296012729228 -10.922880260753805 -10.926636221913839 -
10.928444882084568 -10.928160732911113 -10.925605525780242 -10.92055980191181 -
10.91275122269061 -10.901838175921348 -10.887386204031275 -10.868833162146984 -
10.845435995065067 -10.816186156665902 -10.779668554868717 -10.733811754598165 -
10.675410249927172 -10.599112379520793 -10.494943988918845 -10.340744779777811 -
10.067451840661235 -8.72439919011566 10.06273422356325 10.390724348319985
10.586407425236386 10.728760076636506 10.841922885644651 10.93652992340954
11.018215669989672 11.09032706484118 11.155014146586772 11.213739199730153
11.26754083233128 11.317182171401383 11.36323928855244 11.406156620566922
11.446283130271125 11.483896703158516 11.519221075387097 11.552437860225274
11.583695262761605 11.613114498081156 11.64079457845017 11.666815915675661
11.69124304328868 11.714126669482354 11.735505208037177 11.755405889949559
11.773845526341384 11.790830969048978 11.806359296073119 11.820417732654134
11.832983303403799 11.844022195134553 11.853488792160633 11.861324323960815
11.86745503658278 11.871789760276211 11.874216690868938 11.874599122390196
11.872769748872818 11.868522969872366 11.86160434525441 11.851695875869158
11.838395000527926 11.821183831719198 11.799382668145892 11.77207707839931
11.737998234783035 11.695315220783009 11.641248049268684 11.571276200232731
11.477297805042195 11.342444494208227 11.12081657907327 10.557626405239354 -
10.832867057474905 -11.256650864938782 -11.479367377317812 -11.634567584041985 -
11.755280580520829 -11.85488518142784 -11.94014384265867 -12.014952555288408 -
12.081760760203943 -12.14220797320731 -12.197444465903606 -12.248307266194605 -
12.295423349921636 -12.339273406010044 -12.380232959216816 -12.418599845406364 -
12.454613120190873 -12.488466400255014 -12.520317475504639 -12.550295355288142 -
12.578505505422688 -12.605033780030949 -12.629949390525596 -12.653307147873882 -
12.675149142674764 -12.695505977989734 -12.714397634451224 -12.731834020897121 -
12.747815243285315 -12.76233160755437 -12.775363356595841 -12.786880126019625 -
12.796840086358475 -12.805188718959615 -12.81185714669289 -12.816759905431793 -
12.819791993050744 -12.820824961671859 -12.819701713624012 -12.816229501421716 -
12.810170381858642 -12.8012279720988 -12.789028688389003 -12.773094500915343 -
12.752802184015366 -12.727320182365794 -12.69550654533415 -12.655735067468784 -
12.605579019991772 -12.541184625091772 -12.45587781926729 -12.336501325328841 -
12.150782976083415 -11.770551218427775 11.488407142785382 12.106791734493022
12.366262331956383 12.537169973489497 12.66664919738992 12.771879088314039
12.861076354831324 12.938813824017627 13.007898725534258 13.070177565208764
13.126928739696007 13.179072882883366 13.22729389616318 13.272112615637038
13.313933736490137 13.35307684346074 13.389797582457657 13.424302490294524
13.456759614352444 13.487306258591795 13.516054718280635 13.54309657386024
13.568505929208627 13.592341858908169 13.614650248434266 13.635465155775695
13.654809783824549 13.672697124183813 13.689130311042561 13.704102705838034
13.7175977176054 13.729588348589939 13.74003643831958 13.748891560200413
13.756089500654307 13.761550218892657 13.765175141234117 13.76684358068859
13.766407979620642 13.763687533087303 13.758459533211129 13.750447428918967
13.73930402712827 13.724587295793235 13.705724522732197 13.681957428539413
13.65225467609986 13.615165415753472 13.56855869869876 13.509122039778347
13.431290152589964 13.324591477589118 13.165378684651474 12.877212483080482 -
11.282743259666585 -12.932746056132013 -13.24537111421139 -13.435949641746413 -
13.575740181293128 -13.687354246230619 -13.780918133216206 -13.861849144781534 -
13.933385757390049 -13.997617787165579 -14.055971387773885 -14.10946216455506 -
14.15883789298375 -14.204663997075443 -14.247377286505767 -14.28732110756963 -
14.32476910849485 -14.359941758673578 -14.393018102215176 -14.424144285463425 -
14.453439843569571 -14.481002393005575 -14.506911164286887 -14.531229671761833 -
14.554007726162771 -14.575282933538043 -14.595081780711748 -14.613420375954464 -
14.63030488979777 -14.6457317219658 -14.65968740410676 -14.672148232692575 -
14.683079610574083 -14.692435057602541 -14.700154828477817 -14.706164046891626 -
```

```
14.710370225237785 -14.71265998278834 -14.712894693030705 -14.71090466778921 -
14.70648129664917 -14.699366261632868 -14.689236461554636 -14.675682464490995 -
14.658176883056914 -14.636026473200937 -14.608296785267202 -14.573688072021898 -
14.530318969664501 -14.475321162333113 -14.404004079308104 -14.307891022880593 -
14.169092770074288 -13.937596107306437 -13.302938380702814 13.716838781670516
14.114108927092735 14.33008793181287 14.482195406351678 14.601123108030803
14.69955898825754 14.783988395205732 14.85817453883303 14.92449525249934
14.984548014023174 15.039456779750529 15.090041251898155 15.136916523131722
15.180554850218067 15.221325598980679 15.259521992141261 15.295379551809685
15.329089132415705 15.360806323020366 15.390658347103608 15.418749194985669
15.445163479248711 15.469969346635633 15.493220676651761 15.514958727359854
15.535213340463875 15.55400378313244 15.571339278232912 15.587219254457208
15.601633330897183 15.614561035177354 15.625971238713 15.635821275455799
15.644055689778689 15.65060453252674 15.65538108829859 15.658278866588743
15.659167616505338 15.657888016487247 15.654244525395923 15.647995623009555
15.63884025179366 15.626398579825398 15.61018401196158 15.589561233643742
15.563681032586267 15.531374581652111 15.490972624386309 15.43997484128431
15.374390347072488 15.287256982043393 15.16469140805155 14.971914410762462
14.562330472004968 -14.409941515368738 -14.968331270524159 -15.21846714178979 -
15.385562853976925 -15.512959758775425 -15.616870237526248 -15.70515146846164 -
15.782211895997287 -15.850772914038423 -15.912631816770073 -15.969036750358713 -
16.020888718269543 -16.068858316478007 -16.113457033009166 -16.155082793012248 -
16.194050148066445 -16.23061091457885 -16.264968654066784 -16.29728905702766 -
16.327707525649444 -16.35633479271036 -16.383261131411114 -16.408559531300256 -
16.43228809821859 -16.454491857646143 -16.47520408679968 -16.494447262541083 -
16.512233684037064 -16.528565807469494 -16.543436312369504 -16.556827903413954 -
16.568712836174605 -16.579052138800314 -16.587794482213283 -16.59487462691797 -
16.60021134190762 -16.60370464590061 -16.60523215629596 -16.60464423571523 -
16.601757481578005 -16.5963458799965 -16.588128587489617 -16.57675271514472 -
16.561768486591426 -16.542592363024923 -16.51845042914934 -16.488287871457555 -
16.450616872480733 -16.40324466691805 -16.342746956474407 -16.263333211101546 -
16.15400034723494 -15.98947416880309 -15.684769551998073 14.714564602376177
15.800755065739223 16.09950444052725 16.285260634515605 16.422588689929086
16.532700329664205 16.62524629018666 16.705437796141688 16.776409514343445
16.840193267143505 16.898180253403112 16.9513637438553 17.00047656474668
17.046073746420134 17.08858463804631 17.128347082292464 17.165630568746472
17.200652356061376 17.23358895988599 17.264584497852706 17.293756847615388
17.321202246754307 17.346998757274346 17.371208883980113 17.393881547324884
17.415053550826997 17.434750640688712 17.45298822444267 17.469771792116084
17.48509706469008 17.49894987846071 17.511305798603615 17.522129439266067
17.53137344921363 17.538977099429903 17.54486437937531 17.548941467883154
17.551093386854244 17.55117956144302 17.54902788373929 17.544426681929043
17.537113688406027 17.526760597526813 17.512950956643206 17.49514765195172
17.47264354160249 17.44448357585783 17.409336080616047 17.365267375895602
17.309317023755035 17.236615705806848 17.138288126163165 16.99533828175311
16.75310705471476 16.02195231934097 -16.59668055965065 -16.970851604604412 -
17.18052193101746 -17.32966914079015 -17.44686909149752 -17.5441663167892 -
17.627784121493328 -17.70135892853203 -17.76719973156458 -17.826863391848114 -
17.881448341239043 -17.93175742455328 -17.978394142854487 -18.021822509912553 -
18.06240585960531 -18.100432889244562 -18.1361356495363 -18.16970227751307 -
18.201286194344508 -18.23101286217601 -18.25898481427382 -18.285285435617343 -
18.30998181880843 -18.33312691977498 -18.354761169818534 -18.374913653320647 -
18.39360292652811 -18.410837527536998 -18.426616207699293 -18.440927897908722 -
18.453751407815407 -18.46505484041597 -18.474794687080845 -18.482914547045922 -
18.489343388248408 -18.493993229611153 -18.496756073186088 -18.49749983968564 -
18.496062949519388 -18.492247021340294 -18.485806893320763 -18.476436741734442 -
18.463750353644638 -18.44725236996073 -18.42629507972179 -18.400011117155685 -
18.367203934611364 -18.326159694811835 -18.274301476084105 -18.207494666797604 -
18.118472015219172 -17.992568714659537 -17.792221926864976 -17.34818812763639
17.318425926108553 17.828804958963442 18.07030726982869 18.233772725216497
18.359160043383515 18.461787343365067 18.549173155849505 18.625569373349148
```

```
18.693615026301234 18.75505992843252 18.81112291697495 18.86268591267495
18.9104065422183 18.95478716103501 18.996219067461432 19.035011878040443
19.071413648640295 19.10562501500409 19.13780934694942 19.168100171744427
19.196606679811563 19.223417852309876 19.248605575949714)
          (0 0.4536114001301142 0.6841183679442987 0.8433616730853838
0.9667946430330369 1.0684938048891988 1.1554985188456184 1.2318372599722287
1.3000313530164012 1.3617631009826594 1.4182101648358685 1.4702281918975262
1.5184574893933405 1.5633887437192635 1.6054053018971703 1.644811367383014
1.6818513764417313 1.7167236557855179 1.7495902576524418 1.780584170212422
1.809814681552654 1.8373714151925908 1.8633273889503261 1.8877413401450258
1.9106594870136806 1.9321168458631288 1.9521381878293598 1.9707386930479207
1.9879243401710203 2.0036920531693987 2.018029613520357 2.030915332829012
2.04231746739664 2.052193340905047 2.0604881226204834 2.067133184185083
2.0720439250776588 2.075116910560115 2.076226099174876 2.075217838067382
2.071904154514271 2.0660536386686754 2.057378838948846 2.045518474712703
2.0300117164525644 2.0102599077064136 1.9854676060822993 1.9545479348022858
1.915962749222925 1.8674350622724467 1.805387543952958 1.723718657622348
1.6106801789634613 1.4386519188434856 1.1093939375177784 -0.4777363522339745 -
1.3047963321595912 -1.5873231195827413 -1.7671040555398405 -1.9012744883481263 -
2.0094016620901507 -2.1005634441003784 -2.1797190826039157 -2.2498749916738108 -
2.312990764171465 -2.3704137603336317 -2.4231090555872195 -2.471790476999466 -
2.516999747980736 -2.5591565850182403 -2.5985916529978708 -2.6355689513515546 -
2.6703014361636814 -2.7029621719506993 -2.7336924441859733 -2.7626077522181682 -
2.7898022887383576 -2.815352313960965 -2.83931870416725 -2.861748868631272 -
2.882678170368819 -2.9021309449001036 -2.9201211811470653 -2.9366529056624775 -
2.95172029283653 -2.9653075074994883 -2.977388270760347 -2.987925123416227 -
2.996868342115638 -3.0041544395155224 -3.0097041479883506 -3.0134197426830864 -
3.0151814972770383 -3.0148429740134097 -3.0122247113029026 -3.007105658086562 -
2.9992113634862774 -2.988197371421335 -2.9736253213130657 -2.954927582048723 -
2.931353155876609 -2.901881571577957 -2.8650789994673063 -2.818842806625423 -
2.7599114399744127 -2.6828214933095187 -2.577339374533186 -2.4205333477082975 -
2.139564143781716 -0.09723687321299573 2.1668082411051164 2.4864691289634844
2.679423028359266 2.8204172846554347 2.9327652950866443 3.0268279260119213
3.108123179328447 3.17993995214873 3.2443961124933285 3.3029346738178083
3.356581993779646 3.4060930368114017 3.452038223329493 3.4948579507137305
3.5348982149841546 3.572434670660278 3.607689340915878 3.6408424987608727
3.67204128225835 3.701406042903436 3.729035082754304 3.755008220144366
3.7793894844823317 3.8022291482972705 3.8235652418577963 3.8434246517417194
3.861823872962457 3.8787694603194423 3.8942582055756794 3.9082770507372855
3.9208027324165693 3.9318011364629557 3.941226324127211 3.949019169030421
3.9551055155148713 3.95939372976725 3.961771459670818 3.962101338607162
3.9602152476949617 3.95590656566333 3.9489195433895063 3.938934465693639
3.9255464669023743 3.9082344802431277 3.8863142805062467 3.8588647602550217
3.8246067967776773 3.7816927056458427 3.7273132010813104 3.656891509082458
3.562202246200322 3.426042792127391 3.201184478376109 2.6160522894174685 -
2.9362076065069753 -3.350644131998086 -3.5710429298556643 -3.72518639232443 -
3.845291081368423 -3.9444979161634457 -4.029474414503489 -4.104071221576832 -
4.170713377559907 -4.231026096407853 -4.286150707564868 -4.336918356726548 -
4.38395198792008 -4.427729430419979 -4.468624120806953 -4.506932330983709 -
4.5428919184809935 -4.576695563092844 -4.6085003078234426 -4.638434555434799 -
4.6666032699920414 -4.69309188277704 -4.717969241897745 -4.741289839719442 -
4.763095481277228 -4.783416507643533 -4.802272653068773 -4.8196735886051565 -
4.8356191845311285 -4.850099506860382 -4.863094547733597 -4.874573673984995 -
4.884494761078317 -4.892802959096534 -4.899429011170354 -4.904287009276823 -
4.9072714226924505 -4.908253162698937 -4.907074340798379 -4.903541215805671 -
4.897414572093 -4.888396364117078 -4.8761107863562705 -4.8600767645504 -
4.839666778648223 -4.814043005672188 -4.7820539638798625 -4.742058199778889 -
4.691602962481953 -4.626786100604099 -4.540832235171192 -4.420330070248463 -
4.232138939101227 -3.841901862304941 3.6074455914907206 4.202660987099854
4.4587020925317224 4.628222743197348 4.756947639582552 4.861700814903807
4.950567696470854 5.028061300606497 5.09695770114522 5.159085573134944
5.2157123674871695 5.267751622758129 5.315882435433737 5.360622268344514
```

```
5.402373375300119 5.441453527076711 5.478116990174835 5.512569230248818
5.544977446598274 5.575478259227432 5.604183401781888 5.631183985111261
5.65655371306653 5.680351312724182 5.702622361316991 5.723400637240819
5.742709083654318 5.760560444704812 5.776957612545119 5.791893705449732
5.8053518815517355 5.8173048773889375 5.8277142440248015 5.836529234281845
5.843685270427436 5.849101889484243 5.852680018775998 5.854298370549883
5.853808650688131 5.851029134824721 5.845735945513135 5.8376510139301825
5.826425134111348 5.811613538856271 5.792639694850703 5.768739808627124
5.73887427204234 5.701579226824229 5.65470200490785 5.59489091587849
5.516501923596961 5.4088784594140185 5.247811368091197 4.953967632871074 -
3.679261967661479 -5.030951219297728 -5.338481867621914 -5.52731394192849 -
5.666217427328632 -5.7772918508177815 -5.870490793126016 -5.951157018532732 -
6.022491557274777 -6.086563346501418 -6.144785973813906 -6.198167024583621 -
6.247448911249072 -6.29319333195704 -6.335834423720285 -6.375713562315494 -
6.4131029077523865 -6.448221780939044 -6.481248321928554 -6.512327951981063 -
6.541579614067893 -6.569100432209001 -6.594969219766952 -6.619249130836001 -
6.64198965859229 -6.663228122960248 -6.682990746842108 -6.701293388928638 -
6.718141977508066 -6.733532670819557 -6.747451753250685 -6.759875261367935 -
6.770768317854732 -6.780084133277659 -6.787762613217797 -6.79372847900053 -
6.797888770132407 -6.800129539671627 -6.8003114707569905 -6.798264018174658 -
6.7937774876545145 -6.786592163521896 -6.7763831037428535 -6.762738394500068 -
6.745127212222464 -6.722851406677654 -6.694969262559251 -6.660169784541536 -
6.616553203412517 -6.561218853953914 -6.4894135583051495 -6.3925215425756905 -
6.252269893781711 -6.017056987151835 -5.3516199970956775 5.819481106862743
6.208174388265714 6.421860616725928 6.572899535864832 6.69120572824905
6.789232875452671 6.8733713683444755 6.947338612654607 7.013487625207693
7.073401264915993 7.128194312553131 7.178680416406634 7.225470509685962
7.269033893324328 7.30973777144702 7.347873745828797 7.383676095750376
7.417334703184952 7.449004381705802 7.478811724991072 7.506860202572145
7.533233988437098 7.558000852873589 7.5812143457126115 7.6029154300492205
7.62313367753847 7.641888101989694 7.659187682375049 7.675031606283995
7.689409247988802 7.702299879852269 7.713672100247388 7.723482943890233
7.731676619662686 7.738182794192718 7.742914303211072 7.745764121824888
7.746601351240527 7.74526587007776 7.741561131586395 7.735244326760656
7.726012712099236 7.713484199707929 7.6971690978986365 7.676427715249706
7.650404435384958 7.617920662239299 7.577291449242982 7.525989562724506
7.459973735841377 7.372175220915389 7.248439130978102 7.053029064250865
6.631836711248405 -6.523618195323422 -7.063762058510443 -7.3107486643228405 -
7.476532023936461 -7.603205464285251 -7.706655110360732 -7.794615283072252 -
7.871437827878074 -7.939814431902807 -8.00152529379101 -8.057808020920074 -
8.10955676230591 -8.157437459413229 -8.201958325061803 -8.243514909308074 -
8.28241999593879 -8.318924054940656 -8.3532296007541 -8.385501493697847 -
8.41587446551409 -8.444458697676962 -8.471344001745155 -8.496602973395257 -
8.520293375728388 -8.542459929648004 -8.563135635549276 -8.582342712581388 -
8.600093213822701 -8.616389354192066 -8.631223570263494 -8.644578315445424 -
8.656425578627374 -8.666726097837778 -8.675428220960688 -8.682466340917253 -
8.687758799853624 -8.69120511123703 -8.692682283316095 -8.692039930929049 -
8.689093716692735 -8.683616435917852 -8.675325697517414 -8.663866556759409 -
8.648786438671522 -8.629497886464204 -8.605221317027711 -8.574893389799874 -
8.537012826303007 -8.48936427986137 -8.428482432097022 -8.348493619308535 -
8.238196757606731 -8.071701593661823 -7.760651441127218 6.904503000097749
7.898316198277789 8.192444081084991 8.376547336033774 8.51302176618364
8.622609374129908 8.714798898722762 8.794730751732853 8.865503723007125
8.929129509486854 8.98698714211845 9.040062097955245 9.089081976445472
9.134598167999235 9.177037405454044 9.216735599086144 9.253960776778307
9.288929067560339 9.321816096699374 9.352765266813226 9.381893870878686
9.409297659658874 9.435054282249933 9.459225886386443 9.481861077319834
9.50299637412947 9.522657260220516 9.540858894181085 9.55760652398047
9.572895628863261 9.586711797163714 9.599030332961975 9.609815568489182
9.619019840811173 9.626582068554079 9.632425834524245 9.636456839008929
9.638559530202336 9.638592632909742 9.636383168676495 9.631718363312046
```

```
9.624334525637677 9.613901472198645 9.60000021424279 9.58209011975512
9.559459011878326 9.531144363562623 9.495802885006306 9.451481806086957
9.395186942504965 9.321983143139258 9.222847127716648 9.078366429793098
8.832082485134988 8.057313745121942 -8.698030901115338 -9.06467543032329 -
9.272195763700411 -9.420319567932852 -9.536917854111046 -9.63381679029123 -
9.717149843419328 -9.79050968702433 -9.85618145289679 -9.915707875911073 -
9.970178487338742 -10.020390241384685 -10.066942581590872 -10.110296632159908 -
10.150813609835463 -10.188780621817575 -10.22442849783769 -10.257944418277226 -
10.289481040440768 -10.319163205332549 -10.347092931945035 -10.373353171589718 -
10.398010644135548 -10.42111797862348 -10.442715313427742 -10.46283146430424 -
10.481484735033188 -10.49868342023003 -10.514426030104545 -10.52870125023429 -
10.541487634025147 -10.55275300990446 -10.562453567841407 -10.570532568636523 -
10.576918592085077 -10.58152320304785 -10.58423786231155 -10.584929833524605 -
10.583436724946244 -10.579559132789054 -10.573050583072405 -10.563603532965237 -
10.550829465384632 -10.534229852561058 -10.513152494792482 -10.486723441180645 -
10.453736066596958 -10.412460282498063 -10.36029114353782 -10.2930412031503 -
10.203330266138488 -10.076200959053546 -9.87302531499156 -9.415215692148552
9.428407799757984 9.923880779315967 10.162463945447568 10.324678523440735
10.449367323006529 10.551546313425268 10.638618228879261 10.714781036067114
10.782645251625858 10.84394419130613 10.89988647970341 10.95134737336486
10.998979961483133 11.043283399436875 11.084646660497798 11.123377627734516
11.159723034740262 11.193882487535435 11.226018538455198 11.256264053744037
11.28472767958773 11.311497940839239 11.336646334370684 11.360229666210492
11.382291805886284 11.402864979152112 11.421970683148782 11.439620280671136
11.455815309039982 11.470547521612351 11.483798664331676 11.49553997432848
11.505731370879081 11.51432028925538 11.521240082898565 11.52640788576747
11.52972177996129 11.531057046541852 11.530261178278513 11.52714718268847З
11.521484469786397 11.51298624450369 11.501291705536898 11.48594029547479
11.466333366301656 11.441675118118289 11.410877762647733 11.372401329127815
11.323965349941906 11.261985691050194 11.180346385993408 11.067263672179797
10.895002640177186 10.564643656310846 -9.945016977903729 -10.763994645358641 -
11.045933856121403 -11.225560893006898 -11.359698732912832 -11.467843389064729 -
11.559047700765715 -11.638259935183775 -11.70848098847144 -11.771667422914568 -
11.829164833551708 -11.881937230273566 -11.930697787841117 -11.97598783520647 -
12.018226863295594 -12.05774542725096 -12.094807499935674 -12.129626073603688 -
12.162374298865524 -12.193193589377216 -12.22219961033803 -12.249486756020099 -
12.275131523948286 -12.299195065081188 -12.321725103888975 -12.342757363777727 -
12.362316592176844 -12.380417249651963 -12.39706390461196 -12.412251356785411 -
12.425964496619839 -12.438177892454673 -12.448855081196253 -12.457947519612924 -
12.465393130205806 -12.471114345073968 -12.475015509156254 -12.476979444384822 -
12.476862888629109 -12.4744903915183 -12.469646045850089 -12.46206211069416 -
12.451403055002379 -12.437242659195178 -12.419030246616321 -12.396039242219002 -
12.367285695500454 -12.331392957045924 -12.286353265111282 -12.229074836063507 -
12.154431062192879 -12.052964699414186 -11.904024646536664 -11.645630524823863 -
10.70234137171854 11.573361056587368 11.920533916254985 12.122263377706187
12.267593120439845 12.382537238388526 12.478336407349104 12.560880865108095
12.63364409234992 12.698845922950504 12.757990307534959 12.812142278030247
12.86208254105531 12.90839932963761 12.951545772966266 12.991877222737486
13.029676376293684 13.06517067389913 13.098544637815248 13.129948801054663
13.159506275777295 13.187317648360438 13.213464660985586 13.238012993330171
13.261014361291327 13.282508084103647 13.302522225495991 13.321074381618658
13.338172163798623 13.353813404665358 13.367986099620891 13.38066808026721
13.391826400703655 13.40141639996465 13.409380382375609 13.41564582972256
13.420123021212177 13.422701883735247 13.423247817298169 13.421596124676146
13.417544497076749 13.410842728874979 13.401178383267954 13.388156376216674
13.371269137265353 13.349851637395977 13.323011070530363 13.28951189261513
13.247577249056457 13.194521256900865 13.126003111491777 13.034312401846273
12.903626255552819 12.692091446454233 12.19022450029022 -12.324660611423178 -
12.783063199491483 -13.013841311039455 -13.172651581245683 -13.295423354512598 -
13.396365889600254 -13.48256920203857 -13.5580845896049 -13.625444636500221 -
13.686337405027562 -13.741943429502854 -13.793119596071088 -13.840506463490188 -
13.884594336783552 -13.925765750507125 -13.964323781849638 -14.000511494811994 -
```

```
14.034525635052926 -14.066526482111238 -14.096645062863322 -14.124988507815036 -
14.151644069921703 -14.17668215844496 -14.200158630746802 -14.222116511175622 -
14.242587255232422 -14.261591640899882 -14.279140342171324 -14.295234218969467 -
14.30986434036379 -14.323011742429555 -14.33464690666072 -14.344728927993302 -
14.353204321432319 -14.360005390701874 -14.36504804801564 -14.368228926168799 -
14.369421555194261 -14.368471273804364 -14.365188390937103 -14.359338871229944 -
14.350631430934618 -14.338699289897601 -14.323073726679974 -14.303144623162716 -
14.278099516744284 -14.246825422533375 -14.207742340528176 -14.158502101913095 -
14.095396236139852 -14.012052837906095 -13.896063684541637 -13.717664100592051 -
13.36545368215904 12.921426830366556 13.628184225046656 13.898983793418315
14.074366115060252 14.206254001867855 14.312997270839757 14.403239455916646
14.481746200533841 14.551424600148463 14.614178078077371 14.671319860724475
14.72379310350697 14.772296928532384 14.817362839722897 14.859403242100385
14.898743441903656 14.935643431253006 14.970313118734035 15.002923218987162
15.033613185319485 15.062497076563309 15.089667946644992 15.115201153727135
15.139156861188784 15.161581919548462 15.182511261451772 15.20196890165559
15.2199686045953 15.236514259711283 15.251599986541878 15.265209975668569
15.277318056289753 15.287886964960816 15.296867271185315 15.304195891970172
15.309794096275194 15.31356485725074 15.315389348757458 15.315122292564396
15.31258572691764 15.307560557364786 15.299774917264914 15.288887819169224
15.274465652579785 15.255947453042298 15.232591863673727 15.20339287663713
15.16693937421008 15.121166519038333 15.062880624280417 14.986753516358695
14.882865329828398 14.729216001013192 14.457513552596595 13.18031458869579 -
14.445961157745133 -14.77579591529981 -14.97207583161562 -15.114725113821182 -
15.22806628572041 -15.322793060055137 -15.404565238154932 -15.47674234431877 -
15.541481380667948 -15.600248798687877 -15.654085853981583 -15.703757437590687 -
15.749840842978104 -15.792781381162028 -15.832928658108761 -15.870561044087603 -
15.905902648701746 -15.939135378699191 -15.970407674176203 -15.999840941850762 -
16.027534353098286 -16.053568454285994 -16.07800789495094 -16.10090348535449 -
16.122293731060495 -16.142205947547676 -16.160657025650863 -16.177653894396844 -
16.19319370855418 -16.20726377178363 -16.219841190941274 -16.2308922413047 -
16.240371404650098 -16.248220020263673 -16.254364460526073 -16.258713703916037 -
16.26115612346258 -16.261555228908907 -16.259743981667654 -16.255517118887738 -
16.24862063502884 -16.23873710224509 -16.22546472796928 -16.20828668626896 -
16.186524787267796 -16.15926683024077 -16.12524742644357 -16.08264125687433 -
16.028678104523067 -15.958856177915317 -15.865114627968985 -15.730695629293052 -
15.510135227748675 -14.953927429289868 15.214353352368073 15.641390363581053
15.864898703472896 16.020457815399652 16.141376697191696 16.241115568070743
16.32646925524565 16.40134917655058 16.46821306076377 16.528705275495494
16.583979122914222 16.63487362274064 16.682017114992583 16.725891255433805
16.766872274278523 16.80525853530894 16.841289498171886 16.875159095160445
16.907025367463604 16.937017528190673 16.965241211390282 16.99178241264047
17.016710464566643 17.04008028409861 17.0619340564634 17.08230247118125
17.10120558982621 17.118653398980083 17.134646081278824 17.1491740203434
17.162217539880363 17.173746361759157 17.183718750856592 17.19208029409542
17.198762235041784 17.203679250341057 17.206726505194563 17.207775754281492
17.20667014959577 17.20321725705576 17.1971795344346 17.188261122492104
17.176089136666864 17.16018650463327 17.139931350461588 17.114494087388547
17.082735755940828 17.043034931354303 16.99297303351398 16.92871140765991
16.843607590299733 16.72458323873563 16.53963418566148 16.16245715239904 -
15.864883480556331 -16.491044832413312 -16.751620397468592 -16.922971686764665 -
17.052691430315523 -17.158073056505117 -17.2473753672487 -17.325190316039166 -
17.394335077672167 -17.456661842236926 -17.51345249231297 -17.565629919700317 -
17.613879553875606 -17.658723303177354 -17.700566638482474 -17.739729720669153 -
17.776468633634494 -17.810990254220048 -17.843462898964866 -17.874024088861148 -
17.902786297365772 -17.92984125389439 -17.955263189221576 -17.97911128816103 -
18.001431533963792 -18.022258073307746 -18.041614191477414 -18.059512958577375 -
18.075957585583392 -18.090941511080963 -18.104448223712378 -18.11645081002962 -
18.1269112010879 -18.135779072008866 -18.142990324747522 -18.148465052448703 -
18.152104839711292 -18.153789190077386 -18.153370779444895 -18.150669094350903 -
18.145461797607698 -18.137472819046295 -18.126355603156448 -18.111668983898046 -
```

```
18.09284145837079 -18.069116491234887 -18.039465359662454 -18.002441322213837 -
17.955920281679763 -17.896602102562035 -17.818946253530004 -17.712539149825428 -
17.55390854125176 -17.267504754639774 15.495920247971952 17.316217601879373
17.630503069140957 17.82164489845545 17.96172040144551 18.0735074467373
18.16718815316713 18.248203840802592 18.319805027702433 18.384088234420304
18.4424836453412 18.49600943613055 18.545415099889425 18.591267253243597
18.634003560945423 18.67396799946089 18.711434692770816 18.746624477288343
18.779716685938943 18.810857696634564 18.840167233590893 18.867743070454566)
          (1.0 0.9986133168019985 0.9964089521980072 0.9933174076997987
0.9892618980076057 0.9841569439712616 0.977906628893901 0.970402416655308
0.9615203921191385 0.9511177291421135 0.9390281100079143 0.92505569718736
0.9089670686629543 0.890480227882003 0.8692493105352523 0.8448427878732332
0.8167115299361626 0.7841404729318451 0.7461726101974508 0.701483783688533
0.6481642757887104 0.58330911517505 0.502172426517505 0.3961744292011242
0.24716942268148373 0.004133433989857941 0 0 -0.16262488492662167 -0.3681418185895976
-0.5105754686162159 -0.6201071200537452 -0.7092592242186215 -0.7844394256458623 -
0.8493635678076535 -0.9063743980154273 -0.9570395431329056 -1.002455098855808 -
1.0434131585254927 -1.0805003681339105 -1.1141588949763346 -1.1447256774557555 -
1.172458491563213 -1.1975536629671788 -1.2201582700075622 -1.2403785671370344 -
1.2582857010173616 -1.2739193852227244 -1.2872899335771806 -1.2983788637297053 -
1.3071381328087865 -1.313487928210263 -1.317312784118602 -1.3184555997976621 -
1.31670885834891 -1.3118019179970786 -1.3033825539117725 -1.290989744691329 -
1.2740125821481132 -1.2516262075395346 -1.2226877609685727 -1.1855584380925313 -
1.1377784993628672 -1.075420453691926 -0.9916418035800518 -0.8728451455302357 -
0.6852230864344686 -0.28947339098956154 0.09227780404945321 0.6735776090795457
0.9292724083039241 1.0998781194476934 1.2301015395181225 1.3365110755652543
1.427100089019829 1.5063419472916557 1.5769940648439738 1.6408761694946794
1.6992517446159634 1.753033122819814 1.8028998293055853 1.8493707833024786
1.8928503828422107 1.9336590370783322 1.9720540360815975 2.0082441972052933
2.0424003762936933 2.0746631550632686 2.105148552320244 2.1339523208615847
2.161153210561715 2.186815460029862 2.210990700379594 2.23371940075231
2.25503194746601 2.274949421448929 2.2934841183402197 2.3106398398553223
2.326411971997382 2.3407873541428117 2.35374393184725 2.3652501743692853
2.3752642242590962 2.383732729507653 2.3905892868082064 2.3957523947270696
2.3991227739362797 2.400579851803371 2.399977120480483 2.3971359446669247
2.391837189598307 2.3838097131810785 2.3727142316477634 2.358120163468329
2.3394714653860023 2.316034549506723 2.286815703548985 2.2504237473222046
2.20482762847652 2.1468948489307658 2.0714193620645567 1.9687633688165251
1.8177542231980721 1.5540112942224464 0.511121612525214 -1.5054687705775212 -
1.8453251320039659 -2.044650738687436 -2.1887045063980852 -2.3028108238402356 -
2.3979886901821805 -2.480035101939596 -2.5523773574966366 -2.617209706044863 -
2.6760207000356493 -2.729865482690954 -2.779518062787307 -2.8255619254394935 -
2.8684466954122314 -2.9085250401175364 -2.946077527623321 -2.9813298512397988 -
3.0144650519308183 -3.0456323653742645 -3.0749537309763255 -3.1025286419745592 -
3.128437791357286 -3.1527458237321824 -3.1755034076147632 -3.1967487776335943 -
3.21650885075807 -3.2347999878781395 -3.2516284473796095 -3.2669905576929312 -
3.280872618925582 -3.2932505278478565 -3.3040891040809535 -3.3133410766350537 -
3.320945666899012 -3.3268266740038093 -3.3308899271222936 -3.3330199105621543 -
3.3330752817164937 -3.330882873184753 -3.3262295720228683 -3.3188511549931894 -
3.3084166462396603 -3.2945058993404324 -3.2765765908304187 -3.2539140383788374 -
3.225551908206844 -3.1901409069396416 -3.145718298246682 -3.0892721586104885 -
3.0158325507663077 -2.916297619302923 -2.771032381142282 -2.522604876787438 -
1.7223895763974086 2.4002068005299133 2.763073072324696 2.9695124296031916
3.117130617290528 3.23343981916626 3.330154266144592 3.4133612179887582
3.486630811899501 3.5522358857257013 3.611711904077256 3.6661438285309194
3.7163256009940127 3.762854602309466 3.806190499869681 3.846693455093463
3.884649794587867 3.920289760801405 3.9538000862502085 3.9853330831989755
4.015013325049635 4.042942622698164 4.069203766021054 4.093863350825047
4.116973912719235 4.138575522405714 4.158696950276006 4.177356474717788
4.194562383510515 4.210313197974538 4.224597632890046 4.237394289867599
4.248671066284871 4.258384244522179 4.266477205163746 4.27287868061244
4.277500428663353 4.280234153680353 4.280947427805571 4.279478252684996
```

```
4.275627731206641 4.269150050501206 4.2597385442930635 4.247005880424774
4.230455170505695 4.209436547019928 4.183079490577633 4.150182638305626
4.109024400406931 4.057014503662567 3.989993192130225 3.9006395297546237
3.7741498836062815 3.57245759587251 3.1219488254305974 -3.1127242389254652 -
3.6158524348943777 -3.855947584351095 -4.0188100832448574 -4.1438599816694355 -
4.24626999930944 -4.333503029060307 -4.409785075988321 -4.477741489202837 -
4.539114168334515 -4.59511705162425 -4.646628862290462 -4.694305048866784 -
4.7386464294778365 -4.7800431842608315 -4.818804094798664 -4.85517657875481 -
4.889360773530765 -4.921519651939347 -4.951786418800875 -4.980269997575635 -
5.007059144044123 -5.032225550746797 -5.055826192521369 -5.0779050873590315 -
5.098494594316266 -5.117616332931691 -5.135281781119984 -5.151492587273634 -
5.16624061480036 -5.179507721673413 -5.191265262180273 -5.20147328136903 -
5.210079352954857 -5.217016986413303 -5.222203495507853 -5.225537173908768 -
5.2268935566366554 -5.226120447259371 -5.223031241061928 -5.217395841511566 -
5.208928094695037 -5.197268051391601 -5.181956315375563 -5.162395866985548 -
5.137793267364802 -5.107064291171265 -5.068674614508965 -5.02035329218719 -
4.958533611019431 -4.877135269164561 -4.764460787620988 -4.593051406444126 -
4.265614626233867 3.6169157193433024 4.4560075403313535 4.739631896777766
4.91988827084465 5.054356981374448 5.162706835575394 5.254051673714721
5.3333667957271675 5.4036669194659295 5.466916411767129 5.524465609982432
5.577281573547812 5.626079529029421 5.671402238172976 5.713670224238118
5.7532148064743085 5.790300536886883 5.825140856158002 5.857909269032029
5.888747474112862 5.91777137009722 5.945075546121536 5.970736665414388
5.9948160226413405 6.017361469537697 6.038408844764744 6.05798300265401
6.076098505461135 6.092760020901124 6.107962448310882 6.1216907807483905
6.133919695036579 6.144612845654477 6.15372181980352 6.161184687874481
6.1669240530979454 6.170844462271132 6.17282897983329 6.172734640261785
6.170386362540787 6.1655687080097685 6.158014541878364 6.147389134148923
6.133267349323288 6.115100018094648 6.092162728368435 6.063474751959103
6.027664464685403 5.9827324044466845 5.925601554796613 5.851174377004021
5.750059859347495 5.601797618092214 5.345274290445617 4.433031311155056 -
5.26390837495436 -5.613829757509022 -5.81639719856154 -5.962134673227781 -
6.077321225377984 -6.173282002019108 -6.2559425481572415 -6.328793715013532 -
6.394064879112012 -6.453265670754457 -6.5074647153868055 -6.557445105859192 -
6.60379672339005 -6.6469738736514525 -6.6873327720471964 -6.725156765759175 -
6.7606737948178885 -6.79406877362338 -6.8254925487268885 -6.855068487577535 -
6.882897388193135 -6.909061171458458 -6.9336256708538615 -6.95664273735528 -
6.978151811427279 -6.998181068150769 -7.016748208507632 -7.03386094510353 -
7.049517211054229 -7.063705104170722 -7.076402563213618 -7.0875767572977635 -
7.097183151914869 -7.1051641936003165 -7.1114475274697355 -7.115943624059272 -
7.118542638636548 -7.119110248808711 -7.117482100932939 -7.113456319367062 -
7.106783255203225 -7.097151202213415 -7.084166057363499 -7.067321602127692 -
7.045954727352438 -7.019175451769233 -6.985752569859067 -6.943916255332898 -
6.890992816821946 -6.822664621921129 -6.731271381900273 -6.601122564438705 -
6.39086534850894 -5.896085158575589 6.012399099755716 6.475892159397009
6.707790525441155 6.867094838142896 6.990146277027778 7.091269753817739
7.177600441011294 7.253210910634431 7.32064508583091 7.381597628057656
7.437253175064327 7.488471300261424 7.535894397464244 7.580014069044707
7.6212137941615685 7.659797355563087 7.696008356335805 7.7300439625920285
7.7620647881431815 7.792202130538963 7.820563343447526 7.847235867197723
7.872290271369844 7.89578255325093 7.917755861933954 7.938241766687459
7.957261151796011 7.974824793149091 7.990933650960801 8.00557889569524
8.018741668697297 8.030392563604096 8.040490797779889 8.048983022997179
8.055801699084366 8.060862920048987 8.064063534461148 8.065277333191286
8.064349975998075 8.061092174244092 8.055270406654246 8.046594059657492
8.034697246382047 8.019112466105664 7.999231317317665 7.974243833824348
7.94304081061432 7.904048262446942 7.854928214533435 7.791990970828305
7.708903427147052 7.59335262294385 7.415887076118202 7.067093610496298 -
6.599363509075123 -7.320302141646116 -7.592690740371816 -7.768687799424599 -
7.900903622214605 -8.007851881029337 -8.098235129617297 -8.176845480065772 -
8.24660366961081 -8.309420868915232 -8.366615032405514 -8.419132366177903 -
```

```
8.467674050488345 -8.512773029668319 -8.554842748460233 -8.594209283849187 -
8.631133212871019 -8.665824897486353 -8.698455410684268 -8.729164494308868 -
8.758066443754062 -8.785254510438875 -8.810804220498932 -8.834775882995725 -
8.857216477450079 -8.87816105330936 -8.897633733630276 -8.915648385824635 -
8.932208999849216 -8.947309796019226 -8.96093506868906 -8.973058756736068 -
8.983643715562186 -8.992640646511848 -8.999986616094239 -9.005603066312181 -
9.009393174513573 -9.011238360014202 -9.010993645009789 -9.008481442178741 -
9.00348313253634 -8.995727465279 -8.984874267966598 -8.970491034890868 -
8.95201834073767 -8.928717042355897 -8.899584439263288 -8.863214589854639 -
8.817551244361027 -8.759416087431461 -8.683511458900787 -8.579987902602923 -
8.42705256359536 -8.157403644249738 -6.952416917858284 8.136902531848081
8.46918463361365 8.666251242594388 8.80929058245217 8.922865997795554
9.01774989704761 9.099635433133109 9.171898687126403 9.23670581488733
9.295528742155899 9.349412200997575 9.399123397887388 9.44524122955319
9.488212153839852 9.528386620189833 9.566043634522345 9.601407796018949
9.634661396105658 9.665953182863472 9.695404814110157 9.7231156696702
9.749166472168342 9.773622023068254 9.796533266278942 9.817938827520374
9.837866132845285 9.85633217739615 9.873343991182518 9.88889882937832
9.902984098186478 9.915577011980087 9.92664396166037 9.936139556359162
9.944005278822315 9.950167666449879 9.954535891307712 9.956998557811108
9.95741945733285 9.955631900311069 9.951431064498555 9.944563511454511
9.93471255867638 9.921477416144699 9.904342642240099 9.882632017411558
9.855436247871207 9.821494424417574 9.778988515279067 9.725161013660756
9.655534468318013 9.562101262537302 9.428246682165419 9.20906809093362
8.661710117548605 -8.90296297167692 -9.334340282881397 -9.558893642867625 -
9.71492496987928 -9.836114329537056 -9.936029435279114 -10.021507771806073 -
10.09648105907978 -10.163417923119594 -10.22396910660531 -10.279291891676527 -
10.330227912784757 -10.37740730014335 -10.421312977927089 -10.462322101624293 -
10.500733727282341 -10.536787844261202 -10.57067879856131 -10.602564960704283 -
10.6325758108055 -10.660817203326133 -10.687375319144307 -10.71231964964065 -
10.735705250500775 -10.75757443084039 -10.777957993348801 -10.796876105523463 -
10.814338855660843 -10.830346526692715 -10.844889603824873 -10.857948516420109 -
10.869493099096557 -10.879481740020381 -10.887860164053372 -10.89455977242712 -
10.899495425644767 -10.902562507411343 -10.903633036871698 -10.902550491937873 -
10.899122847583463 -10.893113084847636 -10.884226027635055 -10.872089703436965 -
10.856228288608056 -10.83602166693468 -10.810642817303176 -10.77895667706005 -
10.739348046430251 -10.689408948735494 -10.625320390442267 -10.540480937864174 -
10.421914529699688 -10.23796294384327 -9.864706986669766 9.54665776038075
10.183350625916548 10.44538705343973 10.617322461775345 10.74735830310822
10.85293913585815 10.942379260302747 11.020295809777856 11.0895190385999
11.151908607140518 11.208750976639003 11.26097200328797 11.309259122050719
11.354135648067563 11.396008069681766 11.435197303319006 11.471960007137787
11.506503503720557 11.538996462383794 11.569576688530043 11.598356889086395
11.625428988628366 11.650867384139127 11.674731404786552 11.697067161839314
11.717908918071576 11.737280066597565 11.755193780235409 11.771653370406357
11.786652376589773 11.80017439151259 11.812192611930538 11.82266908851502
11.831553629291559 11.838782287144545 11.844275330154467 11.847934549620993
11.849639697864363 11.849243755657515 11.846566589997954 11.841386347464184
11.833427585370325 11.82234457993756 11.807697294591343 11.788915803180172
11.765245843680663 11.735662099568582 11.698723179688582 11.6523139044702 8
11.593150198982643 11.51572368882442 11.409695967767078 11.25182112765057
10.967691124712834 -8.744332480386529 -11.007484884924306 -11.323971489003732 -
11.515855635700573 -11.656305813974297 -11.768320019752982 -11.862154016957842 -
11.94328075717494 -12.014966599566117 -12.079316877484784 -12.137767070387737 -
12.191338728925166 -12.240783600678663 -12.28666986714607 6 -12.32943631475936 -
12.369427746443161 -12.406918909991646 -12.44213112297047 -12.475244096867693 -
12.506404513070247 -12.535732343526337 -12.563325567795744 -12.589263723831511 -
12.613610591364711 -12.63641621494228 -12.657718411172475 -12.677543860990767 -
12.695908856120747 -12.712819745043685 -12.728273104764416 -12.742255648338576 -
12.75474386280751 -12.765703356330503 -12.775087875283447 -12.782837929939236 -
12.788878938407178 -12.79311875895101 -12.795444424802065 -12.795717813960348 -
12.793769864330189 -12.78939275694364 -12.78232919399414 -12.772257417279128 -
```

```
12.758769804563658 -12.74134147244876 -12.7192827495135 -12.691664472340491 -
12.657195068803766 -12.614006534311374 -12.55925399202509 -12.488292124170464 -
12.392744398425632 -12.25499628576026 -12.026157462892401 -11.412099191256214
11.788157156998468 12.191995108080134 12.409691521251617 12.562594005318456
12.681982547325042 12.7807208725923 12.865365242525593 12.939712971161416
13.006160330775224 13.066315644368276 13.121309667125708 13.171966582755216
13.218904560581239 13.262598042284989 13.303417990676522 13.341658825373907
13.377556985768237 13.411304043719804 13.44305616018006 13.472941022903166
13.501063005974189 13.52750704504388 13.552341564037537 13.57562068509157
13.597385883254569 13.61766719878553 13.636484085037736 13.653845944008898
13.669752381369676 13.684193195817372 13.697148102138955 13.708586171843132
13.718464958060705 13.72672925078475 13.733309382030923 13.73811896474484
13.741051899171561 13.74197840799051 13.740739754027125 13.737141130598916
13.7309419583602 13.721842410051398 13.709464299124575 13.693323287563107
13.67278724902022 13.647011632339288 13.614834713565473 13.574598630757535
13.523822578778113 13.458552122598626 13.371901975087054 13.250183662701298
13.059298275369962 12.657824449565124 -12.47246123165893 -13.045186701478084 -
13.29769068448642 -13.46576439745476 -13.593698247966346 -13.697949939377578 -
13.786468518192754 -13.863704607961443 -13.932401689708557 -13.994369746770738 -
14.050864737620008 -14.102792741657982 -14.150827798792912 -14.1954838187235 -
14.237160481683473 -14.276173644886738 -14.312776118698432 -14.34717223728079 -
14.37952830347896 -14.409980213815881 -14.43863910740188 -14.465595597551507 -
14.490922963878587 -14.51467956452325 -14.53691064906763 -14.557649698307406 -
14.576919378529947 -14.594732169682642 -14.611090705081967 -14.625987842534823 -
14.639406470988245 -14.651319041481624 -14.66168679470126 -14.670458638107672 -
14.677569601239913 -14.682938765369189 -14.686466518703444 -14.688030923939454 -
14.68748289012812 -14.684639697510027 -14.679276201651108 -14.671112688537432 -
14.659797768885817 -14.644883706643736 -14.625789817566732 -14.601746312658763 -
14.571704580577308 -14.534186583507342 -14.487015937768103 -14.426798044518325 -
14.347802397145125 -14.239168260173626 -14.07606097181903 -13.77585477857831
12.703127874192313 13.875991458970498 14.17823562442632 14.365226040523146
14.503188281249354 14.613688166543938 14.706497738461822 14.786881040713265
14.857999431505828 14.921899678186788 14.979982002803274 15.03324544919722
15.082426711890784 15.12808351310239 15.170647135674564 15.210456849684315
15.247783223924081 15.28284435070324 15.315817402499126 15.346847024198984
15.37605152443382 15.403527499486659 15.429353315484088 15.45359174011317
15.476291925779774 15.4974908852148 15.517214557814329 15.535478534020907
15.552288481610834 15.567640298976295 15.581520004289617 15.593903354130553
15.604755169205415 15.614028326548862 15.621662355070303 15.62758154177321
15.63169241549403 15.633880417571865 15.634005484990645 15.631896145803399
15.627341533193468 15.620080418649527 15.609785866527819 15.596043273461241
15.578318089484373 15.555906839275465 15.52785991340489 15.492854080035691
15.448969515035886 15.393270230184998 15.320934485715213 15.223193541999667
15.081349407057674 14.842013271001768 14.139312195731838 -14.669002218347073 -
15.04891919794094 -15.260199004877023 -15.410107459247179 -15.527753491893737 -
15.625345422715165 -15.70917363525351 -15.782907123667327 -15.848872610453258 -
15.908637436528286 -15.96330662887063 -16.01368739297488 -16.060386231252924 -
16.103869293483445 -16.144501477673145 -16.1825726553142 -16.21831577830595 -
16.251919689484627 -16.283538373365623 -16.31329775000515 -16.34130073164976 -
16.367631022731423 -16.39235599030309 -16.415528830879147 -16.437190191250636 -
16.45736935331044 -16.47608505883517 -16.49334602474655 -16.50915117940309 -
16.52348963366585 -16.53634038506438 -16.547671737803423 -16.557440404013477 -
16.56559023069971 -16.572050469834366 -16.576733472477343 -16.57953163644848 -
16.580313362713483 -16.578917665058718 -16.575146908857988 -16.56875689018401 -
16.559443039714377 -16.546820824945605 -16.530397196372686 -16.509527712544113 -
16.48334980024126 -16.450674238720445 -16.409798991479725 -16.3581674528573 -
16.291683229187683 -16.2031605020797 -16.078144312555747 -15.879828058134336 -
15.445366685551873 15.383936561045346 15.905928864623126 16.149624100547314
16.314021382806114 16.43992702178112 16.542886216325012 16.63050405994797
16.7070726242611 16.775252136802873 16.836804655567285 16.89295658765452
16.944594788504123 16.99238024194564 17.036817669388036 17.078300087202333
```

```
17.117138392164982 17.153581616239013 17.187831154978387 17.22005098116199
17.250375109428692 17.278913131310084 17.30575436413585 17.33097098170046
17.354620379800394 17.37674695275108 17.39738340394354 17.41655167584963
17.434263557174706 17.450521003471483 17.465316189947345 17.478631299520732
17.490438033811973 17.500696818140945 17.50935565200245 17.516348531674804
17.521593338480542 17.524989040157216 17.526411986694978 17.525710984491138
17.52270068505479 17.51715259505488 17.508782647745974 17.49723367115617
17.482050056324617 17.462640095649096 17.438218051094722 17.40771131191832
17.36960394530667 17.321655977753196 17.26035724894776 17.179742910427535
17.06839357568685 16.899733438255478 16.58156333237697 -15.826889422709623 -
16.74268185230196 -17.032011278054455 -17.21437279150364 -17.349942383872115 -
17.458972702124534 -17.55078237536426))))

; And in table form:
0       0       1.0
0.743590923721095       0.4536114001301142      0.9986133168019985
1.01766219063027        0.6841183679442987      0.9964089521980072
1.193813999706169       0.8433616730853838      0.9933174076997987
1.3257484974510294      0.9667946430330369      0.9892618980076057
1.432236226973832       1.0684938048891988      0.9841569439712616
1.5220708975099817      1.1554985188456184      0.977906628893901
1.6000845586478505      1.2318372599722287      0.970402416655308
1.669217769523681       1.3000313530164012      0.9615203921191385
1.7313923030092346      1.3617631009826594      0.9511177291421135
1.7879320004848647      1.4182101648358685      0.9390281100079143
1.839786333524481       1.4702281918975262      0.92505569718736
1.8876581880450238      1.5184574893933405      0.9089670686629543
1.93208122913197        1.5633887437192635      0.890480227882003
1.9734689643558088      1.6054053018971703      0.8692493105352523
2.0121470679589546      1.644811367383014       0.8448427878732332
2.048375363718498       1.6818513764417313      0.8167115299361626
2.0823631781162586      1.7167236557855179      0.7841404729318451
2.114280304915592       1.7495902576524418      0.7461726101974508
2.144264981284994       1.780584170212422       0.701483783688533
2.1724297761532547      1.809814681552654       0.6481642757887104
2.198865984806183       1.8373714151925908      0.58330911517505
2.223646929710347       1.8633273889503261      0.502172426517505
2.2468304413467015      1.8877413401450258      0.3961744292011242
2.2684607085155686      1.9106594870136806      0.24716942268148373
2.2885696296458096      1.9321168458631288      0.004133433989857941
2.307177755569855       1.9521381878293598      0
2.3242948839456736      1.9707386930479207      0
2.339920341996758       1.9879243401710203      -0.16262488492662167
2.3540429746818132      2.0036920531693987      -0.3681418185895976
2.3666408376419525      2.018029613520357       -0.5105754686162159
2.37768057644211        2.030915332829012       -0.6201071200537452
2.3871164538517626      2.04231746739664        -0.7092592242186215
2.394888963007749       2.052193340905047       -0.7844394256458623
2.4009229333814903      2.0604881226204834      -0.8493635678076535
2.405124994397533       2.067133184185083       -0.9063743980154273
2.4073802020551125      2.0720439250776588      -0.9570395431329056
2.4075475471176615      2.075116910560115       -1.002455098855808
2.4054539332847433      2.076226099174876       -1.0434131585254927
2.400886013267928       2.075217838067382       -1.0805003681339105
2.3935789531228244      2.071904154514271       -1.1141588949763346
2.383200676787571       2.0660536386686754      -1.1447256774557555
2.3693292676296163      2.057378838948846       -1.172458491563213
2.3514196688886018      2.045518474712703       -1.1975536629671788
2.3287530113143213      2.0300117164525644      -1.2201582700075622
2.300356464346572       2.0102599077064136      -1.2403785671370344
2.2648703498326 1.9854676060822993      -1.2582857010173616
2.2203145402592104      1.9545479348022858      -1.2739193852227244
```

2.1636459704522326	1.915962749222925	-1.2872899335771806
2.08983335306259	1.8674350622724467	-1.2983788637297053
1.9896356396127306	1.805387543952958	-1.3071381328087865
1.8430213381021932	1.723718657622348	-1.313487928210263
1.59080319936432	1.6106801789634613	-1.317312784118602
0.7407356142438424	1.4386519188434856	-1.3184555997976621
-1.4886794929522706	1.1093939375177784	-1.31670885834891
-1.8451921290129314	-0.4777363522339745	-1.3118019179970786
-2.0498099956953095	-1.3047963321595912	-1.3033825539117725
-2.196589279845079	-1.5873231195827413	-1.290989744691329
-2.3124312110945304	-1.7671040555398405	-1.2740125821481132
-2.408858207389486	-1.9012744883481263	-1.2516262075395346
-2.491878097677662	-2.0094016620901507	-1.2226877609685727
-2.565022220752951	-2.1005634441003784	-1.1855584380925313
-2.630542243042043	-2.1797190826039157	-1.1377784993628672
-2.6899611190990593	-2.2498749916738108	-1.075420453691926
-2.744355962903856	-2.312990764171465	-0.9916418035800518
-2.794515560618568	-2.3704137603336317	-0.8728451455302357
-2.8410337804264905	-2.4231090555872195	-0.6852230864344686
-2.884367826716676	-2.471790476999466	-0.28947339098956154
-2.9248760945954877	-2.516999747980736	0.09227780404945321
-2.9628436222608334	-2.5591565850182403	0.6735776090795457
-2.998499701286641	-2.5985916529978708	0.9292724083039241
-3.032030358167704	-2.6355689513515546	1.0998781194476934
-3.0635873813442824	-2.6703014361636814	1.2301015395181225
-3.0932949594381407	-2.7029621719506993	1.3365110755652543
-3.121254627499104	-2.7336924441859733	1.427100089019829
-3.1475489873533204	-2.7626077522181682	1.5063419472916557
-3.1722445197713807	-2.7898022887383576	1.5769940648439738
-3.1953937082130404	-2.815352313960965	1.6408761694946794
-3.2170366275348976	-2.83931870416725	1.6992517446159634
-3.237202104837507	-2.861748868631272	1.753033122819814
-3.2559085264283953	-2.882678170368819	1.8028998293055853
-3.2731643400657178	-2.9021309449001036	1.8493707833024786
-3.2889682821115684	-2.9201211811470653	1.8928503828422107
-3.3033093427422617	-2.9366529056624775	1.9336590370783322
-3.316166467204251	-2.95172029283653	1.9720540360815975
-3.3275079757445165	-2.9653075074994883	2.0082441972052933
-3.337290667727791	-2.977388270760347	2.0424003762936933
-3.3454585547483777	-2.987925123416227	2.0746631550632686
-3.351941140836871	-2.996868342115638	2.105148552320244
-3.356651131707685	-3.0041544395155224	2.1339523208615847
-3.3594814042148395	-3.0097041479883506	2.161153210561715
-3.3603009936954913	-3.0134197426830864	2.186815460029862
-3.3589497476540284	-3.0151814972770383	2.210990700379594
-3.3552311276514617	-3.0148429740134097	2.23371940075231
-3.3489023803360753	-3.0122247113029026	2.25503194746601
-3.3396608779537984	-3.007105658086562	2.274949421448929
-3.327124728769123	-2.9992113634862774	2.2934841183402197
-3.3108045503912025	-2.988197371421335	2.3106398398553223
-3.2900611278898406	-2.9736253213130657	2.326411971997382
-3.264039581231984	-2.954927582048723	2.3407873541428117
-3.2315624773538767	-2.931353155876609	2.35374393184725
-3.1909467791360266	-2.901881571577957	2.3652501743692853
-3.1396685736005625	-2.8650789994673063	2.3752642242590962
-3.073692863575478	-2.818842806625423	2.383732729507653
-2.9859642405934785	-2.7599114399744127	2.3905892868082064
-2.862361955381825	-2.6828214933095187	2.3957523947270696
-2.667270627376725	-2.577339374533186	2.3991227739362797
-2.247547474940148	-2.4205333477082975	2.400579851803371
2.1338610541983196	-2.139564143781716	2.399977120480483
2.6761884286402653	-0.09723687321299573	2.3971359446669247

2.923550954946462	2.1668082411051164	2.391837189598307
3.089485407015026	2.4864691289634844	2.3838097131810785
3.2162376306472362	2.679423028359266	2.3727142316477634
3.319733831284352	2.8204172846554347	2.358120163468329
3.407723451281857	2.9327652950866443	2.3394714653860023
3.4845653028059154	3.0268279260119213	2.316034549506723
3.5529547239884613	3.108123179328447	2.286815703548985
3.614674007709349	3.17993995214873	2.2504237473222046
3.670962045487803	3.2443961124933285	2.20482762847652
3.722713814925774	3.3029346738178083	2.1468948489307658
3.7705958138093174	3.356581993779646	2.0714193620645567
3.8151166403367633	3.4060930368114017	1.9687633688165251
3.8566721181219306	3.452038223329493	1.8177542231980721
3.895575228753132	3.4948579507137305	1.5540112942224464
3.932076587090917	3.5348982149841546	0.511121612525214
3.9663788144223346	3.572434670660278	-1.5054687705775212
3.998646849860823	3.607689340915878	-1.8453251320039659
4.029015482705157	3.6408424987608727	-2.044650738687436
4.05759493547027	3.67204128225835	-2.1887045063980852
4.0844750475380325	3.701406042903436	-2.3028108238402356
4.109728431489631	3.729035082754304	-2.3979886901821805
4.13341285798713	3.755008220144366	-2.480035101939596
4.155573047180666	3.7793894844823317	-2.5523773574966366
4.176241990996307	3.8022291482972705	-2.617209706044863
4.19544189263588	3.8235652418577963	-2.6760207000356493
4.213184781665453	3.8434246517417194	-2.729865482690954
4.229472841525539	3.861823872962457	-2.779518062787307
4.24429846861552	3.8787694603194423	-2.8255619254394935
4.25764406637356	3.8942582055756794	-2.8684466954122314
4.269481562385851	3.9082770507372855	-2.9085250401175364
4.279771619963276	3.9208027324165693	-2.946077527623321
4.288462496078413	3.9318011364629557	-2.9813298512397988
4.295488472844871	3.941226324127211	-3.0144650519308183
4.300767756756516	3.949019169030421	-3.0456323653742645
4.30419969411508	3.9551055155148713	-3.0749537309763255
4.305661085401154	3.95939372976725	-3.1025286419745592
4.305001284515281	3.961771459670818	-3.128437791357286
4.302035622294472	3.962101338607162	-3.1527458237321824
4.296536466081843	3.9602152476949617	-3.1755034076147632
4.288220863451978	3.95590656566333	-3.1967487776335943
4.2767331189722535	3.9489195433895063	-3.21650885075807
4.261619630709511	3.938934465693639	-3.2347999878781395
4.242291498949984	3.9255464669023743	-3.2516284473796095
4.217967047554955	3.9082344802431277	-3.2669905576929312
4.18757978033449	3.8863142805062467	-3.280872618925582
4.149623428211775	3.8588647602550217	-3.2932505278478565
4.101874258083447	3.8246067967776773	-3.3040891040809535
4.040851688371752	3.7816927056458427	-3.3133410766350537
3.96065104805589	3.7273132010813104	-3.320945666899012
3.8499971446518213	3.656891509082458	-3.3268266740038093
3.6827677305218107	3.562202246200322	-3.3308899271222936
3.3693165795135567	3.426042792127391	-3.3330199105621543
-2.5499293769700393	3.201184478376109	-3.3330752817164937
-3.514996745139801	2.6160522894174685	-3.330882873184753
-3.8074633416560624	-2.9362076065069753	-3.3262295720228683
-3.9909673360027442	-3.350644131998086	-3.3188511549931894
-4.12713164609098	-3.5710429298556643	-3.308416646239603
-4.236528909235187	-3.72518639232443	-3.2945058993404324
-4.328589147584981	-3.845291081368423	-3.2765765908304187
-4.408427029161088	-3.9444979161634457	-3.2539140383788374
-4.479128275509111	-4.029474414503489	-3.225551908206844

```
-4.5426972431513395     -4.104071221576832      -3.1901409069396416
-4.600508520999946      -4.170713377559907      -3.145718298246682
-4.653544739299014      -4.231026096407853      -3.0892721586104885
-4.702531587720051      -4.286150707564868      -3.0158325507663077
-4.748019126651269      -4.336918356726548      -2.916297619302923
-4.790433133142877      -4.38395198792008       -2.771032381142282
-4.830108811903571      -4.427729430419979      -2.522604876787438
-4.867313657564834      -4.468624120806953      -1.7223895763974086
-4.90226338757086       -4.506932330983709      2.4002068005299133
-4.935133303336088      -4.5428919184809935     2.763073072324696
-4.966066548014144      -4.576695563092844      2.9695124296031916
-4.995180203055304      -4.6085003078234426     3.117130617290528
-5.022569843778609      -4.638434555434799      3.23343981916626
-5.0483129711887065     -4.6666032699920414     3.330154266144592
-5.0724716057113435     -4.69309188277704       3.4133612179887582
-5.095094241011623      -4.717969241897745      3.486630811899501
-5.116217296305433      -4.741289839719442      3.5522358857257013
-5.135866163598655      -4.763095481277228      3.611711904077256
-5.154055915791404      -4.783416507643533      3.6661438285309194
-5.1707917184478305     -4.802272653068773      3.7163256009940127
-5.186068969437787      -4.8196735886051565     3.762854602309466
-5.199873174543177      -4.8356191845311285     3.806190499869681
-5.212179551822583      -4.850099506860382      3.846693455093463
-5.222952341503789      -4.863094547733597      3.884649794587867
-5.232143779772866      -4.874573673984995      3.920289760801405
-5.2396926720197365     -4.884494761078317      3.9538000862502085
-5.2455224711216655     -4.892802959096534      3.9853330831989755
-5.249538725174227      -4.899429011170354      4.015013325049635
-5.251625700572926      -4.904287009276823      4.042942622698164
-5.251641900797947      -4.9072714226924505     4.069203766021054
-5.2494140728342344     -4.908253162698937      4.093863350825047
-5.24472909530361       -4.907074340798379      4.116973912719235
-5.237322829100649      -4.903541215805671      4.138575522405714
-5.226864500246938      -4.897414572093 4.158696950276006
-5.212934322649606      -4.888396364117078      4.177356474717788
-5.194990558175308      -4.8761107863562705     4.194562383510515
-5.1723194466046145     -4.8600767645504        4.210313197974538
-5.143956108444713      -4.839666778648223      4.224597632890046
-5.108553597012589      -4.814043005672188      4.237394289867599
-5.0641531265192        -4.7820539638798625     4.248671066284871
-5.007749864660341      -4.742058199778889      4.258384244522179
-4.934387849553118      -4.691602962481953      4.266477205163746
-4.834996889728672      -4.626786100604099      4.27287868061244
-4.690031872179976      -4.540832235171192      4.277500428663353
-4.442455122453738      -4.420330070248463      4.280234153680353
-3.652707038232502      -4.232138939101227      4.280947427805571
4.315346289539543       -3.841901862304941      4.279478252684996
4.679707226873661       3.6074455914907206      4.275627731206641
4.886565597343564       4.202660987099854       4.269150050501206
5.034372723739953       4.4587020925317224      4.2597385442930635
5.150784322470198       4.628222743197348       4.247005880424774
5.247559445382907       4.756947639582552       4.230455170505695
5.33080385256656        4.861700814903807       4.209436547019928
5.404096695064507       4.950567696470854       4.183079490577633
5.469715717723319       5.028061300606497       4.150182638305626
5.529199277175606       5.09695770114522        4.109024400406931
5.583634143549693       5.159085573134944       4.057014503662567
5.633815442269407       5.2157123674871695      3.989993192130225
5.680341356367166       5.267751622758129       3.9006395297546237
5.723672111848381       5.315882435433737       3.7741498836062815
5.764168267005693       5.360622268344514       3.57245759587251
5.802116434235923       5.402373375300119       3.1219488254305974
```

5.8377470631294495	5.441453527076711	-3.1127242389254652
5.871247036052415	5.478116990174835	-3.6158524348943777
5.902768772202625	5.512569230248818	-3.855947584351095
5.932436918830747	5.544977446598274	-4.0188100832448574
5.960353334381957	5.575478259227432	-4.1438599816694355
5.986600834657938	5.604183401781888	-4.24626999930944
6.0112460229322515	5.631183985111261	-4.333503029060307
6.034341425863786	5.65655371306653	-4.409785075988321
6.055927089950372	5.680351312724182	-4.477741489202837
6.076031746559887	5.702622361316991	-4.539114168334515
6.09467362002383	5.723400637240819	-4.59511705162425
6.1118609281970935	5.742709083654318	-4.646628862290462
6.127592105124749	5.760560444704812	-4.694305048866784
6.141855758759333	5.776957612545119	-4.7386464294778365
6.154630361282352	5.791893705449732	-4.7800431842608315
6.165883653948304	5.8053518815517355	-4.818804094798664
6.175571730897117	5.8173048773889375	-4.85517657875481
6.183637745190159	5.8277142440248015	-4.889360773530765
6.190010152926611	5.836529234281845	-4.921519651939347
6.194600374130886	5.843685270427436	-4.951786418800875
6.197299696803534	5.849101889484243	-4.980269997575635
6.197975174701973	5.852680018775998	-5.007059144044123
6.196464156504752	5.854298370549883	-5.032225550746797
6.192566911462276	5.853808650688131	-5.055826192521369
6.186036545784904	5.851029134824721	-5.0779050873590315
6.176564966377091	5.845735945513135	-5.098494594316266
6.163762918277493	5.8376510139301825	-5.117616332931691
6.147130858468315	5.826425134111348	-5.135281781119984
6.126015148241738	5.811613538856271	-5.151492587273634
6.099539724260189	5.792639694850703	-5.16624061480036
6.066494726126497	5.768739808627124	-5.179507721673413
6.025144842942116	5.73887427204234	-5.191265262180273
5.972876112640977	5.701579226824229	-5.20147328136903
5.9054840598517115	5.65470200490785	-5.210079352954857
5.815551997349065	5.59489091587849	-5.217016986413303
5.688027646057652	5.516501923596961	-5.222203495507853
5.483934810830967	5.4088784594140185	-5.225537173908768
5.021543338348691	5.247811368091197	-5.2268935566366554
-5.048410816006276	4.953967632871074	-5.226120447259371
-5.53933848019666	-3.679261967661479	-5.223031241061928
-5.77700737591667	-5.030951219297728	-5.217395841511566
-5.938827712100194	-5.338481867621914	-5.208928094695037
-6.063295713764958	-5.52731394192849	-5.197268051391601
-6.165332754762839	-5.666217427328632	-5.181956315375563
-6.252305169584405	-5.7772918508177815	-5.162395866985548
-6.32839393140638	-5.870490793126016	-5.137793267364802
-6.396200555974099	-5.951157018532732	-5.107064291171265
-6.457453144623656	-6.022491557274777	-5.068674614508965
-6.513357091003548	-6.086563346501418	-5.02035329218719
-6.564785541719162	-6.144785973813906	-4.958533611019431
-6.612390148464143	-6.198167024583621	-4.877135269164561
-6.65666905230293	-6.247448911249072	-4.764460787620988
-6.698010489213224	-6.29319333195704	-4.593051406444126
-6.736721792359023	-6.335834423720285	-4.265614626233867
-6.773049275732815	-6.375713562315494	3.6169157193433024
-6.807192218493238	-6.4131029077523865	4.4560075403313535
-6.839312913413947	-6.44822178093904	4.739631896777766
-6.869544016802517	-6.481248321928554	4.91988827084465
-6.897994001956981	-6.512327951981063	5.054356981374448
-6.924751248741484	-6.541579614067893	5.162706835575394
-6.949887130130266	-6.569100432209001	5.254051673714721

-6.973458344160320	-6.594969219766952	5.3333667957271675
-6.995508664223877	-6.619249130836001	5.4036669194659295
-7.016070228536522	-6.64198965859229	5.466916411767129
-7.03516445258172	-6.663228122960248	5.524465609982432
-7.052802621018474	-6.682990746842108	5.577281573547812
-7.068986194400691	-6.701293388928638	5.626079529029421
-7.083706848613963	-6.718141977508066	5.671402238172976
-7.096946249311341	-6.733532670819557	5.713670224238118
-7.108675548234611	-6.747451753250685	5.7532148064743085
-7.118854571587634	-6.759875261367935	5.790300536886883
-7.127430650821707	-6.770768317854732	5.825140856158002
-7.1343370210399515	-6.780084133277659	5.857909269032029
-7.139490678562863	-6.787762613217797	5.888747474112862
-7.142789542318351	-6.79372847900053	5.91777137009722
-7.144108696345915	-6.797888770132407	5.945075546121536
-7.1432953911879435	-6.800129539671627	5.970736665414388
-7.140162331083304	-6.8003114707569905	5.9948160226413405
-7.134478539079963	-6.798264018174658	6.017361469537697
-7.125956716299082	-6.7937774876545145	6.038408844764744
-7.114235390860164	-6.786592163521896	6.05798300265401
-7.098853090444145	-6.7763831037428535	6.076098505461135
-7.079209882854189	-6.762738394500068	6.092760020901124
-7.054508104754528	-6.745127212222464	6.107962448310882
-7.023657153911895	-6.722851406677654	6.1216907807483905
-6.985112596718306	-6.694969262559251	6.133919695036579
-6.936586434146526	-6.660169784541536	6.144612845654477
-6.874480741190235	-6.616553203412517	6.15372181980352
-6.7926512830527	-6.561218853953914	6.161184687874481
-6.679245778634511	-6.4894135583051495	6.1669240530979454
-6.506309280797947	-6.3925215425756905	6.170844462271132
-6.173613778771936	-6.252269893781711	6.17282897983329
5.576041266914911	-6.017056987151835	6.172734640261785
6.380040436547238	-5.3516199970956775	6.170386362540787
6.660668778633563	5.819481106862743	6.1655687080097685
6.8398040743468345	6.208174388265714	6.158014541878364
6.9736829799272915	6.421860616725928	6.147389134148923
7.081666956801822	6.572899535864832	6.133267349323288
7.172761218637834	6.69120572824905	6.115100018094648
7.2518928944759935	6.789232875452671	6.092162728368435
7.322052061606485	6.8733713683444755	6.063474751959103
7.385189168944012	6.947338612654607	6.027664464685403
7.442646094885387	7.013487625207693	5.9827324044466845
7.495384458415738	7.073401264915993	5.925601554796613
7.544115825195638	7.128194312553131	5.851174377004021
7.589380400321739	7.178680416406634	5.750059859347495
7.631596864896959	7.225470509685962	5.601797618092214
7.671095174643723	7.269033893324328	5.345274290445617
7.708138848137196	7.30973777144702	4.433031311155056
7.742940525936823	7.347873745828797	-5.26390837495436
7.775673080989805	7.383676095750376	-5.613829757509022
7.806477703626378	7.417334703184952	-5.81639719856154
7.83546987614625	7.449004381705802	-5.962134673227781
7.862743840295438	7.478811724991072	-6.077321225377984
7.888375964018224	7.506860202572145	-6.173282002019108
7.912427286023002	7.533233988437098	-6.2559425481572415
7.934945431509074	7.558000852873589	-6.328793715013532
7.955966034123311	7.5812143457126115	-6.394064879112012
7.975513758190673	7.6029154300492205	-6.453265670754457
7.993602985378317	7.62313367753847	-6.5074647153868055
8.010238207203235	7.641888101989694	-6.557445105859192
8.025414146425977	7.659187682375049	-6.60379672339005
8.039115614362789	7.675031606283995	-6.6469738736514525

8.051317095845643	7.689409247988802	-6.6873327720471964
8.061982037424318	7.702299879852269	-6.725156765759175
8.071061795767358	7.713672100247388	-6.7606737948178885
8.078494180010729	7.723482943890233	-6.79406877362338
8.084201491193866	7.731676619662686	-6.8254925487268885
8.08808791977472	7.738182794192718	-6.855068487577535
8.090036102194775	7.742914303211072	-6.882897388193135
8.089902549536154	7.745764121824888	-6.909061171458458
8.08751152908583	7.746601351240527	-6.9336256708538615
8.082646775525273	7.74526587007776	-6.95664273735528
8.075040084655798	7.741561131586395	-6.978151811427279
8.064355313165736	7.735244326760656	-6.998181068150769
8.050165412788765	7.726012712099236	-7.016748208507632
8.031918554417805	7.713484199707929	-7.03386094510353
8.008886508887732	7.6971690978986365	-7.049517211054229
7.9800828605958305	7.676427715249706	-7.063705104170722
7.944127114095245	7.650404435384958	-7.076402563213618
7.899005151914184	7.617920662239299	-7.0875767572977635
7.841613873244281	7.577291449242982	-7.097183151914869
7.766804368602924	7.525989562724506	-7.1051641936003165
7.665068497381794	7.459973735841377	-7.1114475274697355
7.515608190955049	7.372175220915389	-7.115943624059272
7.255770717478437	7.248439130978102	-7.118542638636548
6.286813724629085	7.053029064250865	-7.119110248808711
-7.190417327901858	6.631836711248405	-7.117482100932939
-7.53552721764736	-6.523618195323422	-7.113456319367062
-7.7366231299504475	-7.063762058510443	-7.106783255203225
-7.881643819302885	-7.3107486643228405	-7.097151202213415
-7.996403924805664	-7.476532023936461	-7.084166057363499
-8.092080322060907	-7.603205464285251	-7.067321602127692
-8.17453653258181	-7.706655110360732	-7.045954727352438
-8.247232877412253	-7.794615283072252	-7.019175451769233
-8.312381954962397	-7.871437827878074	-6.985752569859067
-8.371483406661362	-7.939814431902807	-6.943916255332898
-8.42559953666462	-8.00152529379101	-6.890992816821946
-8.47550923951956	-8.057808020920074	-6.822664621921129
-8.521799496218065	-8.10955676230591	-6.731271381900273
-8.56492254198327	-8.157437459413229	-6.601122564438705
-8.60523307204226	-8.201958325061803	-6.39086534850894
-8.643013289703648	-8.243514909308074	-5.896085158575589
-8.678490255203974	-8.28241999593879	6.012399099755716
-8.711848192552093	-8.318924054940656	6.475892159397009
-8.743237396235767	-8.3532296007541	6.707790525441155
-8.772780784157233	-8.385501493697847	6.867094838142896
-8.8005787816084	-8.41587446551409	6.990146277027778
-8.826712994716228	-8.444458697676962	7.091269753817739
-8.851248986024231	-8.471344001745155	7.177600441011294
-8.874238368519134	-8.496602973395257	7.253210910634431
-8.895720369039395	-8.520293375728388	7.32064508583091
-8.915722966409406	-8.542459929648004	7.381597628057656
-8.934263676809385	-8.563135635549276	7.437253175064327
-8.951350034268629	-8.582342712581388	7.488471300261424
-8.966979794685368	-8.600093213822701	7.535894397464244
-8.981140875223602	-8.616389354192066	7.580014069044707
-8.993811025579951	-8.631223570263494	7.6212137941615685
-9.004957211904324	-8.644578315445424	7.659797355563087
-9.014534676493659	-8.656425578627374	7.696008356335805
-9.022485614846351	-8.666726097837778	7.7300439625920285
-9.028737383719704	-8.675428220960688	7.7620647881431815
-9.033200115816514	-8.682466340917253	7.792202130538963
-9.035763563111914	-8.687758799853624	7.820563343447526

-9.03629291294688	-8.69120511123703	7.847235867197723
-9.03462320482752	-8.692682283316095	7.872290271369844
-9.030551797991796	-8.692039930929049	7.89578255325093
-9.023828060053459	-8.689093716692735	7.917755861933954
-9.014138994052846	-8.683616435917852	7.938241766687459
-9.001088763604146	-8.675325697517414	7.957261151796011
-8.984168761164979	-8.663866556759409	7.974824793149091
-8.9627124845152	-8.648786438671522	7.990933650960801
-8.935824959194248	-8.629497886464204	8.00557889569524
-8.902267309353626	-8.605221317027711	8.018741668697297
-8.860257280040186	-8.574893389799874	8.030392563604096
-8.807099613955922	-8.537012826303007	8.040490797779889
-8.738435671529686	-8.48936427986137	8.048983022997179
-8.646516409457696	-8.428482432097022	8.055801699084366
-8.51541671613074	-8.348493619308535	8.060862920048987
-8.30289497506852	-8.238196757606731	8.064063534461148
-7.795458667144319	-8.071701593661823	8.065277333191286
7.943822581079266	-7.760651441127218	8.064349975998075
8.398415887579361	6.904503000097749	8.061092174244092
8.628345143209811	7.898316198277789	8.055270406654246
8.786779836329005	8.192444081084991	8.046594059657492
8.909338671907353	8.376547336033774	8.034697246382047
9.010143309977071	8.51302176618364	8.019112466105664
9.09624948919797	8.622609374129908	7.999231317317665
9.171692337166023	8.714798898722762	7.974243833824348
9.238995811543798	8.794730751732853	7.94304081061432
9.299842950642354	8.865503723007125	7.904048262446942
9.355411163156527	8.929129509486854	7.854928214533435
9.406555287461558	8.98698714211845	7.791990970828305
9.453914483472984	9.040062097955245	7.708903427147052
9.497978066977462	9.089081976445472	7.59335262294385
9.539127852088482	9.134598167999235	7.415887076118202
9.577666377753827	9.177037405454044	7.067093610496298
9.613836296650582	9.216735599086144	-6.599363509075123
9.647834033574341	9.253960776778307	-7.320302141646116
9.679819612926941	9.288929067560339	-7.592690740371816
9.709923854958337	9.321816096699374	-7.768687799424599
9.738253719790675	9.352765266813226	-7.900903622214605
9.76489631729797	9.381893870878686	-8.007851881029337
9.789921934300414	9.409297659658874	-8.098235129617297
9.813386321275987	9.435054282249933	-8.176845480065772
9.835332407267881	9.459225886386443	-8.24660366961081
9.855791560841952	9.481861077319834	-8.309420868915232
9.874784478734338	9.50299637412947	-8.366615032405514
9.892321757042886	9.522657260220516	-8.419132366177903
9.908404179005267	9.540858894181085	-8.467674050488345
9.923022736145738	9.55760652398047	-8.512773029668319
9.936158384013861	9.572895628863261	-8.554842748460233
9.94778151830257	9.586711797163714	-8.594209283849187
9.957851140259642	9.599030332961975	-8.631133212871019
9.966313660209666	9.609815568489182	-8.665824897486353
9.97310126237553	9.619019840811173	-8.698455410684268
9.978129719781009	9.626582068554079	-8.729164494308868
9.981295499987748	9.632425834524245	-8.758066443754062
9.982471933256479	9.636456839008929	-8.785254510438875
9.981504112375472	9.638559530202336	-8.810804220498932
9.978202037971975	9.638592632909742	-8.834775882995725
9.972331280755709	9.636383168676495	-8.857216477450079
9.963600043347293	9.631718363312046	-8.87816105330936
9.95164086082806	9.624334525637677	-8.897633733630276
9.935984075766761	9.613901472198645	-8.915648385824635
9.916018253456514	9.60000021424279	-8.932208999849216

```
9.890929015989661       9.58209011975512        -8.947309796019226
9.859600477287282       9.559459011878326       -8.96093506868906
9.820448018485386       9.531144363562623       -8.973058756736068
9.771115637467481       9.495802885006306       -8.983643715562186
9.707880424327826       9.451481806086957       -8.992640646511848
9.624340594676946       9.395186942504965       -8.999986616094239
9.508014424609849       9.321983143139258       -9.005603066312181
9.32889471232094        9.222847127716648       -9.009393174513573
8.974026327228026       9.078366429793098       -9.011238360014202
-8.547703642940181      8.832082485134988       -9.010993645009789
-9.244072653581846      8.057313745121942       -9.008481442178741
-9.513670961240262      -8.698030901115338      -9.00348313253634
-9.688586483004016      -9.06467543032329       -8.995727465279
-9.82022487593611       -9.272195763700411      -8.984874267966598
-9.926812019801458      -9.420319567932852      -8.970491034890868
-10.016946692043135     -9.536917854111046      -8.95201834073767
-10.095374433249042     -9.63381679029123       -8.928717042355897
-10.16499196727388      -9.717149843419328      -8.899584439263288
-10.227696822857743     -9.79050968702433       -8.863214589854639
-10.284798627037524     -9.85618145289679       -8.817551244361027
-10.337238211755981     -9.915707875911073      -8.759416087431461
|#
```

```
(defvar n 0)
(defvar k 0)
(defvar successor)
(defvar R)
(defvar Q)
(defvar gamma .9)
(defvar alpha .1)
(defvar epsilon .1)
(defvar randomness)
(defvar max-num-tasks 2000)
(defvar policy)
(defvar V)

(defun setup (n-arg k-arg)
  (setq n n-arg)
  (setq k k-arg)
  (setq successor (make-array (list n 2 k)))
  (setq R (make-array (list n (+ k 1) 2)))
  (setq Q (make-array (list n 2)))
  (setq policy (make-array n))
  (setq V (make-array n))
  (setq randomness (make-array max-num-tasks))
  (standardize-random-state)
  (advance-random-state 0)
  (loop for task below max-num-tasks do
        (loop repeat 17 do (random 2))
        (setf (aref randomness task) (make-random-state))))

(defun init (task-num)
  (setq *random-state* (make-random-state (aref randomness task-num)))
  (loop for s below n do
        (loop for a below 2 do
              (setf (aref Q s a) 0.0)
              (setf (aref R s k a) (random-normal))
              (loop for sp in (random-k-of-n k n)
                    for i below k
                    do (setf (aref successor s a i) sp)
                    do (setf (aref R s i a) (random-normal))))))

(defun random-k-of-n (k n)
  (loop for i = (random n)
        unless (member i result) collect i into result
        until (= k (length result))
        finally (return result)))

(defun next-state (s a)
  (with-prob gamma
    (aref successor s a (random k))
    n))

(defun full-backup (s a)
  (+ (* (- 1 gamma) (aref R s k a))
     (* gamma (/ k)
        (loop for i below k
              for sp = (aref successor s a i)
              sum (aref R s i a)
              sum (* gamma (loop for ap below 2
                                 maximize (aref Q sp ap)))))))

(defun runs-sweeps (n-arg k-arg num-runs num-sweeps sweeps-per-measurement)
  (unless (and (= n n-arg) (= k k-arg)) (setup n-arg k-arg))
  (loop with backups-per-measurement = (truncate (* sweeps-per-measurement 2 n))
        with backups-per-sweep = (* n 2)
        with num-backups = (* num-sweeps backups-per-sweep)
        with num-measurements = (truncate num-backups backups-per-measurement)
        with perf = (make-array num-measurements :initial-element 0.0)
        for run below num-runs do
        (init run)
        (format t "~A " run)
        (loop with backups = 0
              repeat num-sweeps do
              (loop for s below n do
                    (loop for a below 2 do
                          (when (= 0 (mod backups backups-per-measurement))
                            (incf (aref perf (/ backups backups-per-measurement))
                                  (measure-performance)))
                          (setf (aref Q s a) (full-backup s a))
                          (incf backups))))
        finally (record n k num-runs num-sweeps sweeps-per-measurement gamma 1 nil
                        (loop for i below num-measurements
                              collect (/ (aref perf i) num-runs)))))

(defun runs-trajectories (n-arg k-arg num-runs num-sweeps sweeps-per-measurement)
  (unless (and (= n n-arg) (= k k-arg)) (setup n-arg k-arg))
  (loop with backups-per-measurement = (truncate (* sweeps-per-measurement 2 n))
        with backups-per-sweep = (* n 2)
        with num-backups = (* num-sweeps backups-per-sweep)
        with num-measurements = (truncate num-backups backups-per-measurement)
        with perf = (make-array num-measurements :initial-element 0.0)
        for run below num-runs do
        (init run)
        (format t "~A " run)
        (loop named run with backups = 0 do
              (loop for state = 0 then next-state
                    for action = (with-prob epsilon
                                   (random 2)
                                   (if (>= (aref Q state 0) (aref Q state 1)) 0 1))
                    for next-state = (next-state state action) do
                    (when (= 0 (mod backups backups-per-measurement))
                      (incf (aref perf (/ backups backups-per-measurement))
                            (measure-performance)))
                    (setf (aref Q state action) (full-backup state action))
                    (incf backups)
                    (when (= backups num-backups) (return-from run))
                    until (= next-state n)))
        finally (record n k num-runs num-sweeps sweeps-per-measurement gamma 1
epsilon
                        (loop for i below num-measurements
                              collect (/ (aref perf i) num-runs)))))

(defun measure-performance ()
  (loop for s below n do
        (setf (aref V s) 0.0)
        (setf (aref policy s)
              (if (>= (aref Q s 0) (aref Q s 1))
                0 1)))
  (loop for delta = (loop for s below n
                          for old-V = (aref V s)
                          do (setf (aref V s) (full-backup s (aref policy s)))
                          sum (abs (- old-V (aref V s))))
        until (< delta .001))
  (aref V 0))

(defun both (n-arg k-arg runs-arg sweeps-arg measure-arg)
    (runs-sweeps n-arg k-arg runs-arg sweeps-arg measure-arg)
    (runs-trajectories n-arg k-arg runs-arg sweeps-arg measure-arg)
    (graph-data :n n-arg :k k-arg :runs runs-arg :sweeps sweeps-arg :sweeps-per-
measurement measure-arg))

(defun big-exp ()
  (both 10 1 200 10 1)
  (both 10 3 200 10 1)
  (both 100 1 200 10 .5)
  (both 100 3 200 10 .5)
  (both 100 10 200 10 .5)
  (both 1000 1 200 10 .2)
  (both 1000 3 200 10 .2)
  (both 1000 10 200 10 .2)
  (both 1000 20 200 10 .2)
  (both 10000 1 100 10 .1)
  (both 10000 3 200 10 .1)
  (both 10000 10 200 10 .1)
  (both 10000 20 200 10 .1)
  (both 10000 50 200 10 .1))
```

```
;;;The structure of this file is acrobot-window, pole dynamics stuff,
;;; acrobot-display stuff, top-level stuff, agents


;;; The acrobot-WINDOW is a basic simulation-window with just a few specializations.

(defclass acrobot-WINDOW
  (stop-go-button step-button quiet-button simulation-window)
  ((world :accessor acrobot)))

(defmethod window-close :before ((window acrobot-window))
  (window-close (3D-graph-window (acrobot window))))

(defmethod view-draw-contents :after ((w acrobot-window))
  (when (and (slot-boundp w 'world) (slot-boundp (acrobot w) 'flip))
    (draw-acrobot-background (acrobot w))))

(defclass acrobot (terminal-world displayable-world)
  ((acrobot-position1 :reader acrobot-position1 :initarg :acrobot-position1 :initform
0.0)
   (acrobot-velocity1 :accessor acrobot-velocity1 :initarg :acrobot-velocity1
:initform 0.0)
   (acrobot-position2 :reader acrobot-position2 :initarg :acrobot-position2 :initform
0.0)
   (acrobot-velocity2 :accessor acrobot-velocity2 :initarg :acrobot-velocity2
:initform 0.0)
   (side-view :accessor side-view :initarg :side-view)
   (phase-view1 :accessor phase-view1 :initarg :phase-view1)
   (phase-view2 :accessor phase-view2 :initarg :phase-view2)
   (3D-Graph-window :accessor 3D-graph-window)
   (last-action :accessor last-action :initform nil)
   white
   black
   flip
   fat-flip))

(defmethod world-state ((p acrobot))
  (list (acrobot-position1 p) (acrobot-velocity1 p) (acrobot-position2 p) (acrobot-
velocity2 p)))

(defvar PI/2 (coerce (/ PI 2) 'long-float))
(defvar 2PI (coerce (* PI 2) 'long-float))
(defvar -PI (coerce (- PI) 'long-float))
(defvar acrobot-limit1 (coerce PI 'long-float))
(defvar acrobot-limit2 (coerce PI 'long-float))
(defvar acrobot-delta-t 0.1e0)              ; seconds between state updates
(defvar acrobot-max-velocity1 (coerce (/ (* .04 PI) .02) 'long-float))
(defvar acrobot-max-velocity2 (coerce (/ (* .09 PI) .02) 'long-float))
(defvar acrobot-max-force 2e0)
(defvar acrobot-gravity 9.8e0)
(defvar acrobot-mass1 1.0e0)
(defvar acrobot-mass2 1.0e0)
(defvar acrobot-length1 1.0e0)
(defvar acrobot-length2 1.0e0)
(defvar acrobot-length-center-of-mass1 0.5e0)
(defvar acrobot-length-center-of-mass2 0.5e0)
(defvar acrobot-inertia1 1.0e0)
(defvar acrobot-inertia2 1.0e0)

(setq PI/2 (coerce (/ PI 2) 'long-float))
(setq acrobot-limit1 (coerce PI 'long-float))
(setq acrobot-limit2 (coerce PI 'long-float))
(setq acrobot-delta-t 0.2e0)              ; seconds between state updates
```

```
(setq acrobot-max-velocity1 (coerce (/ (* .04 PI) .02) 'long-float))
(setq acrobot-max-velocity2 (coerce (/ (* .09 PI) .02) 'long-float))
(setq acrobot-max-force 1e0)
(setq acrobot-gravity 9.8e0)
(setq acrobot-mass1 1.0e0)
(setq acrobot-mass2 1.0e0)
(setq acrobot-length1 1.0e0)
(setq acrobot-length2 1.0e0)
(setq acrobot-length-center-of-mass1 0.5e0)
(setq acrobot-length-center-of-mass2 0.5e0)
(setq acrobot-inertia1 1.0e0)
(setq acrobot-inertia2 1.0e0)

(defmethod world-transition ((p acrobot) a)
  (let* ((substeps 4)
         (1/substeps (/ substeps)))
    (loop repeat substeps until (terminal-state? p) do
          (let* ((q2 (acrobot-position2 p))
                 (q2-dot (acrobot-velocity2 p))
                 (q1 (- (acrobot-position1 p) PI/2))
                 (q1-dot (acrobot-velocity1 p))
                 (force (* acrobot-max-force (max -1 (min a 1))))
                 (cos-q2 (cos q2))
                 (sin-q2 (sin q2))
                 (cos-q1+q2 (cos (+ q1 q2)))
                 (m1 acrobot-mass1)
                 (m2 acrobot-mass2)
                 (l1 acrobot-length1)
                 (lc1 acrobot-length-center-of-mass1)
                 (lc2 acrobot-length-center-of-mass2)
                 (d11 (+ (* m1 lc1 lc1)
                         (* m2 (+ (* l1 l1)
                                  (* lc2 lc2)
                                  (* 2 l1 lc2 cos-q2)))
                         acrobot-inertia1 acrobot-inertia2))
                 (d22 (+ (* m2 lc2 lc2)
                         acrobot-inertia2))
                 (d12 (+ (* m2 (+ (* lc2 lc2)
                                  (* l1 lc2 cos-q2)))
                         acrobot-inertia2))
                 (h1 (+ (- (* m2 l1 lc2 sin-q2 q2-dot q2-dot))
                        (- (* 2 m2 l1 lc2 sin-q2 q2-dot q1-dot))))
                 (h2 (* m2 l1 lc2 sin-q2 q1-dot q1-dot))
                 (phi1 (+ (* (+ (* m1 lc1) (* m2 l1))
                             acrobot-gravity (cos q1))
                          (* m2 lc2 acrobot-gravity cos-q1+q2)))
                 (phi2 (* m2 lc2 acrobot-gravity cos-q1+q2))
                 (q2-acc (/ (+ force
                               (* d12 (/ d11) (+ h1 phi1))
                               (- h2)
                               (- phi2))
                            (- d22
                               (* d12 d12 (/ d11)))))
                 (q1-acc (/ (+ (* d12 q2-acc) h1 phi1)
                            (- d11))))
            (incf q1-dot (* 1/substeps acrobot-delta-t q1-acc))
            (bound q1-dot (* 2 acrobot-max-velocity1))
            (incf q1 (* 1/substeps acrobot-delta-t q1-dot))
            (incf q2-dot (* 1/substeps acrobot-delta-t q2-acc))
            (bound q2-dot (* 2 acrobot-max-velocity2))
            (incf q2 (* 1/substeps acrobot-delta-t q2-dot))
            ;(print (list q1 q1-dot q2 q2-dot))
            (set-acrobot-state p (+ q1 PI/2) q1-dot q2 q2-dot a))))
  (setf (world-reward p) -1)
  (world-reward p))

(defun acrobot-cm-angle (state)
  (let* ((q1 (first state))
         (q2 (third state))
         (m1 acrobot-mass1)
         (m2 acrobot-mass2)
         (l1 acrobot-length1)
         (lc1 acrobot-length-center-of-mass1)
         (lc2 acrobot-length-center-of-mass2)
         (x- (sin q2))
         (x (/ (* m2 x-)
               (+ m1 m2)))
         (y- (+ l1 lc2 (cos q2)))
         (y (/ (+ (* m1 lc1) (* m2 y-))
               (+ m1 m2))))
    (+ q1 (atan (/ x y)))))

(defmethod terminal-state? ((acrobot acrobot) &optional (state (world-state
acrobot)))
;  (> (abs (acrobot-cm-angle x)) PI))
  (let* ((angle1 (first state))
         (angle2 (third state))
         (x (* acrobot-length1 (sin angle1)))
         (y (- (* acrobot-length1 (cos angle1))))
         (total-angle (+ angle1 angle2))
         (handx (+ x (* acrobot-length2 (sin total-angle))))
         (handy (+ y (- (* acrobot-length2 (cos total-angle))))))
  (and ;(> handx 1)
       ;(< handx 1.45)
       (> handy 1)
       )));(< handy 1.45))))

(defmethod world-reset ((world acrobot))
  (sleep .5)
  (set-acrobot-state world 0 0 0 0 nil)
  (print (world-time world))
  (when (window world)
    (let* ((window (window world))
           (black (g-color-black window))
           (white (g-color-white window)))
      (gd-fill-rect-r window 20 400 200 50 white)
      (gd-draw-text window
                    (format nil "~A" (+ 1 (length (simulation-trial-reward-history
window))))
                    '("monaco" :srcXor 24)
                    20 650 black)
      (gd-draw-text window "" '("chicago" :srcXor 12) 20 650 black)))
  (world-state world))

(defclass acrobot-phase-view2 (g-view) ())

(defmethod g-click-event-handler ((top-view acrobot-phase-view2) x y)
  (let ((state (list (acrobot-position1 *world*) (acrobot-velocity1 *world*) x y)))
    (format t "~A~%" (if (terminal-state? *world* state)
                           0
                           (state-value *agent* (sense *agent* *world* state))))))

(defclass acrobot-phase-view1 (g--view) ())

(defmethod g-click-event-handler ((top-view acrobot-phase-view1) x y)
  (let ((state (list x y (acrobot-position2 *world*) (acrobot-velocity2 *world*))))
    (format t "~A~%" (if (terminal-state? *world* state)
                         0
                         (state-value *agent* (sense *agent* *world* state))))))

(defmethod world-init-display ((acrobot acrobot))
  (with-slots (window side-view phase-view2 phase-view1) acrobot
    (unless (displayp acrobot)
      (setf window (make-instance 'acrobot-window
                     :window-type :document
                     :window-show nil
                     :view-font '("chicago" 12 :plain)
                     :window-title "Acrobot"
                     :window-do-first-click t
                     :gd-viewport-r '(10 40 540 580)))
      (setf (3D-graph-window acrobot) (make-instance '3D-graph-window
                                         :gd-viewport-r '(580 100 400 400)
                                         :window-show nil))
      (let ((button (make-instance 'button-dialog-item
                      :view-container window
                      :dialog-item-text "3D Graph Joint 1"
                      :dialog-item-action
                      #'(lambda (item) (acrobot-3D-graph-button-action window
item)))))
        (set-view-position-y-up button 160 3)
        (add-subviews window button))
      (let ((button (make-instance 'button-dialog-item
                      :view-container window
                      :dialog-item-text "3D Graph Joint 2"
                      :dialog-item-action
                      #'(lambda (item) (acrobot-3D-graph-button-action window
item)))))
        (gd-set-viewport button 340 3 nil nil)
        (add-subviews window button))
      (setf (world window) acrobot))
    (g-set-coordinate-system window 0 0 1 1)
    (setf side-view (make-instance 'g-view :parent window))
    (setf phase-view1 (make-instance 'acrobot-phase-view1 :parent window))
    (setf phase-view2 (make-instance 'acrobot-phase-view2 :parent window))
    (gd-set-viewport side-view 20 80 520 580)
    (gd-set-viewport phase-view2 270 40 520 290)
    (gd-set-viewport phase-view1 20 40 270 290)
    (let ((limit (+ acrobot-length2 acrobot-length2)))
      (g-set-coordinate-system side-view (- limit) (- limit) limit limit))
    (g-set-coordinate-system phase-view1 (- acrobot-limit1) (- acrobot-max-velocity1)
                 acrobot-limit1 acrobot-max-velocity1)
    (g-set-coordinate-system phase-view2 (- acrobot-limit2) (- acrobot-max-velocity2)
                 acrobot-limit2 acrobot-max-velocity2)
    (setf (slot-value acrobot 'white) (g-color-white side-view))
    (setf (slot-value acrobot 'black) (g-color-black side-view))
    (setf (slot-value acrobot 'fat-flip) (g-color-flip side-view))
    (setf (slot-value acrobot 'fat-flip)
          (g-color-set-pen side-view (slot-value acrobot 'fat-flip) nil nil 2 2))
    (setf (slot-value acrobot 'flip) (g-color-flip side-view))))

(defmethod draw-acrobot-background ((p acrobot))
  (with-slots (side-view black white last-drawn-reward phase-view2 phase-view1) p
    (when (displayp p)
;      (g-outline-rect phase-view2 (- acrobot-limit2) (- acrobot-max-velocity2)
;                      acrobot-limit2 acrobot-max-velocity2 black)
;      (g-outline-rect phase-view1 (- acrobot-limit1) (- acrobot-max-velocity1)
;                      acrobot-limit1 acrobot-max-velocity1 black)
;      (g-fill-rect side-view 1 1 1.45 1.45 (g-color-name side-view :gray))
      (let ((limit (+ acrobot-length2 acrobot-length2)))
        (g-draw-line side-view (- limit) 1 limit 1 (g-color-name side-view :gray))
        (loop for y from (- limit) to limit  by (/ limit 10) do
              (g-draw-point side-view 0 y black)))
      (g-draw-disk side-view 0 0 .02 black)
      (let ((window (window p)))
        (gd-fill-rect-r window 20 650 200 50 white)
        (gd-draw-text window
                      (format nil "~A" (+ 1 (length (simulation-trial-reward-history
window))))
                      '("monaco" :srcXor 24)
                      20 650 black)
        (gd-draw-text window "" '("chicago" :srcXor 12) 20 650 black))
      (setf (world-time p) (world-time p))
      (draw-acrobot-state p))))

(defconstant radians-to-degrees (/ 360 PI 2))
(defconstant degrees-to-radians (/ PI 180))

(defmethod draw-acrobot-state ((p acrobot))
  (with-slots (side-view phase-view2 phase-view1 acrobot-position2 acrobot-position1
                         acrobot-velocity2 acrobot-velocity1 last-action
                         fat-flip flip black white) p
;    (g-draw-disk phase-view1 acrobot-position1 acrobot-velocity1 .1 flip)
;    (g-draw-point phase-view1 acrobot-position1 acrobot-velocity1 black)
;    (g-draw-disk phase-view2 (- (mod (+ acrobot-position2 PI) 2PI) PI) acrobot-
velocity2 .07 flip)
;    (g-draw-point phase-view2 (- (mod (+ acrobot-position2 PI) 2PI) PI) acrobot-
velocity2 black)
    (let* ((x (* acrobot-length1 (sin acrobot-position1)))
           (y (- (* acrobot-length1 (cos acrobot-position1))))
           (dx (gd-coord-x side-view x))
           (dy (gd-coord-y side-view y))
           (total-angle (+ acrobot-position1 acrobot-position2))
           (xinc (* acrobot-length2 (sin total-angle)))
           (yinc (- (* acrobot-length2 (cos total-angle))))
           (dradius 20)
           (arc-size 60)
           (fudge .2)
           (radius (g-offset-x side-view dradius)))
      (g-draw-line side-view 0 0 x y fat-flip)
      ;      (g-draw-disk side-view x y .04 flip)
      (g-draw-line-r side-view x y xinc yinc fat-flip)
      (gd-draw-arc side-view dx dy dradius
                   (- (mod (truncate (* radians-to-degrees total-angle)) 360) 90)
                   (* arc-size (or last-action 0)) flip)
      (incf total-angle (* degrees-to-radians arc-size (or last-action 0)))
      (when (member last-action '(1 -1))
        (g-draw-arrowhead side-view (+ x (* fudge xinc)) (+ y (* fudge yinc))
                          (+ x (* radius (sin total-angle)))
                          (- y (* radius (cos total-angle)))
                          0.0 .3 flip)))))


(defclass CMAC-acrobot-AGENT (acrobot-agent random-policy greedy-policy ERFA-Q-
Learning) ())

(defmethod agent-step :after ((agent CMAC-acrobot-agent) x a y r)
  (declare (ignore x a y r))
  (when (update-displayp (world agent))
    (with-slots (side-view black white) (world agent)
      (let* ((base (+ 1 (gd-coord-y side-view 1.0)))
             (time (world-time (world agent)))
             (x (+ 20 (mod time 500)))
             (x+ (+ 20 (mod (+ time 15) 500)))
             (length (min 65 (truncate (* (slot-value agent 'a-value) 0.5)))))
        (gd-draw-line-r side-view x+ base 0 65 white)
        (gd-draw-point side-view x (- base length) black)))))

(defmethod set-acrobot-state ((p acrobot) new-acrobot-position1 new-acrobot-velocity1
                              new-acrobot-position2 new-acrobot-velocity2 new-action)
  (when (update-displayp p) (draw-acrobot-state p))
  (setf (slot-value p 'acrobot-position1) new-acrobot-position1)
  (setf (slot-value p 'acrobot-velocity1) new-acrobot-velocity1)
  (setf (slot-value p 'acrobot-position2) new-acrobot-position2)
  (setf (slot-value p 'acrobot-velocity2) new-acrobot-velocity2)
  (setf (slot-value p 'last-action) new-action)
  (when (update-displayp p) (draw-acrobot-state p)))


;;;TOP-LEVEL STUFF:

;(defun make-acrobot-simulation (&optional (agent-class 'manual-acrobot-agent))
(defun make-acrobot-simulation (&optional (agent-class 'acrobot-sarsa-agent))
  (let ((acrobot (make-instance 'acrobot)))
    (setf (update-displayp acrobot) t)
    (setf (agent (window acrobot)) (make-agent acrobot agent-class))
    (when (typep (agent (window acrobot)) 'sarsa-agent)
      (setf (lambda (agent (window acrobot))) .9))))

(defun make-acrobot-and-run-silently (agent-class num-steps)
  (let* ((acrobot (make-instance 'acrobot))
         (agent (make-agent acrobot agent-class))
         (simulation (make-instance 'simulation :agent agent :world acrobot)))
    (simulation-run simulation num-steps)))

;;; AGENT STUFF BEGINS HERE

(defclass acrobot-AGENT (terminal-agent tabular-action-agent) ())

(defmethod make-agent ((acrobot acrobot) &optional (agent-class 'q-acrobot-agent))
  (cond ((subtypep agent-class 'q-acrobot-agent)
         (make-instance agent-class :num-actions 3))
        ((subtypep agent-class 'cmac-acrobot-agent)
         (make-instance agent-class :world acrobot :num-actions 3))
        ((subtypep agent-class 'manual-agent)
         (make-instance agent-class))))

(defmethod convert-action ((agent tabular-action-agent) (world acrobot) action-
number)
  (- action-number 1))

;;; A Q-agent could be done as follows.  Divide the unit square of acrobot
;;; positions and velocities into a large number of intervals, say 100
;;; for position and 10 for velocity.  Let
;;; each one be a Q-learner state.  Consider 3 actions, +1, 0, and -1.

#|
(defclass Q-acrobot-AGENT (acrobot-agent random-policy tabular-q-learning greedy-
policy)
  ((num-states :accessor num-states :initarg num-states :initform 1000)
   (initial-Q-value :accessor initial-Q-value :initarg :initial-Q-value :initform
1.0)))

(defmethod sense ((agent tabular-q-learning) (world acrobot) &optional (pos-and-vel-
list (world-state world)))
  (let* ((pos (first pos-and-vel-list))
         (vel (second pos-and-vel-list))
         (position (max 0 (min 0.999999 (/ (- pos acrobot-min)
                                           (- acrobot-max acrobot-min)))))
         (velocity (max 0 (min 0.999999 (+ 0.5 (/ vel
                                                 acrobot-max-velocity2
                                                 2.0))))))
    (+ (* 10 (floor (* 100 position)))
       (floor (* 10 velocity)))))
|#
(defclass acrobot-sarsa-agent
  (single-CMAC-acrobot-AGENT ERFA-sarsa-agent) ())

(defclass multi-CMAC-acrobot-AGENT (CMAC-acrobot-AGENT) ())

(defclass single-CMAC-acrobot-AGENT (CMAC-acrobot-AGENT) ())

(defmethod initialize-instance ((agent single-CMAC-acrobot-agent) &rest initargs)
  (apply #'call-next-method agent initargs)
  (with-slots (representer FAs num-actions) agent
    (setf (alpha agent) 0.2e0)
    (setf (gamma agent) 1.0e0)
    (setf (prob-of-random-action agent) 0)
    (gc)
    (setf representer
          (make-instance 'CMAC-representer
            :input-descriptor
            (list (list (truncate (* 1000000 (- acrobot-limit1)))
                        (truncate (* 1000000 (* acrobot-limit1 1.000e0)))
                        6)
                  (list (truncate (* 1000000 (- acrobot-max-velocity1)))
                        (truncate (* 1000000 (* acrobot-max-velocity1 1.333e0)))
                        7)
                  (list (truncate (* 1000000 (- acrobot-limit2)))
                        (truncate (* 1000000 (* acrobot-limit2 1.000e0)))
                        6)
                  (list (truncate (* 1000000 (- acrobot-max-velocity2)))
                        (truncate (* 1000000 (* acrobot-max-velocity2 1.333e0)))
                        7))
          :contraction 1.0
          :num-layers 10))
  (setf FAs
        (loop for a below num-actions
              collect (make-instance 'normalized-step-adaline
                        :num-inputs (num-outputs representer)
                        :initial-weight (coerce (/ 0.0 (num-layers representer))
'long-float))))))

(defmethod initialize-instance ((agent multi-CMAC-acrobot-agent) &rest initargs)
  (apply #'call-next-method agent initargs)
  (with-slots (representer FAs num-actions) agent
    (setf (alpha agent) 0.2e0)
    (setf (gamma agent) 1.0e0)
    (setf (prob-of-random-action agent) 0)
    (gc)
    (setf representer
          (make-instance 'multi-representer
            :num-inputs 4
            :representers
            (let ((limits
                   (list (list (- acrobot-limit1) (* acrobot-limit1 1.000e0))
                         (list (- acrobot-max-velocity1) (* acrobot-max-velocity1
1.333e0))
                         (list (- acrobot-limit2) (* acrobot-limit2 1.000e0))
                         (list (- acrobot-max-velocity2) (* acrobot-max-velocity2
1.333e0))))
                  (intervals '(6 7 6 7)))
              (loop for limits-i in limits do
                    (setf (first limits-i) (truncate (* 1000000 (first limits-i))))
                    (setf (second limits-i) (truncate (* 1000000 (second limits-
i)))))
              (append (make-singleton-representers 'CMAC-representer limits intervals
3)
                      (make-doubleton-representers 'CMAC-representer limits intervals
2)
                      (make-representers 'CMAC-representer (combinations 4 3) limits
intervals 3)
                      (make-representers 'CMAC-representer '((0 1 2 3)) limits
intervals 12)))))
    (setf FAs
        (loop for a below num-actions
              collect (make-instance 'normalized-step-adaline
                        :num-inputs (num-outputs representer)
                        :initial-weight (coerce (/ 0.0 (num-layers representer))
'long-float))))))

(defmethod sense ((agent CMAC-acrobot-agent) (world acrobot)
                  &optional (state-list (world-state world)))
  (if (terminal-state? world state-list)
    :terminal-state
    (let ((array (make-array (length state-list))))
      (setf (aref array 0) (- (mod (+ (first state-list) PI) 2PI) PI))
      (setf (aref array 1) (limit (second state-list) acrobot-max-velocity1))
      (setf (aref array 2) (- (mod (+ (third state-list) PI) 2PI) PI))
      (setf (aref array 3) (limit (fourth state-list) acrobot-max-velocity2))
      (loop for i below (length state-list) do
            (setf (aref array i) (truncate (* 1000000 (aref array i)))))
      array)))

(defclass manual-acrobot-agent (manual-pole-agent) ())

(defun acrobot-3D-graph-button-action (world-window item)
  (when (or (equal "3D Graph Joint 1" (dialog-item-text item))
            (equal "3D Graph Joint 2" (dialog-item-text item)))
    (disable-buttons world-window)
    (let ((old-sim-running (simulation-runningp world-window))
          (text (dialog-item-text item)))
      (setf (simulation-runningp world-window) nil)
      (eval-enqueue `(progn (set-dialog-item-text ,item "Graphing..")
                            (acrobot-3d-graph ,world-window ,text 20 (3D-graph-window
(world ,world-window)))
                            (set-dialog-item-text ,item ,text)
                            (enable-buttons ,world-window)
                            (when ,old-sim-running (simulation-run ,world-
window)))))))


(defmethod acrobot-3d-graph ((world-window acrobot-window) text res
                      &optional (3d-window (make-instance '3D-graph-window
                                             :view-size #@(400 400)
                                             :view-position #@(500 50)
                                             :window-show nil)))
  (with-slots (data-array) 3D-window
    (setf data-array (make-array (list res res)))
    (cond
     ((equal text "3D Graph Joint 1")
      (loop for i below res
            for pos = (* acrobot-limit1 2 (- (/ (+ i 0.5) res) 0.5)) do
            (loop for j below res
                for vel = (* acrobot-max-velocity1 2 (- (/ (+ j 0.5) res) 0.5))
                do (setf (aref data-array i j)
                         (state-value (agent world-window)
                                      (sense (agent world-window) (world world-
window)
                                             (list pos vel
                                                   (acrobot-position2 (world world-
window))
                                                   (acrobot-velocity2 (world world-
window)))))))))
     ((equal text "3D Graph Joint 2")
      (loop for i below res
            for pos = (* acrobot-limit2 2 (- (/ (+ i 0.5) res) 0.5)) do
            (loop for j below res
                for vel = (* acrobot-max-velocity2 2 (- (/ (+ j 0.5) res) 0.5))
                do (setf (aref data-array i j)
                         (state-value (agent world-window)
                                      (sense (agent world-window) (world world-
window)
                                             (list (acrobot-position1 (world world-
window))
                                                   (acrobot-velocity1 (world world-
window))
                                                   pos vel)))))))
     (t (error "Unrecognized button")))
    (g-make-visible 3D-window)
    (g::graph-surface 3D-window data-array)))

(defvar scaling .3)

(defun draw-bot (side-view action position1 position2 black)
    (let* ((x (* acrobot-length1 (sin position1)))
           (y (- (* acrobot-length1 (cos position1))))
           (dx (gd-coord-x side-view x))
           (dy (gd-coord-y side-view y))
           (total-angle (+ position1 position2))
           (xinc (* acrobot-length2 (sin total-angle)))
           (yinc (- (* acrobot-length2 (cos total-angle))))
           (dradius (round (* scaling 20)))
           (arc-size 60)
           (fudge .25)
           (radius (g-offset-x side-view dradius)))
      (g-draw-line side-view 0 0 x y black)
      (g-draw-line-r side-view x y xinc yinc black)
      (gd-draw-arc side-view dx dy dradius
                   (- (mod (truncate (* radians-to-degrees total-angle)) 360) 90)
                   (* arc-size (or action 0)) black)
      (incf total-angle (* degrees-to-radians arc-size (or action 0)))
      (when (member action '(1 -1))
        (g-draw-arrowhead side-view (+ x (* fudge xinc)) (+ y (* fudge yinc))
                          (+ x (* radius (sin total-angle)))
                          (- y (* radius (cos total-angle)))
                          0.0 .3 black))))
#|
(defun segments ()
  (loop for (off start end) in segments
        for offset = (round (* scaling off)) do
        (gd-set-viewport c offset 10 (round (+ offset (* scaling 400))) (round (+ 10
(* scaling 300))))
        (cl))
  (loop for (off start end) in segments
        for offset = (round (* scaling off)) do
        (gd-set-viewport c offset 10 (round (+ offset (* scaling 400))) (round (+ 10
(* scaling 300))))
        (segment offset start end)))

(defun cl () (g-clear c))

(defun segment (offset start end)
  (gd-set-viewport c offset 10 (round (+ offset (* scaling 400))) (round (+ 10 (*
scaling 300))))
  (g-draw-line c -1 1 +2 1 black)
  (g-draw-disk c 0 0 .02 black)
  (setq black (g-color-set-pen c black nil nil 2 2))
  (apply 'draw (nth start data))
  (setq black (g-color-set-size c black 1 1))
  (loop for n from start to end
        for d = (nth n data)
        do (apply 'draw d)))

(defun draw (a p1 p2) (draw-bot c (- a 1) p1 p2 black))


(setq segments '((1690 63 68) (1430 55 62) (1200 48 54) (975 41 47) (750 34 40) (540
28 33) (375 22 27) (200 16 21) (100 10 15) (0 4 9) (-70 0 3)))


(defun scrap-segments ()
    (start-picture c)
    (segments)
    (put-scrap :pict (get-picture c)))


(defun make-acrobot-and-run-trials-silently (agent-class num-trials num-steps)
  (let* ((acrobot (make-instance 'acrobot))
         (agent (make-agent acrobot agent-class))
         (simulation (make-instance 'simulation :agent agent :world acrobot)))
    (simulation-run-trials simulation num-trials num-steps)))

|#
```

Solutions Manual for:

Reinforcement Learning: An Introduction
by Richard S. Sutton and Andrew G. Barto

An instructor's manual containing answers to all the non-programming exercises is available to qualified teachers. Send or fax a letter under your university's letterhead to the Text Manager at MIT Press. Exactly who you should send to depends on your location. Obtain the address as if you were requesting an examination copy.

Readers using the book for self study can obtain answers on a chapter-by-chapter basis after working on the exercises themselves. Send email to rich@richsutton.com with your efforts to answer the exercises for a chapter, and we will send back a postscript file with the answers for that chapter.

We are also collecting overheads, code, exams, and other material useful for teaching from the book. If you have anything that may be useful in this regard that you would like to share with others, please send it in and we'll make it available.

Figures for:

Reinforcement Learning: An Introduction
by Richard S. Sutton and Andrew G. Barto

Below are links to postscript files for the figures of the book.

Errata and Notes for:

Reinforcement Learning: An Introduction
by Richard S. Sutton and Andrew G. Barto

Errata:

- p. xviii, Ben Van Roy should be acknowledged only once in the list. (Ben Van Roy)
- p. 155, the parameter alpha was 0.01, not 0.1 as stated. (Abinav Garg)
- p. 233, last line of caption: "ne-step" should be "one-step". (Michael Naish)
- p. 309, the reference for Tstisiklis and Van Roy (1997b) should be to Technical Report LIDS-P-2390, Massachusetts Institute of Technology. (Ben Van Roy)
- p. 146, the windy gridworld example *may* have used alpha=0.5 rather than alpha=0.1 as stated. Can you confirm this?
- p. 322, in the index entry for TD error, the range listed as "174-165" should be "174-175". (Jette Randlov)
- p. 197, bottom formula last theta_t(2) should be theta_t(n). (Dan Bernstein)
- p. 151, second line of the equation, pi(s_t,a_t) should be pi(s_{t+1},a_t). (Dan Bernstein)
- p. 174, 181, 184, 200, 212, 213: in the boxed algorithms on all these pages, the setting of the eligibility traces to zero should appear not in the first line, but as a new first line inside the first loop (just after the "Repeat..."). (Jim Reggia)
- p. 215, Figure 8.11, the y-axis label. "first 20 trials" should be "first 20 episodes".
- p. 215. The data shown in Figure 8.11 was apparently not generated exactly as described in the text, as its details (but not its overall shape) have defied replication. In particular, several researchers have reported best "steps per episode" in the 200-300 range.
- p. 78. In the 2nd max equation for V*(h), at the end of the first line, "V*(h)" should be "V*(l)". (Christian Schulz)
- p. 29. In the upper graph, the third line is unlabeled, but should be labeled "epsilon=0 (greedy)".
- p. 212-213. In these two algorithms, a line is missing that is recommended, though perhaps not required. A next to the last line should be added, just before ending the loop, that recomputes Q_a. That line would be Q_a <- \sum_{i\in F_a} theta(i).
- p. 127, Figure 5.7. The first two lines of step (c) refer to pairs s,a and times t at or later than time \tau. In fact, it should only treat them for times *later* than \tau, not equal. (Thorsten Buchheim)
- p. 267, Table 11.1. The number of hidden units for TD-Gammon 3.0 is given as 80,

but should be 160. (Michael Naish)
- p. 98, Figure 4.3. Stuart Reynolds points out that for some MDPs the given policy iteration algorithm never terminates. The problem is that there may be small changes in the values computed in step 2 that cause the policy to forever be changing in step 3. The solution is to terminate step 3 not when the policy is stable, but as soon as the largest change in state value due to a policy change is less than some epsilon.
- p. 259, the reference to McCallum, 1992 should be to Chrisman, 1992. And in the references section, on p. 302, the (incorrect) listing for McCallum, 1992 should not be there. (Paul Crook)

Notes:

- p. 212-213. In these two algorithms, it is implicit that the set of features for the *terminal state* (and all actions) is the empty set.
- p. 28. The description of the 10-armed testbed could be clearer. Basically there are 2000 randomly generated 10-armed bandits. The Q*(a) of each of these were selected from a normal distribution with mean 0 and variance 1. Then, on each play with each bandit, the reward was determined by adding to Q*(a) another normally distributed random number with mean 0 and variance 1.
- p. 127, Figure 5.7. This algorithm is only valid if all policies are *proper*, meaning that they produce episodes that always eventually terminate (this assumption is made on the first page of the chapter). This restriction on environments can be lifted if the algorithm is modified to use epsilon-soft policies, which are proper for all environments. Such a modification is a good exercise for the reader! Alternative ideas for off-policy Monte Carlo learning are discussed in this recent research paper.
- John Tsitsiklis has obtained some new results which come very close to solving "one of the most important open theoretical questions in reinforcement learning" -- the convergence of Monte Carlo ES. See here.
- The last equation on page 214 can be a little confusing. The minus sign here is meant to be grouped with the 0.0025 (as the spacing suggests). Thus the consecutive plus and minus signs have the same effect as a single minus sign. (Chris Hobbs)

Presentation Slides for Teaching from:

Reinforcement Learning: An Introduction
by Richard S. Sutton and Andrew G. Barto

Powerpoint slides for teaching each chapter of the book have been prepared and made available by Professor Barto. These are available as powerpoint files and as postscript files. The latter may be the most useful if you don't have all the right fonts installed. It is also possible to preview the slides via the web.

French versions of many of these slides are available here from Tarik Al ANI.

Chapter 1, Introduction: powerpoint, compressed postscript, web preview.

Chapter 2, Evaluative Feedback: powerpoint, compressed postscript, web preview.

Chapter 3, The Reinforcement Learning Problem: powerpoint, compressed postscript, web preview.

Chapter 4, Dynamic Programming: powerpoint, compressed postscript, web preview.

Chapter 5, Monte Carlo Methods: powerpoint, compressed postscript, web preview.

Chapter 6, Temporal-Difference Learning: powerpoint, compressed postscript, web preview.

Chapter 7, Eligibility Traces: powerpoint, compressed postscript, web preview.

Chapter 8, Generalization and Function Approximation: powerpoint, compressed postscript, web preview.

Chapter 9, Planning and Learning: powerpoint, compressed postscript, web preview.

Chapter 10, Dimensions: powerpoint, compressed postscript, web preview.

Chapter 11, Case Studies: powerpoint, compressed postscript, web preview.

Preface

We first came to focus on what is now known as reinforcement learning in late 1979. We were both at the University of Massachusetts working on one of the earliest projects to revive the old idea that networks of neuron-like adaptive elements might prove to be a promising approach to artificial adaptive intelligence. The project explored the ``heterostatic theory of adaptive systems'' developed by A. Harry Klopf. Harry's work was a rich source of ideas, and we were permitted to explore them critically and compare them with the long history of prior work in adaptive systems. Our task became one of teasing the ideas apart and understanding their relationships and relative importance. This continues to this day, but in 1979 we came to realize that perhaps the simplest of the ideas, which had long been taken for granted, had in fact received relatively little attention from a computational perspective. This was simply the idea of a learning system that wants something, that adapts its behavior in order to maximize a special signal from its environment. This was the idea of a ``hedonistic'' learning system, or, as we would say now, the idea of reinforcement learning.

Like others, we had a sense that reinforcement learning had been thoroughly explored in the early days of artificial intelligence and cybernetics. On closer inspection, though, we found that it had been explored only slightly. We found that while reinforcement learning had clearly motivated some of the earliest computational studies of learning, some of which were most impressive, most of these researchers had gone on to other things, such as pattern classification, supervised learning, adaptive control, or they had abandoned the study of learning altogether. As a result, the special issues involved in learning how to get something from the environment received relatively little attention. In retrospect, focusing on this idea was the critical step that set this branch of research in motion. Little progress could be made in the computational study of reinforcement learning until it was recognized that such a fundamental idea had not yet been thoroughly explored.

The field has come a long way since then, evolving and maturing in several directions. Reinforcement learning has gradually become one of the most active research areas in machine learning, artificial intelligence, and neural-network research. The field has developed strong mathematical foundations and impressive applications. The overall problem of learning from interaction to achieve goals is still far from being solved, but our understanding of it has improved significantly. We can now place component ideas, such as temporal-difference learning, dynamic programming, and function approximation, within a coherent perspective with respect to the overall problem. The computational study of reinforcement learning is now a large field, with hundreds of active researchers around the world in diverse disciplines such as psychology, control theory, artificial intelligence, and neuroscience. Particularly important have been the contributions establishing and developing the relationships to the theory of optimal control and dynamic programming.

Our goal in writing this book is to provide a clear and simple account of the key ideas and algorithms of reinforcement learning. We wanted our treatment to be accessible to readers in all of the related disciplines, but we could not cover all of these perspectives in detail. Our treatment takes almost exclusively the point of view of artificial intelligence and engineering, leaving coverage of connections to psychology, neuroscience, and other fields to others or to another time. We also chose not to produce a rigorous formal treatment of reinforcement learning. We did not reach for the highest possible level of mathematical abstraction and did not rely on a theorem-proof format. We tried to choose a level of mathematical detail that points the mathematically inclined in the right directions without distracting from the simplicity and potential generality of the underlying ideas.

The book consists of three parts. Part I is introductory and problem oriented. We focus on the simplest aspects of reinforcement learning and on its main distinguishing features. One full chapter is devoted to introducing the reinforcement learning problem whose solution we explore in the rest of the book. Part II presents what we see as the three most important elementary solution methods: dynamic programming, simple Monte Carlo methods, and temporal-difference learning. The first of these is a planning method and assumes explicit knowledge of all aspects of a problem, whereas the other two are learning methods. Part III is concerned with generalizing these methods and blending them together. Eligibility traces allow unification of Monte Carlo and temporal-difference methods, and function approximators such as artificial neural networks extend all the methods so that they can be applied to much larger problems. We bring planning and learning methods together again and relate them to heuristic search. Finally, we present several case studies, including some of the most impressive applications of reinforcement learning to date, and briefly discuss some of the open problems and near-future prospects for reinforcement learning.

This book was designed to be used as a text in a one-semester course, perhaps supplemented by readings from the literature or by a more mathematical text such as the excellent one by Bertsekas and Tsitsiklis (1996). This book can also be used as part of a broader course on machine learning, artificial intelligence, or neural networks. In this case, it may be desirable to cover only a subset of the material. We recommend covering Chapter 1 for a brief overview, Chapter 2 through Section 2.2, all of the non-starred sections of Chapter 3, and then selecting sections from the remaining chapters according to time and interests. Chapters 4, 5, and 6 build on each other and are best covered in sequence, but of these, Chapter 6 is the most important for the subject and for the rest of the book. A course focusing on machine learning or neural networks should cover Chapter 8, and a course focusing on artificial intelligence or planning should cover Chapter 9. Chapter 10 should almost always be covered because it is very short and summarizes the overall unified view of reinforcement learning methods developed in the book. Throughout the book, sections that are more difficult and not essential to the rest of the book are marked with a *. These can be omitted on first reading without creating problems later on. Some exercises are also marked with a * to indicate that they are more advanced and not essential to understanding the basic material of the chapter.

The book is largely self-contained. The only mathematical background assumed is familiarity with elementary concepts of probability, such as expectations of random variables. Chapter 8 is substantially easier to digest if the reader already has some familiarity with artificial neural networks or some other kind of supervised learning method, but it can also be read without prior background. We strongly recommend working the exercises provided throughout the book. Solution manuals are available to instructors. This and other related and timely material is available via internet web sites.

At the end of most chapters is a section entitled ``Bibliographical and Historical Remarks" wherein we credit the sources of the ideas presented in that chapter, provide pointers to further reading and ongoing research, and describe relevant historical background. Despite our attempts to make these sections authoritative and complete, we have undoubtedly left out some important prior work. For that we apologize and welcome corrections and extensions for incorporation into a later edition.

In some sense we have been working towards this book for twenty years, and we have lots of people to thank. First, we thank those who have personally helped us develop the overall view presented in this book: Harry Klopf for helping us recognize that reinforcement learning needed to be revived; Chris Watkins, Dimitri Bertsekas, John Tsitsiklis, and Paul Werbos for helping us see the value of the relationships to dynamic programming; John Moore and Jim Kehoe for insights and inspirations from animal learning theory; Oliver Selfridge for emphasizing the breadth and importance of adaptation; and, more generally, our colleagues and students who have contributed in countless ways: Ron Williams, Charles Anderson, Satinder Singh, Steve Bradtke, Bob Crites, Peter Dayan, and Leemon Baird. Our view of reinforcement learning has also been significantly enriched by discussions with Paul Cohen, Paul Utgoff, Martha Steenstrup, Gerry Tesauro, Mike Jordan, Leslie Kaelbling, Andrew Moore, Chris Atkeson, Tom Mitchell, Nils Nilsson, Tom Dean, and Bob Narendra. We thank the the Air Force Office of Scientific Research, the National Science Foundation, and GTE Laboratories for their long and far-sighted support.

We also wish to thank the many people who have read drafts of this book and provided valuable comments, including: Tom Kalt, John Tsitsiklis, Pawel Cichosz, Olle Gällmo, Chuck Anderson, Stuart Russell, Paul Cohen, Jette Randlov, Brian Sheppard, Thomas

O'Connell, Richard Coggins, Cristina Versino, John H. Hiett, Andreas Badelt, Jay Ponte, Joe Beck, Justus Piater, Martha Steenstrup, Satinder Singh, Tommi Jaakkola, Dimitri Bertsekas, Ben Van Roy, Torbjörn Ekman, Christina Björkman, Jakob Carlström, and Olle Palmgren. Finally, we thank Gwyn Mitchell for helping in many ways, and Harry Stanton and Bob Prior for being our champions at MIT Press.

Next Up Previous

Next: Chapter 1 **Up:** Contents **Previous:** Contents

Richard Sutton
Fri May 30 21:54:08 EDT 1997

1 Introduction

The idea that we learn by interacting with our environment is probably the first to occur to us when we think about the nature of learning. When an infant plays, waves its arms, or looks about, it has no explicit teacher, but it does have a direct sensori-motor connection to its environment. Exercising this connection produces a wealth of information about cause and effect, about the consequences of actions, and about what to do in order to achieve goals. Throughout our lives, such interactions are undoubtedly a major source of knowledge about our environment and ourselves. Whether we are learning to drive a car or to hold a conversation, we are all acutely aware of how our environment responds to what we do, and we seek to influence what happens through our behavior. Learning from interaction is a foundational idea underlying nearly all theories of learning and intelligence.

In this book we explore a *computational* approach to learning from interaction. Rather than directly theorizing about how people or animals learn, we explore idealized learning situations and evaluate the effectiveness of various learning methods. That is, we adopt the perspective of an artificial intelligence researcher or engineer. We explore designs for machines that are effective in solving learning problems of scientific or economic interest, evaluating the designs through mathematical analysis or computational experiments. The approach we explore, called *reinforcement learning*, is much more focused on goal-directed learning from interaction than are other approaches to machine learning.

Richard Sutton
Sat May 31 14:27:51 EDT 1997

1.1 Reinforcement Learning

Reinforcement learning is learning what to do---how to map situations to actions---so as to maximize a numerical reward signal. The learner is not told which actions to take, as in most forms of machine learning, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward, but also the next situation and, through that, all subsequent rewards. These two characteristics---trial-and-error search and delayed reward---are the two most important distinguishing features of reinforcement learning.

Reinforcement learning is defined not by characterizing learning algorithms, but by characterizing a learning *problem*. Any algorithm that is well suited to solving that problem we consider to be a reinforcement learning algorithm. A full specification of the reinforcement learning problem in terms of optimal control of Markov decision processes must wait until Chapter 3, but the basic idea is simply to capture the most important aspects of the real problem facing a learning agent interacting with its environment to achieve a goal. Clearly such an agent must be able to sense the state of the environment to some extent and must be able ato take actions that affect that state. The agent must also have a goal or goals relating to the state of the environment. Our formulation is intended to include just these three aspects---sensation, action, and goal---in the simplest possible form without trivializing any of them.

Reinforcement learning is different from supervised learning, the kind of learning studied in most current research in machine learning, statistical pattern recognition, and artificial neural networks. Supervised learning is learning from examples provided by some knowledgable external supervisor. This is an important kind of learning, but alone it is not adequate for learning from interaction. In interactive problems it is often impractical to obtain examples of desired behavior that are both correct and representative of all the situations in which the agent has to act. In uncharted territory---where one would expect learning to be most beneficial---an agent must be able to learn from its own experience.

One of the challenges that arises in reinforcement learning and not in other kinds of learning is the tradeoff between exploration and exploitation. To obtain a lot of reward, a reinforcement learning agent must prefer actions that it has tried in the past and found to be effective in producing reward. But to discover such actions it has to try actions that it has not selected before. The agent has to *exploit* what it already knows in order to obtain reward, but it also has to *explore* in order to make better action selections in the future. The dilemma is that neither exploitation nor exploration can be pursued exclusively without failing at the task. The agent must try a variety of actions *and* progressively favor those that appear to be best. On a stochastic task, each action must be tried many times to reliably estimate its expected reward. The exploration--exploitation dilemma has been intensively studied by mathematicians for many decades (see Chapter 2). For now we simply note that the entire issue of balancing exploitation and exploration does not even arise in supervised learning as it is usually defined.

Another key feature of reinforcement learning is that it explicitly considers the *whole* problem of a goal-directed agent interacting with an uncertain environment. This is in contrast with many approaches that address subproblems without addressing how they might fit into a larger picture. For example, we have mentioned that much of machine learning research is concerned with supervised learning without explicitly specifying how such an ability would finally be useful. Other researchers have developed theories of planning with general goals, but without considering planning's role in real-time decision-making, or the question of where the predictive models necessary for planning would come from. Although these approaches have yielded many useful results, their focus on isolated subproblems is a significant limitation.

Reinforcement learning takes the opposite tack, by starting with a complete, interactive, goal-seeking agent. All reinforcement learning agents have explicit goals, can sense aspects of their environments, and can choose actions to influence their environments. Moreover, it is usually assumed from the beginning that the agent has to operate despite significant uncertainty about the environment it faces. When reinforcement learning involves planning, it has to address the interplay between planning and real-time action selection, as well as the question of how environmental models are acquired and improved. When reinforcement learning involves supervised learning, it does so for very specific reasons that determine which capabilities are critical, and which are not. For learning research to make progress, important subproblems have to be isolated and studied, but they should be subproblems that are motivated by clear roles in complete, interactive, goal-seeking agents, even if all the details of the complete agent cannot yet be filled in.

One of the larger trends of which reinforcement learning is a part is that towards greater contact between artificial intelligence and other engineering disciplines. Not all that long ago, artificial intelligence was viewed as almost entirely separate from control theory and statistics. It had to do with logic and symbols, not numbers. Artificial intelligence was large LISP programs, not linear algebra, differential equations, or statistics. Over the last decades this view has gradually eroded. Modern artificial intelligence researchers accept statistical and control-theory algorithms, for example, as relevant competing methods or simply as tools of their trade. The previously ignored areas lying between artificial intelligence and conventional engineering are now among the most active of all, including new fields such as neural networks, intelligent control, and our topic, reinforcement learning. In reinforcement learning we extend ideas from optimal control theory and stochastic approximation to address the broader and more ambitious goals of artificial intelligence.

Next Up Previous
Next: 1.2 Examples **Up:** 1 Introduction **Previous:** 1 Introduction

---

Richard Sutton
Sat May 31 14:27:51 EDT 1997

1.2 Examples

A good way to understand reinforcement learning is to consider some of the examples and possible applications that have guided its development:

- A master chess player makes a move. The choice is informed both by planning---anticipating possible replies and counter-replies---and by immediate, intuitive judgments of the desirability of particular positions and moves.

- An adaptive controller adjusts parameters of a petroleum refinery's operation in real time. The controller optimizes the yield/cost/quality tradeoff based on specified marginal costs without sticking strictly to the set points originally suggested by human engineers.

- A gazelle calf struggles to its feet minutes after being born. Half an hour later it is running at 30 miles per hour.

- A mobile robot decides whether it should enter a new room in search of more trash to collect or start trying to find its way back to its battery recharging station. It makes its decision based on how quickly and easily it has been able to find the recharger in the past.

- Phil prepares his breakfast. When closely examined, even this apparently mundane activity reveals itself as a complex web of conditional behavior and interlocking goal-subgoal relationships: walking to the cupboard, opening it, selecting a cereal box, then reaching for, grasping, and retrieving the box. Other complex, tuned, interactive sequences of behavior are required to obtain a bowl, spoon, and milk jug. Each step involves a series of eye movements to obtain information and to guide reaching and locomotion. Rapid judgments are continually made about how to carry the objects or whether it is better to ferry some of them to the dining table before obtaining others. Each step is guided by goals, such as grasping a spoon, or getting to the refrigerator, and is in service of other goals, such as having the spoon to eat with once the cereal is prepared and of ultimately obtaining nourishment.

These examples share features that are so basic that they are easy to overlook. All involve *interaction* between an active decision-making agent and its environment, within which the agent seeks to achieve a *goal* despite *uncertainty* about its the environment. The agent's actions are permitted to affect the future state of the environment (e.g., the next chess position, the level of reservoirs of the refinery, the next location of the robot), thereby affecting the options and opportunities available to the agent at later times. Correct choice requires taking into account indirect, delayed consequences of actions, and thus may require foresight or planning.

At the same time, in all these examples the effects of actions cannot be fully predicted, and so the agent must frequently monitor its environment and react appropriately. For example, Phil must watch the milk he pours into his cereal bowl to keep it from overflowing. All these examples involve goals that are explicit in the sense that the agent can judge progress toward its goal on the basis of what it can directly sense. The chess player knows whether or not he wins, the refinery controller knows how much petroleum is being produced, the mobile robot knows when its batteries run down, and Phil knows whether or not he is enjoying his breakfast.

In all of these examples the agent can use its experience to improve its performance over time. The chess player refines the intuition he uses to evaluate positions, thereby improving his play; the gazelle calf improves the efficiency with which it can run; Phil learns to streamline his breakfast making. The knowledge the agent brings to the task at the start---either from previous experience with related tasks or built into it by design or evolution---influences what is useful or easy to learn, but interaction with the environment is essential for adjusting behavior to exploit specific features of the task.

Richard Sutton
Sat May 31 14:27:51 EDT 1997

1.3 Elements of Reinforcement Learning

Beyond the agent and the environment, one can identify four main sub-elements to a reinforcement learning system: a *policy*, a *reward function*, a *value function*, and, optionally, a *model* of the environment.

A *policy* defines the learning agent's way of behaving at a given time. Roughly speaking, a policy is a mapping from perceived states of the environment to actions to be taken when in those states. It corresponds to what in psychology would be called a set of stimulus-response rules or associations. In some cases the policy may be a simple function or lookup table, whereas in others it may involve extensive computation such as a search process. The policy is the core of a reinforcement learning agent in the sense that it alone is sufficient to determine behavior. In general, policies may be stochastic.

A *reward function* defines the goal in a reinforcement learning problem. Roughly speaking, it maps perceived states (or state-action pairs) of the environment to a single number, a *reward*, indicating the intrinsic desirability of the state. A reinforcement-learning agent's sole objective is to maximize the total reward it receives in the long run. The reward function defines what are the good and bad events for the agent. In a biological system, it would not be inappropriate to identify rewards with pleasure and pain. They are the immediate and defining features of the problem faced by the agent. As such, the reward function must necessarily be fixed. It may, however, be used as a basis for changing the policy. For example, if an action selected by the policy is followed by low reward then the policy may be changed to select some other action in that situation in the future. In general, reward functions may also be stochastic.

Whereas a reward function indicates what is good in an immediate sense, a *value function* specifies what is good in the long run. Roughly speaking, the *value* of a state is the total amount of reward an agent can expect to accumulate over the future starting from that state. Whereas rewards determine the immediate, intrinsic desirability of environmental states, values indicate the *long-term* desirability of states after taking into account the states that are likely to follow, and the rewards available in those states. For example, a state might always yield a low immediate reward, but still have a high value because it is regularly followed by other states that yield high rewards. Or the reverse could be true. To make a human analogy, rewards are like pleasure (if high) and pain (if low), whereas values correspond to a more refined and far-sighted judgment of how pleased or displeased we are that our environment is in a particular state. Expressed this way, we hope it is clear that value functions formalize a very basic and familiar idea.

Rewards are in a sense primary, whereas values, as predictions of rewards, are secondary. Without rewards there could be no values, and the only purpose of estimating values is to achieve more reward. Nevertheless, it is values with which we are most concerned when making and evaluating decisions. Action choices are made on the basis of value judgments. We seek actions that bring about states of highest value, not highest reward, because these actions obtain for us the greatest amount of reward over the long run. In decision-making and planning, the derived quantity called value is the one with which we are most concerned. Unfortunately, it is also much harder to determine values than it is to determine rewards. Rewards are basically given directly by the environment, but values must be estimated and re-estimated from the sequences of observations an agent makes over its entire lifetime. In fact, the most important component of almost all reinforcement learning algorithms is a method for efficiently estimating values. The importance and centrality of estimating values is perhaps the most important thing we have learned about reinforcement learning in the last two decades.

Although all the reinforcement learning methods we consider in this book are structured around estimating value functions, it is not strictly necessary to do this to solve reinforcement learning problems. For example, search methods such as genetic algorithms, genetic programming, simulated annealing, and other function optimization methods, have been used to solve reinforcement learning problems. These methods search directly in the space of policies without ever appealing to value functions. We call these *evolutionary* methods because their operation is analogous to how biological evolution produces organisms with skilled behavior even though they do not themselves learn during their individual lifetimes. If the space of policies is sufficiently small, or can be structured so that good policies are common or easy to find, then evolutionary methods are often effective. In addition, evolutionary methods have advantages on problems in which the learning agent cannot accurately sense the state of its environment.

Nevertheless, what we mean by reinforcement learning involves learning while interacting with the environment, which evolutionary methods do not do. It is our belief that methods able to take advantage of the details of individual behavioral interactions can be much more efficient than evolutionary methods in a great many cases. Evolutionary methods ignore much of the useful structure of the reinforcement learning problem: they do not use the fact that the policy they are searching for is a function from states to actions; they do not notice which states an individual passes through during its lifetime, or which actions it selects. In some cases this information can be misleading (e.g., when states are misperceived), but more often it should enable more efficient search. Although evolution and learning share many features and can naturally work together as they do in nature, we do not consider evolutionary methods by themselves to be especially well-suited to reinforcement learning problems. For simplicity, in this book when we use the term ``reinforcement learning'' we do not include evolutionary methods.

The fourth and final element of some reinforcement learning systems is a *model* of the environment. This is something that mimics the behavior of the environment. For example, given a state and action, the model might predict the resultant next state and next reward. Models are used for *planning*, by which we mean any way of deciding on a course of action by considering possible future situations before they are actually experienced. The incorporation of models and planning into reinforcement learning systems is a relatively new development. Early reinforcement learning systems were explicitly trial-and-error learners; what they did was viewed as almost the *opposite* of planning. Nevertheless, it gradually became clear that reinforcement learning methods are closely related to dynamic programming methods, which do use models, and that they in turn are closely related to state-space planning methods. In Chapter 9 we explore reinforcement learning systems that simultaneously learn by trial and error, learn a model of the environment, and use the model for planning. Modern reinforcement learning spans the spectrum from low-level, trial-and-error learning to high-level, deliberative planning.

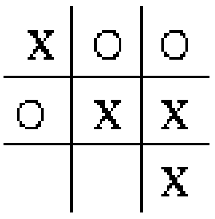
Next Up Previous
Next: 1.4 An Extended Example: **Up:** 1 Introduction **Previous:** 1.2 Examples

Richard Sutton
Sat May 31 14:27:51 EDT 1997

1.4 An Extended Example: Tic-Tac-Toe

To illustrate the general idea of reinforcement learning and contrast it with other approaches, we next consider a single example in more detail.

Consider the familiar child's game of Tic-Tac-Toe. Two players take turns playing on a three-by-three board. One player plays X s and the other Ø s until one player wins by placing three of his marks in a row, horizontally, vertically, or diagonally, as the `X ' player has in this game:



If the board fills up with neither player getting three in a row, the game is a draw. Because a skilled player can play so that he never loses, let us assume that we are playing against an imperfect player, one whose play is sometimes incorrect and allows us to win. For the moment, in fact, let us consider draws and losses to be equally bad for us. How might we construct a player that will find the imperfections in its opponent's play and learn to maximize its chances of winning?

Although this is a very simple problem, it cannot readily be solved in a satisfactory way by classical techniques. For example, the classical ``minimax" solution from game theory is not correct here because it assumes a particular way of playing by the opponent. For example, a minimax player would never reach a game state from which it could lose, even if in fact it always won from that state because of incorrect play by the opponent. Classical optimization methods for sequential decision problems, such as dynamic programming, can *compute* an optimal solution for any opponent, but require as input a complete specification of that opponent, including the probabilities with which the opponent makes each move in each board state. Let us assume that this information is not available a priori for this problem, as it is not for the vast majority of problems of practical interest. On the other hand, such information can be estimated from experience, in this case by playing many games against the opponent. About the best one can do on this problem is to first learn a model of the opponent's behavior, up to some level of confidence, and then apply dynamic programming to compute an optimal solution given the approximate opponent model. In the end, this is not that different from some of the reinforcement learning methods we examine later in this book.

An evolutionary approach to this problem would directly search the space of possible policies for one with a high probability of winning against the opponent. Here, a policy is a rule that tells the player what move to make for every state of the game---every possible configuration of X s and Ø s on the three-by-three board. For each policy considered, an estimate of its winning probability would be obtained by playing some number of games against the opponent. This evaluation would then direct which policy or policies were next considered. A typical evolutionary method would hillclimb in policy space, successively generating and evaluating policies in an attempt to obtain incremental improvements. Or, perhaps, a genetic-style algorithm could be used which would maintain and evaluate a population of policies. Literally hundreds of different optimization algorithms could be applied. By *directly* searching the policy space we mean that *entire policies* are proposed and compared on the basis of scalar evaluations.

Here is how the Tic-Tac-Toe problem would be approached using reinforcement learning and approximate value functions. First we set up a table of numbers, one for each possible state of the game. Each number will be the latest estimate of the probability of our winning from that state. We treat this estimate as the state's current *value*, and the whole table is the learned value function. State A has higher value than state B, or is considered ``better'' than state B, if the current estimate of the probability of our winning from A is higher than it is from B. Assuming we always play X s, then for all states with three X s in a row the probability of winning is 1, because we have already won. Similarly, for all states with three Ø s in a row, or that are ``filled up'', the correct probability is 0, as we cannot win from them. We set the initial values of all the other states, the nonterminals, to 0.5, representing an informed guess that we have a 50% chance of winning.

Now we play many games against the opponent. To select our moves we examine the states that would result from each of our possible moves (one for each blank space on the board) and look up their current values in the table. Most of the time we move *greedily*, selecting the move that leads to the state with greatest value, that is, with the highest estimated probability of winning. Occasionally, however, we select randomly from one of the other moves instead; these are called *exploratory* moves because they cause us to experience states that we might otherwise never see. A sequence of moves made and considered during a game can be diagrammed as in Figure 1.1.

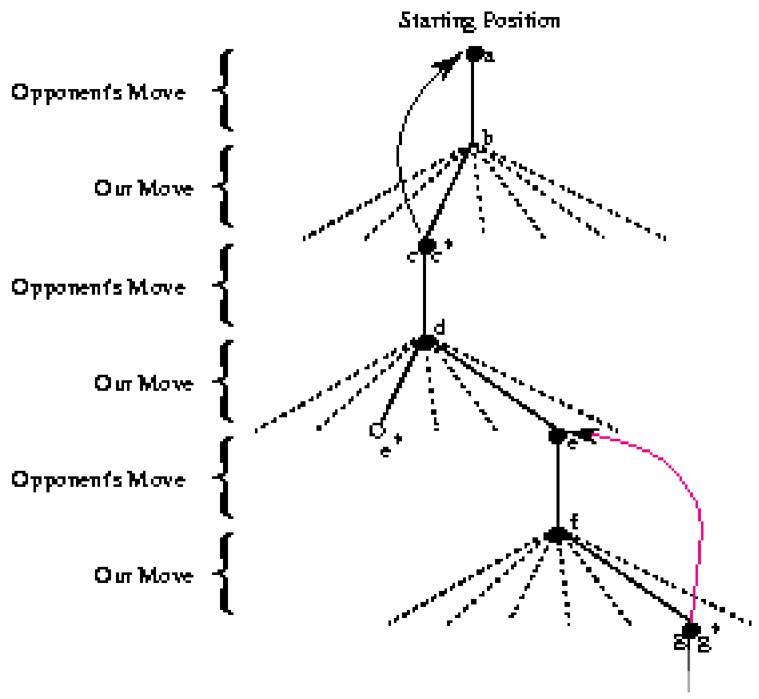


Figure 1.1: A sequence of Tic-Tac-Toe moves. The solid lines represent the moves taken during a game; the dashed lines represent moves that we (our algorithm) considered but did not make. Our second move was an exploratory move, meaning that is was taken even though another sibling move, that leading to e^*, was ranked higher. Exploratory moves do not result in any learning, but each of our other moves do, causing *backups* as suggested by the curved arrows and detailed in the text.

While we are playing, we change the values of the states in which we find ourselves during the game. We attempt to make them more accurate estimates of the probabilities of winning from those states. To do this, we ``back up'' the value of the state after each greedy move to the state before the move, as suggested by the arrows in Figure 1.1. More precisely, the current value of the earlier state is adjusted to be closer to the value of the later state. This can be done by moving the earlier state's value a fraction of the way toward the value of the later state. If we let **s** denote the state before the greedy move, and s' the state after, then the update to the estimated value of **s**, denoted $V(s)$, can be written:

$$V(s) \leftarrow V(s) + \alpha[V(s') - V(s)],$$

where α is a small positive fraction called the *step-size parameter*, which influences the rate of learning. This update rule is an example of a *temporal-difference* learning method, so called because it's changes are based on a difference, $V(s') - V(s)$, between estimates at two different times.

The method described above performs quite well on this task. For example, if the step-size parameter is reduced properly over time, this method converges, for any fixed opponent, to the true probabilities of winning from each state given optimal play by our player. Furthermore, the moves then taken (except on exploratory moves) are in fact the optimal moves against the opponent. In other words, the method converges to an optimal policy for playing the game. If the step-size parameter is not reduced all the way to zero over time, then this player also plays well against opponents that change their way of playing slowly over time.

This example illustrates the differences between evolutionary methods and methods that learn value functions. To evaluate a policy, an evolutionary method must hold it fixed and play many games against the opponent, or simulate many games using a model of the opponent. The frequency of wins gives an unbiased estimate of the probability of winning with that policy, and can be used to direct the next policy selection. But each policy change is made only after many games, and only the final outcome of each game is used: what happens *during* the games is ignored. For example, if the player wins, then *all* of its behavior in the game is given credit, independently of how specific moves might have been critical to the win. Credit is even given to moves that never occurred! Value function methods, in contrast, allow individual states to be evaluated. In the end, both evolutionary and value function methods search the space of policies, but learning a value function takes advantage of information available during the course of play.

This example is very simple, but it illustrates some of the key features of reinforcement learning methods. First, there is the emphasis on learning while interacting with an environment, in this case with an opponent player. Second, there is a clear goal, and correct behavior requires planning or foresight that takes into account delayed effects of one's choices. For example, the simple reinforcement learning player would learn to set up multi-move traps for a short-sighted opponent. It is a striking feature of the reinforcement learning solution that it can achieve the effects of planning and lookahead without using a model of the opponent and without carrying out an explicit search over possible sequences of future states and actions.

While this example illustrates some of the key features of reinforcement learning, it is so simple that it might give the impression that reinforcement learning is more limited than it really is. Although Tic-Tac-Toe is a two-person game, reinforcement learning also applies when there is no explicit external adversary, that is, in the case of a ``game against nature.'' \ Reinforcement learning is also not restricted to problems in which behavior breaks down into separate episodes, like the separate games of Tic-Tac-Toe, with reward only at the end of each episode. It is just as applicable when behavior continues indefinitely and when rewards of various magnitudes can be received at any time.

Tic-Tac-Toe has a relatively small, finite state set, whereas reinforcement learning can be applied to very large, or even infinite, state sets. For example, Gerry Tesauro (1992, 1995) combined the algorithm described above with an artificial neural network to learn to play the game of backgammon, which has approximately 10^{20} states. Notice that with this many states it is impossible to ever experience more than a small fraction of them. Tesauro's program learned to play far better than any previous program, and now plays at the level of the world's best human players (see Chapter 11). The neural network provides the program with the ability to generalize from its experience, so that in new states it selects moves based on information saved from similar states faced in the past, as determined by its network. How well a reinforcement learning agent can work in problems with such large state sets is intimately tied to how appropriately it can generalize from past experience. It is in this role that we have the greatest need for supervised learning methods with reinforcement learning. Neural networks are not the only, or necessarily the best, way to do this.

In this Tic-Tac-Toe example, learning started with no prior knowledge beyond the rules of the game, but reinforcement learning by no means entails a *tabula rasa* view of learning and intelligence. On the contrary, prior information can be incorporated into reinforcement learning in a variety of ways that can be critical for efficient learning. We also had access to the true state in the Tic-Tac-Toe example, whereas reinforcement learning can also be applied when part of the state is hidden, or when different states appear to the learner to be the same. That case, however, is substantially more difficult, and we do not cover it significantly in this book.

Finally, the Tic-Tac-Toe player was able to look ahead and know the states that would result from each of its possible moves. To do this, it had to have a model of the game that allows it to ``think about'' how its environment would change in response to moves that it may never make. Many problems are like this, but in others even a short-term model of the effects of actions is lacking. Reinforcement learning can be applied in either case. No model is required, but models can easily be used if they are available or can be learned.

Exercise 1.1

Self Play. Suppose, instead of playing against a random opponent, the reinforcement learning algorithm described above played against itself. What do you think would happen in this case? Would it learn a different way of playing?

Exercise 1.2

Symmetries. Many Tic-Tac-Toe positions appear different but are really the same because of symmetries. How might we amend the reinforcement learning algorithm described above to take advantage of this? In what ways would this improve it? Now think again, suppose the opponent did not take advantage of symmetries. In that case, should we? It is true then that symmetrically equivalent positions should necessarily have the same value?

Exercise 1.3 *Greedy Play*. Suppose the reinforcement learning player was *greedy*, that is, it always played the move that brought it to the position that it rated the best. Would it learn to play better, or worse, than a non-greedy player? What problems might occur?

Exercise 1.4

Learning from Exploration. Suppose learning updates occurred after *all* moves, including exploratory moves. If the step-size parameter is reduced over time appropriately, then the state values would converge to a set of probabilities. What are the two sets of probabilities computed when we do, and when we do not, learn from exploratory moves? Assuming that we do continue to make exploratory moves, which set of probabilities might be better to learn? Which would result in more wins?

Exercise 1.5

Other Improvements. Can you think of other ways to improve the reinforcement learning player? Can you think of any better way to solve the Tic-Tac-Toe problem as posed?

Next Up Previous
Next: 1.5 Summary **Up:** 1 Introduction **Previous:** 1.3 Elements of Reinforcement

---

Richard Sutton
Sat May 31 14:27:51 EDT 1997

1.5 Summary

Reinforcement learning is a computational approach to understanding and automating goal-directed learning and decision-making. It is distinguished from other computational approaches by its emphasis on learning by the individual from direct interaction with its environment, without relying on exemplary supervision or complete models of the environment. In our opinion, reinforcement learning is the first field to seriously address the computational issues that arise when learning from interaction with an environment in order to achieve long-term goals.

Reinforcement learning uses a formal framework defining the interaction between agent and environment in terms of states, actions, and rewards. This framework is intended to be a simple way of representing essential features of the artificial intelligence problem. These features include a sense of cause and effect, of uncertainty and nondeterminism, and the existence of explicit goals. Most relevant is the formalism of Markov decision processes, which provides a precise, and relatively neutral, way of including the key features.

The concepts of value and value functions are the key features of the reinforcement learning methods that we consider in this book. We take the position that value functions are essential for efficient search in the space of policies. Their use of value functions distinguish reinforcement learning methods from evolutionary methods that search directly in policy space guided by scalar evaluations of entire policies.

1.6 History of Reinforcement Learning

The history of reinforcement learning has two main threads, both long and rich, which were pursued independently before intertwining in modern reinforcement learning. One thread concerns learning by trial and error and started in the psychology of animal learning. This thread runs through some of the earliest work in artificial intelligence and led to the revival of reinforcement learning in the early 1980s. The other thread concerns the problem of optimal control and its solution using value functions and dynamic programming. For the most part, this thread did not involve learning. Although the two threads were largely independent, the exceptions revolve around a third, less distinct thread concerning temporal-difference methods such as used in the Tic-Tac-Toe example in this chapter. All three threads came together in the late 1980s to produce the modern field of reinforcement learning as we present it in this book.

The thread focusing on trial-and-error learning is the one with which we are most familiar and about which we have the most to say in this brief history. Before doing that, however, we briefly discuss the optimal control thread.

The term ``optimal control" came into use in the late 1950s to describe the problem of designing a controller to minimize a measure of a dynamical system's behavior over time. One of the approaches to this problem was developed in the mid-1950s by Richard Bellman and colleagues by extending a 19th century theory of Hamilton and Jacobi. This approach uses the concept of a dynamical system's state and of a value function, or ``optimal return function," to define a functional equation, now often called the Bellman equation. The class of methods for solving optimal control problems by solving this equation came to be known as dynamic programming (Bellman, 1957a). Bellman (1957b) also introduced the discrete stochastic version of the optimal control problem known as Markovian decision processes (MDPs), and

Ron Howard (1960) devised the policy iteration method for MDPs. All of these are essential elements underlying the theory and algorithms of modern reinforcement learning.

Dynamic programming is widely considered the only feasible way of solving general stochastic optimal control problems. It suffers from what Bellman called ``the curse of dimensionality," meaning that its computational requirements grow exponentially with the number of state variables, but it is still far more efficient and more widely applicable than any other method. Dynamic programming has been extensively developed in the last four decades, including extensions to partially observable MDPs (surveyed by Lovejoy, 1991), many applications (surveyed by White, 1985, 1988, 1993), approximation methods (surveyed by Rust, 1996), and asynchronous methods (Bertsekas, 1982, 1983). Many excellent modern treatments of dynamic programming are available (e.g., Bertsekas, 1995; Puterman, 1994; Ross, 1983; and Whittle, 1982, 1983). Bryson (1996) provides a detailed authoritative history of optimal control.

In this book, we consider all of the work on optimal control to also be work in reinforcement learning. We define reinforcement learning as any effective way of solving reinforcement learning problems, and it is now clear that these problems are very closely related to optimal control problems, particularly those formulated as MDPs. Accordingly we must consider the solution methods of optimal control, such as dynamic programming, to also be reinforcement learning methods. Of course, almost all of these methods require complete knowledge of the system to be controlled, and for this reason it feels a little unnatural to say that they are part of reinforcement *learning*. On the other hand, many dynamic programming methods are incremental and iterative. Like true learning methods, they gradually reach the correct answer through successive approximations. As we show in the rest of this book, these similarities are far more than superficial. The theories and solution methods for the cases of complete and incomplete knowledge are so closely related that we feel they must be considered together as part of the same subject matter.

Let us return now to the other major thread leading to the modern field of reinforcement learning, that centered around the idea of trial-and-error learning. This thread began in psychology, where ``reinforcement" theories of learning are common. Perhaps the first to succintly express the essence of trial-and-error learning was Edward Thorndike. We take this essence to be the idea that actions followed by good or bad outcomes have their tendency to be re-selected altered accordingly. In Thorndike's words:

> Of several responses made to the same situation, those which are accompanied or closely followed by satisfaction to the animal will, other things being equal, be more firmly connected with the situation, so that, when it recurs, they will be more likely to recur; those which are accompanied or closely followed by discomfort to the animal will, other things being equal, have their connections with that situation weakened, so that, when it recurs, they will be less likely to occur. The greater the satisfaction or discomfort, the greater the strengthening or weakening of the bond. (Thorndike, 1911, p. 244)

Thorndike called this the ``Law of Effect'' because it describes the effect of reinforcing events on the tendency to select actions. Although sometimes controversial (e.g., see Kimble, 1961, 1967; Mazur, 1994) the Law of Effect is widely regarded as an obvious basic principle underlying much behavior (e.g., Hilgard and Bower, 1975; Dennett, 1978; Campbell, 1958; Cziko, 1995).

The Law of Effect includes the two most important aspects of what we mean by trial-and-error learning. First, it is *selectional* rather than instructional in the sense of Monod, meaning that it involves trying alternatives and selecting among them by comparing their consequences. Second, it is *associative*, meaning that the alternatives found by selection are associated with particular situations. Natural selection in evolution is a prime example of a selectional process, but it is not associative. Supervised learning is associative, but not selectional. It is the combination of these two that is essential to the Law of Effect and to trial-and-error learning. Another way of saying this is that the Law of Effect is an elementary way of combining *search* and *memory*: search in the form of trying and selecting among many actions in each situation, and memory in the form of remembering what actions worked best, associating them with the situations in which they were best. Combining search and memory in this way is essential to reinforcement learning.

In early artificial intelligence, before it was distinct from other branches of engineering, several researchers began to explore trial-and-error learning as an engineering principle. The earliest computational investigations of trial-and-error learning were perhaps by Minsky and Farley and Clark, both in 1954. In his Ph.D. dissertation, Minsky discussed computational models of reinforcement learning and described his construction of an analog machine, composed of components he called SNARCs (Stochastic Neural-Analog Reinforcement Calculators). Farley and Clark described another neural-network learning machine designed to learn by trial-and-error. In the 1960s one finds the terms ``reinforcement'' and ``reinforcement learning'' being widely used in the engineering literature for the first time (e.g., Waltz and Fu, 1965; Mendel, 1966; Fu, 1970; Mendel and McClaren, 1970). Particularly influential was Minsky's paper ``Steps Toward Artificial Intelligence'' (Minsky, 1961), which discussed several issues relevant to reinforcment learning, including what he called the *credit-assignment problem*: how do you distribute credit for success among the many decisions that may have been involved in producing it? All of the methods we discuss in this book are, in a sense, directed toward solving this problem.

The interests of Farley and Clark (1954; Clark and Farley, 1955) shifted from trial-and-error learning to generalization and pattern recognition, that is, from reinforcement learning to supervised learning. This began a pattern of confusion about the relationship between these types of learning. Many researchers seemed to believe that they were studying reinforcement learning when they were actually studying supervised learning. For example, neural-network pioneers such as Rosenblatt (1958) and Widrow and Hoff (1960) were clearly motivated by reinforcement learning---they used the language of rewards and punishments---but the systems they studied were supervised learning systems suitable for pattern recognition and perceptual learning. Even today, reseachers and textbooks often minimize or blur the distinction between these types of learning. Some modern neural-network textbooks use the term trial-and-error to describe networks that learn from training examples because they use error information to update connection weights. This is an understandable confusion, but it substantually misses the essential selectional character of trial-and-error learning.

Partly as a result of these confusions, research into genuine trial-and-error learning became rare in the the 1960s and 1970s. In the next few paragraphs we discuss some of the exceptions and partial exceptions to this trend.

One of these was the work by a New Zealand researcher named John Andreae. Andreae (1963) developed a system called STeLLA that learned by trial and error in interaction with its environment. This system included an internal model of the world and, later, an ``internal monologue'' to deal with problems of hidden state (Andreae, 1969a). Andreae's later work placed more emphasis on learning from a teacher, but still included trial-and error (Andreae, 1977). Unfortunately, Andreae's pioneering research was not well-known, and did not greatly impact subsequent reinforcement learning research.

More influential was the work of Donald Michie. In 1961 and 1963 he described a simple trial-and-error learning system for learning how to play Tic-Tac-Toe (or Noughts and Crosses) called MENACE (for Matchbox Educable Noughts and Crosses Engine). It consisted of a matchbox for each possible game position, each containing a number of colored beads, a color for each possible move from that position. By drawing a bead at random from the matchbox corresponding to the current game position, one could determine MENACE's move. When a game was over, beads were added or removed from the boxes used during play to reinforce or punish MENACE's decisions. Michie and Chambers (1968) described another Tic-Tac-Toe reinforcement learner called GLEE (Game Learning Expectimaxing Engine) and a reinforcement learning controller called BOXES. They applied BOXES to the task of learning to balance a pole hinged to a movable cart on the basis of a failure signal occuring only when the pole fell or the cart reached the end of a track. This task was adapted from the earlier work of Widrow and Smith (1964), who used supervised learning methods, assuming instruction from a teacher already able to balance the pole. Michie and Chamber's version of pole-balancing is one of the best early examples of a reinforcement learning task under conditions of incomplete knowledge. It influenced much later work in reinforcement learning, beginning with some of our own studies (Barto, Sutton and Anderson, 1983; Sutton, 1984). Michie has consistently emphasized the role of trial-and-error and learning as essential aspects of artificial intelligence (Michie, 1974).

Widrow, Gupta, and Maitra (1973) modified the LMS algorithm of Widrow and Hoff (1960) to produce a reinforcement learning rule that could learn from success and failure signals instead of from training examples. They called this form of learning ``selective bootstrap adaptation" and described it as ``learning with a critic" instead of ``learning with a teacher." They analyzed this rule and showed how it could learn to play blackjack. This was an isolated foray into reinforcement learning by Widrow, whose contributions to supervised learning were much more influential.

Research on *learning automata* had a more direct influence on the trial-and-error thread of modern reinforcement learning research. These were methods for solving a non-associative, purely selectional learning problem known as the ***n*** *-armed bandit* by anology to a slot-machine, or ``one-armed bandit," except with **n** levers (see Chapter 2). Learning automata were simple, low-memory machines for solving this problem. Tsetlin (1973) introduced these machines in the west, and excellent surveys are provided by Narendra and Thatachar (1974, 1989). Barto and Anadan (1985) extended these methods to the associative case.

John Holland (1975) outlined a general theory of adaptive systems based on selectional principles. His early work concerned trial and error primarily in its non-associative form, as in evolutionary methods and the **n** -armed bandit. In 1986 he introduced *classifier systems*, true reinforcement-learning systems including association and value functions. A key component of Holland's classifer systems was always a *genetic algorithm*, an evolutionary method whose role was to evolve useful representations. Classifier systems have been extensively developed by many researchers to form a major branch of reinforcement learning research (e.g., see text by Goldberg, 1989; and Wilson, 1994), but genetic algorithms---which by themselves are not reinforcement learning systems---have received much more attention.

The individual most responsible for reviving the trial-and-error thread to reinforcement learning within artificial intelligence is Harry Klopf (1972, 1975, 1982). Klopf recognized that essential aspects of adaptive behavior were being lost as learning researchers came to focus almost exclusively on supervised learning. What was missing, according to Klopf, were the hedonic aspects of behavior, the drive to achieve some result from the environment, to control the environment toward desired ends and away from undesired ends. This is the essential idea of reinforcement learning. Klopf's ideas were especially influential on the authors because our assessment of them (Barto and Sutton, 1981b) led to our appreciation of the distinction between supervised and reinforcement learning, and to our eventual focus on reinforcement learning. Much of the early work that we and colleagues accomplished was directed toward showing that reinforcement learning and supervised learning were indeed very different (Barto, Sutton, and Brouwer, 1981; Barto and Sutton, 1981a; Barto and Anandan, 1985). Other studies showed how reinforcement learning could address important problems in neural-network learning, in particular, how it could produce learning algorithms for multi-layer networks (Barto, Anderson, and Sutton, 1982; Barto and Anderson, 1985; Barto and Anandan, 1985; Barto, 1985; Barto, 1986; Barto and Jordan, 1986).

We turn now to the third thread to the history of reinforcement learning, that concerning temporal-difference learning. Temporal-difference learning methods are distinctive in being driven by the difference between temporally successive estimates of the same quantity, for example, of the probability of winning in the Tic-Tac-Toe example. This thread is smaller and less distinct than the other two, but has played a particularly important role in the field, in part because temporal-difference methods seem to be new and unique to reinforcement learning.

The origins of temporal-difference learning are in part in animal learning psychology, in particular, in the notion of *secondary reinforcers*. A secondary reinforcer is a stimulus that has been paired with a primary reinforcer such as food or pain and, as a result, has come to take on similar reinforcing properties. Minsky (1954) may have been the first to realize that this psychological principle could be important for artificial learning systems. Arthur Samuel (1959) was the first to propose and implement a learning method that included temporal-difference ideas, as part of his celebrated checkers-playing program. Samuel made no reference to Minsky's work or to possible connections to animal learning. His inspiration apparently came from Claude Shannon's (1950a) suggestion that a computer could be programmed to use an evaluation function to play chess, and that it might be able to to improve its play by modifying this function online. (It is possible that these ideas of Shannon also influenced Bellman, but we know of no evidence for this.) Minsky (1961) extensively discussed Samuel's work in his ``Steps'' paper, suggesting the connection to secondary reinforcement theories, natural and artificial.

As we have discussed, in the decade following the work of Minsky and Samuel, little computational work was done on trial-and-error learning, and apparently no computational work at all was done on temporal-difference learning.

In 1972, Klopf brought trial-and-error learning together with an important component of temporal-difference learning. Klopf was interested in principles that would scale to learning in very large systems, and thus was intrigued by notions of local reinforcement, whereby subcomponents of an overall learning system could reinforce one another. He developed the idea of ``generalized reinforcement,'' whereby every component (nominally, every neuron) viewed all of its inputs in reinforcement terms, excitatory inputs as rewards and inhibitory inputs as punishments. This is not the same idea as what we now know as temporal-difference learning, and in retrospect it is farther from it than was Samuel's work. Nevertheless, Klopf's work also linked the idea with trial-and-error learning and with the massive empirical database of animal learning psychology.

Sutton (1978a--c) developed Klopf's ideas further, particularly the links to animal learning theories, describing learning rules driven by changes in temporally successive predictions. He and Barto refined these ideas and developed a psychological model of classical conditioning based on temporal-difference learning (Sutton and Barto, 1981; Barto and Sutton, 1982). There followed several other influential psychological models of classical conditioning based on temporal-difference learning (e.g., Klopf, 1988; Moore et al., 1986; Sutton and Barto, 1987, 1990). Some neuroscience models developed at this time are well interpreted in terms of temporal-difference learning (Hawkins and Kandel, 1984; Byrne, 1990; Gelperin et al., 1985; Tesauro, 1986; Friston et al, 1994), although in most cases there was no historical connection. A recent summary of links between temporal-difference learning and neuroscience ideas is provided by Schultz, Dayan, and Montague, 1997).

In our work on temporal-difference learning up until 1981 we were strongly influenced by animal learning theories and by Klopf's work. Relationships to Minsky's ``Steps'' paper and to Samuel's checkers players appear to have been recognized only afterwards. By 1981, however, we were fully aware of all the prior work mentioned above as part of the temporal-difference and trial-and-error threads. At this time we developed a method for using temporal-difference learning in trial-and-error learning, known as the actor-critic architecture, and applied this method to Michie and Chambers' pole-balancing problem (Barto, Sutton and Anderson, 1983). This method was extensively studied in Sutton's (1984) PhD thesis and extended to use backpropagation neural networks in Anderson's (1986) PhD thesis. Around this time, Holland (1986) incorporated temporal-difference methods explicitly into his classifier systems. A key step was taken by Sutton in 1988 by separating temporal-difference learning from control, treating it as a general prediction method. That paper also introduced the TD(λ) \ algorithm and proved some of its convergence properties.

As we were finalizing our work on the actor-critic architecture in 1981, we discovered a paper by Ian Witten (1977) which contains the earliest known publication of a temporal-difference learning rule. He proposed the method that we now call TD(0) for use as part of an adaptive controller for solving MDPs. Witten's work was a descendant of Andreae's early experiments with STeLLA and other trial-and-error learning systems. Thus, Witten's 1977 paper spanned both major threads of reinforcement-learning research---trial-and-error and optimal control---while making a distinct early contribution to temporal-difference learning.

Finally, the temporal-difference and optimal control threads were fully brought together in 1989 with Chris Watkins's development of Q-learning. This work extended and integrated prior work in all three threads of reinforcement learning research. Paul Werbos (1987) also contributed to this integration by arguing for the convergence of trial-and-error learning and dynamic programming since 1977. By the time of Watkins's work there had been tremendous growth in reinforcement learning research, primarily in the machine learning subfield of artificial intelligence, but also in neural networks and artificial intelligence more broadly. In 1992, the remarkable success of Gerry Tesauro's backgammon playing program, TD-Gammon, brought additional attention to the field. Other important contributions made in the recent history of reinforcement learning are too numerous to mention in this brief account; we cite these at the end of the individual chapters in which they arise.

Richard Sutton
Sat May 31 14:27:51 EDT 1997

1.7 Bibliographical Remarks

For additional general coverage of reinforcement learning, we refer the reader to the books by Bertsekas and Tsitsiklis (1996) and Kaelbling (1993). Two special issues of the journal *Machine Learning* focus on reinforcement learning: Sutton (1992) and Kaelbling (1996). Useful surveys are provided by Barto (1995), Kaelbling, Littman, and Moore (1996), and Keerthi and Ravindran (1997).

The example of Phil's breakfast in this chapter was inspired by Agre (1988). We refer the reader to Chapter 6 for references to the kind of temporal difference method we used in the Tic-Tac-Toe example.

Modern attempts to relate the kinds of algorithms used in reinforcement learning to the nervous system are made by Barto (1995), Friston et al. (1994), Hampson (1989), Houk et al. (1995), Montague et al. (1996), and Schultz et al. (1997).

Richard Sutton
Sat May 31 14:27:51 EDT 1997

2 Evaluative Feedback

The most important feature distinguishing reinforcement learning from other types of learning is that it uses training information that *evaluates* the actions taken rather than *instructs* by giving correct actions. This is what creates the need for active exploration, for an explicit trial-and-error search for good behavior. Purely evaluative feedback indicates how good the action taken was, but not whether it was the best or the worst action possible. Evaluative feedback is the basis of methods for function optimization, including evolutionary methods. Purely instructive feedback, on the other hand, indicates the correct action to take, independently of the action actually taken. This kind of feedback is the basis of supervised learning, which includes large parts of pattern classification, artificial neural networks, and system identification. In their pure forms, these two kinds of feedback are quite distinct: evaluative feedback depends entirely on the action taken, whereas instructive feedback is independent of the action taken. There are also interesting cases between these two in which evaluation and instruction blend together.

In this chapter we study the evaluative aspect of reinforcement learning in a simplified setting, one that does not involve learning to act in more than one situation. This *non-associative* setting is the one in which most prior work involving evaluative feedback has been done, and it avoids much of the complexity of the full reinforcement learning problem. Studying this case will enable us to see most clearly how evaluative feedback differs from, and yet can be combined with, instructive feedback.

The particular non-associative, evaluative-feedback problem that we explore is a simple version of the **n** -armed bandit problem. We use this problem to introduce a number of basic learning algorithms that we extend in later chapters to apply to the full reinforcement learning problem. We end this chapter by taking a step closer to the full reinforcement learning problem by discussing what happens when the bandit problem becomes associative, that is, when actions are taken in more than one situation.

Next Up Previous

Next: 2.1 An n-armed Bandit **Up:** Contents **Previous:** 1 Introduction

Richard Sutton
Sat May 31 12:02:11 EDT 1997

2.1 An n-armed Bandit Problem

Consider the following learning problem. You are faced repeatedly with a choice among **n** different options, or actions. After each choice you receive a numerical reward chosen from a stationary probability distribution dependent on the action you selected. Your objective is to maximize the expected total reward over some time period, for example, over 1000 action selections. Each action selection is called a *play*.

This is the original form of the ***n*** *-armed bandit problem*, so named by analogy to a slot machine, or ``one-armed bandit,'' except with **n** levers instead of one. Each action selection is like a play of one of the slot machine's levers, and the rewards are the payoffs for hitting the jackpot. Through repeated plays you are to maximize your winnings by concentrating your plays on the best levers. Another analogy is that of a doctor choosing between experimental treatments for a series of seriously ill patients. Each play is a treatment selection, and each reward is the survival or well being of the patient. Today the term ``**n** -armed bandit problem'' is often used for a generalization of the problem described above, but in this book we use it to refer just to this simple case.

In our **n** -armed bandit problem, each action has an expected or mean reward given that that action is selected; let us call this the *value* of that action. If you knew the value of each action, then it would be trivial to solve the **n** -armed bandit problem: you would simply always select the action with highest value. We assume that you do not know the action values with certainty, although you may have estimates.

If you maintain estimates of the action values, then at any time there is at least one action whose estimated value is greatest. We call this a *greedy* action. If you select a greedy action, we say that you are *exploiting* your current knowledge of the value of the actions. If instead you select one of the non-greedy actions, then we say you are *exploring* because this enables you to improve your estimate of the non-greedy action's value. Exploitation is the right thing to do to maximize expected reward on the one play, but exploration may produce greater total reward in the long run. For example, suppose the greedy action's value is known with certainty, while several other actions are estimated to be nearly as good but with substantial uncertainty. The uncertainty is such that at least one of these other actions is probably actually better than the greedy action, only you don't know which. If you have many plays yet to make, then it may be better to explore the non-greedy actions and discover which of them are better than the greedy action. Reward is lower in the short run, during exploration, but higher in the long run because after you have discovered the better actions, you can exploit *them*. Because it is not possible to both explore and exploit with any single action selection, one often refers to the ``conflict'' between exploration and exploitation.

In any specific case, whether it is better to exploit or explore depends in a complex way on the precise values of the estimates, uncertainties, and the number of remaining plays. There are many sophisticated methods for balancing exploration and exploitation for particular mathematical formulations of the **n** -armed bandit and related problems. However, most of these methods make strong assumptions about stationarity and prior knowledge that are either violated or impossible to verify in applications and in the full reinforcement learning problem that we consider in subsequent chapters. The guarantees of optimality or bounded loss for these methods are of little comfort when the assumptions of their theory do not apply.

In this book we do not worry about balancing exploitation and exploration in a sophisticated way; we worry only about balancing them at all. In this chapter we present several simple balancing methods for the **n** -armed bandit problem and show that they work much better than methods that always exploit. In addition, we point out that supervised learning methods (or rather the methods closest to supervised learning methods when adapted to this problem) perform poorly on this problem because they do not balance exploitation and exploration at all. The need to balance exploration and exploitation is a distinctive challenge that arises in reinforcement learning; the simplicity of the **n** -armed bandit problem enables us to show this in a particularly clear form.

Next Up Previous

Next: 2.2 Action-Value Methods **Up:** 2 Evaluative Feedback **Previous:** 2 Evaluative Feedback

Richard Sutton
Sat May 31 12:02:11 EDT 1997

2.2 Action-Value Methods

We begin by looking more closely at some simple algorithms for estimating the value of actions and for using the estimates to make action-selection decisions. In this chapter, we denote the true (actual) value of action **a** as $Q^*(a)$, and the estimated value after **t** plays as $Q_t(a)$. Recall that the true value of an action is the mean reward given that the action is selected. One natural way to estimate this is by averaging the rewards actually received when the action was selected. In other words, if, after **t** decisions, action **a** has been chosen k_a times, yielding rewards $r_1, r_2, \ldots, r_{k_a}$, then its value is estimated to be

$$Q_t(a) = \frac{r_1 + r_2 + \cdots + r_{k_a}}{k_a}. \tag{2.1}$$

If $k_a = 0$, then we define $Q_t(a)$ instead as some default value, e.g., $Q_0(a) = 0$. As $k_a \rightarrow \infty$, by the law of large numbers $Q_t(a)$ converges to $Q^*(a)$. We call this the *sample-average* method for estimating action values because each estimate is a simple average of the sample of relevant rewards. Of course this is just one way to estimate action values, not necessarily the best one. Nevertheless, for now let us stay with this simple estimation algorithm and turn to the question of how the estimates might be used to select actions.

The simplest action selection rule is to select the action (or one of the actions) with highest estimated action value, i.e., to select on the **t**th play one of the greedy actions, a_t^*, for which $Q_{t-1}(a_t^*) = \max_a Q_{t-1}(a)$. This method always exploits current knowledge to maximize immediate reward; it spends no time at all sampling apparently inferior actions to see if they might really be better. A simple alternative is to behave greedily most of the time, but every once in a while, say with small probability ϵ , instead select an action at random, uniformly, independently of the action-value estimates. We call methods using this near-greedy action selection rule ϵ *-greedy* methods. An advantage of these methods is that in the limit as the number of plays increases, every action will be sampled an infinite number of times, guaranteeing that $k_a \rightarrow \infty$ for all **a**, and thus ensuring that all the $Q_t(a)$ converge to $Q^*(a)$. This of course implies that the probability of selecting the optimal action converges to $1 - \epsilon$, i.e., to near certainty. These are just asymptotic guarantees, however, and say little about the practical effectiveness of the methods.

To roughly assess the relative effectiveness of the greedy and ϵ -greedy methods, we compared them numerically on a suite of test problems. This is a set of 2000 randomly generated **n** -armed bandit tasks with **n=10**. For each action, **a**, the rewards were selected from a normal (Gaussian) probability distribution with mean $Q^*(a)$ and variance **1**. The 2000 **n** -armed bandit tasks were generated by re-selecting the $Q^*(a)$ 2000 times, each according to a normal distribution with mean **0** and variance **1**. Averaging over tasks, we can plot the performance and behavior of various methods as they improve with experience over 1000 plays, as in Figure 2.1. We call this suite of test tasks the *10-armed testbed*.

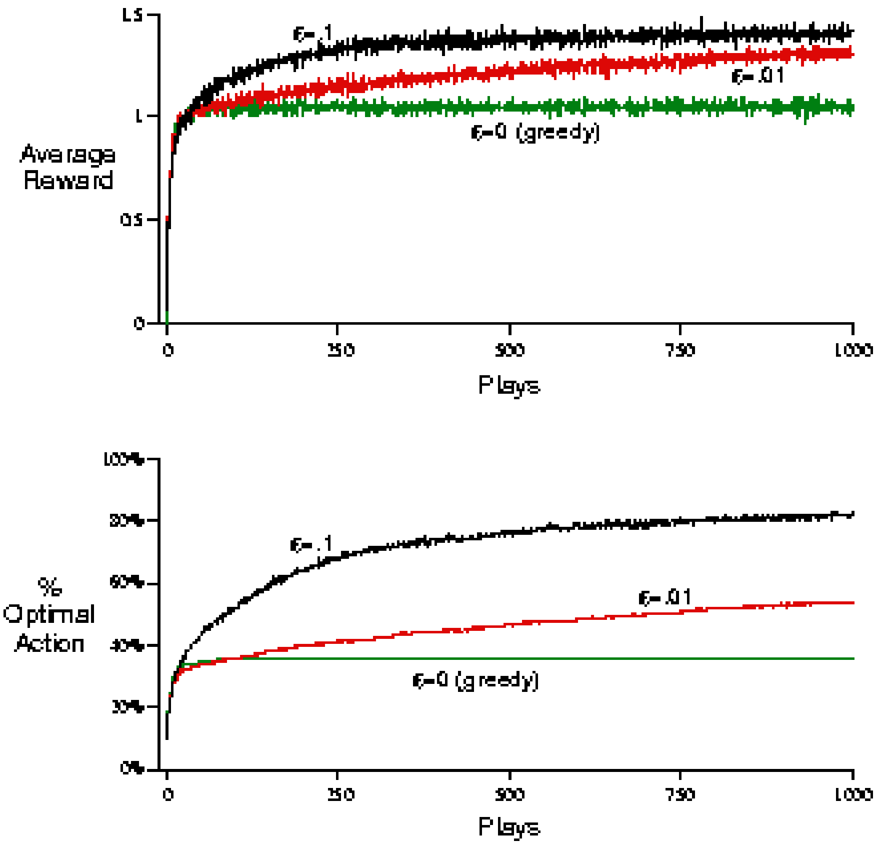


Figure 2.1: Average performance of ϵ -greedy action-value methods on 10-armed testbed.

These data are averages over 2000 tasks. All methods used sample averages as their action-value estimates.

Figure 2.1 compares a greedy method with two ϵ-greedy methods ($\epsilon = .01$ and $\epsilon = .1$), as described above, on the 10-armed testbed. Both methods formed their action-value estimates using the sample-average technique. The upper graph shows the increase in expected reward with experience. The greedy method improves slightly faster than the other methods at the very beginning, but then levels off at a lower level. It achieves a reward-per-step of only about 1 compared to the best possible of about 1.55 on this testbed. The greedy method performs significantly worse in the long run because it often gets stuck performing suboptimal actions. The lower graph shows that the greedy method found the optimal action only in approximately one-third of the tasks. In the other two-thirds, its initial samples of the optimal action were disappointing, and it never returned to it. The ϵ-greedy methods eventually perform better because they continue to explore, and continue to improve their chances of recognizing the optimal action. The $\epsilon = .1$ method explores more, and usually finds the optimal action earlier, but never selects it more than 91% of the time. The $\epsilon = .01$ method improves more slowly, but eventually performs better than the $\epsilon = .1$ method by both performance measures. It is also possible to reduce ϵ over time to try to get the best of both high and low values.

The advantage of ϵ-greedy over greedy methods depends on the problem. For example, suppose the reward variance had been larger, say 10 instead of 1. With noisier rewards it takes more exploration to find the optimal action, and ϵ-greedy methods should fare even better relative to the greedy method. On the other hand, if the reward variances were zero, then the greedy method would know the true value of each action after trying it once. In this case the greedy method might actually perform best because it would soon find the optimal action and then never explore. But even in the deterministic case, there is a large advantage to exploring if we weaken some of the other assumptions. For example, suppose the bandit problem were nonstationary, that is, that the true values of the actions changed over time. In this case exploration is needed even in the deterministic case to make sure one of the non-greedy actions has not changed to become better than the greedy one. As we will see in the next few chapters, effective nonstationarity is the case most commonly encountered in reinforcement learning. Even if the underlying problem is stationary and deterministic, the learner faces a set of bandit-like decision problems each of which changes over time due to the learning process itself. Reinforcement learning problems have a strong requirement for some sort of balance between exploration and exploitation.

Exercise 2.1

In the comparison shown in Figure 2.1, which method will perform best in the long run, in terms of cumulative reward and cumulative probability of selecting the best action? How much better will it be?

Richard Sutton
Sat May 31 12:02:11 EDT 1997

2.3 Softmax Action Selection

Although ϵ -greedy action-selection is an effective and popular means of balancing exploration and exploitation in reinforcement learning, one drawback is that when it explores it chooses equally among all actions. This means that it is just as likely to choose the worst appearing action as it is to choose the next-to-best. In tasks where the worst actions are very bad, this may be unsatisfactory. The obvious solution is to vary the action probabilities as a graded function of estimated value. The greedy action is still given the highest selection probability, but all the others are ranked and weighted according to their value estimates. These are called *softmax* action selection rules. The most common softmax method uses a Gibbs, or Boltzmann, distribution. It chooses action **a** on the **t**th play with probability

$$\frac{e^{Q_{t-1}(a)/\tau}}{\sum_b e^{Q_{t-1}(b)/\tau}}, \qquad (2.2)$$

where τ is a positive parameter called the *temperature*. High temperatures cause the actions to be all (nearly) equi-probable. Low temperatures cause a greater difference in selection probability for actions that differ in their value estimates. In the limit as $\tau \longrightarrow 0$, softmax action selection becomes the same as greedy action selection. Of course, the softmax effect can be produced in a large number of ways other than by a Gibbs distribution. For example, one could simply add a random number from a long-tailed distribution to each $Q_{t-1}(a)$, and then pick the action whose sum was largest.

Whether softmax action selection or ϵ -greedy action selection is better is unclear and may depend on the task and on human factors. Both methods have only one parameter that must be set. Most people find it easier to set the ϵ \ parameter with confidence; setting τ requires knowledge of the likely action values, and of powers of **e**. We know of no careful comparative studies of these two simple action-selection rules.

Exercise 2.2 **(programming)**

How does the softmax action selection method (using the Gibbs distribution) fare on the 10-armed testbed? Implement the method and run it at several temperatures to produce graphs similar to those in Figure 2.1. To verify your code, first implement the ϵ -greedy methods and reproduce some specific aspect of the results in Figure 2.1.

Exercise 2.3*

Show that in the case of two actions, the softmax operation using the Gibbs distribution becomes the logistic, or sigmoid, function commonly used in artificial neural networks. What effect does the temperature parameter τ have on the function?

Next Up Previous

Next: 2.4 Evaluation versus Instruction **Up:** 2 Evaluative Feedback **Previous:** 2.2 Action-Value Methods

Richard Sutton
Sat May 31 12:02:11 EDT 1997

2.4 Evaluation versus Instruction

The **n** -armed bandit problem we considered above is a case in which the feedback is purely evaluative. The reward received after each action gives some information about how good the action was, but it says nothing at all about whether the action was correct or incorrect, that is, whether it was a best action or not. Here, correctness is a relative property of actions that can only be determined by trying them all and comparing their rewards. In this sense the problem is inherently one requiring explicit search among the alternative actions. You have to perform some form of the generate-and-test method whereby you try actions, observe the outcomes, and selectively retain those that are the most effective. This is learning by selection, in contrast to learning by instruction, and all reinforcement learning methods have to use it in one form or another.

This contrasts sharply with supervised-learning, where the feedback from the environment directly indicates what the correct action should have been. In this case there is no need to search: whatever action you try, you will be told what the right one would have been. There is no need to try a variety of actions; the instructive ``feedback'' is typically *independent* of the action selected (so is not really feedback at all). It might still be necessary to search in the parameter space of the supervised learning system (e.g., the weight-space of a neural network), but searching in the space of actions is not required.

Of course, supervised learning is usually applied to problems that are much more complex in some ways than the **n** -armed bandit. In supervised learning there is not one situation in which action is taken, but a large set of different situations, each of which must be responded to correctly. The main problem facing a supervised learning system is to construct a mapping from the situations to actions that mimics the correct actions specified by the environment and that generalizes correctly to new situations. A supervised learning system cannot be said to learn to control its environment because it follows, rather than influences, the instructive information it receives. Instead of trying to make its environment behave in a certain way, it tries to make *itself* behave as instructed by its environment.

Focusing on the special case of a single situation that is encountered repeatedly helps make plain the distinction between evaluation and instruction. Suppose there are 100 possible actions and you select action number 32. Evaluative feedback would give you a score, say 0.9, for that action whereas instructive training information would say what other action, say action number 67, would actually have been correct. The latter is clearly much more informative training information. Even if instructional information is noisy, it is still more informative than evaluative feedback. It is always true that a *single* instruction can be used to advantage to direct changes in the action selection rule, whereas evaluative feedback must be compared with that of other actions before any inferences can be made about action selection.

The difference between evaluative feedback and instructive information remains significant even if there are only two actions and two possible rewards. For these *binary bandit* tasks, let us call the two rewards *success* and *failure*. If you received success, then you might reasonably infer that whatever action you selected was correct, and if you received failure, then you might infer that whatever action you did *not* select was correct. You could then keep a tally of how often each action was (inferred to be) correct and select the action that was correct most often. Let us call this the *supervised* algorithm because it corresponds most closely to what supervised learning methods might do in the case of only a single input pattern. If the rewards are deterministic, then the inferences of the supervised algorithm are all correct and it performs excellently. If the rewards are stochastic, then the picture is more complicated.

In the stochastic case, a particular binary bandit task is defined by two numbers, the probabilities of success for each possible action. The space of all possible tasks is thus a unit square, as shown in Figure 2.2. The upper-left and lower-right quadrants correspond to relatively easy tasks for which the supervised algorithm would work well. For these, the probability of success for the better action is greater than a half and the probability of success for the poorer action is less than a half. For these tasks, the action inferred to be correct as described above will actually be the correct action more than half the time.

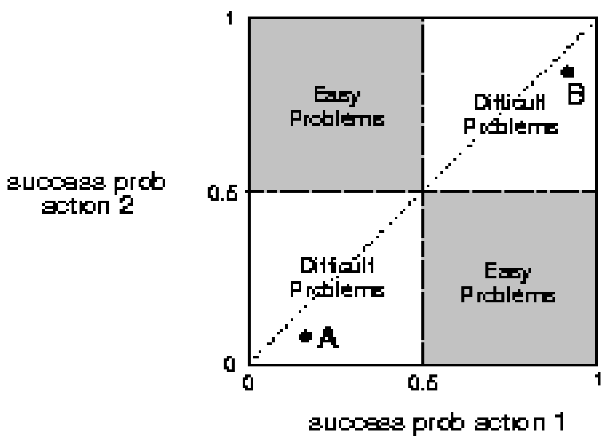
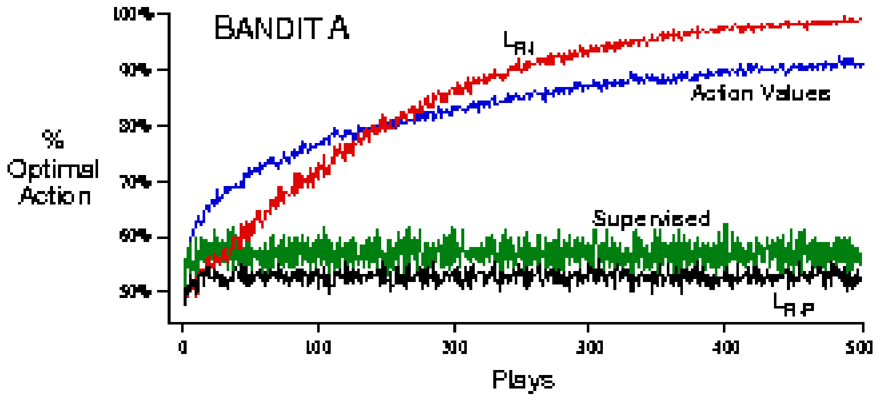


Figure 2.2: The easy and difficult regions in the space of all binary bandit tasks.

However, binary bandit tasks in the other two quadrants of Figure 2.2 are more difficult and cannot be solved effectively by the supervised algorithm. For example, consider a task with success probabilities .1 and .2, corresponding to point A in the lower-left difficult quadrant of Figure 2.2. Because both actions produce failure at least 80% of the time, any method that takes failure as an indication that the other action was correct will oscillate between the two actions, never settling on the better one. Now consider a task with success probabilities .8 and .9, corresponding to point B in the upper-right difficult quadrant of Figure 2.2. In this case both actions produce success almost all the time. Any method that takes success as an indication of correctness can easily become stuck selecting the wrong action.

Figure 2.3 shows the average behavior of the supervised algorithm and several other algorithms on the binary bandit tasks corresponding to points A and B. For comparison, also shown is the behavior of an ϵ -greedy action-value method ($\epsilon = 0.1$) as described in Section 2.2. In both tasks, the supervised algorithm learns to select the better action only slightly more than half the time.



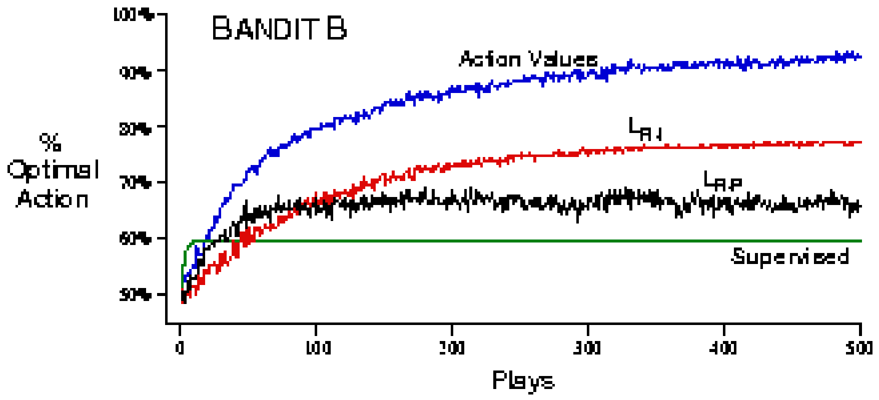


Figure 2.3: Performance of selected algorithms on the binary bandit tasks corresponding to points A and B in Figure 2.2. These data are averages over 2000 runs.

The graphs in Figure 2.3 also show the average behavior of two other algorithms, known as L_{R-I} and L_{R-P} . These are classical methods from the field of *learning automata* that follow a logic similar to that of the supervised algorithm. Both methods are stochastic, updating the probabilities of selecting each action, denoted $\pi_t(1)$ and $\pi_t(2)$. The L_{R-P} method infers the correct action just as the supervised algorithm does, and then adjusts its probabilities as follows. If the action inferred to be correct on play **t** was d_t, then $\pi_t(d_t)$ is incremented a fraction, α, of the way from its current value towards one:

$$\pi_{t+1}(d_t) = \pi_t(d_t) + \alpha(1 - \pi_t(d_t)). \qquad (2.3)$$

The probability of the other action is adjusted inversely, so that the two probabilities sum to one. For the results shown in Figure 2.3, $\alpha = 0.1$. The idea of L_{R-P} is very similar to that of the supervised algorithm, only it is stochastic. Rather than committing totally to the action inferred to be best, L_{R-P} gradually increases its probability.

The name L_{R-P} stands for ``linear, reward-penalty," meaning that the update (2.3) is linear in the probabilities and that the update is performed on both success (reward) plays and failure (penalty) plays. The name $L_{R-I}$ stands for ``linear, reward-inaction." This algorithm is identical to L_{R-P} except that it only updates its probabilities upon success plays; failure plays are ignored entirely. The results in Figure 2.3 show that L_{R-P} performs little, if any, better than the supervised algorithm on the binary bandit tasks corresponding to points A and B in Figure 2.2. L_{R-I} eventually performs very well on the A task, but not on the B task, and learns slowly in both cases.

Binary bandit problems are an instructive special case blending aspects of supervised and reinforcement learning problems. Because the rewards are binary, it is possible to infer something about the correct action given just a single reward. On some instances of such problems, these inferences are quite reasonable and lead to effective algorithms. In other instances, however, such inferences are less appropriate and lead to poor behavior. In bandit problems with non-binary rewards, such as the 10-armed test bandit introduced in Section 2.2, it is not at all clear how the ideas behind these inferences could be applied to produce effective algorithms. All of these are very simple problems, but already we see the need for capabilities beyond those of supervised learning methods.

Exercise 2.4

Consider a simplified supervised learning problem in which there is only one situation (input pattern) and two actions. One action, say **a**, is correct and the other, **b**, is incorrect. The instruction signal is noisy: it instructs the wrong action with probability **p**; that is, with probability **p** it says that **b** is correct. You can think of this as a binary bandit problem if you treat agreeing with the (possibly wrong) instruction signal as *success*, and disagreeing with it as *failure*. Discuss the resulting class of binary bandit problems. Is anything special about these problems? How does the supervised algorithm perform on this type of problem?

Next Up Previous

Next: 2.5 Incremental Implementation **Up:** 2 Evaluative Feedback **Previous:** 2.3 Softmax Action Selection

---

Richard Sutton
Sat May 31 12:02:02:11 EDT 1997

2.5 Incremental Implementation

The action-value methods we have discussed so far all estimate their action values as the sample averages of observed rewards. The obvious implementation is to maintain, for each action, **a**, a record of all the rewards that have followed the selection of that action. Then, whenever estimate of the action value is needed, it can be computed according to (2.1), which we repeat here:

$$Q_t(a) = \frac{r_1 + r_2 + \cdots + r_{k_a}}{k_a}, \qquad (2.1)$$

where $r_1, \ldots, r_{k_a}$ are all the rewards received following all selections of action **a** up until play **t**. A problem with this straightforward implementation is that its memory and computational requirements grow over time without bound. That is, each additional reward following a selection of action **a** requires more memory to store it and results in more computation being required to compute $Q_t(a)$.

As you might suspect, this is not really necessary. It is easy to devise incremental update formulas for computing averages with small, constant computation required to process each new reward. For some action, let Q_k denote the average of its first **k** rewards. Given this average and a $(k+1)$st reward, r_{k+1}, the average of all **k+1** rewards can be computed by:

$$
\begin{aligned}
Q_{k+1} &= \frac{1}{k+1}\sum_{i=1}^{k+1} r_i \\
&= \frac{1}{k+1}\left[r_{k+1} + \sum_{i=1}^{n} r_i\right] \\
&= \frac{1}{k+1}\left[r_{k+1} + kQ_k + Q_k - Q_k\right] \\
&= \frac{1}{k+1}\left[r_{k+1} + (k+1)Q_k - Q_k\right] \\
&= Q_k + \frac{1}{k+1}\left[r_{k+1} - Q_k\right], \qquad (2.4)
\end{aligned}
$$

which holds even for **k=0**, obtaining $Q_1 = r_1$ for arbitrary Q_0. This implementation requires memory only for Q_k and **k**, and only the small computation (2.4) for each new reward.

The update rule (2.4) is of a form that occurs frequently throughout this book. The general form is:

$$\mathit{NewEstimate} \leftarrow \mathit{OldEstimate} + \mathit{StepSize}\left[\mathit{Target} - \mathit{OldEstimate}\right].$$

The expression $\left[\mathit{Target} - \mathit{OldEstimate}\right]$ is an *error* in the estimate. It is reduced by taking a step toward the ``target". The target is presumed to indicate a desirable direction in which to move, though it may be noisy. In the case above, for example, the target is the $(k+1)\mathrm{st}$ reward.

Note that the step size used in the incremental method described above changes from time step to time step. In processing the **k**th reward for action **a**, that method uses a step-size of $\frac{1}{k}$. In this book we denote the step size by the symbol α , or, more generally, by $\alpha_k(a)$. For example, the above incremental implementation of the sample average method is described by the equation $\alpha_k(a) = \frac{1}{k_a}$. Accordingly, we sometimes use the informal shorthand $\alpha = 1/k$ to refer to this case, leaving the action dependence implicit.

Exercise 2.5

Give pseudo-code for a complete algorithm for the **n** -armed bandit problem. Use greedy action selection and incremental computation of action values with $\alpha = 1/k$ step size. Assume a function $bandit(a)$ that takes an action and returns a reward. Use arrays and variables; do not subscript anything by the time index **t**. Indicate how the action values are initialized and updated after each reward.

Richard Sutton
Sat May 31 12:02:11 EDT 1997

2.6 Tracking a Nonstationary Problem

The averaging methods discussed so far are appropriate in a stationary environment, but not if the bandit is changing over time. As we noted earlier, we often encounter reinforcement learning problems that are effectively nonstationary. In such cases it makes sense to weight recent rewards more heavily than long past ones. One of the most popular ways of doing this is to use a fixed step size. For example, the incremental update rule (2.4) for updating an average Q_k of the **k** past rewards is modified to be

$$Q_{k+1} = Q_k + \alpha[r_{k+1} - Q_k] \tag{2.5}$$

where the step size, α, $0 < \alpha \leq 1$, is constant. This results in Q_k being a weighted average of past rewards and the initial estimate Q_0:

$$\begin{aligned} Q_k &= Q_{k-1} + \alpha[r_k - Q_{k-1}] \\ &= \alpha r_k + (1-\alpha)Q_{k-1} \\ &= \alpha r_k + (1-\alpha)\alpha r_{k-1} + (1-\alpha)^2 Q_{k-2} \\ &= \alpha r_k + (1-\alpha)\alpha r_{k-1} + (1-\alpha)^2 \alpha r_{k-2} + \\ &\qquad \cdots + (1-\alpha)^{k-1}\alpha r_1 + (1-\alpha)^k Q_0 \\ &= (1-\alpha)^k Q_0 + \sum_{i=1}^{k} \alpha(1-\alpha)^{k-i} r_i \end{aligned} \tag{2.6}$$

We call this a weighted average because the sum of the weights is

$(1-\alpha)^k + \sum_{i=1}^{k} \alpha(1-\alpha)^{k-i} = 1$, as you can check yourself. Note that the weight, $\alpha(1-\alpha)^{k-i}$, given to the reward, r_i, depends on how many rewards ago, **k-i**, it was observed. The quantity $1-\alpha$ is less than **1**, and thus the weight given to r_i decreases as the number of intervening rewards increases. In fact, the weight decays exponentially according to the exponent on $1-\alpha$. Accordingly, this is sometimes called an *exponential, recency-weighted average*.

Sometimes it is convenient to vary the step size from step to step. Let $\alpha_k(a)$ denote the step size used to process the reward received after the **k**th selection of action **a**. As we have noted, the choice $\alpha_k(a) = 1/k$ results in the sample-average method, which is guaranteed to converge to the true action values by the law of large numbers. But of course convergence is not guaranteed for all choices of the sequence $\{\alpha_k(a)\}$. A classic result in stochastic approximation theory gives us the conditions required to assure convergence with probability one:

$$\sum_{k=1}^{\infty} \alpha_k(a) = \infty \qquad \text{and} \qquad \sum_{k=1}^{\infty} \alpha_k^2(a) < \infty. \tag{2.7}$$

The first condition is required to guarantee that the steps are large enough to eventually overcome any initial conditions or random fluctuations. The second condition guarantees that eventually the steps become small enough to assure convergence.

Note that both convergence conditions are met for the sample-average case, $\alpha_k(a) = 1/k$, but not for the case of constant step size, $\alpha_k(a) = \alpha$. In the latter case, the second condition is not met, indicating that the estimates never completely converge but continue to vary in response to the most recently received rewards. As we mentioned above, this is actually desirable in a nonstationary environment, and problems that are effectively nonstationary are the norm in reinforcement learning. In addition, step-size sequences that meet the conditions (2.7) often converge very slowly or need considerable tuning in order to obtain a satisfactory convergence rate. Although step-size sequences that meet these convergence conditions are often used in theoretical work, they are seldom used in applications and empirical research.

Exercise 2.6

If the step sizes, $\alpha_k(a)$, are not constant, then the estimate $Q_k(a)$ is a weighted average of previously received rewards with a weighting different from that given by (2.6). What is the weighting on each prior reward for the general case?

Exercise 2.7 (programming)

Design and conduct an experiment to demonstrate the difficulties that sample-average methods have on nonstationary problems. Use a modified version of the 10-armed testbed in which all the $Q^*(a)$ start out equal and then take independent random walks. Prepare plots like Figure 2.1 for an action-value method using sample averages, incrementally computed by $\alpha = 1/k$, and another action-value method using a constant step size, $\alpha = 0.1$. Use $\epsilon = .1$ and, if necessary, runs longer than 1000 plays.

Next Up Previous
Next: 2.7 Optimistic Initial Values **Up:** 2 Evaluative Feedback **Previous:** 2.5 Incremental Implementation

---

Richard Sutton
Sat May 31 12:02:11 EDT 1997

2.7 Optimistic Initial Values

All the methods we have discussed so far are dependent to some extent on the initial action-value estimates, $Q_0(a)$. In the language of statistics, these methods are *biased* by their initial estimates. For the sample-average methods, the bias disappears once all actions have been selected at least once, but for methods with constant α, the bias is permanent, though decreasing over time as given by (2.6). In practice, this kind of bias is usually not a problem, and can sometimes be very helpful. The downside is that the initial estimates become, in effect, a whole set of parameters that must be picked by the user, if only to set them all to zero. The upside is that they provide an easy way to supply some prior knowledge about what level of rewards can be expected.

Initial action values can also be used as a simple way of encouraging exploration. Suppose that instead of setting the initial action values to zero, as we did in the 10-armed testbed, we set them all to +5. Recall that the $Q^*(a)$ in this problem are selected from a normal distribution with mean 0 and variance 1. An initial estimate of +5 is thus wildly optimistic. But this optimism encourages action-value methods to explore. Whichever actions are initially selected, the reward is less than the starting estimates; the learner switches to other actions, being ``disappointed" with the rewards it is receiving. The result is that all actions are tried, several times, before the value estimates converge. The system does a fair amount of exploration even if greedy actions are selected all the time.

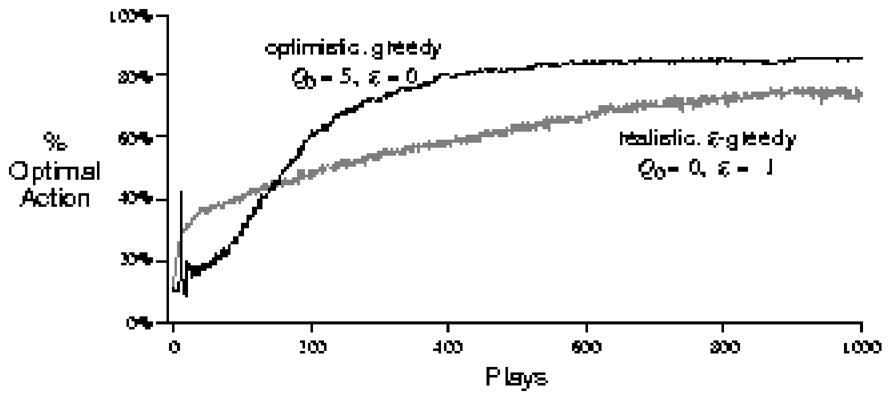


Figure 2.4: The effect of optimistic initial action-values estimates on the 10-armed

testbed. Both algorithms used constant step size, $\alpha = .1$.

Figure 2.4 shows the performance on the 10-armed bandit testbed of a greedy method using $Q_0(a) = +5$, for all **a**. For comparison, also shown is an ϵ -greedy method with $Q_0(a) = 0$. Both methods used a constant step size, $\alpha = .1$. Initially, the optimistic method performs worse, because it explores more, but eventually it performs better because its exploration decreases with time. We call this technique for encouraging exploration *optimistic initial values*. We regard it as a simple trick that can be quite effective on stationary problems, but it is far from being a generally useful approach to encouraging exploration. For example, it is not well suited to nonstationary problems because its drive for exploration is inherently temporary. If the task changes, creating a renewed need for exploration, this method cannot help. Indeed, any method that focuses on the initial state in any special way is unlikely to help with the general non-stationary case. The beginning of time occurs only once, and thus we should not focus on it too much. This criticism applies as well to the sample-average methods, which also treat the beginning of time as a special event, averaging with equal weights all subsequent rewards. Nevertheless, all of these methods are very simple, and one or some simple combination of them is often adequate in practice. In the rest of this book we make frequent use of many of the exploration techniques that we explore here within the simpler non-associative setting of the **n** -armed bandit problem.

Exercise 2.8

The results shown in Figure 2.4 should be quite reliable because they are averages over 2000 individual, randomly chosen 10-armed bandit tasks. Why, then, are there oscillations and spikes in the early part of the curve for the optimistic method? What might make this method perform particularly better or worse, on average, on particular early plays?

Richard Sutton
Sat May 31 12:02:11 EDT 1997

2.8 Reinforcement Comparison

A central intuition underlying reinforcement learning is that actions followed by large rewards should be made more likely to recur, whereas actions followed by small rewards should be made less likely to recur. But how is the learner to know what constitutes a large or a small reward? If an action is taken and the environment returns a reward of 5, is that large or small? To make such a judgment one must compare the reward with some standard or reference level, called the *reference reward*. A natural choice for the reference reward is an average of previously received rewards. In other words, a reward is interpreted as large if it is higher than average, and small if it is lower than average. Learning methods based on this idea are called *reinforcement comparison* methods. These methods are sometimes more effective than action-value methods. They are also the precursors to actor-critic methods, an important class of methods, which we present later, for solving the full reinforcement learning problem.

Reinforcement-comparison methods typically do not maintain estimates of action values but only of an overall reward level. In order to pick among the actions, they maintain a separate measure of their preference for each action. Let us denote the preference for action **a** on play **t** by $p_t(a)$. The preferences might be used to determine action-selection probabilities according to a softmax relationship, e.g.:

$$\pi_t(a) = Pr\left\{a_t = a\right\} = \frac{e^{p_{t-1}(a)}}{\sum_b e^{p_{t-1}(b)}}, \qquad (2.8)$$

where $\pi_t(a)$ denotes the probability of selecting action **a** on the **t**th play. The reinforcement comparison idea is used in updating the action preferences. After each play, the preference for the action selected on that play, a_t, is incremented by the difference between the reward, r_t, and the reference reward, $\bar{r}_t$:

$$p_{t+1}(a_t) = p_t(a_t) + \beta(r_t - \bar{r}_t), \qquad (2.9)$$

where β is a positive step-size parameter. This equation implements the idea that high rewards should increase the probability of re-selecting the action taken and low rewards should decrease its probability.

The reference reward is an incremental average of *all* recently received rewards, whichever actions were taken. After the update (2.9), the reference reward is updated:

$$\bar{r}_{t+1} = \bar{r}_t + \alpha(r_t - \bar{r}_t), \qquad (2.10)$$

where α, $0 < \alpha \leq 1$, is a step size as usual. The initial value of the reference reward, $\bar{r}_0$, can be set either optimistically, to encourage exploration, or according to prior knowledge. The initial values of the action preferences can simply all be set to zero. Constant α is a good choice here because the distribution of rewards is changing over time as action selection improves. We see here the first case in which the learning problem is effectively nonstationary even though the underlying problem is stationary.

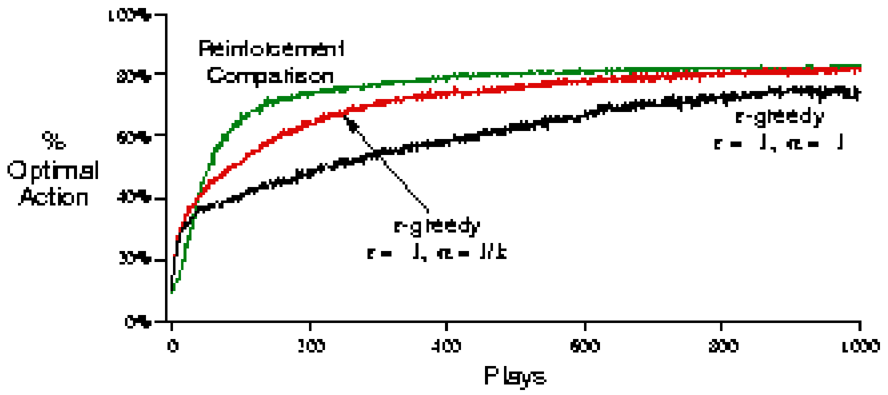


Figure 2.5: Reinforcement comparison methods versus action-value methods on the 10-armed testbed.

Reinforcement-comparison methods can be very effective, sometimes performing even better than action-value methods. Figure 2.5 shows the performance of the above algorithm ($\alpha = .1$) on the 10-armed testbed. The performances of ϵ -greedy ($\epsilon = 0.1$) action-value methods with $\alpha = 0.1$ and $\alpha = 1/k$ are also shown for comparison.

Exercise 2.9

The softmax action-selection rule given for reinforcement-comparison methods (2.8) lacks the temperature parameter, τ, used in the earlier softmax equation (2.2). Why do you think this was done? Has any important flexibility been lost here by omitting τ?

Exercise 2.10

The reinforcement-comparison methods described here have two step-size parameters, α and β. Could we, in general, reduce this to just one parameter by choosing $\alpha = \beta$? What would be lost by doing this?

Exercise 2.11 (programming)

Suppose the initial reference reward, $\bar{r}$, is far too low. Whatever action is selected first will then probably increase in its probability of selection. Thus it is likely to be selected again, and increased in probability again. In this way an early action that is no better than any other could crowd out all other actions for a long time. To counteract this effect, it is common to add a factor of $(1 - \pi_t(a_t))$ to the increment in (2.9). Design and implement an experiment to determine whether or not this really improves the performance of the algorithm.

Richard Sutton
Sat May 31 12:02:11 EDT 1997

2.9 Pursuit Methods

Another class of effective learning methods for the **n** -armed bandit problem are *pursuit* methods. Pursuit methods maintain both action-value estimates *and* action preferences, with the preferences continually ``pursuing'' the action that is greedy according to the current action-value estimates. In the simplest pursuit method, the action preferences, $p_t(a)$, are the probabilities with which each action, **a**, is selected on play **t**.

Just before selecting the action, the probabilities are updated so as to make the greedy action more likely to be selected. Suppose $a_t^* = \arg\max_a Q_{t-1}(a)$ is the greedy action (or a random sample from the greedy actions if there are more than one) just prior to selecting action a_t. Then the probability of selecting $a_t = a_t^*$ is incremented a fraction, β, of the way toward one:

$$\pi_t(a_t^*) = \pi_{t-1}(a_t^*) + \beta(1 - \pi_{t-1}(a_t^*)), \qquad (2.11)$$

while the probabilities of selecting the other actions are decremented toward zero:

$$\pi_t(a) = \pi_{t-1}(a) + \beta(0 - \pi_{t-1}(a)), \qquad \text{for all } \; a \neq a_t^*. \qquad (2.12)$$

After a_t is taken and a new reward observed, the action value, $Q_t(a_t)$, is updated in one of the ways discussed in the preceding sections, e.g., to be a sample average of the observed rewards, using (2.1).

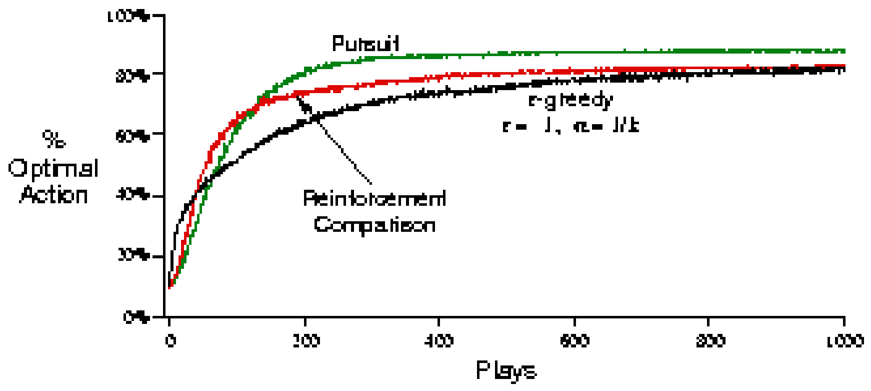


Figure 2.6: Performance of the pursuit method vis-a-vis action-value and reinforcement-comparison methods on the 10-armed testbed.

Figure 2.6 shows the performance of the pursuit algorithm described above when the action values were estimated using sample averages (incrementally computed using $\alpha = 1/k$). In these results, the initial action probabilities were $\pi_0(a) = 1/n$, for all **a** and the parameter β was .01. For comparison, we also show the performance of an ϵ -greedy method ($\epsilon = .1$) with action values also estimated using sample averages. The performance of the reinforcement-comparison algorithm from the previous section is also shown. Although the pursuit algorithm performs the best of these three on this task at these parameter settings, the ordering could well be different in other cases. All three of these methods appear to have their uses and advantages.

Exercise 2.12

An ϵ -greedy method always selects a random action on a fraction, ϵ , of the time steps. How about the pursuit algorithm? Will it eventually select the optimal action with probability approaching certainty?

Exercise 2.13

For many of the problems we will encounter later in this book it is not feasible to directly update action probabilities. To use pursuit methods in these cases it is necessary to modify them to use action preferences that are not probabilities, but which determine action probabilities according to a softmax relationship such as the Gibbs distribution (2.8). How can the pursuit algorithm described above be modified to be used in this way? Specify a complete algorithm, including the equations for action-values, preferences, and probabilities at each play.

Exercise 2.14 **(programming)**

How well does the algorithm you proposed in

Exercise 2.13

perform? Design and run an experiment assessing the performance of your method. Discuss the role of parameter value settings in your experiment.

Exercise 2.15

The pursuit algorithm described above is suited only for stationary environments because the action probabilities converge, albeit slowly, to certainty. How could you combine the pursuit idea with the ϵ -greedy idea to obtain a method with performance close to that of the pursuit algorithm, but which always continues to explore to some small degree?

Richard Sutton
Sat May 31 12:02:11 EDT 1997

2.10 Associative Search

So far in this chapter we have considered only non-associative tasks, in which there is no need to associate different actions with different situations. In these tasks the learner either tries to find a single best action when the task is stationary, or tries to track the best action as it changes over time when the task is nonstationary. However, in a general reinforcement learning task there is more than one situation, and the goal is to learn a policy: a mapping from situations to the actions that are best in those situations. To set the stage for the full problem, we briefly discuss the simplest way in which non-associative tasks extend to the associative setting.

As an example, suppose there are several different **n** -armed bandit tasks, and that on each play you confront one of these chosen at random. Thus, the bandit task changes randomly from play to play. This would appear to you as a single, nonstationary **n** -armed bandit task, whose true action values change randomly from play to play. You could try using one of the methods described in this chapter that can handle nonstationarity, but unless the true action values change slowly, these methods will not work very well. Now suppose, however, that when a bandit task is selected for you, you are given some distinctive clue about its identity (but not its action values). Maybe you are facing an actual slot machine that changes the color of its display as it changes its action values. Now you can learn a policy associating each task, signaled by the color you see, with the best action to take when facing that task, e.g., if red, play arm 1; if green, play arm 2. With the right policy you can usually do much better than you could in the absence of any information distinguishing one bandit task from another.

This is an example of an *associative search* task, so called because it involves both trial-and-error learning in the form of *search* for the best actions and *association* of these actions with the situations in which they are best. Associative search tasks are intermediate between the **n** -armed bandit problem and the full reinforcement learning problem. They are like the full reinforcement learning problem in that they involve learning a policy, but like the **n** -armed bandit problem in that each action affects only the immediate reward. If actions are allowed to affect the *next situation* as well as the reward, then we have the full reinforcement learning problem. We present this problem in the next chapter and consider its ramifications throughout the rest of the book.

Exercise 2.16

Suppose you face a binary bandit task whose true action values change randomly from play to play. Specifically, suppose that for any play the true values of actions **1** and **2** are respectively .1 and .2 with probability .5 (case A), and .9 and .8 with probability .5 (case B). If you are not able to tell which case you face at any play, what is the best expectation of success you can achieve and how should you behave to achieve it? Now suppose that on each play you are told if you are facing case A or case B (although you still don't know the true action values). This is an associative search task. What is the best expectation of success you can achieve in this task, and how should you behave to achieve it?

Next Up Previous
Next: 2.11 Conclusion **Up:** 2 Evaluative Feedback **Previous:** 2.9 Pursuit Methods

---

Richard Sutton
Sat May 31 12:02:11 EDT 1997

2.11 Conclusion

We have presented in this chapter some simple ways of balancing exploration and exploitation. The ϵ -greedy methods simply choose randomly a small fraction of the time, the softmax methods grade their action probabilities according to the current action-value estimates, and the pursuit methods just keep taking steps toward the current greedy action. Are these simple methods really the best we can do in terms of practically useful algorithms? So far, the answer appears to be ``yes''. Despite their simplicity, in our opinion the methods presented in this chapter can fairly be considered the state-of-the-art. There are more sophisticated methods, but their complexity and assumptions make them impractical for the full reinforcement learning problem that is our real focus. Starting in Chapter 5 we present learning methods for solving the full reinforcement learning problem that use in part the simple methods explored in this chapter.

Although the simple methods explored in this chapter may be the best we can do at present, they are far from a fully satisfactory solution to the problem of balancing exploration and exploitation. We conclude this chapter with a brief look at some of the current ideas that, while not as yet practically useful, may point the way toward better solutions.

One promising idea is to use estimates of the uncertainty of the action-value estimates to direct and encourage exploration. For example, suppose there are two actions estimated to have values slightly less than that of the greedy action, but which differ greatly in their degree of uncertainty. One estimate is very certain; perhaps that action has been tried many times and many rewards have been observed. The uncertainty for this action value is so low that its true value is very unlikely to be higher than the value of the greedy action. The other action is known less well, and the estimate of its value is very uncertain. The true value of this action could easily be better than that of the greedy action. Obviously, it makes more sense to explore the second action than the first.

This line of thought leads to *interval estimation* methods. These methods estimate for each action the $(1-\epsilon)$ confidence interval of the action value. That is, rather than learning that the action value is approximately 10, they learn that it is between 9 and 11 with 95% confidence. The action selected is then the action whose confidence interval has the highest upper limit. This encourages exploration of actions that are uncertain *and* have a chance of ultimately being the best action. In some cases one can obtain guarantees that the optimal action has been found with confidence equal to the confidence factor (e.g., the 95%). Interval estimation methods are problematic in practice because of the complexity of the statistical methods used to estimate the confidence intervals. Moreover, the underlying statistical assumptions required by these methods are often not satisfied. Nevertheless, the idea of using confidence intervals, or some other measure of uncertainty, to encourage exploration of particular actions is sound and appealing.

There is also a well-known algorithm for computing the Bayes optimal way to balance exploration and exploitation. This method is computationally intractable when done exactly, but there may be efficient ways to approximate it. In this optimal method we assume that we know the distribution of problem instances, i.e., the probability of each possible set of true action values. Given any action selection, we can then compute the probability of each possible immediate reward and the resultant posterior probability distribution over action values. This evolving distribution becomes the *information state* of the problem. Given a horizon, say 1000 plays, one can consider all possible actions, all possible resulting rewards, all possible next actions, all next rewards, and so on for all 1000 plays. Given the assumptions, the rewards and probabilities of each possible chain of events can be determined, and one need only pick the best. But the tree of possibilities grows extremely rapidly; even if there are only two actions and two rewards the tree will have 2^{2000} leaves. This approach effectively turns the bandit problem into an instance of the full reinforcement learning problem. In the end, we may be able to use reinforcement learning methods to approximate this optimal solution. But that is a topic for current research and beyond the scope of this introductory book.

The classical solution to balancing exploration and exploitation in $\mathbf{n}$ -armed bandit problems is to compute special functions called *Gittins indices*. These provide an optimal solution to a certain kind of bandit problem more general than that considered here, but which assumes the prior distribution of possible problems is known. Unfortunately, this method does not appear to generalize either in its theory or computational tractability to the full reinforcement learning problem that we consider in the rest of the book.

Richard Sutton
Sat May 31 12:02:11 EDT 1997

2.12 Bibliographical and Historical Remarks

2.1

Bandit problems have been studied in statistics, engineering, and psychology. In statistics, bandit problems they fall under the heading of the ``sequential design of experiments," having been introduced by Thompson (1933, 1934) and Robbins (1952), and studied by Bellman (1956). Berry and Fristedt (1985) provide an extensive treatment of bandit problems from the perspective of statistics. Narendra and Thathachar (1989) treat bandit problems from the engineering perspective, providing a good discussion of the various theoretical traditions that have focussed on them. In psychology, bandit problems played roles in statistical learning theory (e.g., Bush and Mosteller, 1958; Estes, 1950).

The term *greedy* is often used in the heuristic search literature (e.g., Pearl, 1984). The conflict between exploration and exploitation is known in control engineering as the conflict between identification (or estimation) and control (e.g., Witten, 1976)). Feldbaum (1965) called it the *dual control* problem, referring to the need to solve the two problems of identification and control simultaneously when trying to control a system under uncertainty. In discussing aspects of genetic algorithms, Holland (1975) emphasized the importance of this conflict, referring to it as the conflict between exploitation and new information.

2.2

Action-value methods for our n-armed bandit problem were first proposed by Thathachar and Sastry (1985). These are often called *estimator algorithms* in the learning automata literature. The term *action value* is due to Watkins (1989). The first to use ϵ -greedy methods may also have been Watkins (1989, p. 187), but the idea is so simple that some earlier use seems likely.

2.3

The term *softmax* for the action selection rule (2.2) is due to Bridle (1990). This rule appears to have been first proposed by Luce (1959). The parameter τ is called temperature in simulated annealing algorithms (Kirkpatrick, Gelatt and Vecchi, 1983).

2.4

The main argument and results in this section were first presented by Sutton (1984). Further analysis of the relationship between evaluation and instruction has been presented by Barto (1985, 1991, 1992), and Barto and Anandan (1985). The unit-square representation of a binary bandit task used in Figure 2.2 has been called a contingency space in experimental psychology (e.g., Staddon, 1983).

Narendra and Thathachar (1989) provide a comprehensive treatment of modern learning automata theory and its applications. They also discuss similar algorithms from the statistical learning theory of psychology. Other methods based on converting reinforcement-learning experience into target actions were developed by Widrow, Gupta, and Maitra (1973) and by Gällmo and Asplund (1995).

2.5 **and 2 .6**

This material falls under the general heading of stochastic iterative algorithms, which is well covered by Bertsekas and Tsitsiklis (1996).

2.8

Reinforcement comparison methods were extensively developed by Sutton (1984) and further refined by Williams (1986, 1992), Kaelbling (1993), and Dayan (1991). These authors analyzed many variations of the idea including various eligibility terms that may significantly improve performance. Perhaps the earliest use of reinforcement comparison was by Barto, Sutton, and Brouwer (1981).

2.9

The pursuit algorithm is due to Thathachar and Sastry (1986).

2.10

The term *associative search* and the corresponding problem were introduced by Barto, Sutton, and Brouwer (1981). The term *associative reinforcement learning* has also been used for associative search (Barto and Anandan, 1985), but we prefer to reserve that term as a synonym for the full reinforcement learning problem (as in Sutton, 1984). We note that Thorndike's Law of Effect (quoted in Chapter 1) describes associative search by referring to the formation of associative links between situations and actions. According to the terminology of operant, or instrumental, conditioning (e.g., Skinner, 1938), a discriminative stimulus is a stimulus that signals the presence of a particular reinforcement contingency. In our terms, different discriminative stimuli correspond to different situations.

2.11

Interval estimation methods are due to Lai (1987) and Kaelbling (1993). Bellman (1956) was the first to show how dynamic programming could be used to compute the optimal balance between exploration and exploitation within a Bayesian formulation of the problem. The survey by Kumar (1985) provides a good discussion of Bayesian and non-Bayesian approaches to these problems. The term *information state* comes from the literature on partially observable MDPs, see, e.g., Lovejoy (1991). The Gittins index approach is due to Gittins and Jones (1974). Duff (1996) showed how it is possible to learn Gittins indices for bandit problems through reinforcement learning.

Richard Sutton
Sat May 31 12:02:11 EDT 1997

...selection.

The difference between instruction and evaluation can be clarified by contrasting two types of function optimization algorithms. One type is used when information about the *gradient* of the function being minimized (or maximized) is directly available. The gradient instructs the algorithm as to how it should move in the search space. The errors used by many supervised learning algorithms are gradients (or approximate gradients). The other type of optimization algorithm uses only function values, corresponding to evaluative information, and has to actively probe the function at additional points in the search space in order to decide where to go next. Classical examples of these types of algorithms are respectively the Robbins-Monro and the Kiefer-Wolfowitz stochastic approximation algorithms (see, e.g., Kashyap, Blaydon, and Fu, 1970).

...probability.

This is actually a considerable simplification of these learning automata algorithms. For example, they are defined as well for **n>2** and often use a different step-size parameter (70#70) on success and on failure. Nevertheless, the limitations identified in this section still apply.

Richard Sutton
Sat May 31 12:02:11 EDT 1997

3 The Reinforcement Learning Problem

In this chapter we introduce the problem that we try to solve in all the rest of the book. For us, this problem defines the field of reinforcement learning: any method that is suited to solving this problem we consider to be a reinforcement learning method.

Our objective in this chapter is to describe the reinforcement learning problem in a broad sense. We try to convey the wide range of possible applications that can be framed as reinforcement learning problems. We also describe mathematically idealized forms of the reinforcement learning problem for which precise theoretical statements can be made. We introduce key elements of the problem's mathematical structure, such as value functions and Bellman equations. As in all of artificial intelligence, there is a tension between breadth of applicability and mathematical tractability. In this chapter we introduce this tension and discuss some of the tradeoffs and challenges that it implies.

Richard Sutton
Sat May 31 13:56:52 EDT 1997

Next Up Previous
Next: 3.2 Goals and Rewards **Up:** 3 The Reinforcement Learning **Previous:** 3 The Reinforcement Learning

3.1 The Agent-Environment Interface

The reinforcement learning problem is meant to be a straightforward framing of the problem of learning from interaction to achieve a goal. The learner and decision-maker is called the *agent*. The thing it interacts with, comprising everything outside the agent, is called the *environment*. These interact continually, the agent selecting actions and the environment responding to those actions and presenting new situations to the agent.

The environment also gives rise to rewards, a special signal whose values the agent tries to maximize over time. A complete specification of an environment defines a *task*, one instance of the reinforcement learning problem.

More specifically, the agent and environment interact at each of a sequence of discrete time steps, $t = 0, 1, 2, 3, \ldots$. At each time step, **t**, the agent receives some representation of the environment's *state*, $s_t \in \mathcal{S}$, where $\mathcal{S}$ is the set of possible states, and on that basis selects an *action*, $a_t \in \mathcal{A}(s_t)$, where $\mathcal{A}(s_t)$ is the set of actions available in state s_t. One time step later, in part as a consequence of its action, the agent receives a numerical *reward*, $r_{t+1} \in \Re$, and finds itself in a new state, s_{t+1}.

Figure 3.1 diagrams the agent-environment interaction.

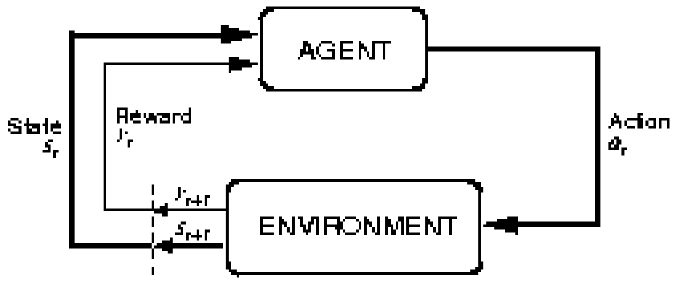


Figure 3.1: The reinforcement learning framework

At each time step, the agent implements a mapping from state representations to probabilities of selecting each possible action. This mapping is called the agent's *policy* and denoted π_t, where $\pi_t(s, a)$ is the probability that $a_t = a$ if $s_t = s$. Reinforcement learning methods specify how the agent changes its policy as a result of its experience. The agent's goal, roughly speaking, is to maximize the total amount of reward it receives over the long run.

This framework is abstract and very flexible, allowing it be applied to many different problems in many different ways. For example, the time steps need not refer to fixed intervals of real time; they can refer to arbitrary successive stages of decision making and acting. The actions can be low-level controls such as the voltages applied to the motors of a robot arm, or high-level decisions such as whether or not to have lunch or to go to graduate school. Similarly, the states can take a wide variety of forms. They can be completely determined by low-level sensations, such as direct sensor readings, or they can be more high-level and abstract, such as symbolic descriptions of objects in a room. Some of what makes up a state could be based on memory of past sensations or even be entirely mental or subjective. For example, an agent could be in ``the state'' of not being sure where an object is, or of having just been ``surprised'' in some clearly defined sense. Similarly, some actions might be totally mental or computational. For example, some actions might control what an agent chooses to think about, or where it focuses its attention. In general, actions can be any decisions we want to learn how to make, and the state representations can be anything we can know that might be useful in making them.

In particular, it is a mistake to think of the interface between the agent and the environment as the physical boundary of a robot's or animal's body. Usually, the boundary is drawn closer to the agent than that. For example, the motors and mechanical linkages of a robot and its sensing hardware should usually be considered parts of the environment rather than parts of the agent. Similarly, if we apply the framework to a person or animal, the muscles, skeleton, and sensory organs should all be considered part of the environment. Rewards too are presumably computed inside the physical bodies of natural and artificial learning systems, but are considered external to the agent.

The general rule we follow is that anything that cannot be changed arbitrarily by the agent is considered to be outside of it and thus part of its environment. We do not assume that everything in the environment is unknown to the agent. For example, the agent often knows quite a bit about how its rewards are computed as a function of its actions and the states in which they are taken. But we always consider the reward computation to be external to the agent because it defines the task facing the agent and thus must be beyond its ability to change arbitrarily. In fact, in some cases the agent may know *everything* about how its environment works and still face a difficult reinforcement learning task, just as we may know exactly how a puzzle like Rubik's cube works, but still be unable to solve it. The agent-environment boundary represents the limit of the agent's *absolute control,* not of its knowledge.

The agent-environment boundary can be placed at different places for different purposes. In a complicated robot, many different agents may be operating at once, each with its own boundary. For example, one agent may make high-level decisions which form part of the states faced by a lower-level agent that implements the high-level decisions. In practice, the agent-environment boundary is determined once one has selected particular states, actions, and rewards, and thus identified a specific decision-making task of interest.

The reinforcement learning framework is a considerable abstraction of the problem of goal-directed learning from interaction. It proposes that whatever the details of the sensory, memory, and control apparatus, and whatever objective one is trying to achieve, any problem of learning goal-directed behavior can be reduced to three signals passing back and forth between an agent and its environment: one signal to represent the choices made by the agent (the actions), one signal to represent the basis on which the choices are made (the states), and one signal to define the agent's goal (the rewards). This framework may not be sufficient to usefully represent all decision-learning problems, but it has proven itself widely useful and applicable.

Of course, the state and action representations vary greatly from application to application and strongly affect performance. In reinforcement learning, as in other kinds of learning, such representational choices are at present more art than science. In this book we offer some advice and examples regarding good choices of state and action representations, but our primary focus is on the general principles useful for learning how to behave once the state and action representations have been selected.

Example 3.1

Bioreactor. Suppose reinforcement learning is being applied to determine moment-by-moment temperatures and stirring rates for a bioreactor (a large vat of nutrients and bacteria used to produce useful chemicals). The actions in such an application might be *target* temperatures and *target* stirring rates that are passed to lower-level control systems which, in turn, directly activate heating elements and motors to attain the targets. The state representation is likely to be thermocouple and other sensory readings, perhaps filtered and delayed, plus symbolic inputs representing the ingredients in the vat and the target chemical. The reward might be a moment-by-moment measure of the rate at which the useful chemical is produced by the bioreactor. Notice that here each state representation is a list, or vector, of sensor readings and symbolic inputs, and each action is a vector consisting of a target temperature and a stirring rate. It is typical of reinforcement learning tasks to have such state and action representations with this kind of structure. Rewards, on the other hand, are always single numbers. ◇

Example 3.2

Pick-and-Place Robot. Consider using reinforcement learning to control the motion of a robot arm in a repetitive pick-and-place task. If we want to learn movements that are fast and smooth, the learning agent will have to directly control the motors and have low-latency information about the current positions and velocities of the mechanical linkages. The actions in this case might be the currents applied to each motor at each joint, and the state representation might be the latest readings of joint angles and velocities. The reward might be simply **+1** for each object successfully picked up and placed. To encourage smooth movements, on each time step a small, negative reward can be given dependent on a measure of the moment-to-moment ``jerkiness" of the motion. ◇

Example 3.3

Recycling Robot. A mobile robot has the job of collecting empty soda cans in an office environment. It has sensors for detecting cans, an arm and gripper that that can pick them up and place them in an onboard bin, and it runs on rechargable batteries. The robot's control system has components for interpreting sensory information, for navigating, and for controlling the arm and gripper. High-level decisions about how to search for cans are made by a reinforcement learning agent on the basis of the current charge level of the battery. This agent has to decide whether the robot should 1) actively search for a can for a certain period of time, 2) remain stationary and wait for someone to bring it a can, or 3) head back to its home base to recharge its batteries. This decision has to be made either periodically or whenever certain events occur, such as finding an empty can. The agent therefore has three actions, and its state is determined by the state of the battery. The rewards might be zero most of the time, but then become positive when the robot secures an empty can, or large and negative if the battery runs all the way down. In this example, the reinforcement learning agent is not the entire robot. The states it monitors describe conditions within the robot itself, not conditions of the robot's external environment. The agent's environment therefore includes the rest of the robot, which might contain other complex decision-making systems, as well as the robot's external environment. ◇

Exercise 3.1

Devise three example tasks of your own that fit into the reinforcement learning framework, identifying for each its state representations, actions, and rewards. Make the three examples as *different* from each other as possible. The framework is very abstract and flexible and can be applied in many different ways. Stretch its limits in some way in at least one of your examples.

Exercise 3.2

Is the reinforcement learning framework adequate to usefully represent *all* goal-directed learning tasks? Can you think of any clear exceptions?

Exercise 3.3

Consider the problem of driving. You could define the actions in terms of the accelerator, steering wheel, and brake, i.e., where your body meets the machine. Or you could define them farther out, say where the rubber meets the road, considering your actions to be tire torques. Or, you could define them farther in, say where your brain meets your body, the actions being muscle twitches to control your limbs. Or you could go to a really high level and say that your actions are your choices of *where* to drive. What is the right level, the right place to draw the line between agent and environment? On what basis is one location of the line to be preferred over another? Is there any fundamental reason for preferring one location over another, or is it just a free choice?

Next Up Previous
Next: 3.2 Goals and Rewards **Up:** 3 The Reinforcement Learning **Previous:** 3 The Reinforcement Learning

Richard Sutton
Sat May 31 13:56:52 EDT 1997

3.2 Goals and Rewards

In reinforcement learning, the purpose or goal of the agent is formalized in terms of a special signal, called the *reward*, that passes from the environment to the agent. The reward is just a single number whose value varies from step to step. Informally, the agent's goal is to maximize the total amount of reward it receives. This means maximizing not just immediate reward, but cumulative reward in the long run.

The use of a reward signal to formalize the idea of a goal is one of the most distinctive features of reinforcement learning. Although this way of formulating goals might at first appear limiting, in practice it has proven to be flexible and widely applicable. The best way to see this is to consider examples of how it has been, or could be, used. For example, to make a robot learn to walk, researchers have provided reward on each time step proportional to the robot's forward motion. In making a robot learn how to escape from a maze, the reward is often zero until it escapes, when it becomes **+1**. Another common approach in maze learning is to give a reward of **-1** for every time step that passes prior to escape; this encourages the agent to escape as quickly as possible. To make a robot learn to find and collect empty soda cans for recycling, one might give it a reward of zero most of the time, and then a reward of **+1** for each can collected (and confirmed as empty). One might also want to give the robot negative rewards when it bumps into things, or when somebody yells at it! For an agent to learn to play checkers or chess, the natural rewards are **+1** for winning, **-1** for losing, and 0 for drawing and for all non-terminal positions.

You can see what is happening in all of these examples. The agent always learns to maximize its reward. If we want it to do something for us, we must provide rewards to it in such a way that in maximizing them the agent will also achieve our goals. It is thus critical that the rewards we set up truly indicate what we want accomplished. In particular, the reward signal is not the place to impart to the agent prior knowledge about *how* to achieve what we want it to do.

For example, a chess playing agent should be rewarded only for actually winning, not for achieving subgoals such taking its opponent's pieces or gaining control of the center of the board. If achieving these sorts of subgoals were rewarded, then the agent might find a way to achieve them without achieving the real goal. For example, it might find a way to take the opponent's pieces even at the cost of losing the game. The reward signal is your way of communicating to the robot *what* you want it to achieve, not *how* you want it achieved.

Newcomers to reinforcement learning are sometimes surprised that the rewards---which define of the goal of learning---are computed in the environment rather than in the agent. Certainly most ultimate goals for animals are recognized by computations occurring inside their bodies, e.g., by sensors for recognizing food and hunger, pain and pleasure, etc. Nevertheless, as we discussed in the previous section, one can simply redraw the agent-environment interface such that these parts of the body are considered to be outside of the agent (and thus part of the agent's environment). For example, if the goal concerns a robot's internal energy reservoirs, then these are considered to be part of the environment; if the goal concerns the positions of the robot's limbs, then these too are considered to be part of the environment---the agent's boundary is drawn at the interface between the limbs and their control systems. These things are considered internal to the robot but external to the learning agent. For our purposes, it is convenient to place the boundary of the learning agent not at the limit of its physical body, but at the limit of its control.

The reason we do this is that the agent's ultimate goal should be something over which it has imperfect control: it should not be able, for example, to simply *decree* that the reward has been received in the same way that it might arbitrarily change its actions.

Therefore, we place the reward source outside of the agent. This does not preclude the agent from defining for itself a kind of internal reward, or a sequence of internal rewards. Indeed, this is exactly what many reinforcement learning methods do.

Next Up Previous

Next: 3.3 Returns **Up:** 3 The Reinforcement Learning **Previous:** 3.1 The Agent-Environment Interface

---

Richard Sutton
Sat May 31 13:56:52 EDT 1997

3.3 Returns

So far we have been imprecise regarding the objective of learning. We have said that the agent's goal is to maximize the reward it receives in the long run. How might this be formally defined? If the sequence of rewards received after time step **t** is denoted $r_{t+1}, r_{t+2}, r_{t+3}, \ldots$, then what precise aspect of this sequence do we wish to maximize? In general, we seek to maximize the *expected return*, where the return, R_t, is defined as some specific function of the reward sequence. In the simplest case the return is just the sum of the rewards:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \cdots + r_T, \qquad (3.1)$$

where **T** is a final time step. This approach makes sense in applications in which there is a natural notion of final time step, that is, when the agent-environment interaction breaks naturally into subsequences, which we call *episodes*, such as plays of a game, trips through a maze, or any sort of repeated interactions. Each episode ends in a special state called the *terminal state*, followed by a reset to a standard starting state or to a standard distribution of starting states. Tasks with episodes of this kind are called *episodic tasks*. In episodic tasks we sometimes need to distinguish the set of all non-terminal states, denoted $\mathcal{S}$, and the set of all states plus the terminal state, denoted $\mathcal{S}^+$.

On the other hand, in many cases that the agent-environment interaction does not break naturally into identifiable episodes, but simply goes on and on without limit. For example, this would be the natural way to formulate a continual process-control task, or an application to a robot with a long lifespan. We call these *continual tasks*. The return formulation (3.1) is problematic for continual tasks because the final time step would be $T = \infty$, and the return, which is what we are trying to maximize, could itself easily be infinite. (For example, suppose the agent receives a reward of **+1** at each time step.) Thus, in this book we use a definition of return that is slightly more complex conceptually but much simpler mathematically.

The additional concept that we need is that of *discounting*. According to this approach, the agent tries to select actions so that the sum of the discounted rewards it receives over the future is maximized. In particular, it chooses a_t to maximize the expected *discounted return*:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \qquad (3.2)$$

where γ is a parameter, $0 \leq \gamma \leq 1$, called the *discount rate*.

The discount rate determines the present value of future rewards: a reward received **k** time steps in the future is worth only γ^{k-1} times what it would be worth if it were received immediately. If $\gamma < 1$, the infinite sum has a finite value as long as the reward sequence $\{r_k\}$ is bounded. If $\gamma = 0$, the agent is ``myopic" in being only concerned with maximizing immediate rewards: its objective in this case is to learn how to choose a_t so as maximize only r_{t+1}. If each of the agent's actions happened only to influence the immediate reward, not future rewards as well, then a myopic agent could maximize (3.2) by separately maximizing each immediate reward. But in general, acting to maximize immediate reward can reduce access to future rewards so that the return may actually be reduced. As γ approaches one, the objective takes future rewards into account more strongly: the agent becomes more far-sighted.

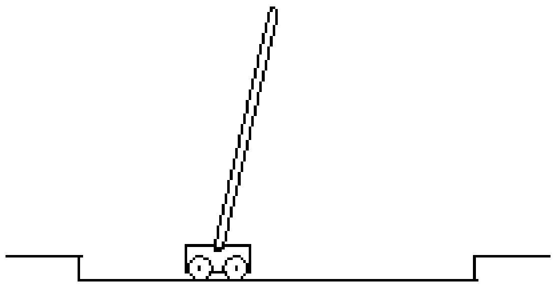


Figure 3.2: The pole-balancing task

Example 3.4

Pole Balancing. Figure 3.2 shows a task that served as an early illustration of reinforcement learning. The objective here is to apply forces to a cart moving along a track so as to keep a pole hinged to the cart from falling over. A failure is said to occur if the pole falls past a given angle from vertical or if the cart exceeds the limits of the track. The pole is reset to vertical after each failure. This task could be treated as episodic, where the natural episodes are the repeated attempts to balance the pole. The reward in this case could be **+1** for every time step on which failure did not occur, so that the return at each time would be the number of steps until failure. Alternatively, we could treat pole balancing as a continual task, using discounting. In this case the reward would be **-1** on each failure and zero at all other times. The return at each time would then be related to $-\gamma^k$, where **k** is the number of time steps before failure. In either case, the return is maximized by keeping the pole balanced for as long as possible. $\diamond$

Exercise 3.4

Suppose you treated pole-balancing as an episodic task but also used discounting, with all rewards zero except for a **-1** upon failure. What then would the return be at each time? How does this return differ from that in the discounted, continual formulation of this task?

Exercise 3.5

Imagine that you are designing a robot to run a maze. You decide to give it a reward of **+1** for escaping from the maze and a reward of zero at all other times. The task seems to break down naturally into episodes---the successive runs through the maze---so you decide to treat it as an episodic task, where the goal is to maximize expected total reward (3.1). After running the learning agent for a while, you find that it is showing no improvement in escaping from the maze. What is going wrong? Have you effectively communicated to the agent what you want it to achieve?

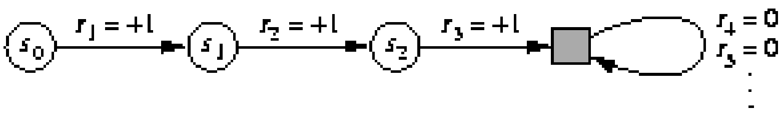
Richard Sutton
Sat May 31 13:56:52 EDT 1997

3.4 A Unified Notation for Episodic and Continual Tasks

In the preceding section we described two kinds of reinforcement learning tasks, one in which the agent-environment interaction naturally breaks down into a sequence of separate episodes (episodic tasks), and one in which it does not (continual tasks). The former case is mathematically easier because each action affects only the finite number of rewards subsequently received during the episode. In this book we consider sometimes one kind of problem and sometimes the other, but often both. It is therefore useful to establish one notation that enables us to talk precisely about both cases simultaneously.

To be precise about episodic tasks requires some additional notation. Rather than one long sequence of time steps, we need to consider a series of episodes, each of which consists of a finite sequence of time steps. We number the time steps of each episode starting anew from zero. Therefore, we have to refer not just to s_t, the state representation at time **t**, but to $s_{t,i}$, the state representation at time **t** of episode **i** (and similarly for $a_{t,i}$, $r_{t,i}$, $\pi_{t,i}$, T_i, etc.). However, it turns out that when we discuss episodic tasks we will almost never have to distinguish between different episodes. We will almost always be considering a particular single episode, or stating something that is true for all episodes. Accordingly, in practice we will almost always abuse notation slightly by dropping the explicit reference to episode number. That is, we will simply write s_t to refer to $s_{t,i}$, etc.

We need one other convention to obtain a single notation that covers both episodic and continual tasks. We have defined the return as a sum over a finite number of terms in one case (3.1) and as a sum over an infinite number of terms in the other (3.2). These can be unified by considering episode termination to be the entering of a special *absorbing state* that transitions only to itself and which generates only rewards of zero. For example, consider the state transition diagram:



Here the solid square represents the special absorbing state corresponding to the end of an episode. Starting from s_0 we get the reward sequence $+1, +1, +1, 0, 0, 0, \ldots$. Summing these up, we get the same return whether we sum over the first **T** rewards (here **T=3**) or we sum over the full infinite sequence. This remains true even if we introduce discounting. Thus, we can define the return, in general, as

$$R_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1}, \tag{3.3}$$

including the possibility that either $T = \infty$ or $\gamma = 1$ (but not both), and using the convention of omitting episode numbers when they are not needed. We use these conventions throughout the rest of the book to simplify notation and to express the close parallels between episodic and continual tasks.

Next Up Previous
Next: 3.5 The Markov Property **Up:** 3 The Reinforcement Learning **Previous:** 3.3 Returns

Richard Sutton
Sat May 31 13:56:52 EDT 1997

3.5 The Markov Property

In the reinforcement learning framework, the agent makes its decisions on the basis of a signal from the environment called the environment's *state*. In this section we discuss what is required of the state signal, and what kind of information we should and should not expect it to provide. In particular, we formally define a property of environment's and their state signals that is of particular interest, called the Markov property.

In this book, by the state we mean just whatever information is available to the agent. We assume that the state is given by some preprocessing system which is nominally part of the environment. We do not address the issues of constructing, changing, or learning the state signal in this book. We take this approach not because we consider state representation to be unimportant, but in order to focus fully on the decision-making issues. In other words, our main concern is not with designing the state signal, but on deciding what action to take as a function of whatever state signal is available.

Certainly the state signal should include immediate sensations such as sensory measurements, but it can contain much more than that. State representations can be highly processed versions of original sensations, or they can be complex structures built up over time from the sequence of sensations. For example, as people we move our eyes over a scene, with only a tiny spot (the fovea) visible in detail at any one time, and yet build up a rich and detailed representation of a scene. Or, more obviously, we can look at an object, then look away, and still know that it is there. We can hear the word ``Yes,'' and consider ourselves to be in totally different states depending on the question that came before and which is no longer audible. At a more mundane level, a control system can measure position at two different times to produce a state representation including velocity. In all of these cases the state is constructed and maintained based on immediate sensations together with the previous state or some other memory of past sensations. In this book, we do not explore how that is done, but certainly it could be and has been done. There is no reason to restrict the state representation to immediate sensations; in typical applications we should expect the state representation to be able to inform the agent of more than that.

On the other hand, the state signal should not be expected to inform the agent of everything about the environment, or even everything that could be useful to it in making decisions. If the agent is playing blackjack, we should not expect it to know what the next card is in the deck. If the agent is answering the phone, we should not expect it to know in advance who the caller is. If the agent is a paramedic called to a road accident, we should not expect it to know immediately the internal injuries of an unconscious victim. In all of these cases there is hidden state information in the environment, and that information would be useful if the agent knew it, but the agent could not know it because it has never received any relevant sensations. In short, we don't fault an agent for not knowing something that matters, but only for having known something and then forgotten it!

What we would like, ideally, is a state signal that summarizes past sensations compactly yet in such a way that all relevant information is retained. This normally requires more than the immediate sensations, but never more than the complete history of all past sensations. A state signal that succeeds in retaining all relevant information is said to be *Markov*, or to have *the Markov property* (we define this formally below). For example, a checkers position---the current configuration of all the pieces on the board---would serve as a Markov state because it summarizes everything important about the complete sequence of positions that led to it. Much of the information about the sequence is lost, but all that really matters for the future of the game is retained. Similarly, the current position and velocity of a cannon ball is all that matters for its future flight. It doesn't matter how that position and velocity came about. This is sometimes also referred to as an ``independence of path'' property because all that matters is in the current state signal; its meaning is independent of the ``path,'' or history, of signals that have led up to it.

We now formally define the Markov property for the reinforcement learning problem. To keep the mathematics simple, we assume here that their are a finite number of states and reward values. This enables us to work in terms of sums and probabilities rather than integrals and probability densities, but the argument could easily be extended to include continuous states and rewards. Consider how a general environment might respond at time **t+1** to the action taken at time **t**. In the most general, causal case this response may depend on everything that has happened earlier. In this case the dynamics can be defined only by specifying the complete probability distribution:

$$Pr\left\{s_{t+1} = s', r_{t+1} = r \middle| s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0 \right|, \}$$

(3.4)

for all s', **r**, and all possible values of the past events: $s_t, a_t, r_t, \ldots, r_1, s_0, a_0$. If the state signal has the *Markov property*, on the other hand, then the environment's response at **t+1** depends only on the state and action representations at **t**, in which case the environment's dynamics can be defined by specifying only

$$Pr\{s_{t+1} = s', r_{t+1} = r | s_t, a_t\}, \qquad (3.5)$$

for all s', r, s_t, and a_t. In other words, a state representation has the Markov property, and is a Markov state, if and only if (3.5) is equal to (3.4) for all s', r, and histories, $s_t, a_t, r_t, \ldots, r_1, s_0, a_0$.

If an environment has the Markov property, then its one-step dynamics (3.5) enable us to predict the next state and expected next reward given the current state and action. One can show that by iterating this equation one can predict all future states and expected rewards from knowledge only of the current state as well as is possible given the complete history up to the current time. It also follows that Markov states provide the best possible basis for choosing actions. That is, the best policy for choosing actions as a function of a Markov state signal is just as good as the best policy for choosing actions as a function of complete histories.

Even when the state signal is non-Markov, it is still appropriate to think of the state in reinforcement learning as an approximation to a Markov state. In particular, we always want the state to be a good basis for predicting future rewards and for selecting actions. In cases in which a model of the environment is learned (see Chapter 9), we also want the state to be a good basis for predicting subsequent states. Markov states provide an unsurpassed basis for doing all of these things. To the extent that the state approaches the ability of Markov states in these ways, one will obtain better performance from reinforcement learning systems. For all of these reasons, it is useful to think of the state at each time step as an approximation to a Markov state, although one should remember that it may not fully satisfy the Markov property.

The Markov property is important in reinforcement learning because decisions and values are assumed to be a function only of the current state. In order for these to be effective and informative, the state representation must be informative. All of the theory presented in this book assumes Markov state signals. This means that not all the theory strictly applies to cases in which the Markov property does not strictly apply. However, the theory developed for the Markov case still helps us to understand the behavior of the algorithms, and the algorithm can and have been successfully applied to many tasks that were not strictly Markov. A full understanding of the theory of the Markov case is an essential foundation for extending it to the more complex and realistic non-Markov case. Finally, we note that the assumption of Markov state representations is not unique to reinforcement learning, but is also present in most if not all other approaches to artificial intelligence.

Example 3.5

Pole-Balancing State. In the pole-balancing task introduced earlier, a state signal would be Markov if it exactly specified, or made it possible to exactly reconstruct, the position and velocity of the cart along the track, the angle between the cart and the pole, and the rate at which this angle is changing (the angular velocity). In an idealized cart-pole system, this information would be sufficient to exactly predict the future behavior of the cart and pole, given the actions taken by the controller. In practice, however, it is never possible to know this information exactly because any real sensor would introduce some distortion and delay in its measurements. Furthermore, in any real cart-pole system there are always other effects, such as the bending of the pole, the temperatures of the wheel and pole bearings, and various forms of backlash, which slightly affect the behavior of the system. These factors would cause violations of the Markov property if the state signal were only the positions and velocities of the cart and the pole.

However, often the positions and velocities serve quite well as states. Some early studies of learning to solve the pole-balancing task used a very coarse state signal which divided cart positions into three regions: right, left, and middle (and similar rough quantizations of the other three intrinsic state variables). This distinctly non-Markov state was sufficient to allow the task to be solved easily by reinforcement learning methods. In fact, this coarse representation may have facilitated rapid learning by forcing the learning agent to ignore fine distinctions that would not have been useful in solving the task. ◇

Example 3.6

Draw Poker. In draw poker, each player is dealt a hand of five cards. There is a round of betting, in which each player exchanges some of his cards for new ones, and then there is a final round of betting. At each round of betting, each player must match the highest bets of the other players or else drop out (fold). After the second round of betting, the player with the best hand who has not folded is the winner and collects all the bets.

The state signal in draw poker is different for each player. Each player knows the cards in his own hand, but can only guess at those in the other players' hands. A common mistake is to think that a Markov state signal should include the contents of all the players' hands and the cards remaining in the deck. In a fair game, however, we assume that the players are in principle unable to determine these things from their past observations. If a player did know them, then she could predict some future events (such as the cards one could exchange for) *better* than by remembering all past observations.

In addition to knowledge of one's own cards, the state in draw poker should include the bets and the numbers of cards drawn by the other players. For example, if one of the other players drew three new cards, you may suspect he retained a pair and adjust your guess at the strength of his hand accordingly. The players' bets also influence your assessment of their hands. In fact, much of your past history with these particular players is part of the

Markov state. Does Ellen like to bluff, or does she play conservatively? Does her face or demeanor provide clues to the strength of her hand? How does Joe's play change when it is late at night, or when he has already won a lot of money?

Although everything ever observed about the other players may have an effect on the probabilities that they are holding various kinds of hands, in practice this is far too much to remember and analyze, and most of it will have no clear effect on one's predictions and decisions. Very good poker players are adept at remembering just the key clues, and at sizing up new players quickly, but no one remembers everything that is relevant. As a result, the state representations people use to make their poker decisions are undoubtedly sub-Markov, and the decisions themselves are presumably imperfect. Nevertheless, people still make very good decisions in such tasks. We conclude that the inability to have access to a *perfect* Markov state representation is probably not a severe problem for a reinforcement learning agent. $\diamond$

Exercise 3.6

Broken Vision System. Imagine that you are a vision system. When you are first turned on for the day, an image floods into your camera. You can see lots of things, but not all things. You can't see objects that are occluded, and of course you can't see objects that are behind you. After seeing that first scene, do you have access to the Markov state of the environment? Suppose your camera was broken that day and you received no images at all, all day. Would you have access to the Markov state then?

Next Up Previous
Next: 3.6 Markov Decision Processes **Up:** 3 The Reinforcement Learning **Previous:** 3.4 A Unified Notation

---

Richard Sutton
Sat May 31 13:56:52 EDT 1997

3.6 Markov Decision Processes

A reinforcement learning task that satisfies the Markov property is called a *Markov decision process*, or *MDP*. If the state and action spaces are finite, then it is called a *finite Markov decision process (finite MDP)*. Finite MDPs are particularly important to the theory of reinforcement learning. We treat them extensively throughout this book; they are all you need to understand for 90% of modern reinforcement learning.

A particular finite MDP is defined by its state and action sets and by the one-step dynamics of the environment. Given any state and action, **s** and **a**, the probability of each possible next state, s', is

$$\mathcal{P}^a_{ss'} = Pr\left\{s_{t+1}=s' \middle| s_t=s, a_t=a\right\}. \tag{3.6}$$

Similarly, given any current state and action, **s** and **a**, together with any next state, s', the expected value of the next reward is

$$\mathcal{R}^a_{ss'} = E\left\{r_{t+1} \middle| s_t=s, a_t=a, s_{t+1} = s'\right\}. \tag{3.7}$$

These two quantities, $\mathcal{P}^a_{ss'}$ and $\mathcal{R}^a_{ss'}$, completely specify the most important aspects of the dynamics of a finite MDP (only information about the distribution of rewards around the expected value is lost). Most of the theory we present in the rest of this book implicitly assumes the environment is a finite MDP.

Example 3.7

Recycling Robot MDP. The recycling robot (Example 3.3) can be turned into a simple example of an MDP by simplifying it and providing some more details. (Our aim is to produce a simple example, not a particularly realistic one.) Recall that the agent makes a decision at times determined by external events (or by other parts of the robot's control system). At each such time the robot decides whether it should 1) actively search for a can,

2) remain stationary and wait for someone to bring it a can, or 3) go back to home base to recharge its battery. Suppose the environment works as follows. The best way to find cans is to actively search for them, but this runs down the robot's battery, whereas just waiting does not. Whenever the robot is searching, the possibility exists that its battery will become depleted. In this case the robot must shut down and wait to be rescued (producing a low reward).

The agent makes its decisions solely on the basis of the energy level of the battery. It can distinguish two levels, `high` and `low`, so that the state set is $\mathcal{S} = \{\texttt{high}, \texttt{low}\}$. Let us call the possible decisions---the agent's actions--- `wait`, `search`, and `recharge`. When the energy level is `high`, recharging would always be foolish, so we do not include it in the action set for this state. The agent's action sets are:

$$\begin{aligned} \mathcal{A}(\texttt{high}) &= \{\texttt{search}, \texttt{wait}\} \\ \mathcal{A}(\texttt{low}) &= \{\texttt{search}, \texttt{wait}, \texttt{recharge}\}. \end{aligned}$$

If the energy level is `high`, then a period of active search can always be completed without risk of depleting the battery. A period of searching beginning with a `high` energy level leaves the energy level `high` with probability α and reduces it to `low` with probability $1-\alpha$. On the other hand, a period of searching undertaken when the energy level is `low` leaves it `low` with probability β and depletes the battery with probability $1-\beta$. In the latter case, the robot must be rescued, but then the battery is then recharged back to `high`. Each can collected by the robot counts as a unit reward, whereas a reward of **-3** results whenever the robot has to be rescued. Let $\mathcal{R}^{\texttt{search}}$ and $\mathcal{R}^{\texttt{wait}}$, with $\mathcal{R}^{\texttt{search}} > \mathcal{R}^{\texttt{wait}}$, respectively denote the expected number of cans the robot will collect (and hence the expected reward) while searching and while waiting. Finally, to keep things simple, suppose that no cans can be collected during a run home for recharging, and that no cans can be collected on a step in which the battery is depleted. This system is then a finite MDP, and we can write down the transition probabilities and the expected rewards, as in Table 3.1.

$s = s_t$	$s' = s_{t+1}$	$a = a_t$	$\mathcal{P}^a_{ss'}$	$\mathcal{R}^a_{ss'}$
high	high	search	α	$\mathcal{R}^{\mathtt{search}}$
high	low	search	$1-\alpha$	$\mathcal{R}^{\mathtt{search}}$
low	high	search	$1-\beta$	-3
low	low	search	β	$\mathcal{R}^{\mathtt{search}}$
high	high	wait	1	$\mathcal{R}^{\mathtt{wait}}$
high	low	wait	0	$\mathcal{R}^{\mathtt{wait}}$
low	high	wait	0	$\mathcal{R}^{\mathtt{wait}}$
low	low	wait	1	$\mathcal{R}^{\mathtt{wait}}$
low	high	recharge	1	0
low	low	recharge	0	0.

Table 3.1: Transition probabilities and expected rewards for the finite MDP of the recycling-robot example. There is a row for each possible combination of current state, **s**, next state,s', and action possible in the current state, $a \in \mathcal{A}(s)$.

A *transition graph* is a useful way to summarize the dynamics of a finite MDP. Figure 3.3 shows the transition graph for the recycling robot example. There are two kinds of nodes: *state nodes* and *action nodes*. There is a state node for each possible state (a large open circle labeled by the name of the state), and an action node for each state-action pair (s, a) (a small solid circle labeled by **a** and connected by a line to the state node **s**). Starting in state **s** and taking action **a** moves you along the line from state node **s** to action node (s, a). Then the environment responds with a transition to the next state's node via one of the arrows leaving action node (s, a). Each arrow corresponds to a triple (s, s', a), where s' is the next state, and we label the arrow with the transition probability, $\mathcal{P}^a_{ss'}$, and the expected reward for that transition, $\mathcal{R}^a_{ss'}$. Note that the transition probabilities labeling the arrows leaving an action node always sum to one.

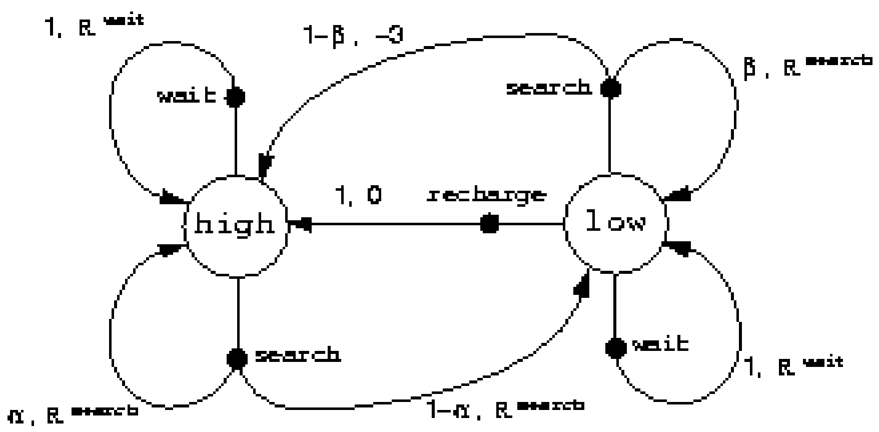


Figure 3.3: Transition graph for the recycling-robot example. The graph has a node for each state, shown as a large circle, and a node for each state-action pair, shown as a small solid circle. Each arrow is labeled by the transition probability and the expected reward for that transition.

◇

Exercise 3.7

Assuming a finite MDP with a finite number of reward values, write an equation for the transition probabilities and the expected rewards in terms of the joint conditional distribution in (3.5).

Richard Sutton
Sat May 31 13:56:52 EDT 1997

3.7 Value Functions

Almost all reinforcement learning algorithms are based on estimating *value functions*---functions of states (or of state-action pairs) that estimate *how good* it is for the agent to be in a given state (or how good it is to perform a given action in a given state). The notion of ``how good" here is defined in terms of future rewards that can be expected, or, to be precise, in terms of expected return. Of course the rewards the agent can expect to receive in the future depends on what actions it will take. Accordingly, value functions are defined with respect to particular policies.

Recall that a policy, π, is a mapping from states, $s \in \mathcal{S}$, and actions, $a \in \mathcal{A}(s)$, to the probability $\pi(s, a)$ of taking action **a** when in state **s**. Informally, the *value* of a state **s** under a policy π, denoted $V^\pi(s)$, is the expected return when starting in **s** and following π thereafter. For MDPs, we can define $V^\pi(s)$ formally as:

$$V^\pi(s) = E_\pi\{R_t | s_t = s\} = E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, s_t = s\right\}, \qquad (3.8)$$

where $E_\pi\{\}$ denotes the expected value given that the agent follows policy π, and **t** is any time step. Note that the value of the terminal state, if any, is always zero. We call the function V^π the *state-value function for policy* π.

Similarly, we define the value of taking action **a** in state **s** under a policy π, denoted $Q^\pi(s, a)$, as the expected return starting from **s**, taking the action **a**, and thereafter following policy π:

$$Q^\pi(s,a) = E_\pi\{R_t | s_t = s, a_t = a\}$$
$$= \mathbf{E}_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, s_t = s, a_t = a\right\}. (3.9)$$

We call Q^π the *action-value function for policy* π.

The value functions V^π and Q^π can be estimated from experience. For example, if an agent follows policy π and maintains an average for each state encountered of the actual returns that have followed that state, then the average will converge to the state's value, $V^\pi(s)$, as the number of times that state is encountered approaches infinity. If separate averages are kept for each action taken in a state, then these averages will similarly converge to the action values, $Q^\pi(s,a)$. We call estimation methods of this kind *Monte Carlo methods* because they involve averaging over many random samples of actual returns. These kinds of methods are presented in Chapter 5. Of course if there are very many states, then it may not be practical to keep separate averages for each state individually. Instead, the agent would have to maintain V^π and Q^π as parameterized functions and adjust the parameters to better match the observed returns. This can also produce accurate estimates, although much depends on the nature of the parameterized function approximation (Chapter 8).

A fundamental property of value functions used throughout reinforcement learning and dynamic programming is that they satisfy particular recursive relationships. For any policy π and any state **s**, the following consistency condition holds between the value of **s** and the value of its possible successor states:

$$\begin{aligned}
V^\pi(s) &= E_\pi\{R_t \mid s_t = s\} \\
&= E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \;\middle|\; s_t = s\right\} \\
&= E_\pi\left\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \;\middle|\; s_t = s\right\} \\
&= \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \;\middle|\; s_{t+1} = s'\right\}\right] \\
&= \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')\right], \qquad (3.10)
\end{aligned}$$

where it is implicit that the actions, **a**, are taken from the set $\mathcal{A}(s)$, and the next states, s', are taken from the set $\mathcal{S}$, or from $\mathcal{S}^+$ in the case of an episodic problem. The last equation is the *Bellman equation for* V^π. It expresses a relationship between the value of a state and the values of its successor states. Think of looking ahead from one state to its possible successor states, as suggested by Figure 3.4a. Each open circle represents a state and each solid circle represents a state-action pair. Starting from state **s**, the root node at the top, the agent could take any of some set of actions---three are shown in Figure 3.4a. From each of these, the environment could respond with one of several next states, s', along with a reward, **r**. The Bellman equation (3.10) simply averages over all the possibilities, weighting each by its probability of occurring. It states that the value of the start state **s** must equal the (discounted) value of the expected next state, plus the reward expected along the way.

The value function V^π is the unique solution to its Bellman equation. We show in subsequent chapters how this Bellman equation forms the basis of a number of ways to compute, approximate, and learn V^π. We call diagrams like those shown in Figure 3.4 *backup diagrams* because they diagram relationships that form the basis of the update or *backup* operations that are at the heart of all reinforcement learning methods. These operations transfer value information *back* to a state (or a state-action pair) from its successor states (or state-action pairs). We use backup diagrams throughout the book to provide graphical summaries of the algorithms we discuss. (Note that unlike state-transition diagrams, the state nodes of backup diagrams do not necessarily represent distinct states; for example, a state might be its own successor. We also omit explicit arrowheads because time always flows downward in a backup diagram.)

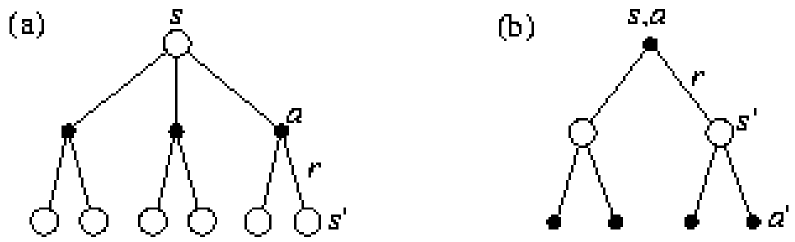
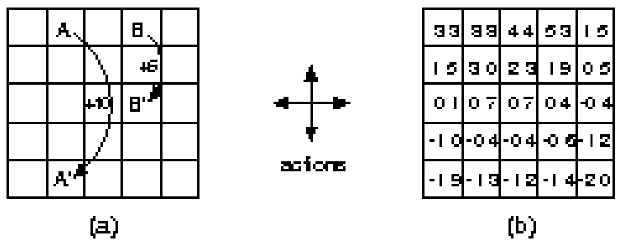


Figure 3.4: Backup diagrams for a) V^π and b) Q^π. The top node is the root node. Below it are all possible actions and states that might occur.

Example 3.8

Gridworld. Figure 3.5a uses a rectangular grid to illustrate value functions for a simple finite MDP. The cells of the grid correspond to the states of the environment. At each cell, four actions are possible: `north`, `south`, `east`, and `west`, which deterministically cause the agent to move one cell in the respective direction in the grid. Actions that would take the agent off the grid leave its location unchanged, but also result in a reward of **-1**. Other actions result in a reward of 0, except those that move the agent out of the special states A and B. From state A, all four actions yield a reward of **+10** and take the agent to $\mathbf{A'}$. From state B, all actions yield a reward of **+5** and take the agent to $\mathbf{B'}$.



33	88	44	53	15
15	30	23	19	05
01	07	07	04	-04
-10	-04	-04	-06	-12
-19	-13	-12	-14	-20

Figure 3.5: Grid Example: a) exceptional reward dynamics; b) state-value function for the equiprobable random policy.

Suppose the agent selects all four actions with equal probability in all states. Figure 3.5b shows the value function, V^π, for this policy, for the discounted-reward case with $\gamma = 0.9$. This value function was computed by solving the system of equations (3.10). Notice the negative values near the lower edge; these are the result of the high probability of hitting the edge of the grid there under the random policy. Notice that A is the best state to be in under this policy, but that its expected return is less than 10, its immediate reward, because from A the agent is taken to $\mathbf{A'}$, from which it is likely to run into the edge of the grid. State B, on the other hand, is valued more than 5, its immediate reward, because from B the agent is taken to $\mathbf{B'}$, which has a positive value. From $\mathbf{B'}$ the expected penalty (negative reward) for possibly running into an edge is more than compensated for by the expected gain for possibly stumbling onto A or B. ◊

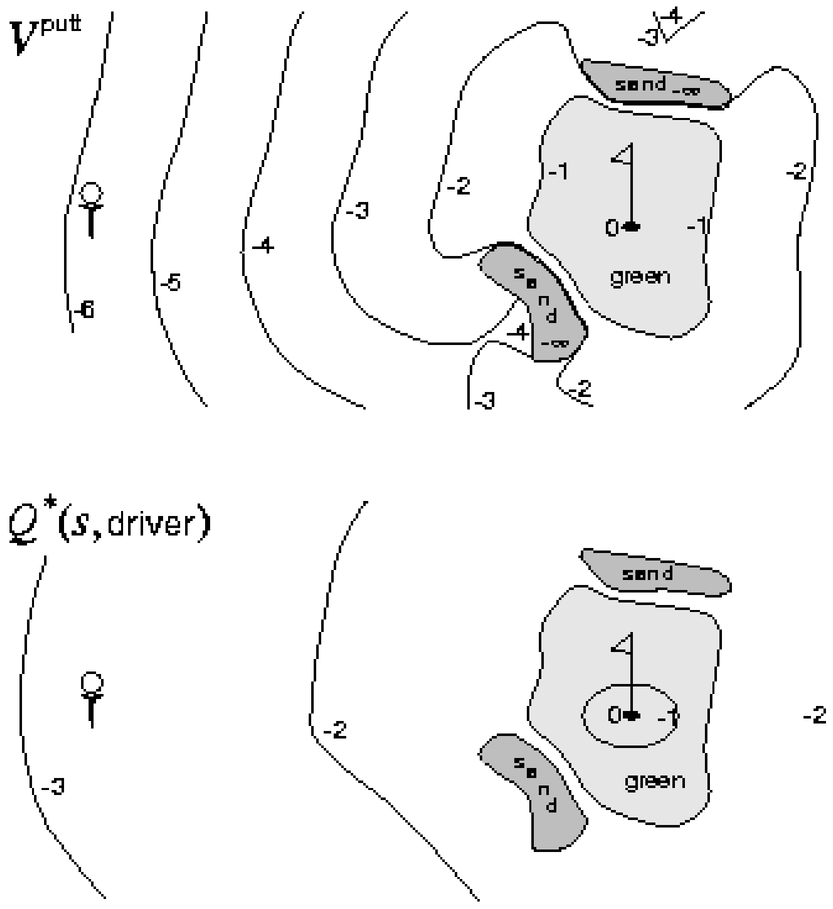


Figure 3.6: A golf example: the state-value function for putting (above) and the optimal action-value function for using the driver (below).

Example 3.9 *Golf*. To formulate playing a hole of golf as a reinforcement learning task, we count a penalty (negative reward) of -1 for each stroke until we hit the ball into the hole. The state is the location of the ball. The value of a state is the negative of the number of strokes to the hole from that location. Our actions are how we aim and swing at the ball, of course, and which club we select. Let us take the former as given and consider just the choice of club, which we assume is either a putter or a driver. The upper part of Figure 3.6 shows a possible state-value function, $V^{\text{putt}}(s)$, for the policy that always uses the putter. The terminal state *in-the-hole* has a value of **0**. From anywhere on the green we assume we can make a putt; these states have value **-1**. Off the green we cannot reach the hole by putting and the value is greater. If we can reach the green from a state by putting, then that state must have value one less than the green's value, i.e., **-2**. For simplicity, let us assume we can putt very precisely and deterministically, but with a limited range. This gives us the sharp contour line labeled **-2** in the figure; all locations between that line and the green require exactly two strokes to complete the hole. Similarly, any location within putting range of the **-2** contour line must have a value of **-3**, and so on to get all the contour lines shown in the figure. Putting doesn't get us out of sand traps, so they have a value of $-\infty$. Overall, it takes us 6 strokes to get from the tee to the hole by putting. $\Diamond$

Exercise 3.8

What is the Bellman equation for action values, that is, for Q^π? It must give the action value $Q^\pi(s,a)$ in terms of the action values, $Q^\pi(s',a')$, of possible successors to the state-action pair (s,a). As a hint, the backup diagram corresponding to this equation is given in Figure 3.4b. Show the sequence of equations analogous to (3.10), but for action values.

Exercise 3.9

The Bellman equation (3.10) must hold for each state for the value function V^π shown in Figure 3.5b. As an example, show numerically that this equation holds for the center state, valued at $+0.7$, with respect to its four neighboring states, valued at $+2.3$, $+0.4$, -0.4, and $+0.7$. (These numbers are accurate only to one decimal place.)

Exercise 3.10

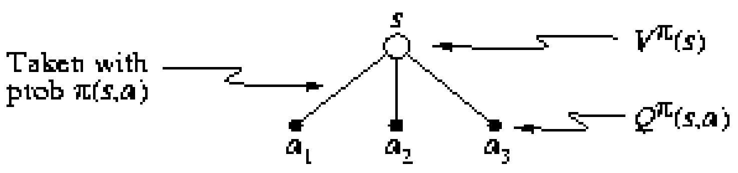
In the gridworld example, rewards are positive for goals, negative for running into the edge of the world, and zero all the rest of the time. Are the signs of these rewards important, or only the intervals between them? Prove using (3.2) that adding a constant **C** to all the rewards simply adds a constant, **K**, to the values of all states, and thus does not affect the relative value of any policies. What is **K** in terms of **C** and γ?

Exercise 3.11

Now consider adding a constant **C** to all the rewards in an episodic task, such as maze running. Would this have any effect, or would it leave the task unchanged as in the infinite-horizon case above? Why or why not? Give an example.

Exercise 3.12

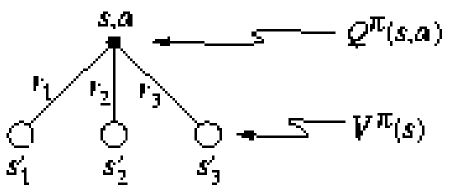
The value of a state depends on the the value of the actions possible in that state and on how likely each action is to be taken under the current policy. We can think of this in terms of a small backup diagram rooted at the state and considering each possible action:



Give the equation corresponding to this intuition and diagram for the value at the root node, $V^\pi(s)$, in terms of the value at the expected leaf node, $Q^\pi(s,a)$, given $s_t = s$. This will expectation depend on the policy, π. Then give a second equation in which the expected value is written out explicitly in terms of $\pi(s,a)$ such that no expected value notation appears in the equation.

Exercise 3.13

The value of an action, $Q^\pi(s,a)$, can be divided into two parts, the expected next reward, which does not depend on the policy π, and the expected sum of the remaining rewards, which depends on the next state and the policy. Again we can think of this in terms of a small backup diagram, this one rooted at an action (state-action pair) and branching based on the next state:



Give the equation corresponding to this intuition and diagram for the action value, $Q^\pi(s,a)$, in terms of the expected next reward, r_{t+1}, and the expected next state value, $V^\pi(s_{t+1})$, given that $s_t = s$ and $a_t = a$. Then give a second equation, writing out the expected value explicitly in terms of $\mathcal{P}^a_{ss'}$ and $\mathcal{R}^a_{ss'}$, defined respectively by (3.6) and (3.7), such that no expected value notation appears in the equation.

Richard Sutton
Sat May 31 13:56:52 EDT 1997

3.8 Optimal Value Functions

Solving a reinforcement learning task means, roughly, finding a policy that achieves a lot of reward over the long run. For finite MDPs, we can precisely define an optimal policy in the following way. Value functions define a partial ordering over policies. A policy π is defined to be better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states. In other words, $\pi \geq \pi'$ if and only if $V^{\pi}(s) \geq V^{\pi'}(s)$ for all $s \in \mathcal{S}$. There is always at least one policy that is better than or equal to all other policies. This is an *optimal policy*. Although there may be more than one, we denote all the optimal policies by π^*. They share the same value function, called the *optimal value function*, denoted V^*, defined as

$$V^*(s) = \max_{\pi} V^{\pi}(s), \qquad (3.11)$$

for all $s \in \mathcal{S}$.

Optimal policies also share the same *optimal action-value function*, denoted Q^*, defined as

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a), \qquad (3.12)$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$. For state-action pair (s,a), this function gives the expected return for taking action **a** in state **s** and thereafter following an optimal policy. Thus, we can write Q^* in terms of V^* as follows:

$$Q^*(s,a) = E\left\{r_{t+1} + \gamma V^*(s_{t+1}) \middle| s_t = s, a_t = a\right\}. \qquad (3.13)$$

Example 3.10 *Optimal Value Functions for Golf*. The lower part of Figure 3.6 shows the contours of a possible optimal action-value function $Q^*(s, \texttt{driver})$. These are the values of each state if we first play a stroke with the driver and then afterwards select either the driver or the putter, whichever is best. The driver enables us to hit the ball farther, but with less accuracy. We can reach the hole in one shot using the driver only if we are already very close; thus the **-1** contour for $Q^*(s, \texttt{driver})$ covers only a small portion of the green. If we have two strokes, however, then we can reach the hole from much farther away, as shown by the **-2** contour. In this case we don't have to drive all the way to within the small **-1** contour, but only to anywhere on the green; from there we can use the putter. The optimal action-value function gives the values after committing to a particular *first* action, in this case, to the driver, but afterwards using whichever actions are best. The **-3** contour is still farther out and includes the starting tee. From the tee, the best sequence of actions is two drives and one putt, sinking the ball in three strokes. $\diamond$

Because V^* is the value function for a policy, it must satisfy the self-consistency condition given by the Bellman equation for state values (3.10). Because it is the optimal value function, however, V^*'s consistency condition can be written in a special form without reference to any specific policy. This is the Bellman equation for V^*, or the *Bellman optimality equation*. Intuitively, the Bellman optimality equation expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state:

$$\begin{aligned}
V^*(s) &= \max_a Q^{\pi^*}(s,a) \\
&= \max_a E_{\pi^*}\left\{ R_t \,\middle|\, s_t = s, a_t = a \right\} \\
&= \max_{a \in \mathcal{A}(s)} E_{\pi^*}\left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, s_t = s, a_t = a \right\} \\
&= \max_{a \in \mathcal{A}(s)} E_{\pi^*}\left\{ r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \,\middle|\, s_t = s, a_t = a \right\} \\
&= \max_{a \in \mathcal{A}(s)} E\left\{ r_{t+1} + \gamma V^*(s_{t+1}) \middle| s_t = s, a_t = a \right\} && (3.14) \\
&= \max_{a \in \mathcal{A}(s)} \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^*(s') \right]. && (3.15)
\end{aligned}$$

These last two equations are two forms of the Bellman optimality equation for V^*. The

Bellman optimality equation for Q^* is

$$\begin{aligned} Q^*(s,a) &= E\left\{ r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') \,\middle|\, s_t = s, a_t = a \right\} \\ &= \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma \max_{a'} Q^*(s', a') \right] \end{aligned}$$

The backup diagrams in Figure 3.7 show graphically the spans of future states and actions considered in the Bellman optimality equations for V^* and Q^*. These are the same as the backup diagrams for V^π and Q^π except that arcs have been added at the agent's choice points to represent that the maximum over that choice is taken rather than the expected value given some policy. Figure 3.7a represents graphically the Bellman optimality equation (3.15).

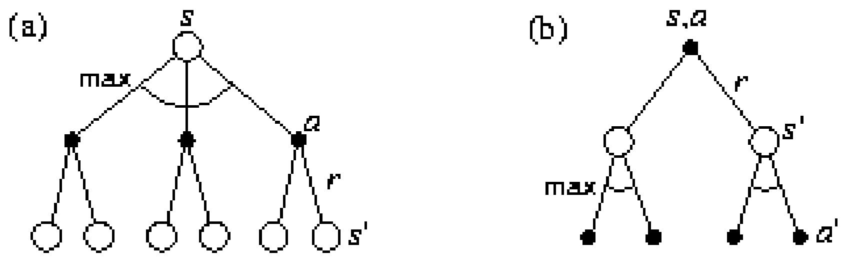


Figure 3.7: Backup diagrams for a) V^* and b) Q^*

For finite MDPs, the Bellman optimality equation (3.15) has a unique solution independent of the policy. The Bellman optimality equation is actually a system of equations, one for each state, so if there are **N** states, then there are **N** equations in **N** unknowns. If the dynamics of the environment are known ($\mathcal{R}^a_{ss'}$ and $\mathcal{P}^a_{ss'}$), then in principle one can solve this system of equations for V^* using any one of a variety of methods for solving systems of nonlinear equations. One can solve a related set of equations for Q^*.

Once one has V^*, it is relatively easy to determine an optimal policy. For each state **s**, there will be one or more actions at which the maximum is obtained in the Bellman optimality equation. Any policy that assigns non-zero probability only to these actions is an optimal policy. You can think of this as a one-step search. If you have the optimal value function, V^*, then the actions that appear best after just a one-step search will be optimal actions. Another way of saying this is that any policy that is *greedy* with respect to the optimal evaluation function V^* is an optimal policy. The term greedy is used in computer science to describe any search or decision procedure that selects alternatives based only on local or immediate considerations, without considering the possibility that such a selection may prevent future access to even better alternatives. Consequently, it describes policies that select actions based only on their short-term consequences. The beauty of V^* is that if one uses it to evaluate the short-term consequences of actions, specifically, the one-step consequences, then a greedy policy is actually optimal in the long-term sense in which we are interested because V^* already takes into account the reward consequences of all possible future behavior. By means of V^*, the optimal expected long-term return is turned into a quantity that is locally and immediately available for each state. Hence, a one-step ahead search yields the long-term optimal actions.

Having Q^* makes choosing optimal actions still easier. With Q^*, the agent does not even have to do a one-step ahead search: for any state **s**, it can simply find any action which maximizes $Q^*(s, a)$. The action-value function effectively caches the results of all one-step ahead searches. It provides the optimal expected long-term return as a value that is locally and immediately available for each state-action pair. Hence, at the cost of representing a function of state-action pairs, instead of just of states, the optimal action value function allows optimal actions to be selected without having to know anything about possible successor states and their values, that is, without having to know anything about the environment's dynamics.

Example 3.11

Bellman Optimality Equations for the Recycling Robot.

Using (3.15), we can explicitly give the the Bellman optimality equation for the recycling robot example. To make things more compact, we abbreviate the states `high` and `low`, and the actions `search`, `wait`, and `recharge` respectively by `h` , `l` , `s` , `w` , and `re` . Since there are only two states, the Bellman optimality equation consists of just two equations. The equation for $V^*(\mathtt{h})$ can be written as follows:

$$\begin{aligned} V^*(\mathtt{h}) &= \max \left\{ \begin{array}{l} \mathcal{P}^{\mathtt{s}}_{\mathtt{hh}}[\mathcal{R}^{\mathtt{s}}_{\mathtt{hh}} + \gamma V^*(\mathtt{h})] + \mathcal{P}^{\mathtt{s}}_{\mathtt{hl}}[\mathcal{R}^{\mathtt{s}}_{\mathtt{hl}} + \gamma V^*(\mathtt{l})], \\ \mathcal{P}^{\mathtt{w}}_{\mathtt{hh}}[\mathcal{R}^{\mathtt{w}}_{\mathtt{hh}} + \gamma V^*(\mathtt{h})] + \mathcal{P}^{\mathtt{w}}_{\mathtt{hl}}[\mathcal{R}^{\mathtt{w}}_{\mathtt{hl}} + \gamma V^*(\mathtt{l})] \end{array} \right\} \\ &= \max \left\{ \begin{array}{l} \alpha[\mathcal{R}^{\mathtt{s}} + \gamma V^*(\mathtt{h})] + (1-\alpha)[\mathcal{R}^{\mathtt{s}} + \gamma V^*(\mathtt{h})], \\ 1[\mathcal{R}^{\mathtt{w}} + \gamma V^*(\mathtt{h})] + 0[\mathcal{R}^{\mathtt{w}} + \gamma V^*(\mathtt{l})] \end{array} \right\} \\ &= \max \left\{ \begin{array}{l} \mathcal{R}^{\mathtt{s}} + \gamma[\alpha V^*(\mathtt{h}) + (1-\alpha) V^*(\mathtt{l})], \\ \mathcal{R}^{\mathtt{w}} + \gamma V^*(\mathtt{h}) \end{array} \right\}. \end{aligned}$$

Following the same procedure for $V^*(\mathtt{l})$ yields the equation:

$$V^*(\mathtt{l}) = \max \left\{ \begin{array}{l} \beta \mathcal{R}^{\mathtt{s}} - 3(1-\beta) + \gamma[(1-\beta)V^*(\mathtt{h}) + \beta V^*(\mathtt{l})] \\ \mathcal{R}^{\mathtt{w}} + \gamma V^*(\mathtt{l}), \\ \gamma V^*(\mathtt{h}) \end{array} \right\}.$$

For any choice of $\mathcal{R}^{\mathtt{s}}$, $\mathcal{R}^{\mathtt{w}}$, α, β, and γ, with $0 \leq \gamma < 1$, $0 \leq \alpha, \beta \leq 1$, there is exactly one pair of numbers, $V^*(\mathtt{h})$ and $V^*(\mathtt{l})$, that simultaneously satisfy these two nonlinear equations.

Example 3.12

Solving the Gridworld. Suppose we solve the Bellman equation for V^* for the simple grid task introduced in the previous example and shown again in Figure 3.8a. Recall that state A is followed by a reward of **+10** and transition to state A', while state B is followed by a reward of **+5** and transition to state B'. Figure 3.8b shows the optimal value function, and Figure 3.8c shows the corresponding optimal policies. Where there are multiple arrows in a cell, any of the corresponding actions is optimal.

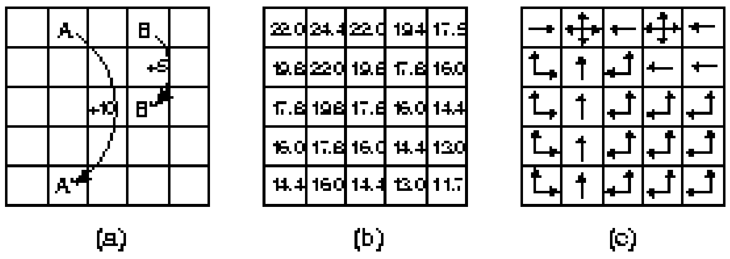


Figure:

Explicitly solving the Bellman optimality equation provides one route for finding an optimal policy and thus to solving the reinforcement learning problem. However, this solution is rarely directly useful. It is akin to an exhaustive search, looking ahead at all possibilities, computing their probabilities of occurrence, and their desirabilities in terms of expected rewards. This solution relies on at least three assumptions that are never completely true in practice: 1) we accurately know the dynamics of the environment, 2) we have enough computational resources to complete the computation of the solution, and 3) the Markov property. For the kinds of tasks in which we are interested, one is generally not able to implement this solution exactly because various combinations of these assumptions are violated. For example, although the first and third assumptions present no problems for the game of backgammon, the second is a major impediment. Since the game has about 10^{20} states, it would take thousands of years on today's fastest computers to solve the Bellman equation for V^*, and the same is true for finding Q^*. In reinforcement learning one typically has to settle for approximate solutions.

Many different decision-making methods can be viewed as ways of approximately solving the Bellman optimality equation. For example, heuristic search methods can be viewed as expanding the right-hand side of (3.15) several times, up to some depth, forming a ``tree'' of possibilities, and then using a heuristic evaluation function to approximate $V^*$ at the ``leaf'' nodes. (Heuristic search methods such as A^* are almost always based on the total-reward case.) The methods of dynamic programming can be related even more closely to the Bellman optimality equation. Many reinforcement learning methods can be clearly understood as approximately solving the Bellman optimality equation, using actual experienced transitions in place of knowledge of the expected transitions. We consider a variety of such methods in the following chapters.

Exercise 3.14

Draw or describe the optimal state-value function for the golf example.

Exercise 3.15

Draw or describe the contours of the optimal action-value function for putting, $Q^*(s, \mathtt{putter})$.

Exercise 3.16

Give the Bellman equation for Q^* for the recycling robot.

Exercise 3.17

Figure 3.8 gives the optimal value of the best state of the gridworld as 24.4, to one decimal place. Use your knowledge of the optimal policy and (3.2) to compute this value symbolically, and then to three decimal places.

Next Up Previous
Next: 3.9 Optimality and Approximation **Up:** 3 The Reinforcement Learning **Previous:** 3.7 Value Functions

Richard Sutton
Sat May 31 13:56:52 EDT 1997

3.9 Optimality and Approximation

We have defined optimal value functions and optimal policies. Clearly, an agent that learns an optimal policy has done very well, but in practice this rarely happens. For the kinds of tasks in which we are interested, optimal learning strategies can be generated only with extreme computational cost. A well-defined notion of optimality organizes the approach to learning we describe in this book and provides a way to understand the theoretical properties of various learning algorithms, but it is an ideal that agents can only approximate to varying degrees. As we discussed above, even if we have a complete and accurate model of its environment's dynamics, it is usually not possible to simply compute an optimal policy by solving the Bellman optimality equation. For example, board games such as chess are a tiny fraction of human experience, yet large, custom-designed computers still cannot compute the optimal moves. A critical aspect of the problem facing the agent is always the computational power available to it, in particular, the amount of computation it can perform in a single time step.

The memory available is also an important constraint. A large amount of memory is often required to build up approximations of value functions, policies, and models. In tasks with small, finite state sets, it is possible to form these approximations using arrays or tables with one entry for each state (or state-action pair). This we call the *tabular* case, and the corresponding methods we call tabular methods. In many cases of practical interest, however, there are far more states than could possibly be entries in a table. In these cases the functions must be approximated using some sort of more compact parameterized function representation.

Our framing of the reinforcement learning problem forces us to settle for approximations. However, it also presents us with some unique opportunities for achieving useful approximations.

For example, in approximating optimal behavior, there may be many states that the agent faces with such a low probability that selecting suboptimal actions for them has little impact on the amount of reward it receives. Tesauro's backgammon player, for example, plays with exceptional skill even though it might make very bad decisions on board configurations that never occur in games against experts. In fact, it is possible that TD-Gammon makes bad decision in a large fraction of the game's state set. The on-line nature of reinforcement learning makes it possible to approximate optimal policies in ways that put more effort into learning to make good decisions for frequently encountered states, at the expense of less effort for infrequently encountered states. This is one key property that distinguishes reinforcement learning from other approaches to approximately solving MDPs.

Next Up Previous

Next: 3.10 Summary **Up:** 3 The Reinforcement Learning **Previous:** 3.8 Optimal Value Functions

---

Richard Sutton
Sat May 31 13:56:52 EDT 1997

3.10 Summary

Let us summarize the elements of the reinforcement learning problem that we have presented in this chapter. Reinforcement learning is about learning from interaction how to behave in order to achieve a goal. The reinforcement learning *agent* and its *environment* interact over a sequence of discrete time steps. The specification of their interface defines a particular task: the *actions* are the choices made by the agent; the *states* are basis for making the choices; and the *rewards* are the basis for evaluating the choices. Everything inside the agent is completely known and controllable by the agent; everything outside is incompletely controllable but may or may not be completely known. A *policy* is a stochastic rule by which the agent selects actions as a function of states. The agent's objective is to maximize the amount of reward it receives over time.

The *return* is the function of future rewards that the agent seeks to maximize. It has several different definitions depending upon whether one is interested in *total reward* or *discounted reward*. The first is appropriate for *episodic tasks*, in which the agent-environment interaction breaks naturally into *episodes*; the second is appropriate for *continual tasks*, in which the interaction does not naturally break into episodes but continues without limit.

An environment satisfies the *Markov property* if its state compactly summarizes the past without degrading the ability to predict the future. This is rarely exactly true, but often nearly so; the state signal should be chosen or constructed so that the Markov property approximately holds. In this book we assume that this has already been done and focus on the decision-making problem: how to decide what to do as a function of whatever state signal is available. If the Markov property does hold, then the environment is called a *Markov decision process* (MDP). A *finite MDP* is an MDP with finite state and action sets. Most of the current theory of reinforcement learning is restricted to finite MDPs, but the methods and ideas apply more generally.

A policy's *value function* assigns to each state the expected return from that state given that the agent uses the policy. The *optimal value function* assigns to each state the largest expected return achievable by any policy. A policy whose value function is the optimal value function is an *optimal policy*. Whereas there is only one optimal value function for a given MDP, there can be many optimal policies. Any policy that is greedy with respect to the optimal value function is an optimal policy. The *Bellman optimality equation* is a special consistency condition that the optimal value function must satisfy and that can, in principle, be solved for the optimal value function, from which an optimal policy can be determined with relative ease.

A reinforcement learning problem can be posed in a variety of different ways depending on assumptions about the level of knowledge initially available to the agent. In problems of *complete knowledge*, the agent has a complete and accurate model of the environment's dynamics. In the environment is an MDP, then such a model consists of the one-step *transition probabilities* and *expected rewards* for all states and their allowable actions. In problems of *incomplete knowledge*, a complete and perfect model of the environment is not available.

Even if the agent has a complete and accurate environment model, the agent is typically unable to perform enough computation per time step to fully use it. The memory available is often an important constraint. Memory may be required to build up accurate approximations of value functions, policies, and models. In most cases of practical interest there are far more states than could possibly be entries in a table, and approximations must be made.

A well-defined notion of optimality organizes the approach to learning we describe in this book and provides a way to understand the theoretical properties of various learning algorithms, but it is an ideal that reinforcement learning agents can only approximate to varying degrees. In reinforcement learning we are very much concerned with cases in which optimal solutions cannot be found but must be approximated in some way.

Next Up Previous

Next: 3.11 Bibliographical and Historical **Up:** 3 The Reinforcement Learning **Previous:** 3.9 Optimality and Approximation

Richard Sutton
Sat May 31 13:56:52 EDT 1997

3.11 Bibliographical and Historical Remarks

The reinforcement learning problem is obviously deeply indebted to the idea of Markov decision processes from the field of optimal control. These historical influences and other major influences from psychology are described in the brief history given in Chapter 1. Reinforcement learning adds to MDPs a focus on approximation and incomplete information for realistically large problems. MDPs and the reinforcement learning problem are only weakly linked to traditional learning and decision-making problems in artificial intelligence. However, artificial intelligence is now vigorously exploring MDP formulations for planning and decision making from a variety of perspectives. MDPs are more general than previous formulations used in artificial intelligence problem in that they permit more general kinds of goals and uncertainty.

Our presentation of the reinforcement learning problem was influenced by Watkins (1989).

3.1

The bio-reactor example is based on the work of Ungar (1990) and Miller and Williams (1992). The recycling robot example was inspired by the can-collecting robot built by Jonathan Connell (1989).

3.3 -- 3.4

The terminology of *episodic* and *continual* tasks is different from that usually used in the MDP literature. In that literature it is common to distinguish three types of tasks: 1) finite-horizon tasks, in which interaction terminates after a particular *fixed* number of time steps; 2) indefinite-horizon tasks, in which interaction can last arbitrarily long but must eventually terminate; and 3) infinite-horizon tasks, in which interaction does not terminate. Our episodic and continual tasks are similar to indefinite-horizon and infinite-horizon tasks, respectively, but we prefer to emphasize the difference in the nature of the interaction. This difference seems more fundamental than the difference in the objective functions emphasized by the usual terms. Often episodic tasks use an indefinite-horizon objective function and continual tasks an infinite-horizon objective function, but we see this as a common coincidence rather than a fundamental difference.

The pole-balancing example is from Barto, Sutton, and Anderson (1983) and Michie and Chambers (1968).

3.5

For an excellent discussion of the concept of state see Minsky (1967).

3.6

The theory of Markov decision processes (MDPs) is treated by, e.g., Bertsekas (1995), Ross (1983), White (1969), and Whittle (1982, 1983). This theory is also studied under the heading of stochastic optimal control, where *adaptive* optimal control methods are most closely related to reinforcement learning (e.g., Kumar, 1985; Kumar and Varaiya, 1986).

The theory of MDPs evolved from efforts to understand the problem of making sequences of decisions under uncertainty, where each decision can depend on the previous decisions and their outcomes. It is sometimes called the theory of multi-stage decision processes, or sequential decision processes, and has roots in the statistical literature on sequential sampling beginning with the papers by Thompson (1933, 1934) and Robbins (1952) that we cited in Chapter 2 in connection with bandit problems (which are prototypical MDPs if formulated as multiple-situation problems).

The earliest instance of which we are aware in which reinforcement learning was discussed using the MDP formalism is Andreae's (1969b) description of a unified view of learning machines. Witten and Corbin (1973) experimented with a reinforcement learning system later analyzed by Witten (1977) using the MDP formalism. Although he did not explicitly mention MDPs, Werbos (1977) suggested approximate solution methods for stochastic optimal control problems that are related to modern reinforcement learning methods (see also Werbos, 1982, 1987, 1988, 1989,1992). Although Werbos' ideas were not widely recognized at the time, they were prescient in emphasizing the importance of approximately solving optimal control problems in a variety of domains, including artificial intelligence. The most influential integration of reinforcement learning and MDPs is due to Watkins (1989). His treatment of reinforcement learning using the MDP formalism has been widely adopted.

Our characterization of the reward dynamics of an MDP in terms of $\mathcal{R}^a_{ss'}$ is slightly unusual. It is more common in the MDP literature to describe the reward dynamics in terms of the expected next reward given just the current state and action, i.e., by $\mathcal{R}^a_s = E\{r_{t+1} | s_t = s, a_t = a\}$. This quantity is related to our $\mathcal{R}^a_{ss'}$ as follows:

$$\mathcal{R}^a_s = \sum_{s'} \mathcal{P}^a_{ss'} \mathcal{R}^a_{ss'}.$$

In conventional MDP theory, $\mathcal{R}^a_{ss'}$ always appears in an expected value sum like this one, and therefore it is easier to use R^a_s. In reinforcement learning, however, we more often have to refer to individual actual or sample outcomes. For example, in the recycling robot, the expected reward for the action `search` in state `low` depends on the next state (Table 3.1). In teaching reinforcement learning, we have found the $\mathcal{R}^a_{ss'}$ notation to be more straightforward conceptually and easier to understand.

3.7 and 3.8

Assigning value based on what is good or bad in the long run has ancient roots.

In control theory, mapping states to numerical values representing the long-term consequences of control decisions is a key part of optimal control theory, which was developed in the 1950s by extending 19th century state-function theories of classical mechanics (see, for example, Schultz and Melsa, 1967). In describing how a computer could be programmed to play chess, Shannon (1950b) suggested using an evaluation function that took into account the long-term advantages and disadvantages of a chess position.

Watkins's (1989) Q-learning algorithm for estimating Q^* (Chapter 6) made action-value functions an important part of reinforcement learning, and consequently these functions are often called *Q-functions*. But the idea of an action-value function is much older than this. Shannon (1950b) suggested that a function $h(P, M)$ could be used by a chess-playing program to decide whether a move **M** in position **P** is worth exploring. Michie's (1961, 1963) MENACE system and Michie and Chambers (1968) BOXES system can be understood as estimating action-value functions. In classical physics, Hamilton's principle function is an action-value function: Newtonian dynamics are greedy with respect to this function (e.g., Goldstein, 1957). Action-value functions also played a central in Denardo's (1967) theoretical treatment of DP in terms of contraction mappings.

What we call the Bellman equation for V^* was first introduced by Richard Bellman (1957) who called it the ``basic functional equation." The counterpart of the Bellman optimality equation for continuous time and state problems is known as the Hamilton-Jacobi-Bellman equation (or often just the Hamilton-Jacobi equation), indicating its roots in classical physics (e.g., Schultz and Melsa, 1967).

Next Up Previous

Next: Part II: Elementary Solution Methods **Up:** 3 The Reinforcement Learning **Previous:** 3.10 Summary

Richard Sutton
Sat May 31 13:56:52 EDT 1997

...agent.

We use the terms agent, environment, and action instead of the engineers' terms controller, controlled system (or plant), and control signal because they are meaningful to a wider audience.

....

We restrict attention to discrete time to keep things as simple as possible, even though many of the ideas can be extended to the continuous-time case (e.g., see Bertsekas and Tsitsiklis, 1996; Werbos, 1992; Doya, 1996).

....

We use 15#15 to denote the immediate reward for an action taken at time step **t**, instead of the more common 16#16, because it emphasizes that the next reward and the next state are jointly determined.

...do.

Better places for imparting this kind of prior knowledge are the initial policy or value function, or in influences on these. For example, see Lin (1993), Maclin and Shavlik (1994), and Clouse (1996).

...episodes,

Episodes are often called ``trials" in the literature.

...both

Ways to formulate tasks that are both continual and undiscounted are subjects of current research (e.g., Mahadevan, 1996; Schwartz, 1993; Tadepalli and Ok, 1994). Some of the ideas are discussed in Section 6 .7 .

Richard Sutton
Sat May 31 13:56:52 EDT 1997

Part II: Elementary Solution Methods

In this part of the book we describe three fundamental classes of methods for solving the reinforcement learning problem: dynamic programming, Monte Carlo methods, and temporal-difference learning. All of these methods solve the full version of the problem including delayed rewards.

Each of the three classes of methods has its strengths and weaknesses. Dynamic programming methods are very well developed mathematically, but require a complete and accurate model of the environment. Monte Carlo methods don't require a model and are very simple conceptually, but are not suited for step-by-step incremental computation. Finally, temporal-difference methods require no model and are fully incremental, but are more complex to analyze. The methods also differ in several ways with respect to their efficiency and speed of convergence. In the third part of this book we explore how these methods can be combined so as to obtain the best features of each of them.

Richard Sutton
Fri May 30 21:13:22 EDT 1997

4 Dynamic Programming

The term ``Dynamic Programming'' (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process (MDP). Classical DP algorithms are of limited utility in reinforcement learning both because of their assumption of a perfect model and because of their great computational expense, but they are still very important theoretically. DP provides an essential foundation for the understanding of the methods presented in the rest of this book. In fact, all of these methods can be viewed as attempts to achieve much the same effect as DP, only with less computation and without assuming a perfect model of the environment.

Starting with this chapter, we usually assume that the environment is a finite MDP. That is, we assume that its state and action sets, $\mathcal{S}$ and $\mathcal{A}(s)$, for $s \in \mathcal{S}$, are finite, and that its dynamics are given by a set of transition probabilities, $\mathcal{P}^a_{ss'} = Pr\left\{s_{t+1}=s' \middle| s_t=s, a_t=a\right\}$, and expected immediate rewards, $\mathcal{R}^a_{ss'} = E\left\{r_{t+1} \middle| a_t=a, s_t=s, s_{t+1} = s'\right\}$, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, and $s' \in \mathcal{S}^+$ ($\mathcal{S}^+$ is $\mathcal{S}$ plus a terminal state if the problem is episodic). Although DP ideas can be applied to problems with continuous state and action spaces, exact solutions are possible only in special cases. A common way of obtaining approximate solutions for continuous state and action tasks is to quantize the state and action spaces and then apply finite-state DP methods. The methods we explore in Chapter 8 are applicable to continuous problems and are a significant extension of that approach.

The key idea of DP, and of reinforcement learning generally, is the use of value functions to organize and structure the search for good policies. In this chapter we show how DP can be used to compute the value functions defined in Chapter 3. As discussed there, we can easily obtain optimal policies once we have found the optimal value functions, V^* or Q^*, which satisfy the Bellman optimality equations:

$$V^*(s) = \max_a E\left\{r_{t+1} + \gamma V^*(s_{t+1}) \middle| s_t = s, a_t = a\right\} = \max_a \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^*(s')\right], \qquad (4.1)$$

or

$$Q^*(s,a) = E\left\{r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') \middle| s_t = s, a_t = a\right\} = \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma \max_{a'} Q^*(s', a')\right], (4.2)$$

for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, and $s' \in \mathcal{S}^+$. As we shall see, DP algorithms are obtained by turning Bellman equations such as these into assignment statements, that is, into update rules for improving approximations of the desired value functions.

Richard Sutton
Fri May 30 12:39:34 EDT 1997

4.1 Policy Evaluation

First we consider how to compute the state-value function V^π for an arbitrary policy, π. This is called *policy evaluation* in the DP literature. We also refer to it as the *prediction problem*. Recall from Chapter 3 that, for all $s \in \mathcal{S}$,

$$\begin{aligned} V^\pi(s) &= E_\pi\left\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots \middle| s_t = s\right\} \\ &= E_\pi\{r_{t+1} + \gamma V^\pi(s_{t+1}) | s_t = s\} && (4.3) \\ &= \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')\right], && (4.4) \end{aligned}$$

where $\pi(s,a)$ is the probability of taking action **a** in state **s** under policy π, and the expectations are subscripted by π to indicate that they are conditional on π being followed. The existence and uniqueness of V^π is guaranteed as long as either $\gamma < 1$ or eventual termination is guaranteed from all states under the policy π.

If the environment's dynamics are completely known, then (4.4) is a system of $|\mathcal{S}|$ simultaneous linear equations in $|\mathcal{S}|$ unknowns (the $V^\pi(s)$, $s \in \mathcal{S}$). In principle, its solution is a straightforward, if tedious, computation. For our purposes, iterative solution methods are most suitable. Consider a sequence of approximate value functions $V_0, V_1, V_2, \ldots$, each mapping $\mathcal{S}^+$ to $\Re$. The initial approximation, V_0, is chosen arbitrarily (except that the terminal state, if any, must be given value 0), and each successive approximation is obtained by using the Bellman equation for V^π (3 .10) as an update rule:

$$\begin{aligned} V_{k+1}(s) &= E_\pi\{r_{t+1} + \gamma V_k(s_{t+1}) | s_t = s\} \\ &= \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V_k(s')\right], \qquad (4.5) \end{aligned}$$

for all $s \in \mathcal{S}$. Clearly, $V_k = V^\pi$ is a fixed point for this update rule because the Bellman equation for V^π assures us of equality in this case. Indeed, the sequence $\{V_k\}$ can be shown in general to converge to V^π as $k \rightarrow \infty$ under the same conditions that guarantee the existence of V^π. This algorithm is called *iterative policy evaluation*.

To produce each successive approximation, V_{k+1} from V_k, iterative policy evaluation applies the same operation to each state **s**: it replaces the old value of **s** with a new value obtained from the old values of the successor states of **s**, and the expected immediate rewards, along all the one-step transitions possible under the policy being evaluated. We call this kind of operation a *full backup*. Each iteration of iterative policy evaluation *backs up* the value of every state once to produce the new approximate value function V_{k+1}. There are several different kinds of full backups depending on whether a state is being backed up (as here) or a state-action pair, and depending on the precise way the estimated values of the successor states are combined. All the backups done in DP algorithms are called *full* backups because they are based on all possible next states rather than on a sample next state. The nature of a backup can be expressed in an equation, as above, or in a backup diagram like those introduced in Chapter 3. For example, Figure 3 .4 a is the backup diagram corresponding to the full backup used in iterative policy evaluation.

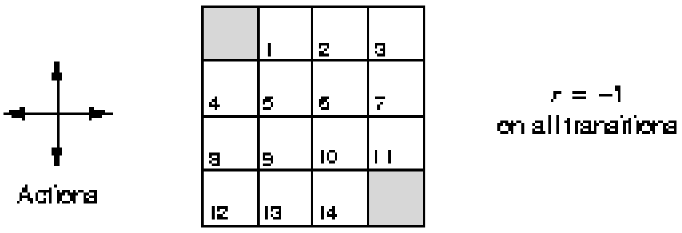
To write a sequential computer program to implement iterative policy evaluation, as given by (4.5), you would have to use two arrays, one for the old values, $V_k(s)$, and one for the new values, $V_{k+1}(s)$. This way the new values can be computed one by one from the old values without the old values being changed. Of course it is easier simply to use one array and update the values ``in place", that is, with each new backed-up value immediately overwriting the old one. Then, depending on the order in which the states are backed up, sometimes new values are used instead of old ones on the righthand side of (4.5). This slightly different algorithm also converges to V^π; in fact, it usually converges faster than the two-array version, as you might expect since it uses new data as soon as it is available. We think of the backups as being done in a *sweep* through the state space. The order in which states are backed up during the sweep has a significant influence on the rate of convergence. We usually have the in-place version in mind when we think of DP algorithms.

Another implementation point concerns the termination of the algorithm. Formally, iterative policy evaluation converges only in the limit, but in practice it must be halted short of this. A typical stopping condition for iterative policy evaluation is to test the quantity $\max_{s\in\mathcal{S}} |V_{k+1}(s) - V_k(s)|$ after each sweep and stop when it is sufficiently small. Figure 4.1 gives a complete algorithm for iterative policy evaluation with this stopping criterion.

```
Input $\pi$, the policy to be evaluated
Initialize $V(s) = 0$, for all $s \in \mathcal{S}^+$
Repeat
   $\Delta \leftarrow 0$
   For each $s \in \mathcal{S}$:
       $v \leftarrow V(s)$
       $V(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V(s')]$
       $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)
Output $V \approx V^\pi$
```

Figure 4.1: Iterative policy evaluation.

Example 4.1 Consider the 4×4 gridworld shown below.



The nonterminal states are $\mathcal{S} = \{1, 2, \ldots, 14\}$. There are 4 actions possible in each state, $\mathcal{A} = \{\texttt{up}, \texttt{down}, \texttt{right}, \texttt{left}\}$, which deterministically cause the corresponding state transitions, except that actions that would take the agent off the grid in fact leave the state unchanged. Thus, e.g., $\mathcal{P}^{\mathtt{right}}_{5,6} = 1$, $\mathcal{P}^{\mathtt{right}}_{5,10} = 0$, and $\mathcal{P}^{\mathtt{right}}_{7,7} = 1$. This is an undiscounted, episodic task. The reward is **-1** on all transitions until the terminal state is reached. The terminal state is shaded in the figure (although it is shown in two places it is formally one state). The expected reward function is thus $\mathcal{R}^{a}_{ss'} = -1$, for all states s, s' and actions **a**. Suppose the agent follows the equiprobable random policy (all actions equally likely). The left side of Figure 4.2 shows the sequence of value functions $\{V_k\}$ computed by iterative policy evaluation. The final estimate is in fact V^{π}, which in this case gives for each state the negation of the expected number of steps from that state until termination. ◇

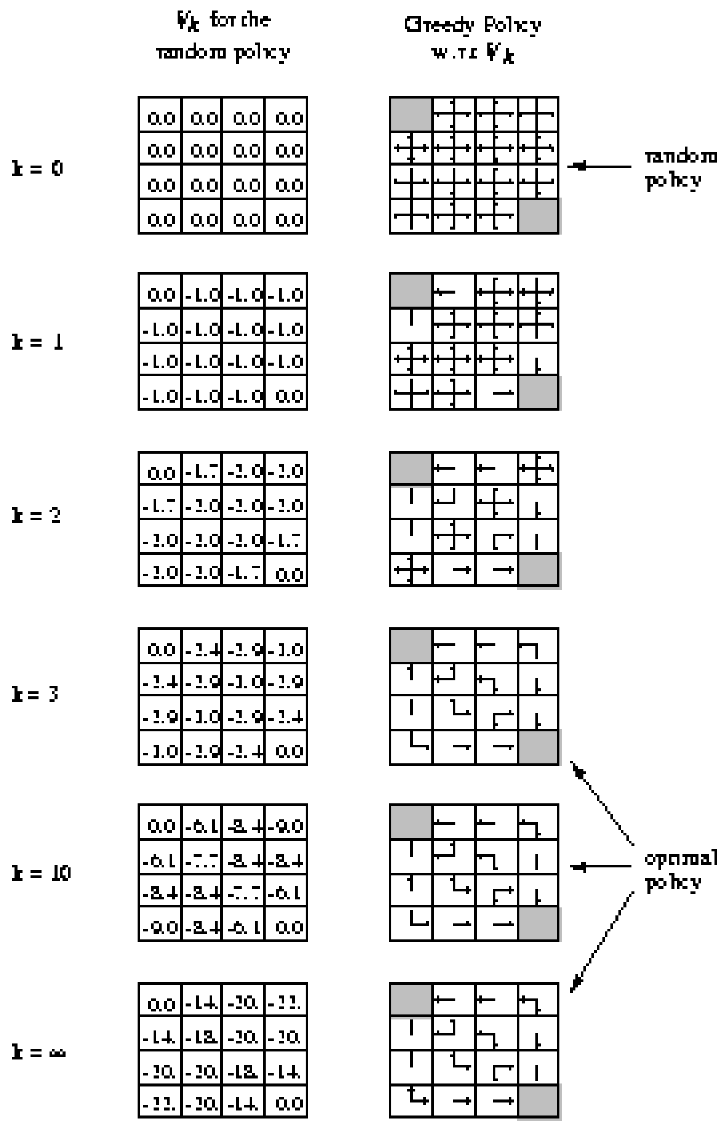


Figure 4.2: Convergence of iterative policy evaluation on a small gridworld. The left column is the sequence of approximations of the value function for the random policy (all actions equal). The right column is the sequence of greedy policies corresponding to the value function estimates (arrows are shown for all actions achieving the maximum). The

last policy is guaranteed only to be an improvement over the random policy, but in this case it and all policies after the third iteration are optimal.

Exercise 4.1

If π is the equiprobable random policy, what is $Q^\pi(11, \mathtt{down})$? What is $Q^\pi(7, \mathtt{down})$?

Exercise 4.2

Suppose a new state 15 is added to the gridworld just below state 13, whose actions, `left`, `up`, `right`, and `down`, take the agent to states 12, 13, 14, and 15 respectively. Assume that the transitions *from* the original states are unchanged. What then is $V^\pi(15)$ for the equiprobable random policy? Now suppose the dynamics of state 13 are also changed, such that action `down` from state 13 takes the agent to the new state 15. What is $V^\pi(15)$ for the equiprobable random policy in this case?

Exercise 4.3

What are the equations analogous to (4.3), (4.4) and (4.5) for the action-value function Q^π and its successive approximation by a sequence of functions $Q_0, Q_1, Q_2, \ldots$?

Exercise 4.4

In some undiscounted episodic tasks there may be some policies for which eventual termination is not guaranteed. For example, in the grid problem above it is possible to go back and forth between two states forever. In a task that is otherwise perfectly sensible, $V^\pi(s)$ may be negative infinity for some policies and states, in which case the algorithm for iterative policy evaluation given in Figure 4.1 will not terminate. As a purely practical matter, how might we amend this algorithm to assure termination even in this case. Assume that eventual termination *is* guaranteed under the optimal policy.

Programming

Richard Sutton
Sat May 31 11:09:21 EDT 1997

4.2 Policy Improvement

Our reason for computing the value function for a policy is to help find better policies. Suppose we have determined the value function V^π for an arbitrary deterministic policy π. For some state **s** we would like to know whether or not we should change the policy to deterministically choose an action $a \neq \pi(s)$. We know how good it is to follow the current policy from **s**---that is just $V^\pi(s)$---but would it be better or worse to change to the new policy? One way to answer this question is to consider selecting **a** in **s** and then thereafter following the existing policy, π. The value of this way of behaving is

$$\begin{aligned} Q^\pi(s,a) &= E_\pi\{r_{t+1} + \gamma V^\pi(s_{t+1}) \mid s_t = s, a_t = a\} \\ &= \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')\right]. \end{aligned} \tag{4.6}$$

The key criterion is whether this is greater than or less than $V^\pi(s)$. If it is greater, that is, if it is better to select **a** once in **s** and thereafter follow π than it would be to follow π all the time, then one would expect it to be better still to select **a** every time **s** is encountered, and that the new policy would in fact be a better one overall.

That this is true is a special case of a general result called the *policy improvement theorem*. Let π and π' be any pair of deterministic policies such that, for all $s \in \mathcal{S}$,

$$Q^\pi(s, \pi'(s)) \geq V^\pi(s). \tag{4.7}$$

Then the policy π' must be as good as, or better than, π. That is, it must obtain greater or equal expected return from all states $s \in \mathcal{S}$:

$$V^{\pi'}(s) \geq V^\pi(s). \tag{4.8}$$

Moreover, if there is strict inequality of (4.7) at any state, then there must be strict inequality of (4.8) at at least one state. This result applies in particular to the two policies that we considered in the previous paragraph, an original deterministic policy, π, and a changed policy, π', that is identical to π except that $\pi'(s) = a \neq \pi(s)$. Obviously, (4.7) holds trivially at all states other than **s**. Thus, if $Q^\pi(s,a) > V^\pi(s)$, then the changed policy is indeed better than π.

The idea behind the proof of the policy improvement theorem is easy to understand. Starting from (4.7), we keep expanding the Q^π side and re-applying (4.7) until we get $V^{\pi'}(s)$:

$$\begin{aligned}
V^\pi(s) &\leq Q^\pi(s,\pi'(s)) \\
&\leq E_{\pi'}\{r_{t+1} + \gamma V^\pi(s_{t+1}) | s_t = s\} \\
&\leq E_{\pi'}\{r_{t+1} + \gamma Q^\pi(s_{t+1},\pi'(s_{t+1})) | s_t = s\} \\
&\leq E_{\pi'}\{r_{t+1} + \gamma E_{\pi'}\{r_{t+2} + \gamma V^\pi(s_{t+2})\} | s_t = s\} \\
&\leq E_{\pi'}\left\{r_{t+1} + \gamma r_{t+2} + \gamma^2 V^\pi(s_{t+2}) \middle| s_t = s\right\} \\
&\leq E_{\pi'}\left\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 V^\pi(s_{t+3}) \middle| s_t = s\right\} \\
&\vdots \\
&\leq E_{\pi'}\left\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \cdots \middle| s_t = s\right\} \\
&\leq V^{\pi'}(s).
\end{aligned}$$

So far we have seen how, given a policy and its value function, we can easily evaluate a change in the policy at a single state to a particular action. It is a natural extension to consider changes at *all* states and to *all* possible actions, selecting at each state the action that appears best according to $Q^\pi(s,a)$. In other words, to consider the new *greedy* policy, π', given by

$$\begin{aligned}\pi'(s) &= \arg\max_a Q^\pi(s,a) \\ &= \arg\max_a E\{r_{t+1} + \gamma\end{aligned}$$

$$V^\pi(s_{t+1}) | s_t = s, a_t = a (4.9)$$
$$= \arg\max_a \textstyle\sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma\right.$$
$$\left. V^\pi(s')\right],$$

where the ``$\arg\max_a$'' notation denotes the value of **a** at which the expression is maximized (with ties broken arbitrarily). The greedy policy takes the action that looks best in the short term---after one step of lookahead---according to V^π. By construction, the greedy policy meets the conditions of the policy improvement theorem (4.7), so we know that it is as good as, or better than, the original policy. The process of making a new policy that improves over an original policy, by making it greedy with respect to the value function of the original policy, is called *policy improvement*.

Suppose the new greedy policy, π', is as good, but not better than, the old policy π. Then $V^\pi = V^{\pi'}$, and from (4.9) it follows that for all $s \in \mathcal{S}$:

$$\begin{aligned}V^{\pi'}(s) &= \max_a E\left\{r_{t+1} + \gamma V^{\pi'}(s_{t+1}) \middle| s_t = s, a_t = a\right\} \\ &= \max_a \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^{\pi'}(s')\right].\end{aligned}$$

But this is the same as the Bellman optimality equation (4.1), and therefore, $V^{\pi'}$ must be V^*, and both π and π' must be optimal policies. Policy improvement thus must give us a strictly better policy except when the original policy is already optimal.

So far in this section we have considered the special case of deterministic policies. In the general case, a stochastic policy π specifies probabilities, $\pi(s,a)$, for taking each action, **a**, in each state, **s**. We will not go through the details, but in fact all the ideas of this section extend easily to the case of stochastic policies. In particular, the policy improvement theorem carries through as stated for the stochastic case, under the natural definition:

$$Q^\pi(s, \pi'(s)) = \sum_a \pi'(s,a) Q^\pi(s,a).$$

In addition, if there are ties in policy improvement steps such as (4.9), that is, if there are several actions at which the maximum is achieved, then in the stochastic case we need not select a single action from among them. Instead, each maximizing action can be given a portion of the probability of being selected in the new greedy policy. Any apportioning scheme is allowed as long as all sub-maximal actions are given zero probability.

The last row of Figure 4.2 shows an example of policy improvement for stochastic policies. Here the original policy, π, is the equiprobable random policy, and the new policy, π', is greedy with respect to V^π. The value function V^π is shown in the bottom-left diagram and the set of possible π' is shown in the bottom-right diagram. The states with multiple arrows in the π' diagram are those in which several actions achieve the maximum in (4.9); any apportionment of probability among these actions is permitted. The value function of any such policy, $V^{\pi'}(s)$, can be seen by inspection to be either **-1**, **-2**, or **-3** at all states, $s \in \mathcal{S}$, whereas $V^\pi(s)$ is at most **-14**. Thus, $V^{\pi'}(s) \geq V^\pi(s)$, for all $s \in \mathcal{S}$, illustrating policy improvement. Although in this case the new policy π' happens to be optimal, in general only an improvement is guaranteed.

Richard Sutton
Sat May 31 11:09:21 EDT 1997

4.3 Policy Iteration

Once a policy, π, has been improved using V^π to yield a better policy, π', we can then compute $V^{\pi'}$ and improve it again to yield an even better π''. We can thus obtain a sequence of monotonically improving policies and value functions:

$$\pi_0 \xrightarrow{E} V^{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V^{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi^* \xrightarrow{E} V^*,$$

where $\xrightarrow{E}$ denotes a policy *evaluation* and $\xrightarrow{I}$ denotes a policy *improvement*. Each policy is guaranteed to be a strict improvement over the previous one (unless it is already optimal). Because a finite MDP has only a finite number of policies, this process must converge to an optimal policy and optimal value function in a finite number of iterations.

This way of finding an optimal policy is called *policy iteration*. A complete algorithm is given in Figure 4.3. Note that each policy evaluation, itself an iterative computation, is started with the value function for the previous policy. This typically results in a great increase in the speed of convergence of policy evaluation (presumably because the value function changes little from one policy to the next).

```
1. Initialization
        $V(s) \in \Re$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
   $\Delta \leftarrow 0$
   For each $s \in \mathcal{S}$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_{s'} \mathcal{P}_{ss'}^{\pi(s)} \left[\mathcal{R}_{ss'}^{\pi(s)} + \gamma V(s')\right]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)
3. Policy Improvement
   $policy\text{-}stable \leftarrow true$
   For each $s \in \mathcal{S}$:
      $b \leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s'} \mathcal{P}_{ss'}^{a} \left[\mathcal{R}_{ss'}^{a} + \gamma V(s')\right]$
      If $b \neq \pi(s)$ then $policy\text{-}stable \leftarrow false$
   If policy-stable, then stop; else go to 2
```

Figure 4.3: Policy iteration (using iterative policy evaluation) for V^*. In the ``$\arg\max$'' step in 3, it is assumed that ties are broken in a consistent order.

Policy iteration often converges in surprisingly few iterations. This is illustrated by the example in the preceding section. The bottom-left diagram in Figure 4.2 shows the value function for the equiprobable random policy, and the bottom-right diagram shows a greedy policy for this value function. The policy improvement theorem assures us that these policies are better than the original random policy. In this case, however, these policies are not just better, but optimal, proceeding to the terminal states in the minimum number of steps. In this example, policy iteration would find the optimal policy after just one iteration.

Example 4.2

Jack's Car Rental. Jack manages two locations for a nationwide car rental company. Each day, some number of customers arrive at each location to rent cars. If Jack has a car available, he rents it out and is credited \$10 by the national company. If he is out of cars at that location, then the business is lost. Cars become available for renting the day after they are returned. To help ensure that cars are available where they are needed, Jack can move them between the two locations overnight, at a cost of \$2 per car moved. We assume that the number of cars requested and returned at each location are poisson random variables, meaning that the probability that the number is **n** is $\frac{\lambda^n}{n!} e^{-\lambda}$, where λ is the expected number. Suppose λ \ is 3 and 4 for rental requests at the first and second locations and 3 and 2 for dropoffs. To simplify the problem slightly, we assume that there can be no more than 20 cars at each location (any additional cars are returned to the nationwide company, and thus disappear from the problem) and a maximum of 5 cars can be moved from one location to the other in one night. We take the discount rate to be $\gamma = .9$ and formulate this as a continual finite MDP, where the time steps are days, the state is the number of cars at each location at the end of the day, and the actions are the net number of cars moved between the two locations overnight. Figure 4.4 shows the sequence of policies found by policy iteration starting from the policy that never moves any cars. $\diamond$

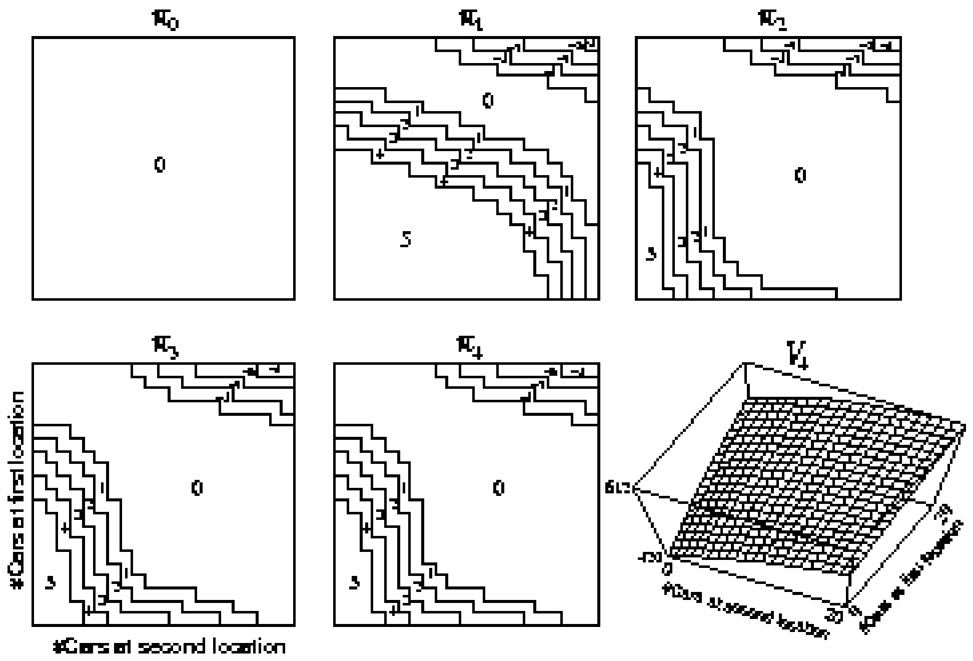


Figure 4.4: The sequence of policies found by policy iteration on Jack's car rental problem, and the final state-value function. The first five diagrams show, for each number of cars at each location at the end of the day, the number of cars to be moved from the first location to the second (negative numbers indicate transfers from the second location to the first). Each successive policy is a strict improvement over the previous policy, and the last policy is optimal.

Exercise 4.5 (programming)

Write a program for policy iteration and re-solve Jack's car rental problem with the following changes. One of Jack's employees at the first location rides a bus home each night and lives very near the second location. She is happy to shuttle one car to the second location for free. Each additional car still costs \$2, as do all cars in the other direction. In addition, Jack has limited parking space at each location. If more than 10 cars are kept overnight at a location (after any moving of cars), then an additional cost of \$4 must be incurred to use a second parking lot (independent of how many cars are kept there). These sort of nonlinearities and arbitrary dynamics often occur in real problems and cannot easily be handled by optimization methods other than dynamic programming. To check your program, first replicate the results given for the original problem. If your computer is too slow for the full problem, cut all the numbers of cars in half.

Exercise 4.6

How would policy iteration be defined for action values? Give a complete algorithm for computing Q^*, analogous to Figure 4.3 for computing V^*. Please pay special attention to this exercise because the ideas involved will be used throughout the rest of the book.

Exercise 4.7

Suppose you are restricted to considering only policies that are ϵ-*soft*, meaning that the probability of selecting each action in each state, **s**, is at least $\epsilon/|\mathcal{A}(s)|$. Describe qualitatively the changes that would be required in each of the steps 3, 2, and 1, in that order, of the policy iteration algorithm for V^* (Figure 4.3).

Richard Sutton
Sat May 31 11:09:21 EDT 1997

4.4 Value Iteration

One drawback to policy iteration is that each of its iterations involves policy evaluation, which may itself be a protracted iterative computation requiring multiple sweeps through the state set. If policy evaluation is done iteratively, then convergence exactly to V^π occurs only in the limit. Must we wait for exact convergence, or can we stop short of that? The example in Figure 4.2 certainly suggests that it may be possible to truncate policy evaluation. In that example policy evaluation iterations beyond the first three have no effect on the corresponding greedy policy.

In fact, the policy-evaluation step of policy iteration can be truncated in several ways without losing the convergence guarantees of policy iteration. One important special case is when policy evaluation is stopped after just one sweep (one backup of each state). This algorithm is called *value iteration*. It can be written as a particularly simple backup operation that combines the policy-improvement and truncated policy-evaluation steps:

$$V_{k+1}(s) = \max_a E\left\{r_{t+1} + \gamma V_k(s_{t+1}) \middle| s_t = s, a_t = a\right\} \qquad (4.10)$$
$$= \max_a \sum_{s'} \mathcal{P}^a_{ss'}\left[\mathcal{R}^a_{ss'} + \gamma\right.$$

$$V_k(s'),$$

for all $s \in \mathcal{S}$. For arbitrary V_0, the sequence $\{V_k\}$ can be shown to converge to V^* under the same conditions that guarantee the existence of V^*.

Another way of understanding value iteration is by reference to the Bellman optimality equation (4.1). Note that value iteration is obtained simply by turning the Bellman optimality equation into an update rule. Also note how the value iteration backup is identical to the policy evaluation backup (4.5) except that it requires the maximum to be taken over all actions. Another way of seeing this close relationship is to compare the backup diagrams for these algorithms: Figure 3 .4 a shows the backup diagram for policy evaluation and Figure 3 .7 a shows the backup diagram for value iteration. These two are the natural backup operations for computing V^π and V^*.

Finally, let us consider how value iteration terminates. Like policy evaluation, value iteration formally requires an infinite number of iteration to converge exactly to V^*. In practice, we stop once the value function changes by only a small amount in a sweep. Figure 4.5 gives a complete value iteration algorithm with this kind of termination condition.

Value iteration effectively combines, in each of its sweeps, one sweep of policy evaluation and one sweep of policy improvement. Faster convergence is often achieved by interposing multiple policy evaluation sweeps between each policy improvement sweep. In general, the entire class of truncated policy iteration algorithms can be thought of as sequences of sweeps, some of which use policy evaluation backups, and some of which use value iteration backups. Since the max operation in (4.10) is the only difference between these backups, this just means that the max operation is added to some sweeps of policy evaluation. All of these algorithms converge to an optimal policy for discounted finite MDPs.

```
Initialize V arbitrarily, e.g., $V(s) = 0$, for all $s \in \mathcal{S}^+$
Repeat
    $\Delta \leftarrow 0$
    For each $s \in \mathcal{S}$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \max_a \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V(s')]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
    until $\Delta < \theta$ (a small positive number)
Output a deterministic policy, $\pi$, such that:
                    $\pi(s) = \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V(s')]$
```

Figure 4.5: Value Iteration.

Example 4.3 *The Gambler's Problem*. A gambler has the opportunity to make bets on the outcomes of a sequence of coin flips. If the coin comes up heads, then he wins as many dollars as he has staked on that flip, but if it is tails then he loses his stake. The game ends when the gambler wins by reaching his goal of 100 dollars, or loses by running out of money. On each flip, the gambler must decide what portion of his capital to stake, in integer numbers of dollars. This problem can be formulated as an undiscounted, episodic, finite MDP. The state is the gambler's capital, $s \in \{1, 2, \ldots, 99\}$ and the actions are stakes, $a \in \{0, 1, \ldots, \min(s, 100 - s)\}$. The reward is zero on all transitions except those on which the gambler reaches his goal, when it is **+1**. The state-value function then gives the probability of winning from each state. A policy is a mapping from levels of capital to stakes. The optimal policy maximizes the probability of reaching the goal. Let **p** denote the probability of the coin coming up heads. If **p** is known, then the entire problem is known and it can be solved, e.g., by value iteration. Figure 4.6 shows the change in the value function over successive sweeps of value iteration, and the final policy found, for the case of $p = .4$. ◇

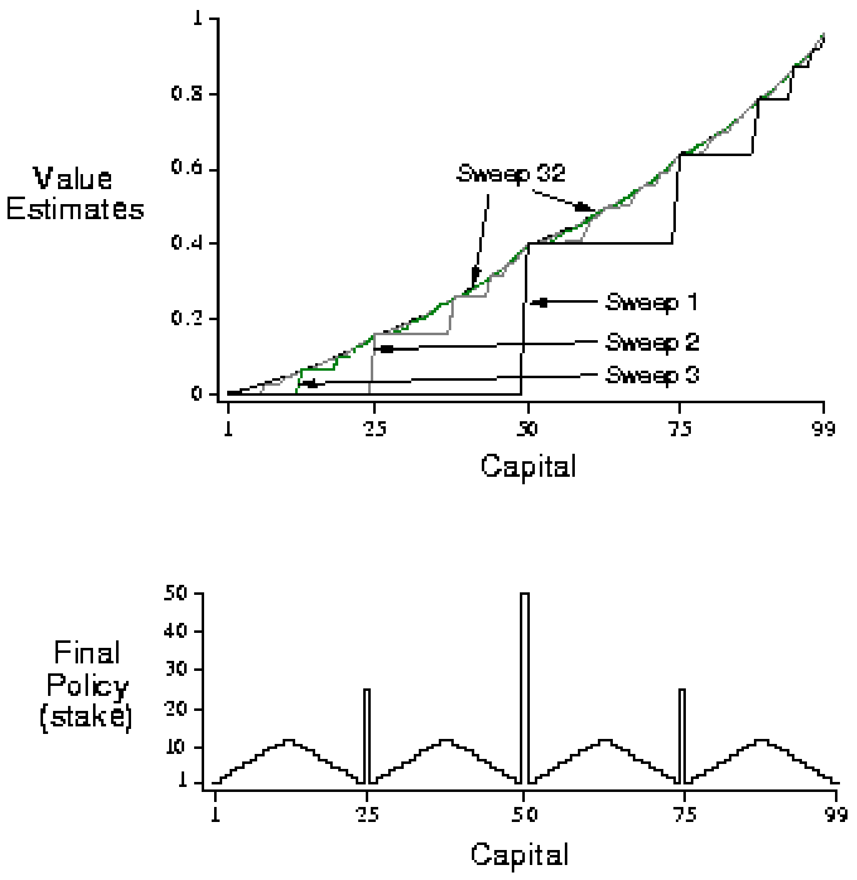


Figure 4.6: The solution to the gamblers problem for $p = 0.4$. The upper graph shows the value function found by successive sweeps of value iteration. The lower graph shows the final policy.

Exercise 4.8

Why does the optimal policy for the gambler's problem have such a curious form? In particular, for capital of 50 it bets it all on one flip, but for capital of 51 it does not. Why is this a good policy?

Exercise 4.9 (programming)

Implement value iteration for the gambler's problem and solve it for $p = .25$ and $p = .55$. In programming, you may find it convenient to introduce two dummy states corresponding to termination with capital of 0 and 100 dollars, giving them values of 0 and 1 respectively. Show your results graphically as in Figure 4.6. Are your results stable as $\theta \rightarrow 0$?

Exercise 4.10

What is the analog of the value iteration backup (4.10) for action values, $Q_{k+1}(s,a)$?

4.5 Asynchronous Dynamic Programming

A major drawback to the DP methods that we have discussed so far is that they involve operations over the entire state set of the MDP, that is, they require sweeps of the state set. If the state set is very large, then even a single sweep can be prohibitively expensive. For example, the game of backgammon has over 10^{20} states. Even if we could perform the value iteration backup on a million states per second, it would take over a thousand years to complete a single sweep.

Asynchronous DP algorithms are in-place iterative DP algorithms that are not organized in terms of systematic sweeps of the state set. These algorithms back up the values of states in any order whatsoever, using whatever values of other states happen to be available. The values of some states may be backed up several times before the values of others are backed up once. To converge correctly, however, an asynchronous algorithm must continue to backup the values of all the states: it can't ignore any state after some point in the computation. Asynchronous DP algorithms allow great flexibility in selecting states to which backup operations are applied.

For example, one version of asynchronous value iteration backs up the value, in place, of only one state, s_k, on each step, **k**, using the value iteration backup (4.10). If $0 \leq \gamma < 1$, asymptotic convergence to V^* is guaranteed given only that all states occur in the sequence $\{s_k\}$ an infinite number of times. (In the undiscounted episodic case, it is possible that there are some orderings of backups that do not result in convergence, but it is relatively easy to avoid these). Similarly, it is possible to intermix policy evaluation and value iteration backups to produce a kind of asynchronous truncated policy iteration. Although the details of this and other more unusual DP algorithms are beyond the scope of this book, it is clear that a few different backups form building blocks that can be used flexibly in a wide variety of sweep-less DP algorithms.

Of course, avoiding sweeps does not necessarily mean that we can get away with less computation. It just means that an algorithm does not need to get locked into any hopelessly long sweep before it can make progress improving a policy. We can try to take advantage of this flexibility by selecting the states to which we apply backups so as to improve the algorithm's rate of progress. We can try to order the backups to let value information propagate from state to state in an efficient way. Some states may not need their values backed up as often as others. We might even try to skip backing up some states entirely if they are not relevant to optimal behavior. Some ideas for doing this are discussed in Chapter 9.

Asynchronous algorithms also make it easier to intermix computation with real-time interaction. To solve a given MDP, we can run an iterative DP algorithm *at the same time that an agent is actually experiencing the MDP*. The agent's experience can be used to determine the states to which the DP algorithm applies its backups. At the same time, the latest value and policy information from the DP algorithm can guide the agent's decision making. For example, we can apply backups to states as the agent visits them. This makes it possible to *focus* the DP algorithm's backups onto parts of the state set that are most relevant to the agent. This kind of focusing is a repeated theme in reinforcement learning.

Richard Sutton
Sat May 31 11:09:21 EDT 1997

4.6 Generalized Policy Iteration

Policy iteration consists of two simultaneous, interacting processes, one making the value function consistent with the current policy (policy evaluation), and the other making the policy greedy with respect to the current value function (policy improvement). In policy iteration, these two processes alternate, each completing before the other begins, but this is not really necessary. In value iteration, for example, only a single iteration of policy evaluation is performed in between each policy improvement process. In asynchronous DP methods, the evaluation and improvement processes are interleaved at an even finer grain. In some cases just a single state is updated in one process before returning to the other. As long as both processes continue to update all states, the ultimate result is typically the same---convergence to the optimal value function and an optimal policy.

We use the term *generalized policy iteration* (GPI) to refer to the general idea of letting policy evaluation and policy improvement processes interact, independent of the granularity and other details of the two processes. Almost all reinforcement learning methods are well described as GPI. That is, all have identifiable policies and value functions, with the policy always being improved with respect to the value function and the value function always being driven towards the value function for the policy. This overall schema for GPI is illustrated in Figure 4.7.

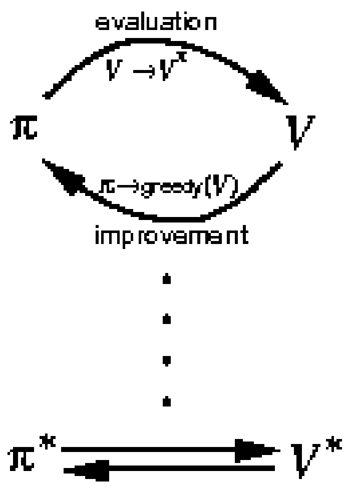
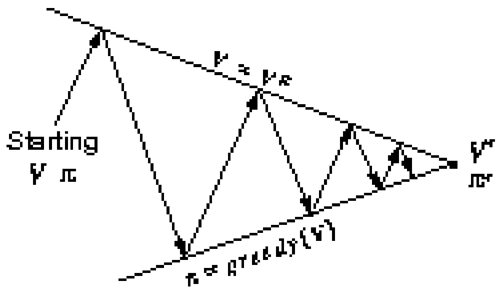


Figure 4.7: Generalized policy iteration.

It is easy to see that if both the evaluation process and the improvement process stabilize, that is, no longer produce changes, then the value function and policy must be optimal. The value function stabilizes only when it is consistent with the current policy, and the policy stabilizes only when it is greedy with respect to the current value function. Thus, both processes stabilize only when a policy has been found that is greedy with respect to its own evaluation function. This implies that the Bellman optimality equation (4.1) holds, and thus that the policy and value functions are optimal.

The evaluation and improvement processes in GPI can be viewed as both competing and cooperating. They compete in the sense that they pull in opposing directions. Making the policy greedy with respect to the value function typically makes the value function incorrect for the changed policy, and making the value function consistent with the policy typically causes that policy to no longer be greedy. In the long run, however, these two convergence processes interact to find a single joint solution: the optimal value function and an optimal policy.

One might also think of the interaction between the evaluation and improvement processes in GPI in terms of two constraints or goals, for example, as two lines in two-dimensional space:



Although the real geometry is much more complicated than this, the diagram suggests what happens even in the real case. Each process drives the value function or policy toward one of the lines representing a solution to one of the two goals. The goals interact because the two lines are not orthogonal. Driving directly toward one goal causes some movement away from the other goal. Inevitably, however, the joint process is brought closer to the overall goal of optimality. The arrows in this diagram correspond to the behavior of policy iteration in that each takes the system all the way to achieving one of the two goals completely. In GPI one could also take smaller, incomplete steps towards each goal. In either case, the two processes together achieve the overall goal of optimality even though neither is attempting to achieve it directly.

Richard Sutton
Sat May 31 11:09:21 EDT 1997

4.7 Efficiency of Dynamic Programming

DP may not be practical for very large problems, but, compared to other methods for solving MDPs, DP is actually quite efficient. If we ignore a few technical details, then the (worst case) time they take to find an optimal policy is polynomial in the number of states and actions. If **n** and **m** respectively denote the number of states and actions, this means that a DP algorithm takes a number of computational operations that is less than some polynomial function of **n** and **m**. A DP algorithm is guaranteed to find an optimal policy in polynomial time even though the total number of (deterministic) policies is m^n. In this sense, DP is exponentially faster than any direct search in policy space could be, because direct search would have to exhaustively examine each policy to provide the same guarantee. Linear-programming methods can also be used to solve MDPs, and in some cases their worst-case convergence guarantees are better than those of DP methods. But linear-programming methods become impractical at a much smaller number of states than DP methods (by a factor of about one hundred). For the largest problems, only DP methods are feasible.

DP is sometimes thought to be of limited applicability because of the *curse of dimensionality* (Bellman, 1957), the fact that the number of states often grows exponentially with the number of state variables. Large state sets do create difficulties, but these are inherent difficulties of the problem, not of DP as a solution method. In fact, DP is comparatively better suited to handling very large state spaces than competitor methods such as direct search and linear programming.

In practice, DP methods can be used with today's computers to solve MDPs with millions of states. Both policy iteration and value iteration are widely used, and it is not clear which if either is better in general. In practice, these methods usually converge much faster their theoretical worst-case run-times, particularly if they are started with good initial value functions or policies.

On problems with very large state spaces, *asynchronous* DP methods are generally prefered. To complete even one sweep of a synchronous method requires computation and memory for every state. For some problems, even this much memory and computation is impractical, yet the problem is still potentially solvable because only a relatively few states occur along optimal solution trajectories. Asynchronous methods and other variations of GPI can often be applied in such cases to find good or optimal policies much faster than synchronous methods.

Richard Sutton
Sat May 31 11:09:21 EDT 1997

4.8 Summary

In this chapter we have become familiar with the basic ideas and algorithms of dynamic programming as they relate to solving finite MDPs. *Policy evaluation* refers to the (typically) iterative computation of the value functions for a given policy. *Policy improvement* refers to the computation of an improved policy given the value function for that policy. Putting these two computations together we obtain *policy iteration* and *value iteration*, the two most popular DP methods. Either of these can be used to reliably compute optimal policies and value functions for finite MDPs given complete knowledge of the MDP.

Classical DP methods operate in sweeps through the state set, performing a *full backup* operation on each state. Each backup updates the value of one state based on the value of all possible successor states and their probability of occurring. Full backups are very closely related to Bellman equations: they are little more than these equations turned into assignment statements. When the backups no longer result in any changes in value, convergence has occurred to values that satisfy the corresponding Bellman equation. Just as there are four primary value functions (V^π, V^*, Q^π, and Q^*), there are four corresponding Bellman equations and four corresponding full backups. An intuitive view of the operation of backups is given by *backup diagrams*.

Insight into DP methods, and, in fact, into almost all reinforcement learning methods, can be gained by viewing them as *generalized policy iteration* (GPI). GPI is the general idea of two interacting convergence processes revolving around an approximate policy and an approximate value function. One process takes the policy as given and performs some form of policy evaluation, changing the value function to be more like the true value function for the policy. The other process takes the value function as given and performs some form of policy improvement, changing the policy to make it better, assuming that the value function is its value function. Although each process changes the basis for the other, overall they work together to find a joint solution: a policy and value function that are unchanged by either process, and which, consequently, are optimal. In some cases, GPI can be proven to converge, most notably for the classical DP methods that we have presented in this chapter. In other cases convergence has not been proven, but still the idea of GPI improves our understanding of the methods.

It is not necessary to perform DP methods in complete sweeps through the state set. *Asynchronous DP* methods are in-place iterative methods that backup states in an arbitrary order, perhaps stochastically determined and using out of date information. Many of these methods can be viewed as fine-grained forms of GPI.

Finally, we note one last special property of DP methods. All of them update their estimates of the value of a state based on estimates of the successor states of that state. That is, they update one estimate based on another estimate. We call this general idea *bootstrapping*. Many reinforcement learning methods perform bootstrapping, even those that do not require, as DP requires, a complete and accurate model of the environment. In the next chapter we explore reinforcement learning methods that do not require a model and do not bootstrap. In the following chapter we explore methods that do not require a model but do bootstrap. These key features and properties are separable, yet can be mixed together in many interesting combinations.

Richard Sutton
Sat May 31 11:09:21 EDT 1997

4.9 Historical and Bibliographical Remarks

The term ``Dynamic Programming'' is due to Bellman (1957), who showed how these methods could be applied to a wide range of problems. Extensive treatments of DP can be found in many texts, including Bertsekas (1995), Bertsekas and Tsitsiklis (1996), Dreyfus and Law (1977), Ross (1983), White (1969), and Whittle (1982, 1983). Our interest in DP is restricted to its use in solving MDPs, but DP also applies to other types of problems. Kumar and Kanal (1988) provide a more general look at DP.

To the best of our knowledge, the first connection between DP and reinforcement learning was made by Minsky (1961) in commenting on Samuel's checkers player. In a footnote, Minsky mentioned that it is possible to apply DP to problems in which Samuel's backing-up process can be handled in closed analytic form. This remark may have mislead artificial intelligence researchers into believing that DP was restricted to analytically tractable problems and therefore largely irrelevant to artificial intelligence. Andreae (1969b) also mentioned DP in the context of reinforcement learning, specifically policy iteration, although he did not make specific connections between DP and learning algorithms. Werbos (1977) suggested an approach to approximating DP called ``heuristic dynamic programming'' that emphasizes gradient-descent methods for continuous-state problems (Werbos, 1982, 1987, 1988, 1989,1992). These methods are closely related to the reinforcement learning algorithms that we discuss in this book. Watkins (1989) was explicit in connecting reinforcement learning to DP, characterizing a class of reinforcement learning methods as ``incremental dynamic programming.''

4.1 -- 4 .4

These sections describe well-established DP algorithms that are covered in any of the general DP references cited above. The policy improvement theorem and the policy iteration algorithm are due to Bellman (1957) and Howard (1960). Our presentation was influenced by the local view of policy improvement taken by Watkins (1989). Our discussion of value iteration as a form of truncated policy iteration is based on the approach of Puterman and Shin (1978) who presented a class of algorithms called *modified policy iteration*, which includes policy iteration and value iteration as special cases. An analysis showing how value iteration can be made to find an optimal policy in finite time is given by Bertsekas (1987).

4.5

Asynchronous DP algorithms are due to Bertsekas (1982, 1983), who also called them distributed DP algorithms. The original motivation for asynchronous DP was its implementation on a multi-processor system with communication delays between processors and no global synchronizing clock. These algorithms are extensively discussed by Bertsekas and Tsitsiklis (1989). Jacobi-style and Gauss-Seidel-style DP algorithms are special cases of the asynchronous version. Williams and Baird (1990) presented DP algorithms that are asynchronous at a finer grain than the ones we have discussed: the backup operations themselves are broken into steps that can be performed asynchronously.

4.7

This section was written with the help of Michael Littman. It is based on Littman, Dean, and Kaelbling (1995).

Next Up Previous
Next: 5 Monte Carlo Methods **Up:** 4 Dynamic Programming **Previous:** 4.8 Summary

Richard Sutton
Fri May 30 12:39:34 EDT 1997

...algorithms.

The iterative policy evaluation algorithm is an example of a classical successive approximation algorithm for solving a system of linear equations (e.g., Varga, 1962). The version of the algorithm that uses two arrays, one holding the old values while the other is updated, is often called a *Jacobi-style* algorithm, after Jacobi's classical use of this method. It is also sometimes called a *synchronous* algorithm because it can be performed in parallel, with separate processors simultaneously updating the values of individual states using input from other processors. The second array is needed to sequentially simulate this parallel computation. The in-place version of the algorithm is often called a *Gauss-Seidel-style* algorithm after the classical Gauss-Seidel algorithm for solving systems of linear equations. In addition to iterative policy evaluation, other DP algorithms can be implemented in these different versions. Bertsekas and Tsitsiklis (1989) provide excellent coverage of these variations and their performance differences.

Richard Sutton
Sat May 31 11:09:21 EDT 1997

5 Monte Carlo Methods

In this chapter we consider our first learning methods for estimating value functions and discovering optimal policies. Unlike the previous chapter, here we do not assume complete knowledge of the environment. Monte Carlo methods require only *experience*---sample sequences of states, actions, and rewards from actual or simulated interaction with an environment. Learning from *actual* experience is striking because it requires no prior knowledge of the environment's dynamics, yet can still attain optimal behavior. Learning from *simulated* experience is also very powerful. Although a model is still required, the model need only generate sample transitions, not the complete probability distributions of all possible transitions that is required by DP methods. In surprisingly many cases it is easy to generate experience sampled according to the desired probability distributions, but infeasible to obtain the distributions in explicit form.

Monte Carlo methods are ways of solving the reinforcement learning problem based on averaging sample returns. To ensure that well-defined returns are available, we define Monte Carlo methods only for episodic tasks. That is, we assume experience is divided into episodes, and that all episodes eventually terminate no matter what actions are selected. It is only upon the completion of an episode that value estimates and policies are changed. Monte Carlo methods are thus incremental in an episode-by-episode sense, but not in a step-by-step sense. The term ``Monte Carlo" is sometimes used more broadly for any estimation method whose operation involves a significant random component. Here we use it specifically for methods based on averaging complete returns (as opposed to methods that learn from partial returns, considered in the next chapter).

Despite the differences between Monte Carlo and dynamic programming (DP) methods, the most important ideas carry over from DP to the Monte Carlo case. Not only do we compute the same value functions, but they interact to attain optimality in essentially the same way. Just as in the DP chapter, we consider first policy evaluation, the computation of V^π and Q^π for a fixed arbitrary policy π, then policy improvement, and, finally, generalized policy iteration. Each of these ideas taken from DP is extended to the Monte Carlo case in which only sample experience is available.

Next Up Previous
Next: 5.1 Monte Carlo Policy **Up:** Contents **Previous:** 4 Dynamic Programming

Richard Sutton
Fri May 30 13:20:35 EDT 1997

5.1 Monte Carlo Policy Evaluation

We begin by considering Monte Carlo methods for learning the state-value function for a given policy. Recall that the value of a state is the expected return---expected cumulative future discounted reward---starting from that state. An obvious way to estimate it from experience, then, is simply to average the returns observed after visits to that state. As more returns are observed, the average should converge to the expected value. This idea underlies all Monte Carlo methods.

In particular, suppose we wish to estimate $V^\pi(s)$, the value of a state **s** under policy π, given a set of episodes obtained by following π and passing through **s**. Each occurrence of state **s** in an episode is called a *visit* to **s**. The *every-visit MC method* estimates $V^\pi(s)$ as the average of the returns following all the visits to **s** in a set of episodes. Within a given episode, the first time **s** is visited is called the *first visit* to **s**. The *first-visit MC method* averages just the returns following first visits to **s**. These two Monte Carlo methods are very similar, but have slightly different theoretical properties. First-visit MC has been most widely studied, dating back to the 1940s, and is the one we focus on in this chapter. We reconsider every-visit MC in Chapter 7 . First-visit MC is shown in procedural form in Figure 5.1.

```
Initialize:
   $\pi \leftarrow$ policy to be evaluated
   $V \leftarrow$ an arbitrary state-value function
   $Returns(s) \leftarrow$ an empty list, for all $s \in S$
Repeat Forever:
   (a) Generate an episode using $\pi$
   (b) For each state s appearing in the episode:
        $R \leftarrow$ return following the first occurrence of $s$
        Append $R$ to $Returns(s)$
        $V(s) \leftarrow$ average($Returns(s)$)
```

Figure 5.1: First-visit MC method for estimating V^π.

Both first-visit MC and every-visit MC converge to $V^\pi(s)$ as the number of visits (or first visits) to **s** goes to infinity. This is easy to see for the case of first-visit MC. In this case each return is an independent, identically distributed estimate of $V^\pi(s)$. By the law of large numbers the sequence of averages of these estimates converges to their expected value. Each average is itself an unbiased estimate, and the standard deviation of its error falls as $1/\sqrt{n}$, where **n** is the number of returns averaged. Every-visit MC is less straightforward, but its estimates also converge asymptotically to $V^\pi(s)$ (Singh and Sutton, 1996).

The use of Monte Carlo methods is best illustrated through an example.

Example 5.1 *Blackjack* is a popular casino card game. The object is to obtain cards the sum of whose numerical values is as great as possible without exceeding 21. All face cards count as 10, and the ace can count either as 1 or as 11. We consider the version in which each player competes independently against the dealer. The game begins with two cards dealt to both dealer and player. One of the dealer's cards is face up and the other is face down. If the player has twenty-one immediately (an ace and a 10-card), it is called a *natural*. He then wins unless the dealer also has a natural, in which case the game is a draw. If the player does not have a natural, then he can request additional cards, one by one (``hits"), until he either stops (``sticks") or exceeds 21 (``goes bust"). If he goes bust he loses, but if he sticks it then becomes the dealer's turn. The dealer hits or sticks according to a fixed strategy without choice: he sticks on any sum of 17 or greater, and hits otherwise. If the dealer goes bust then the player wins, otherwise the outcome---win, lose, or draw---is determined by whose final sum is closest to 21.

Playing blackjack is naturally formulated as an episodic finite MDP. Each game of blackjack is an episode. Rewards of **+1**, **-1**, and **0** are given for winning, losing and drawing respectively. All rewards within a game are zero, and we do not discount ($\gamma = 1$); therefore these terminal rewards are also the returns. The player's actions are to hit or stick. The states depend on the player's cards and the dealer's showing card. We assume that cards are dealt from an infinite deck (i.e., with replacement) so that there is no advantage to keeping track of the cards already dealt. If the player holds an ace that he could count as 11 without going bust, then the ace is said to be *usable*. In this case it is always counted as 11 because counting it as 1 would make the sum 11 or less, in which case there is no decision to be made because obviously the player should always hit. Thus, the player makes decisions on the basis of three variables: his current sum [12--21], the dealers one showing card [Ace--10], and whether or not he holds a usable ace. This makes for a total of 200 states.

Consider the policy that sticks if the player's sum is 20 or 21, and otherwise hits. To find the state-value function for this policy by a Monte Carlo approach one simulates many blackjack games using the policy and averages the returns following each state. Note that in this task the same state never recurs within one episode, so there is no difference between first-visit and every-visit MC methods. In this way, we obtained the estimates of the state-value function shown in Figure 5.2. The estimates for states with a usable ace are less certain and less regular because these states are less common. In any event, after 500,000 games the value function is very well approximated.

Although we have complete knowledge of the environment in this task, it would not be easy to apply DP policy evaluation to compute the value function. DP methods require the distribution of next events, in particular, they require the quantities $\mathcal{P}^a_{ss'}$ and $\mathcal{R}^a_{ss'}$, and it is not easy to determine these for blackjack. For example, suppose the player's sum is 14 and he chooses to stick. What is his expected reward as a function of the dealer's showing card? All of these expected rewards and transition probabilities must be computed *before* DP can be applied, and such computations are often complex and error-prone. In contrast, generating the sample games required by Monte Carlo methods is very easy. This is the case surprisingly often; the ability of Monte Carlo methods to work with sample episodes alone can be a significant advantage even when one does have knowledge of the environment's dynamics. $\diamond$

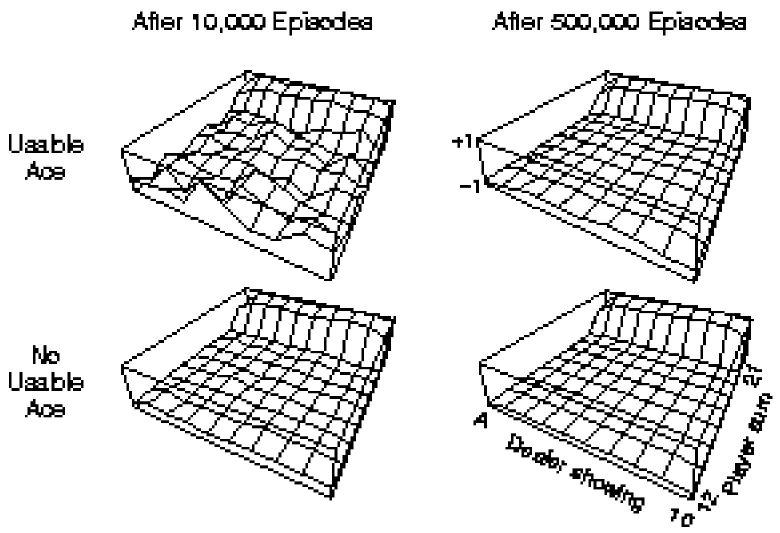


Figure 5.2: Approximate state-value functions for the blackjack policy that sticks only on 20 or 21, computed by Monte Carlo policy evaluation.

Can we generalize the idea of backup diagrams to Monte Carlo algorithms? The general idea of a backup diagram is to show at the top the root node to update and to show below all the transitions and leaf nodes whose rewards and estimated values contribute to the update. For Monte Carlo estimation of V^{π} we have a state node as root and below the entire sequence of transitions along a particular episode, ending at the terminal state, as in Figure 5.3. Whereas the DP diagram (Figure 3 .4a) shows all possible transitions, the Monte Carlo diagram shows only those sampled on the one episode. Whereas the DP diagram includes only one-step transitions, the Monte Carlo diagram goes all the way to the end of the episode. These differences in the diagrams accurately reflect the fundamental differences between the algorithms.

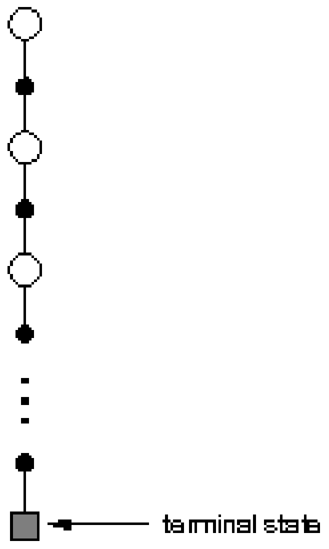


Figure 5.3: The backup diagram for Monte Carlo estimation of V^{π}.

An important fact about Monte Carlo methods is that the estimates for each state are independent. The estimate for one state does not build upon the estimate of any other state, as they do in dynamic programming. In other words, Monte Carlo methods do not ``bootstrap'' as we described it in the previous chapter.

In particular, note that the computational expense of estimating the value of a single state is independent of the number of states. This can make Monte Carlo methods particularly attractive when one requires the value of only a subset of the states. One can generate many sample episodes starting from these states, averaging returns only for these states, and ignoring all the others. This is a third advantage Monte Carlo methods can have over DP methods (after the ability to learn from actual experience and from simulated experience).

Example 5.2 *Soap Bubble*. Suppose a wire frame forming a closed loop is dunked in soapy water to form a soap surface or bubble conforming at its edges to the wire frame. If the geometry of the wire frame is irregular but known, how can you compute the shape of the surface? The shape has the property that the total force on each point exerted by neighboring points is zero (or else the shape would change). This means that the surface's height at any point is the average of its heights at points in a small circle around that point. In addition, the surface must meet up at its boundaries with the wire frame. The usual approach to problems of this kind is to put a grid over the area covered by the surface and solve for its height at the grid points by an iterative computation. Grid points at the boundary are forced to the wire frame, and all others are adjusted toward the average of the heights of their four nearest neighbors. This process then iterates, much like DP's iterative policy evaluation, and ultimately converges to a close approximation to the desired surface.

This is the kind of problem for which Monte Carlo methods were originally designed. Instead of the iterative computation described above, imagine standing on the surface and taking a random walk, stepping randomly from grid point to neighboring grid point, with equal probability, until you reach the boundary. It turns out that the expected value of the height at the boundary is a close approximation to the height of the desired surface at the starting point (if fact, is is exactly the value computed by the iterative method described above). Thus, one can closely approximate the height of the surface at a point by simply averaging together the boundary heights of many walks started at the point. If one is interested in only the value at one point, or any fixed set of points, then this Monte Carlo method is usually far more efficient than the iterative method based on local consistency.
$\diamond$

Exercise 5.1

Consider the diagrams on the right in Figure 5.2. Why does the value function jump up for the last two rows in the rear? Why does it drop off for the whole last row on the left? Why are the frontmost values higher in the upper diagram than in the lower?

Richard Sutton
Fri May 30 13:20:35 EDT 1997

5.2 Monte Carlo Estimation of Action Values

If a model is not available, then it is particularly useful to estimate *action* values rather than *state* values. With a model, state values alone are sufficient to determine a policy; one simply looks ahead one step and chooses whichever action leads to the best combination of reward and next state, as we did in the chapter on DP. Without a model, however, state values alone are not sufficient. One must explicitly estimate the value of each action in order for the values to be useful in suggesting a policy. Thus, one of our primary goals for Monte Carlo methods is to estimate Q^*. To achieve this, we first consider another policy evaluation problem.

The policy evaluation problem for action values is to estimate $Q^\pi(s,a)$, the expected return when starting in state **s**, taking action **a**, and thereafter following policy π. The Monte Carlo methods here are essentially the same as just presented for state values. The every-visit MC method estimates the value of a state-action pair as the average of the returns that have followed visits to the state in which the action was selected. The first-visit MC method averages the returns following the first time in each episode that the state was visited and the action was selected. These methods converge quadratically, as before, to the true expected values as the number of visits to each state-action pair approaches infinity.

The only complication is that many relevant state-action pairs may never be visited. If π is a deterministic policy, then in following π one will observe returns only for one of the actions from each state. With no returns to average, the Monte Carlo estimates of the other actions will not improve with experience. This is a serious problem because the purpose of learning action values is to help in choosing among the actions available in each state. To compare alternatives we need to estimate the value of *all* the actions from each state, not just the one we currently favor.

This is the general problem of *maintaining exploration*, as discussed in the context of the **n** -armed bandit problem in Chapter 2 . For policy evaluation to work for action values, we must assure continual exploration. One way to do this is by specifying that the first step of each episode starts at a state-action *pair*, and that every such pair has a nonzero probability of being selected as the start. This guarantees that all state-action pairs will be visited an infinite number of times in the limit of an infinite number of episodes. We call this the assumption of *exploring starts*.

The assumption of exploring starts is sometimes useful, but of course it cannot be relied upon in general, particularly when learning directly from real interactions with an environment. In that case the starting conditions are unlikely to be that helpful. The most common alternative approach to assuring that all state-action pairs are encountered is simply to consider only policies that are stochastic with a nonzero probability of selecting all actions. We discuss two important variants of this approach in later sections. For now we retain the assumption of exploring starts and complete the presentation of a full Monte Carlo control method.

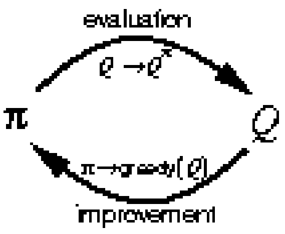
Exercise 5.2

What is the backup diagram for Monte Carlo estimation of Q^{π}?

Richard Sutton
Fri May 30 13:20:35 EDT 1997

5.3 Monte Carlo Control

We are now ready to consider how Monte Carlo estimation can be used in control, that is, to approximate optimal policies. The overall idea is to proceed according to the same pattern as in the DP chapter, that is, according to the idea of generalized policy iteration (GPI). In GPI one maintains both an approximate policy and an approximate value function. The value function is repeatedly altered to more closely approximate the value function for the current policy, and the policy is repeatedly improved with respect to the current value function:



These two kinds of changes work against each other to some extent, as each creates a moving target for the other, but together they cause both policy and value function to approach optimality.

To begin, let us consider a Monte Carlo version of classical policy iteration. In this method, we perform alternating complete steps of policy evaluation and policy improvement, beginning with an arbitrary policy π_0 and ending with the optimal policy and optimal action-value function:

$$\pi_0 \xrightarrow{\mathrm{E}} Q^{\pi_0} \xrightarrow{\mathrm{I}} \pi_1 \xrightarrow{\mathrm{E}} Q^{\pi_1} \xrightarrow{\mathrm{I}} \pi_2 \xrightarrow{\mathrm{E}} \cdots \xrightarrow{\mathrm{I}} \pi^* \xrightarrow{\mathrm{E}} Q^*$$

where $\xrightarrow{\mathrm{E}}$ denotes a complete policy evaluation and $\xrightarrow{\mathrm{I}}$ denotes a complete policy improvement. Policy evaluation is done exactly as described in the preceding section. Many episodes are experienced, with the approximate action-value function approaching the true function asymptotically. For the moment, let us assume that we do indeed observe an infinite number of episodes and that, in addition, the episodes are generated with exploring starts. Under these assumptions, the Monte Carlo methods will compute each Q^{π_k} exactly, for arbitrary π_k.

Policy improvement is done by making the policy greedy with respect to the current value function. In this case we have an *action*-value function, and therefore no model is needed to construct the greedy policy. For any action-value function **Q**, the corresponding greedy policy is the one that, for each $s \in \mathcal{S}$, deterministically chooses an action with maximal **Q**-value:

$$\pi(s) = \arg\max_a Q(s, a). \qquad (5.1)$$

Policy improvement then can be done simply by constructing each π_{k+1} as the greedy policy with respect to Q^{π_k}. The policy improvement theorem (Section 4 .2) then applies to π_k and π_{k+1} because, for all $s \in \mathcal{S}$,

$$\begin{aligned} Q^{\pi_k}(s, \pi_{k+1}(s)) &= Q^{\pi_k}(s, \arg\max_a Q^{\pi_k}(s, a)) \\ &= \max_a Q^{\pi_k}(s, a) \\ &\geq Q^{\pi_k}(s, \pi_k(s)) \\ &\geq V^{\pi_k}(s). \end{aligned}$$

As we discussed in the previous chapter, the theorem assures us that each π_{k+1} is uniformly better than π_k, unless it is equal to π_k, in which case they are both optimal policies. This in turn assures us that the overall process converges to the optimal policy and optimal value function. In this way Monte Carlo methods can be used to find optimal policies given only sample episodes and no other knowledge of the environment's dynamics.

We made two unlikely assumptions above in order to easily obtain this guarantee of convergence for the Monte Carlo method. One was that the episodes have exploring starts, and the other was that policy evaluation could be done with an infinite number of episodes. We consider how to remove the first assumption later, in the next few sections.

For now we focus on the assumption that policy evaluation operates on an infinite number of episodes. This assumption is relatively easy to remove. In fact, the same issue arises even in classical DP methods such as iterative policy evaluation, which also converge only asymptotically to the true value function. In both DP and Monte Carlo cases there are two ways to solve the problem. One is to hold firm to the idea of approximating Q^{π_k} in each policy evaluation. Measurements and assumptions are made to obtain bounds on the magnitude and probability of error in the estimates, and then sufficient steps are taken during each policy evaluation to assure that these bounds are sufficiently small. This approach can probably be made completely satisfactory in the sense of guaranteeing correct convergence up to some level of approximation. However, it is also likely to require far too many episodes to be useful in practice on any but the smallest problems.

The second approach to avoiding the infinite number of episodes nominally required for policy evaluation is to forego trying to complete policy evaluation before returning to policy improvement. On each evaluation step we move the value function *toward* Q^{π_k}, but we do not expect to actually get close except over many steps. We used this idea before when we first introduced the idea of GPI in Section 4 .6 . One extreme form of the idea is value iteration, in which only one iteration of iterative policy evaluation is performed between each step of policy improvement. The in-place version of value iteration is even more extreme; there we alternate between improvement and evaluation steps for single states.

For Monte Carlo policy evaluation it is natural to alternate between evaluation and improvement on an episode-by-episode basis. After each episode, the observed returns are used for policy evaluation, and then the policy is improved at all the states visited in the episode. A complete simple algorithm along these lines is given in Figure 5.4. We call this algorithm *Monte Carlo ES*, for Monte Carlo with Exploring Starts.

```
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
   $Q(s,a) \leftarrow$ arbitrary
   $\pi(s) \leftarrow$ arbitrary
   $Returns(s,a) \leftarrow$ empty list
Repeat Forever:
   (a) Generate an episode using $\pi$
   (b) For each pair s,a appearing in the episode:
           $R \leftarrow$ return following the first occurrence of $s,a$
           Append $R$ to $Returns(s,a)$
           $Q(s,a) \leftarrow \mathrm{average}(Returns(s,a))$
   (c) For each s in the episode:
```

$$\pi(s) \leftarrow \arg\max_a Q(s,a)$$

Figure 5.4: Monte Carlo ES: A Monte Carlo control algorithm assuming exploring starts.

In Monte Carlo ES, all the returns for each state-action pair are accumulated and averaged, irrespective of what policy was in force when they were observed. It is easy to see that Monte Carlo ES cannot converge to any suboptimal policy. If it did, then the value function would eventually converge to the corresponding value function, and that would in turn cause the policy to change unless it was optimal. Convergence to this optimal fixed point seems inevitable as the changes to the action-value function decrease over time, but has not yet been formally proved. In our opinion this is one of the most important open theoretical questions in reinforcement learning.

Example 5.3 *Solving Blackjack*. It is straightforward to apply Monte Carlo ES to blackjack. Since the episodes are all simulated games, it is easy to arrange for exploring starts that include all possibilities. In this case one simply picks the dealer's cards, the player's sum, and whether or not the player has a usable ace, all at random with equal probability. As the initial policy we use the policy evaluated in the previous blackjack example, that which sticks only on 20 or 21. The initial action-value function can simply be zero for all state-action pairs. Figure 5.5 shows the optimal policy for blackjack found by Monte Carlo ES. This policy is the same as the ``basic'' strategy of Thorp (1966) with the sole exception of the leftmost notch in the policy for a usable ace, which is not present in Thorp's strategy. We are uncertain of the reason for this discrepancy, but confident that what is shown here is indeed the optimal policy for the version of blackjack we have described. $\diamond$

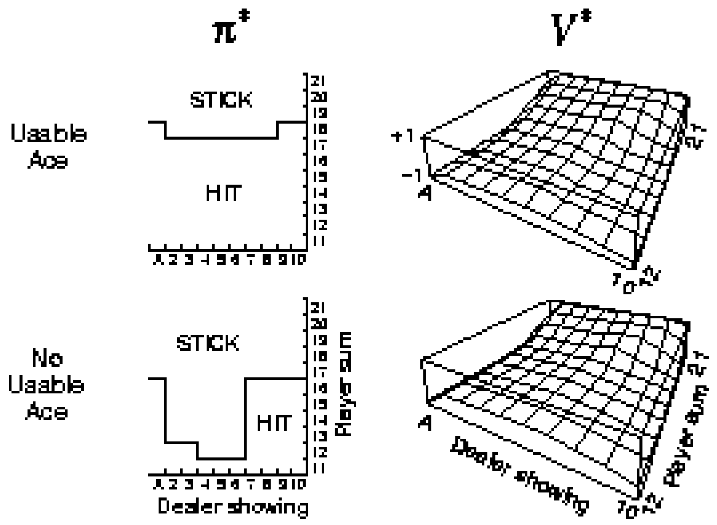


Figure 5.5: The optimal policy and state-value function for blackjack, found by Monte Carlo ES (Figure 5.4). The state-value function shown was computed from the action-

value function found by Monte Carlo ES.

Next Up Previous

Next: 5.4 On-Policy Monte Carlo **Up:** 5 Monte Carlo Methods **Previous:** 5.2 Monte Carlo Estimation

---

Richard Sutton
Fri May 30 13:20:35 EDT 1997

5.4 On-Policy Monte Carlo Control

How can we avoid the unlikely assumption of exploring starts? The only general way to ensure that all actions are selected infinitely often is for the agent to continue to select them. There are two approaches to ensuring this, resulting in what we call *on-policy* methods and *off-policy* methods. In this section we treat on-policy methods, which are distinguished in that they attempt to evaluate and improve the same policy that they use to make decisions. In this section we present an on-policy Monte Carlo control method in order to illustrate the idea.

In on-policy control methods the policy is *soft*, meaning that $\pi(s,a) > 0$ for all $s \in \mathcal{S}$ and all $a \in \mathcal{A}(s)$. There are many possible variations on on-policy methods. One possibility is to gradually shift the policy toward a deterministic optimal policy. Many of the methods discussed in Chapter 2 provide mechanisms for this. The on-policy method we present in this section uses ϵ *-greedy* policies, meaning that most of the time they choose an action that has maximal estimated action value, but with probability ϵ they instead select an action at random. That is, all non-greedy actions are given the minimal probability of selection, $\frac{\epsilon}{|\mathcal{A}(s)|}$, and the remaining bulk of the probability, $1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|}$, is given to the greedy action. The ϵ -greedy policies are examples of ϵ *-soft* policies, defined as policies for which $\pi(s,a) \geq \frac{\epsilon}{|\mathcal{A}(s)|}$ for all states and actions, for some $\epsilon > 0$. Among ϵ -soft policies, ϵ -greedy policies are in some sense those that are closest to greedy.

The overall idea of on-policy Monte Carlo control is still that of GPI. As in Monte Carlo ES, we use first-visit MC methods to estimate the action-value function for the current policy. Without the assumption of exploring starts, however, we cannot simply improve the policy by making it greedy with respect to the current value function, because that would prevent further exploration of non-greedy actions. Fortunately, GPI does not require that the policy be taken all the way to a greedy policy, only that it be moved *toward* a greedy policy. In our on-policy method we will move it only to an ϵ -greedy policy. For any ϵ -soft policy, π, any ϵ -greedy policy with respect to Q^π is guaranteed to be better than or equal to π.

That any ϵ -greedy policy with respect to Q^π is an improvement over any ϵ -soft policy π is assured by the policy improvement theorem. Let π' be the ϵ -greedy policy. The conditions of the policy improvement theorem apply because for any $s \in \mathcal{S}$:

$$\begin{aligned}
Q^\pi(s,\pi'(s)) &= \sum_a \pi'(s,a) Q^\pi(s,a) \\
&= \frac{\epsilon}{|\mathcal{A}(s)|}\sum_a Q^\pi(s,a) \;+\; (1-\epsilon)\max_a Q^\pi(s,a) \qquad (5.2)\\
&\geq \frac{\epsilon}{|\mathcal{A}(s)|}\sum_a Q^\pi(s,a) \;+\; (1-\epsilon)\sum_a \frac{\pi(s,a)-\frac{\epsilon}{|\mathcal{A}(s)|}}{1-\epsilon} Q^\pi(s,a)
\end{aligned}$$

(the sum is a weighted average with non-negative weights summing to one, and as such it must be less than or equal to the largest number averaged)

$$\begin{aligned}
&= \frac{\epsilon}{|\mathcal{A}(s)|}\sum_a Q^\pi(s,a) \;-\; \frac{\epsilon}{|\mathcal{A}(s)|}\sum_a Q^\pi(s,a) \;+\; \sum_a \pi(s,a) Q^\pi(s,a) \\
&= V^\pi(s).
\end{aligned}$$

Thus, by the policy improvement theorem, $\pi' \geq \pi$ (i.e., $V^{\pi'}(s) \geq V^\pi(s), \forall s \in \mathcal{S}$). We now prove that equality can only hold when both π' and π are optimal among the ϵ -soft policies, i.e., that they are better than or equal to all other ϵ -soft policies.

Consider a new environment that is just like the original environment, except with the requirement that policies be ϵ -soft ``moved inside" the environment. The new environment has the same action and state set as the original and behaves as follows. If in state **s** and taking action **a**, then with probability $1-\epsilon$ the new environment behaves exactly like the old environment. With probability ϵ it re-picks the action at random, with equal probabilities, and then behaves like the old environment with the new, random action. The best one can do in this new environment with general policies is the same as the best one could do in the original environment with ϵ -soft policies. Let $\tilde{V}^*$ and $\tilde{Q}^*$ denote the optimal value functions for the new environment. Then a policy π is optimal among ϵ -soft policies if and only if $V^\pi = \tilde{V}^*$. From the definition of $\tilde{V}^*$ we know that it is the unique solution to

$$\begin{aligned}
\tilde{V}^*(s) &= (1-\epsilon)\max_a \tilde{Q}^*(s,a) + \frac{\epsilon}{|\mathcal{A}(s)|}\sum_a \tilde{Q}^*(s,a) \\
&= (1-\epsilon)\max_a \sum_{s'} \mathcal{P}^a_{ss'}\left[\mathcal{R}^a_{ss'} + \gamma \tilde{V}^*(s')\right] \\
&\qquad + \frac{\epsilon}{|\mathcal{A}(s)|}\sum_a \sum_{s'} \mathcal{P}^a_{ss'}\left[\mathcal{R}^a_{ss'} + \gamma \tilde{V}^*(s')\right].
\end{aligned}$$

And from (5.2) we know that, when equality holds and the ϵ -soft policy π is no longer improved, then

$$\begin{aligned}
V^\pi(s) &= (1-\epsilon)\max_a Q^\pi(s,a) + \frac{\epsilon}{|\mathcal{A}(s)|}\sum_a Q^\pi(s,a) \\
&= (1-\epsilon)\max_a \sum_{s'} \mathcal{P}^a_{ss'}\left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')\right] \\
&\qquad + \frac{\epsilon}{|\mathcal{A}(s)|}\sum_a \sum_{s'} \mathcal{P}^a_{ss'}\left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')\right].
\end{aligned}$$

but this equation is the same as the previous, except for the substitution of V^π for $\tilde{V}^*$. Since $\tilde{V}^*$ is the unique solution, $V^\pi = \tilde{V}^*$.

In essence, we have shown in the last few pages that policy iteration works for ϵ -soft policies. Using the natural notion of greedy policy for ϵ -soft policies, one is assured of improvement on every step, except when the best policy has been found among the ϵ -soft policies. This analysis is independent of how the action-value functions are determined at each stage, but it does assume they are computed exactly. This brings us to roughly the same point as in the previous section. Now we achieve the best policy only among the ϵ -soft policies, but, on the other hand, we have eliminated the assumption of exploring starts. The complete algorithm is given in Figure 5.6.

```
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
    $Q(s,a) \leftarrow$ arbitrary
    $Returns(s,a) \leftarrow$ empty list
    $\pi \leftarrow$ an arbitrary $\epsilon$ -soft policy
Repeat Forever:
    (a) Generate an episode using $\pi$
    (b) For each pair s,a appearing in the episode:
            $R \leftarrow$ return following the first occurrence of $s,a$
            Append $R$ to $Returns(s,a)$
            $Q(s,a) \leftarrow \text{average}(Returns(s,a))$
    (c) For each s in the episode:
            $a^* \leftarrow \arg\max_a Q(s,a)$
            For all $a \in \mathcal{A}(s)$:
                $\pi(s,a) \leftarrow \begin{cases} 1-\epsilon+\epsilon/|\mathcal{A}(s)| & \text{if } a = a^* \\ \epsilon/|\mathcal{A}(s)| & \text{if } a \neq a^* \end{cases}$
```

Figure 5.6: An ϵ -soft on-policy Monte Carlo control algorithm.

Next Up Previous
Next: 5.5 Evaluating One Policy **Up:** 5 Monte Carlo Methods **Previous:** 5.3 Monte Carlo Control

Richard Sutton
Fri May 30 13:20:35 EDT 1997

5.5 Evaluating One Policy While Following Another

So far we have considered methods for estimating the value functions for a policy given an infinite supply of episodes generated using that policy. Suppose now that all we have are episodes generated from a different policy. That is, suppose we wish to estimate V^{π} or Q^{π}, but all we have are episodes following π', where $\pi' \neq \pi$. Can we do it?

Happily, in many cases we can. Of course, in order to use episodes from π' to estimate values for π, we require that every action taken under π is also taken, at least occasionally, under π'. That is, we require that $\pi(s,a) > 0$ imply $\pi'(s,a) > 0$. In the episodes generated using π', consider the **i**th first visit to state **s** and the complete sequence of states and actions following that visit. Let $p_i(s)$ and $p_i'(s)$ denote the probabilities of that complete sequence happening given policies π and π' and starting from **s**. Let $R_i(s)$ denote the corresponding observed return from state **s**. To average these to obtain an unbiased estimate of $V^{\pi}(s)$, we need only weight each return by its relative probability of occurring under π and π', that is, by $p_i(s)/p_i'(s)$. The desired Monte Carlo estimate after observing n_s returns from state **s** is then

$$V(s) = \frac{\sum_{i=1}^{n_s} \frac{p_i(s)}{p_i'(s)} R_i(s)}{\sum_{i=1}^{n_s} \frac{p_i(s)}{p_i'(s)}}. \qquad (5.3)$$

This equation involves the probabilities $p_i(s)$ and $p_i'(s)$, which are normally considered unknown in applications of Monte Carlo methods. Fortunately, here we need only their ratio, $p_i(s)/p'_i(s)$, which *can* be determined with no knowledge of the environment's dynamics. Let $T_i(s)$ be the time of termination of the i th episode involving state **s**. Then

$$p_i(s_t) = \prod_{k=t}^{T_i(s)-1} \pi(s_k, a_k)\mathcal{P}_{s_k s_{k+1}}^{a_k}$$

and

$$\frac{p_i(s_t)}{p'_i(s_t)} = \frac{\prod_{k=t}^{T_i(s)-1} \pi(s_k, a_k)\mathcal{P}_{s_k s_{k+1}}^{a_k}}{\prod_{k=t}^{T_i(s)-1} \pi'(s_k, a_k)\mathcal{P}_{s_k s_{k+1}}^{a_k}} = \prod_{k=t}^{T_i(s)-1} \frac{\pi(s_k, a_k)}{\pi'(s_k, a_k)}.$$

Thus the weight needed in (5.3), $p_i(s)/p'_i(s)$, depends only on the two policies and not at all on the environment's dynamics.

Exercise 5.3

What is the Monte Carlo estimate analogous to (5.3) for *action* values, given returns generated using π'?

Richard Sutton
Fri May 30 13:20:35 EDT 1997

5.6 Off-Policy Monte Carlo Control

We are now ready to present an example of the second class of learning control methods we consider in this book: off-policy methods. Recall that the distinguishing feature of on-policy methods is that they estimate the value of a policy while using it for control. In off-policy methods these two functions are separated. The policy used to generate behavior, called the *behavior* policy, may in fact be unrelated to the policy that is evaluated and improved, called the *estimation* policy. An advantage of this separation is that the estimation policy may be deterministic, e.g., greedy, while the behavior policy can continue to sample all possible actions.

Off-policy Monte Carlo control methods use the technique presented in the preceding section for estimating the value function for one policy while following another. They follow the behavior policy while learning about and improving the estimation policy. This technique requires that the behavior policy have a nonzero probability of selecting all actions that might be selected by the estimation policy. To explore all possibilities, we require that the behavior policy be soft.

```
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
        $Q(s,a) \leftarrow$ arbitrary
        $N(s,a) \leftarrow 0$                              ; Numerator and
        $D(s,a) \leftarrow 0$                              ; Denominator of $Q(s,a)$
        $\pi \leftarrow$ an arbitrary deterministic policy
Repeat Forever:
    (a) Select a policy $\pi'$ and use it to generate an episode:
                $s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T, s_T$
    (b) $\tau \leftarrow$ latest time at which $a_\tau \neq \pi(s_\tau)$
    (c) For each pair s,a appearing in the episode after $\tau$:
        $t \leftarrow$ the time of first occurrence (after $\tau$) of $s,a$
        $w \leftarrow \prod_{k=t+1}^{T-1} \frac{1}{\pi'(s_k,a_k)}$
```

```
            $N(s,a) \leftarrow N(s,a) + wR_t$
            $D(s,a) \leftarrow D(s,a) + w$
            $Q(s,a) \leftarrow \frac{N(s,a)}{D(s,a)}$
    (d) For each $s \in \mathcal{S}$:
          $\pi(s) \leftarrow \arg\max_a Q(s,a)$
```

Figure 5.7: An off-policy Monte Carlo control algorithm.

Figure 5.7 shows an off-policy Monte Carlo method, based on GPI, for computing Q^*. The behavior policy π' is maintained as an arbitrary soft policy. The estimation policy π is the greedy policy with respect to **Q**, an estimate of Q^π. The behavior policy chosen in (a) can be anything, but in order to assure convergence of π to the optimal policy, an infinite number of returns suitable for use in (c) must be obtained for each pair of state and action. This can be assured by careful choice of the behavior policy. For example, any ϵ -soft behavior policy will suffice.

A potential problem is that this method only learns from the *tails* of episodes, after the last non-greedy action. If non-greedy actions are frequent, then learning will be very slow, particularly for states appearing in the early portions of long episodes. Potentially, this could greatly slow learning. There has been insufficient experience with off-policy Monte Carlo methods to assess how serious this problem is.

Exercise 5.4 (programming) *Racetrack*. Consider driving a race car around a turn like those shown in Figure 5.8. You want to go as fast as possible, but not so fast as to run off the track. In our simplified racetrack, the car is at one of a discrete set of grid positions, the cells in the diagram. The velocity is also discrete, a number of grid cells moved horizontally and vertically per time step. The actions are increments to the velocity components. Each may be changed by **+1**, **-1**, or **0** in one step, for a total of nine actions. Both components are restricted to be non-negative and less than 5, and they cannot both be zero. Each trial begins in one of the randomly selected start states and ends when the car crosses the finish line. The rewards are **-1** for each step that stays on the track, and **-5** if the agent tries to drive off the track. Actually leaving the track is not allowed, but the state is always advanced at least one cell along the horizontal or vertical axes. With these restrictions and considering only right turns, such as shown in the figure, all episodes are guaranteed to terminate, yet the optimal policy is unlikely to be excluded. To make the task more challenging, we assume that on half of the time steps the position is displaced forward or to the right by one additional cell beyond that specified by the velocity. Apply the on-policy Monte Carlo control method to this task to compute the optimal policy from each starting state. Exhibit several trajectories following the optimal policy.

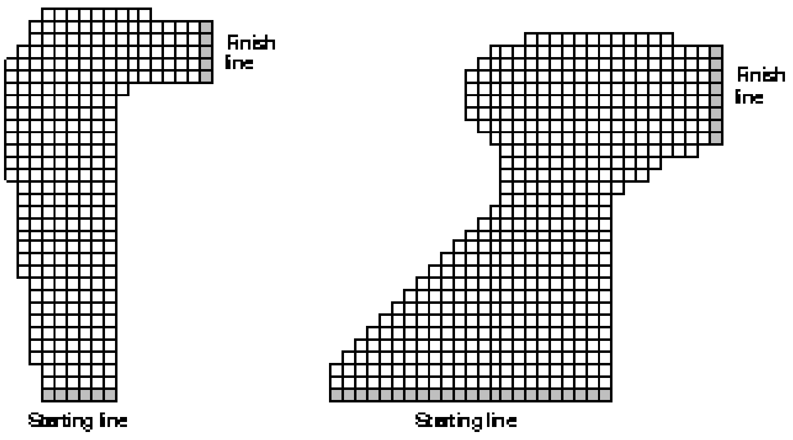


Figure 5.8: A couple of right turns for the racetrack task.

Next Up Previous

Next: 5.7 Incremental Implementation **Up:** 5 Monte Carlo Methods **Previous:** 5.5 Evaluating One Policy

---

Richard Sutton
Fri May 30 13:20:35 EDT 1997

5.7 Incremental Implementation

Monte Carlo methods can be implemented incrementally, on an episode-by-episode basis, using extensions of techniques described in Chapter 2. Monte Carlo methods use averages of *returns* just as some of the methods for solving **n** -armed bandit tasks described in Chapter 2 use averages of *rewards*. The techniques in Sections 2 .5 and 2 .6 extend immediately to the Monte Carlo case. These enable Monte Carlo methods to incrementally process each new return with no increase in computation or memory over episodes.

There are two differences between the Monte Carlo and bandit cases. One is that the Monte Carlo case typically involves multiple situations, that is, a different averaging process for each state, whereas bandit problems involve just one state (at least in the simple form treated in Chapter 2). The other difference is that the reward distributions in bandit problems are typically stationary, whereas in Monte Carlo methods the return distributions are typically nonstationary. This is because the returns depend on the policy, and the policy is typically changing and improving over time.

The incremental implementation described in Section 2 .5 handles the case of simple or arithmetic averages, in which each return is weighted equally. Suppose we instead want to implement a *weighted* average, in which each return R_n is weighted by w_n, and we want to compute

$$V_n = \frac{\sum_{k=1}^{n} w_k R_k}{\sum_{k=1}^{n} w_k} \qquad (5.4)$$

For example, the method described for estimating one policy while following another in Section 5 .5 uses weights of $w_n(s) = p_n(s)/p_n'(s)$. Weighted averages also have a simple incremental update rule. In addition to keeping track of V_n, we must also maintain for each state the cumulative sum W_n of the weights given to the first **n** returns. The update rule for V_n is

$$V_{n+1} = V_n + \frac{w_{n+1}}{W_{n+1}}\left[R_{n+1} - V_n\right] \qquad (5.5)$$

and

$$W_{n+1} = W_n + w_{n+1}$$

where $W_0 = 0$.

Exercise 5.5

Modify the algorithm for first-visit MC policy evaluation (Figure 5.1) to use the incremental implementation for stationary averages described in Section 2 .5 .

Exercise 5.6

Derive the weighted-average update rule (5.5) from (5.4). Follow the pattern of the derivation of the unweighted rule (2 .4) from (2 .1).

Exercise 5.7

Modify the algorithm for the off-policy control algorithm (Figure 5.7) to use the method described above for incrementally computing weighted averages.

Richard Sutton
Fri May 30 13:20:35 EDT 1997

5.8 Summary

The Monte Carlo methods presented in this chapter learn value functions and optimal policies from experience in the form of *sample episodes*. This gives them at least three kinds of advantages over DP methods. First, they can be used to learn optimal behavior directly from interaction with the environment, with no model of the environment's dynamics. Second, they can be used with simulation or *sample models*. For surprisingly many applications it is easy to simulate sample episodes even though it is difficult to construct the kind of explicit model of transition probabilities required by DP methods. Third, it is easy and efficient to *focus* Monte Carlo methods on a small subset of the states. A region of special interest can be accurately evaluated without going to the expense of accurately evaluating the rest of the state set (we explore this much further in Chapter 9).

In designing Monte Carlo control methods we have followed the overall schema of *generalized policy iteration* (GPI) introduced in Chapter 4. GPI involves interacting processes of policy evaluation and policy improvement. Monte Carlo methods provide an alternative policy evaluation process. Rather than use a model to compute the value of each state, they simply average many returns that start in the state. Because a state's value is the expected return, this average converges to a good approximation to the value. In control methods we are particularly interested in approximating action-value functions because these can be used to improve the policy without requiring a model of the environment's transition dynamics. Monte Carlo methods intermix policy evaluation and policy improvement steps on an episode-by-episode basis, and can be incrementally implemented on an episode-by-episode basis.

Maintaining *sufficient exploration* is an issue in Monte Carlo control methods. It is not enough just to select the actions currently estimated to be best, because then no returns will be obtained for alternative actions, and it may never be learned that they are actually better. One approach is to ignore this problem by assuming that episodes begin with state-action pairs randomly selected to cover all possibilities. Such *exploring starts* can sometimes be arranged in applications with simulated episodes, but are unlikely in learning from real experience. Instead, one of two general approaches can be used. In *on-policy* methods, the agent commits to always exploring and tries to find the best policy that still explores. In *off-policy* methods, the agent also explores, but learns a deterministic optimal policy that may be unrelated to the policy followed. More instances of both kinds of methods are presented in the next chapter.

All Monte Carlo methods for reinforcement learning have been explicitly identified only very recently. Their convergence properties are not yet clear, and their effectiveness in practice has been little tested. At present, their primary significance is their simplicity and relationships to other methods.

Monte Carlo methods differ from DP methods in two ways. First, they operate on sample experience, and thus can be used for direct learning without a model. Second, they do not bootstrap. That is, they do not build their value estimates for one state on the basis of the estimates of successor states. These two differences are not tightly linked, but can be separated. In the next chapter we consider methods that learn from experience, like Monte Carlo methods, but also bootstrap, like DP methods.

Next Up Previous
Next: 5.9 Historical and Bibliographical **Up:** 5 Monte Carlo Methods **Previous:** 5.7 Incremental Implementation

5.9 Historical and Bibliographical Remarks

The term ``Monte Carlo" dates from the 1940s when physicists at Los Alamos devised games of chance that they could study to help understand complex physical phenomena relating to the atom bomb. Coverage of Monte Carlo methods in this sense can be found in several textbooks (e.g., Kalos and Whitlock, 1986; Rubinstein, 1981).

An early use of Monte Carlo methods to estimate action values in a reinforcement learning context was by Michie and Chambers (1968). In pole balancing (Example 3.4), they used averages of episode durations to assess the worth (expected balancing ``life") of each possible action in each state, and then used these assessments to control action selections. Their method is similar in spirit to Monte Carlo ES. In our terms, they used a form of every-visit MC method.

Barto and Duff (1994) discussed policy evaluation in the context of classical Monte Carlo algorithms for solving systems of linear equations. They used the analysis of Curtiss (1954) to point out the computational advantages of Monte Carlo policy evaluation for large problems. Singh and Sutton (1996) distinguished between every-visit and first-visit MC methods and proved results relating these methods to reinforcement learning algorithms.

The blackjack example is based on an example used by Widrow, Gupta, and Maitra (1973). The soap bubble example is a classical Dirichlet problem whose Monte Carlo solution was first proposed by Kakutani (1945). (see Hersh and Griego, 1969; Doyle and Snell, 1984). The racetrack exercise is adapted from Barto, Bradtke, and Singh (1995), and from Gardner (1973).

Richard Sutton
Fri May 30 13:20:35 EDT 1997

6 Temporal Difference Learning

If one had to identify one idea as central and novel to reinforcement learning, it would undoubtedly be *temporal difference* (TD) learning. TD learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas. Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics. Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap). The relationship between TD, DP, and Monte Carlo methods is a recurring theme in the theory of reinforcement learning. This chapter is the beginning of our exploration of it. Before we are done, we will see that these ideas and methods blend into each other and can be combined in many ways. In particular, in Chapter 7 we introduce the TD(λ) algorithm, which seamlessly integrates TD and Monte Carlo methods.

As usual, we start by focusing on the policy-evaluation or *prediction* problem, that of estimating the value function V^π for a given policy π. For the *control* problem (finding an optimal policy), all of DP, TD, and Monte Carlo methods use some variation of generalized policy iteration (GPI). The differences in the methods are primarily differences in their approaches to the prediction problem.

Richard Sutton
Fri May 30 13:53:05 EDT 1997

6.1 TD Prediction

Both TD and Monte Carlo methods use experience to solve the prediction problem. Given some experience following a policy π, both methods update their estimate **V** of V^π. If a nonterminal state s_t is visited at time **t**, then both methods update their estimate $V(s_t)$ based on what happens after that visit. Roughly speaking, Monte Carlo methods wait until the return following the visit is known, then use that return as a target for $V(s_t)$. A simple every-visit Monte Carlo method suitable for nonstationary environments is

$$V(s_t) \leftarrow V(s_t) + \alpha\Big[R_t - V(s_t)\Big], \qquad (6.1)$$

where R_t is the actual return following time **t** and α is a constant step-size parameter. Let us call this method *constant-α MC*. Whereas Monte Carlo methods must wait until the end of the episode to determine the increment to $V(s_t)$ (only then is R_t known), TD methods need wait only until the next time step. At time **t+1** they immediately form a target and make a useful update using the observed reward r_{t+1} and the estimate $V(s_{t+1})$. The simplest TD method, known as *TD(0)*, is

$$V(s_t) \leftarrow V(s_t) + \alpha\Big[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\Big]. \qquad (6.2)$$

In effect, the target for the Monte Carlo update is R_t, whereas the target for the TD update is $r_{t+1} + \gamma V_t(s_{t+1})$.

Because the TD method bases its update in part on an existing estimate, we say that it is a *bootstrapping* method, like DP. We know from Chapter 3 that

$$\begin{aligned} V^\pi(s) &= E_\pi\{R_t \mid s_t = s\} && (6.3)\\ &= E_\pi\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1} \;\middle|\; s_t = s\right\} \\ &= E_\pi\left\{r_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k r_{t+k+2} \;\middle|\; s_t = s\right\} \\ &= E_\pi\left\{r_{t+1} + \gamma V^\pi(s_{t+1}) \;\middle|\; s_t = s\right\}. && (6.4) \end{aligned}$$

Roughly speaking, Monte Carlo methods use an estimate of (6.3) as a target, whereas DP methods use an estimate of (6.4) as a target. The Monte Carlo target is an estimate because the expected value in (6.3) is not known; a sample return is used in place of the real expected return. The DP target is an estimate not because of the expected values, which are assumed to be completely provided by a model of the environment, but because $V^\pi(s_{t+1})$ is not known and the current estimate, $V_t(s_{t+1})$, is used instead. The TD target is an estimate for both reasons: it samples the expected values in (6.4) *and* it uses the current estimate V_t instead of the true V^π. Thus, TD methods combine the sampling of Monte Carlo with the bootstrapping of DP. As we shall see, with care and imagination this can take us a long way toward obtaining the advantages of both Monte Carlo and DP methods.

Figure 6.2 specifies TD(0) completely in procedural form, and Figure 6.1 shows its backup diagram. The value estimate for the state node at the top of the backup diagram is updated based on the one sample transition from it to the immediately following state. We refer to TD and Monte Carlo updates as *sample backups* because they involve looking ahead to a sample successor state (or state-action pair), using the value of the successor and the reward along the way to compute a backed-up value, and then changing the value of the original state (or state-action pair) accordingly. *Sample* backups differ from the *full* backups of DP methods in that they are based on a single sample successor rather than on a complete distribution of all possible successors.

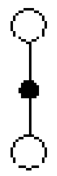


Figure 6.1: The backup diagram for the TD(0) algorithm.

```
Initialize $V(s)$ arbitrarily, $\pi$ to the policy to be evaluated
Repeat (for each episode):
```

```
Initialize s
Repeat (for each step of episode):
   $a \leftarrow$ action given by $\pi$ for $s$
   Take action a; observe reward, r, and next state, $s'$
   $V(s) \leftarrow V(s) + \alpha\left[r + \gamma V(s') - V(s)\right]$
   $s \leftarrow s'$
until s is terminal
```

Figure 6.2: TD(0) algorithm for estimating V^{π}.

Example 6.1

Driving Home. Each day you drive home from work and you try to predict how long it will take to get home. When you leave your office, you note the time, the day of week, and anything else that might be relevant. Say on this friday you are leaving at exactly 6 o'clock, and you estimate that it will take 30 minutes to get home. As you reach your car it is 6:05 and you notice it is starting to rain. Traffic is often slower in the rain so you re-estimate that it will take 35 minutes from then, or a total of 40 minutes. Fifteen minutes later you have completed the highway portion of your journey in good time. As you exit onto a secondary road you cut your estimate of total travel time to 35 minutes. Unfortunately, at this point you get stuck behind a slow truck, and the road is too narrow to pass. You end up having to follow the truck until you turn off onto the side street where you live at 6:40. Three minutes later you are home. The sequence of situations, times, and predictions is thus as follows:

situation	*elapsed time (minutes)*	*predicted time-to-go*	*predicted total time*
leaving office, friday at 6	0	30	30
reach car, raining	5	35	40
exiting highway	20	15	35
2ndary road, behind truck	30	10	40
entering home street	40	3	43
arrive home	43	0	43

The rewards in this example are the elapsed times on each leg of the journey. We are not discounting ($\gamma = 1$) and thus the return for each situation is the actual time-to-go from that situation. The value of each situation is the *expected* time-to-go. The second column of numbers gives the current estimated value for each situation encountered.

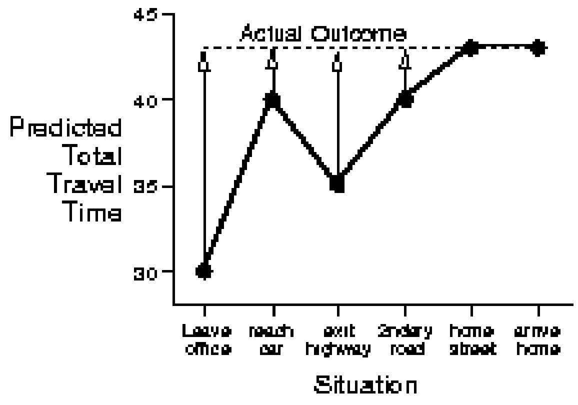


Figure 6.3: Changes recommended by Monte Carlo methods in the driving-home example.

A simple way to view the operation of Monte Carlo methods is to plot the predicted total time (the last column) over the sequence, as in Figure 6.3. The arrows show the changes in predictions recommended by the constant-α MC method (6.1). These are exactly the errors between the estimated value (predicted time-to-go) in each situation and the actual return (actual time-to-go). For example, when you exited the highway you thought it would take only 15 minutes more to get home, but in fact it took 23 minutes. Equation 6.1 applies at this point and determines an increment in the estimate of time-to-go after exiting the highway. The error, $R_t - V_t(s_t)$, at this time is 8 minutes. Suppose the step size α is $1/2$. Then the predicted time-to-go after exiting the highway would be revised upward by 4 minutes as a result of this experience. This is probably too large of a change in this case; the truck was probably just an unlucky break. In any event, the change can only be made offline, that is, after you have reached home. Only at this point do you know any of the actual returns.

Is it necessary to wait until the final outcome is known before learning can begin? Suppose on another day you again estimate when leaving your office that it will take 30 minutes to drive home, but then you become stuck in a massive traffic jam. Twenty-five minutes after leaving the office you are still bumper-to-bumper on the highway. You now estimate that it will take another 25 minutes to get home, for a total of 50 minutes. As you wait in traffic you already know that your initial estimate of 30 minutes was too optimistic. Must you wait until you get home before increasing your estimate for the initial situation? According to the Monte Carlo approach you must, because you don't yet know the true return.

According to a TD approach, on the other hand, you would learn immediately, shifting your initial estimate from 30 minutes toward 50. In fact, each estimate would be shifted toward the estimate that immediately follows it. Returning to our first day of driving, Figure 6.4 shows the same predictions as Figure 6.3 except here showing the changes recommended by the TD rule (6.2) (these are the changes made by the rule if $\alpha = 1$). Each error is proportional to the change over time of the prediction, i.e., learning is driven by the *temporal differences* in prediction.

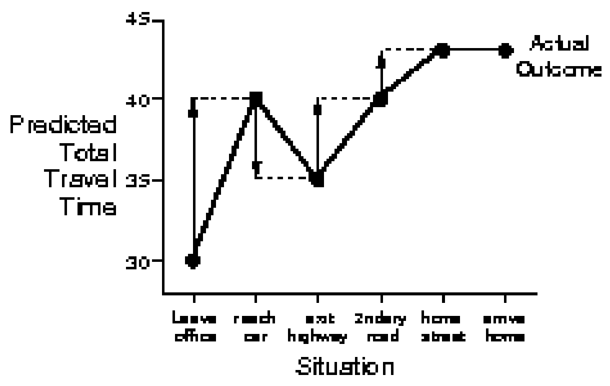


Figure 6.4: Changes recommended by TD methods in the driving-home example.

Besides giving you something to do while waiting in traffic, there are several computational reasons why it is advantageous to learn based in part on your current predictions rather than waiting until termination when you know the actual return. We briefly discuss some of these next.

Next Up Previous

Next: 6.2 Advantages of TD **Up:** 6 Temporal Difference Learning **Previous:** 6 Temporal Difference Learning

---

Richard Sutton
Fri May 30 13:53:05 EDT 1997

6.2 Advantages of TD Prediction Methods

TD methods learn their estimates in part on the basis of other estimates. They learn a guess from a guess---they *bootstrap*. Is this a good thing to do? What advantages do TD methods have over Monte Carlo and DP methods? Developing and answering such questions will take the rest of this book and more. In this section we briefly anticipate some of the answers.

Obviously, TD methods have an advantage over DP methods in that they do not require a model of the environment, of its reward and next-state probability distributions.

The next most obvious advantage of TD methods over Monte Carlo methods is that they are naturally implemented in an online, fully incremental fashion. With Monte Carlo methods one must wait until the end of an episode, because only then is the return known, whereas with TD methods one need only wait one time step. Surprisingly often this turns out to be a critical consideration. Some applications have very long episodes, so that delaying all learning until an episode's end is just too slow. Other applications are continual and have no episodes at all. Finally, as we noted in the previous chapter, some Monte Carlo methods must ignore or discount episodes on which experimental actions are taken, which can greatly slow learning. TD methods are much less susceptible to these problems because they learn from each transition regardless of what subsequent actions are taken.

But are TD methods sound? Certainly it is convenient to learn one guess from the next, without waiting for an actual outcome, but can we still guarantee convergence to the correct answer? Happily, the answer is yes. For any fixed policy π, the TD algorithm described above has been proven to converge to V^π, in the mean for a constant step-size parameter if it is sufficiently small, and with probability one if the step-size parameter decreases according to the usual stochastic approximation conditions (). Most convergence proofs apply only to the table-based case of the algorithm presented above

(6.2), but some also apply to the case of general linear function approximators. These results are discussed in a more general setting in the next two chapters.

If both TD and MC methods converge asymptotically to the correct predictions, then a natural next question is ``Which gets there first?" In other words, which method learns fastest? Which makes the most efficient use of limited data? At the current time this is an open question in the sense that no one has been able to prove mathematically that one method converges faster than the other. In fact, it is not even clear what is the most appropriate formal way to phrase this question! In practice, however, TD methods have usually been found to converge faster than MC methods on stochastic tasks, as illustrated in the following example.

Example 6.2 *Random Walk*. In this example we empirically compare the prediction abilities of TD(0) and constant-α MC applied to the small Markov process shown in Figure 6.5. All episodes start in the center state, **C**, and proceed either left or right by one state on each step, with equal probability. This behavior is presumably due to the combined effect of a fixed policy and an environment's state-transition probabilities, but we do not care which; we are concerned only with predicting returns however they are generated. Episodes terminate either on the extreme left or the extreme right. If they terminate on the right a reward of **+1** occurs, and otherwise all rewards are zero. For example, a typical walk might consist of the following state-and-reward sequence: **C,0,B,0,C,0,D,0,E,1**. Because this task is undiscounted and episodic, the true value of each state is just the probability of terminating on the right if starting from that state. Thus, the true value of the center state is $V^\pi(C) = 0.5$. The true values of all the states, **A** through **E**, are $\frac{1}{6}$, $\frac{2}{6}$, $\frac{3}{6}$, $\frac{4}{6}$, and $\frac{5}{6}$. Figure 6.6 shows the values learned by TD(0) approaching the true values as more episodes are experienced. Averaging over many episode sequences, Figure 6.7 shows the average error in the predictions found by TD(0) and constant-α MC, for a variety of values of α, as a function of number of episodes. In all cases the approximate value function was initialized to the intermediate value $V(s) = 0.5$, for all **s**. The TD method is consistently better than the MC method on this task over this number of episodes. $\diamond$

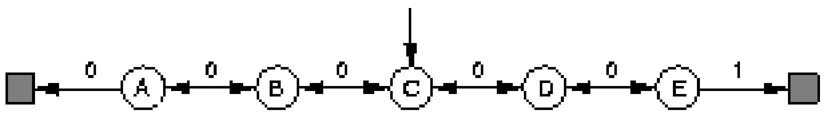


Figure 6.5: A small Markov process for generating random walks. Walks start in the center, then move randomly right or left until entering a terminal state. Termination on the right produces a reward of +1; all other transitions yield zero reward.

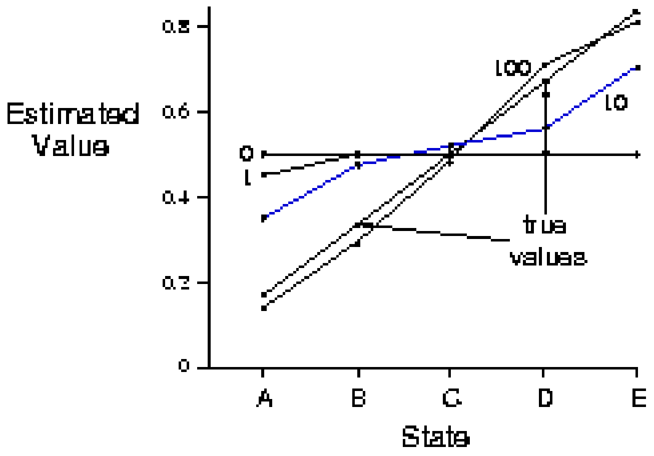


Figure 6.6: Values learned by TD(0) after various numbers of episodes. The final estimate is about as close as the estimates ever get to the true values. With a constant step-size parameter ($\alpha = 0.1$ in this example), the values fluctuate indefinitely in response to the outcomes of the most recent episodes.

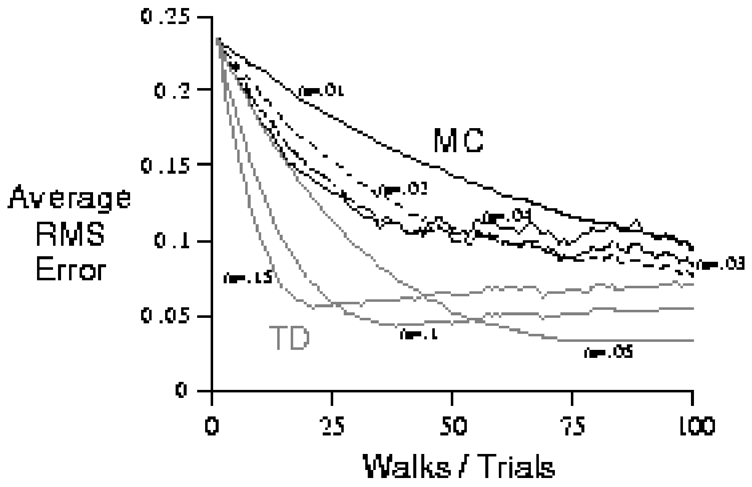


Figure 6.7: Learning curves for TD(0) and constant-α MC methods, for various values of α, on the prediction problem for the random walk. The performance measure shown is the squared error between the value function learned and the true value function, averaged over the five states. These data are averages over 100 different sequences of episodes.

Exercise 6.1

This is a exercise to help develop your intuition about why TD methods are often more efficient than MC methods. Consider the driving-home example and how it is addressed by TD and MC methods. Can you imagine a scenario in which a TD update would be better on average than an MC update? Give an example scenario---a description of past experience and a current situation---in which you would expect the TD update to be better. Here's a hint: Suppose you have lots of experience driving home from work. Then you move to a new building and a new parking lot (but you still enter the highway at the same place). Now you are starting to learn predictions for the new building. Can you see why TD updates are likely to be much better, at least initially, in this case? Might the same sort of thing happen in the original task?

Exercise 6.2

From Figure 6.6, it appears that the first episode results in a change in only $V(A)$. What does this tell you about what happened on the first episode? Why was only the estimate for this one state changed? By exactly how much was it changed?

Exercise 6.3

Do you think that by choosing the step-size parameter, α, differently, either algorithm could have done significantly better than shown in Figure 6.7? Why or why not?

Exercise 6.4

In Figure 6.7, the RMS error of the TD method seems to go down and then up again, particularly at high α 's. What could have caused this? Do you think this always occurs, or might it be a function of how the approximate value function was initialized?

Exercise 6.5

Above we stated that the true values for the random walk task are $\frac{1}{6}$, $\frac{2}{6}$, $\frac{3}{6}$, $\frac{4}{6}$, and $\frac{5}{6}$, for states **A** through **E**. Describe at least two different ways that these could have been computed. Which would you guess we actually used? Why?

Richard Sutton
Fri May 30 13:53:05 EDT 1997

6.3 Optimality of TD(0)

Suppose there is available only a finite amount of experience, say 10 episodes or 100 time steps. In this case, a common approach with incremental learning methods is to repeatedly present the experience until the method converges upon an answer. Given an approximate value function, **V**, the increments specified by (6.1) or (6.2) are computed for every time step **t** at which a nonterminal state is visited, but the value function is only changed once, by the sum of all the increments. Then all the available experience is processed again with the new value function to produce a new overall increment, and so on, until the value function converges. We call this *batch updating* because updates are made only after processing each complete *batch* of training data.

Under batch updating, TD(0) converges deterministically to a single answer independent of the step-size parameter, α , as long as α is chosen to be sufficiently small. The constant-α MC method also converges deterministically under the same conditions, but to a different answer. Understanding these two answers will help us understand the difference between the two methods. Under normal updating the methods do not move all the way to their respective batch answers, but in some sense they take a step in these directions. Before trying to understand the two answers in general, for all possible tasks, we first look at a few examples.

Example 6.3 *Random walk under batch updating*. Batch-updating versions of TD(0) and constant-α MC were applied as follows to the random-walk prediction example (Example 6 .2). After each new episode, all episodes seen so far were treated as a batch. They were repeatedly presented to the algorithm, either TD(0) or constant-α MC, with α \ sufficiently small that the value function converged. The resulting value function was then compared to V^π, and the average mean-squared error across the five states (and across 100 independent repetitions of the whole experiment) was plotted to obtain the learning curves shown in Figure 6.8. Note that the batch TD method was consistently better than the batch MC method. $\diamond$

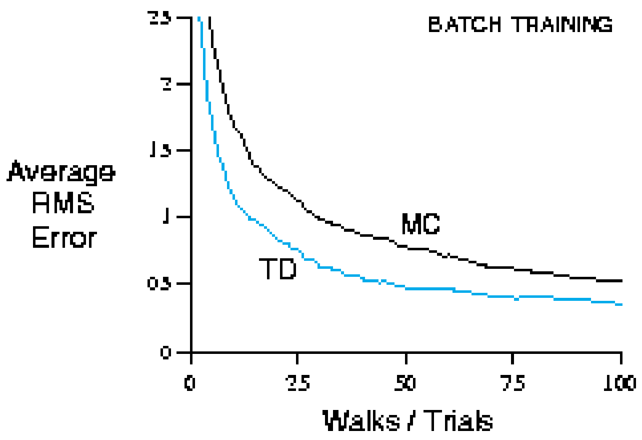


Figure 6.8: Performance of TD(0) and constant-α MC under batch training on the random-walk task.

Under batch training, the MC method converges to values, $V(s)$, that are the sample averages of the actual returns experienced after visiting each state **s**. These are optimal estimates in the sense that they minimize the mean-squared error from the actual returns in the training set. In this sense it is surprising that the batch TD method was able to perform better in the mean-squared error measure shown in Figure 6.8. How is it that batch TD was able to perform better than this optimal method? The answer is that MC is only optimal in a limited way, and that TD is optimal in a way that is more relevant to predicting returns. But first let's develop our intuitions about different kinds of optimality through another example.

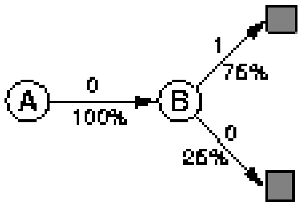
Example 6.4 *You are the Predictor*. Place yourself now in the role of the predictor of returns for a Markov chain. Suppose you observe the following eight episodes:

$A, 0, B, 0$	$B, 1$
$B, 1$	$B, 1$
$B, 1$	$B, 1$
$B, 1$	$B, 0$

This means that the first episode started in state **A**, transitioned to **B** with a reward of 0, and then terminated from **B** with a reward of 0. The other seven episodes were even shorter, starting from **B** and terminating immediately. Given this batch of data, what would you say are the optimal predictions, the best values for the estimates $V(A)$ and $V(B)$?

Everyone would probably agree that the optimal value for $V(B)$ is $\frac{3}{4}$, because six out of the eight times in state **B** the process terminated immediately with a return of 1, and the other two times in **B** the process terminated immediately with a return of 0.

But what is the optimal value for the estimate $V(A)$ given this data? Here there are two reasonable answers. One is to observe that 100% of the times the process was in state **A** it traversed immediately to **B** (with a reward of 0), and we have already decided that **B** has value $\frac{3}{4}$, so therefore **A** must have value $\frac{3}{4}$ as well. One way of viewing this answer is that it is based on first modeling the Markov process, in this case as



and then computing the correct estimates given the model, which indeed in this case gives $V(A) = \frac{3}{4}$. This is also the answer that batch TD(0) gives.

The other reasonable answer is simply to observe that we have seen **A** once and the return that followed it was 0, and therefore estimate $V(A)$ as **0**. This is the answer that batch Monte Carlo methods give. Notice that it is also the answer that gives minimum squared error on the training data. In fact, it gives zero error on the data. But still we expect the first answer to be better. If the process is Markov, we expect that the first answer will produce lower error on *future* data, even though the MC answer is better on the existing data. ◇

The above example illustrates a general difference between the estimates found by batch TD(0) and batch Monte Carlo methods. Batch Monte Carlo methods always find the estimates that minimize mean squared error on the training set, whereas batch TD(0) always finds the estimates that would be exactly correct for the maximum-likelihood model of the Markov process. In general, the *maximum-likelihood estimate* of a parameter is the parameter value whose probability of generating the data is greatest. In this case, the maximum-likelihood estimate is simply the model of the Markov process formed in the obvious way from the observed episodes: the estimated transition probability from **i** to **j** is just the fraction of observed transitions from **i** that went to **j**, and the associated expected reward is just the average of the rewards observed on those transitions. Given this model, we can compute the the estimate of the value function that would be exactly correct if the model were exactly correct. This is called the *certainty-equivalence estimate* because it is equivalent to assuming that the estimate of the underlying process was known with certainty rather than being approximated. In general, batch TD(0) converges to the certainty-equivalence estimate.

This helps explain why TD methods converge more quickly than MC methods. In batch form, TD(0) is faster than Monte Carlo methods because it computes the true certainty-equivalence estimate. This explains the advantage of TD(0) shown in the batch results on the random-walk task (Figure 6.8). The relationship to the certainty-equivalence estimate may also explain in part the speed advantage of non-batch TD(0) (e.g., Figure 6.7). Although the non-batch methods do not achieve either the certainty-equivalence or the minimum-squared-error estimates, they can be understood as moving roughly in these directions. Non-batch TD(0) may be faster than constant-α MC simply because it is moving towards a better estimate, even though it is not getting all the way there. At the current time nothing more definite can be said about the relative efficiency of online TD and MC methods.

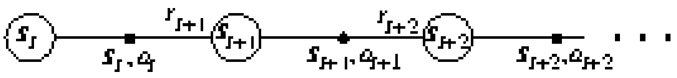
Finally, it is worth noting that although the certainty-equivalence estimate is in some sense an optimal solution, it is almost never feasible to compute it directly. If **N** is the number of states, then simply forming the maximum-likelihood estimate of the process may require N^2 memory, and computing the corresponding value function requires on the order of N^3 computational steps if done conventionally. In these terms it is indeed striking that TD methods can approximate the same solution using memory no more than **N** and repeated computations over the training set. On tasks with large state spaces, TD methods may be the only feasible way of approximating the certainty-equivalence solution.

Richard Sutton
Fri May 30 13:53:05 EDT 1997

6.4 Sarsa: On-Policy TD Control

We turn now to the use of TD prediction methods for the control problem. As usual, we follow the pattern of generalized policy iteration (GPI), only this time using TD methods for the evaluation or prediction part. As with Monte Carlo methods, we face the need to tradeoff exploration and exploitation, and again approaches fall into two main classes: on-policy and off-policy. In this section we present an on-policy TD control method.

The first step is to learn an action-value function rather than a state-value function. In particular, for an on-policy method we must estimate $Q^\pi(s,a)$ for the current behavior policy π and for all states **s** and actions **a**. Happily, we can learn Q^π using essentially the same TD method as described above for learning V^π. Recall that an episode consists of an alternating sequence of states and state-action pairs:



In the previous section we considered transitions from state to state and learned the value of states. But the relationship between states and state-actions pairs is symmetrical. Now we consider transitions from state-action pair to state-action pair, and learn the value of state-action pairs. Formally these cases are identical: they are both Markov chains with a reward process. The theorems assuring the convergence of state values under TD(0) also applies to the corresponding algorithm for action values:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\Big[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\Big]. \qquad (6.5)$$

This update is done after every transition from a nonterminal state s_t. If s_{t+1} is terminal, then $Q(s_{t+1},a_{t+1})$ is defined as zero. This rule uses every element of the quintuple of events, $(s_t,a_t,r_{t+1},s_{t+1},a_{t+1})$, that make up a transition from one state-action pair to the next. This quintuple gives rise to the name *Sarsa* for the algorithm.

It is straightforward to design an on-policy control algorithm based on the Sarsa prediction method. As in all on-policy methods, we continually estimate Q^π for the behavior policy π, and at the same time change π towards greediness with respect to Q^π. The general form of the Sarsa control algorithm is given in Figure 6.9.

```
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
   Initialize s
   Choose a from s using policy derived from Q
         (e.g., $\epsilon$-greedy)
   Repeat (for each step of episode):
      Take action a, observe r, $s'$
      Choose $a'$ from $s'$ using policy derived from Q
         (e.g., $\epsilon$-greedy)
      $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$
      $s \leftarrow s'$; $a \leftarrow a'$;
   until s is terminal
```

Figure 6.9: Sarsa: An on-policy TD control algorithm.

The convergence properties of the Sarsa algorithm depend on the nature of the policy's dependence on **Q**. For example, one could use ϵ-greedy or ϵ-soft policies. According to Satinder Singh (personal communication), Sarsa converges with probability one to an optimal policy and action-value function as long as all state action pairs are visited an infinite number of times and the policy converges in the limit to the greedy policy (which can be arranged, for example, with ϵ-greedy policies by setting $\epsilon = 1/t$), but this result has not yet been published in the literature.

Example 6.5 *Windy Gridworld*. Figure 6.10 shows a standard gridworld, with start and goal states, but with one difference: there is a crosswind upward through the middle of the grid. The actions are the standard four, `up`, `down`, `right`, and `left`, but in the middle region the resultant next states are shifted upward by a ``wind,'' the strength of which varies from column to column. The strength of the wind is given below each column, in number of cells shifted upward. For example, if you are one cell to the right of the goal, then the action `left` takes you to the cell just above the goal. Let us treat this as a undiscounted episodic task, with constant rewards of **-1** until the goal state is reached. Figure 6.11 shows the result of applying ϵ-greedy Sarsa to this task, with $\epsilon = .1$, $\alpha = .1$, and the initial values $Q(s,a) = 0$ for all **s,a**. The increasing slope of the graph shows that the goal is reached more and more quickly over time. By 8000 time steps, the greedy policy (shown inset) was long since optimal; continued ϵ -greedy exploration kept the average episode length at about 17 steps, two less than the minimum of 15. Note that Monte Carlo methods can not easily be used on this task because termination is not guaranteed for all policies. If a policy was ever found that caused the agent to stay in the same state, then the next episode would never end. Step-by-step learning methods such as Sarsa do not have this problem because they quickly learn *during the episode* that such policies are poor, and switch to something else. $\diamondsuit$

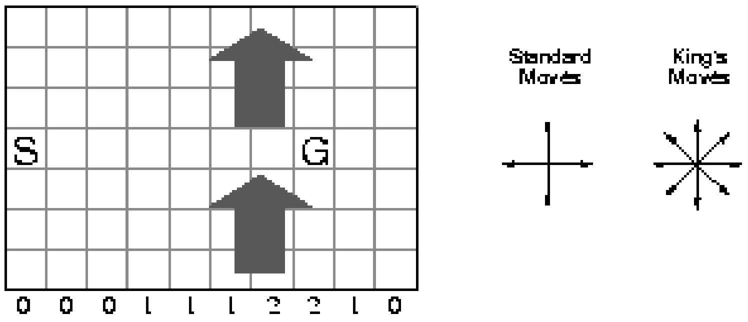


Figure 6.10: In this gridworld, movement is altered by a location-dependent, upward ``wind.''

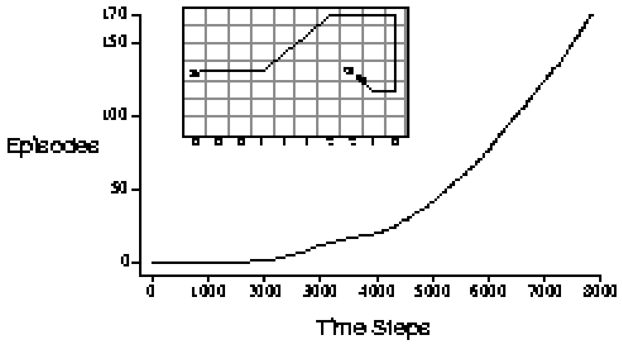


Figure 6.11: Results of Sarsa applied to the windy gridworld. The slope shows the rate at which the goal was reached---approximately once every 17 steps at the end. The inset shows the greedy policy, which is optimal.

Exercise 6.6 *Windy Gridworld with King's Moves.* Re-solve the windy gridworld task assuming eight possible actions, including the diagonal moves, rather than the usual four. How much better can you do with the extra actions? Can you do even better by including a ninth action that causes no movement at all other than that caused by the wind?

Exercise 6.7 *Stochastic Wind.* Re-solve the windy gridworld task with King's moves, assuming that the effect of the wind, if there is any, is stochastic, sometimes varying by one from the mean values given for each column. That is, a third of the time you move exactly according to these values, as in the previous exercise, but also a third of the time you move one cell above that, and another third of the time you move one cell below that. For example, if you are one cell to the right of the goal and you move `left`, then one third of the time you move one cell above the goal, one third of the time you move two cells above the goal, and one third of the time you move to the goal.

Exercise 6.8

What is the backup diagram for Sarsa?

Richard Sutton
Fri May 30 13:53:05 EDT 1997

6.5 Q-learning: Off-Policy TD Control

One of the most important breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as *Q-learning* (Watkins, 1989). Its simplest form, *1-step Q-learning* is defined by

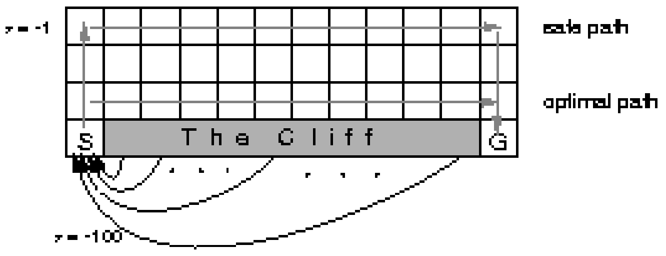
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]. \quad (6.6)$$

In this case, the learned action-value function, **Q**, directly approximates Q^*, the optimal action-value function, independent of the policy being followed. This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs. The policy still has an effect in that it determines which state-action pairs are visited and updated. However, all that is required for correct convergence is that all pairs continue to be updated. As we have discussed before, this is a minimal requirement in the sense that any method guaranteed to find optimal behavior in the general case must require it. Under this assumption and a variant of the usual stochastic approximation conditions on the step-size sequence, Q_t has been shown to converge with probability one to Q^*. The Q-learning algorithm is shown in procedural form in Figure 6.12.

```
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
   Initialize s
   Repeat (for each step of episode):
      Choose a from s using policy derived from Q
         (e.g., $\epsilon$-greedy)
      Take action a, observe r, $s'$
      $Q(s,a) \leftarrow Q(s,a) + \alpha \left[ r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$
      $s \leftarrow s'$;
   until s is terminal
```

Figure 6.12: Q-learning: An off-policy TD control algorithm.

What is the backup diagram for Q-learning? The rule (6.6) updates a state-action pair, so the top node, the root of the backup, must be a small, filled action node. The backup is also *from* action nodes, maximizing over all those actions possible in the next state. Thus the bottom nodes of the backup diagram should be all these action nodes. Finally, remember that we indicate taking the maximum of these ``next action" nodes with an arc across them (Figure 3 .7). Can you guess now what the diagram is? If so, please do make a guess before turning to the answer (on the next page) in Figure 6.14.



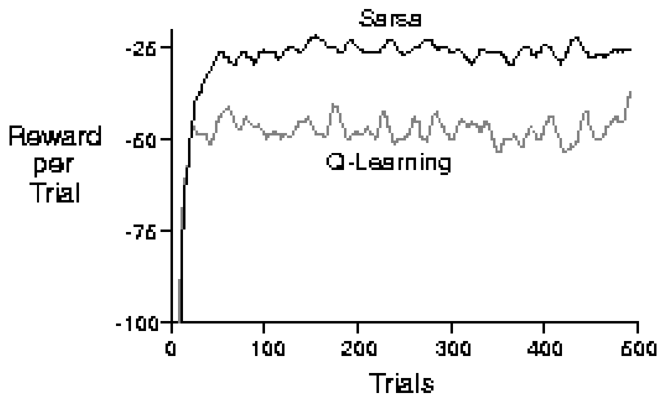


Figure 6.13: The cliff-walking task. Off-policy Q-learning learns the optimal policy, along the edge of the cliff, but then keeps falling off because of the ϵ -greedy action selection. On-policy Sarsa learns a safer policy taking into account the action selection method. These data are from a single run, but smoothed.

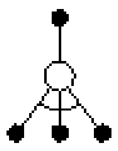


Figure 6.14: The backup diagram for Q-learning.

Example 6.6 *Cliff Walking*. This gridworld example compares Sarsa and Q-learning, highlighting the difference between on-policy (Sarsa) and off-policy (Q-learning) methods. Consider the gridworld shown in the upper part of Figure 6.13. This is a standard undiscounted, episodic task, with start and goal states, and the usual actions causing movement up, down, right, and left. Reward is **-1** on all transitions except into the the region marked ``The Cliff." Stepping into this region incurs a reward of **-100** and sends the agent instantly back to the start. The lower part of the figure shows the performance of the Sarsa and Q-learning methods with ϵ -greedy action selection, $\epsilon = 0.1$. After an initial transient, Q-learning learns values for the optimal policy, that which travels right along the edge of the cliff. Unfortunately, this results in it occasionally falling off the cliff because of the ϵ -greedy action selection. Sarsa, on the other hand, takes into account the action selection and learns the longer but safer path through the upper part of the grid. Although Q-learning actually learns the values of the optimal policy, its online performance is worse than that of Sarsa, which learns the roundabout policy. Of course, if ϵ were gradually reduced, then both methods would asymptotically converge to the optimal policy. $\diamond$

Exercise 6.9

Why is Q-learning considered an *off-policy* control method?

Exercise 6.10

Consider the learning algorithm that is just like Q-learning except that instead of the maximum over next state-action pairs it uses the expected value, taking into account how likely each action is under the current policy. That is, consider the algorithm otherwise like Q-learning except with the update rule

$$\begin{aligned} Q(s_t, a_t) &\leftarrow Q(s_t, a_t) + \alpha\Big[r_{t+1} + \gamma E\{Q(s_{t+1}, a_{t+1}) | s_t\} - Q(s_t, a_t)\Big] \\ &\leftarrow Q(s_t, a_t) + \alpha\Big[r_{t+1} + \gamma \sum_a \pi(s_t, a) Q(s_{t+1}, a) - Q(s_t, a_t)\Big] \end{aligned}$$

Is this new method an on-policy or off-policy method? What is the backup diagram for this algorithm? Given the same amount of experience, would you expect this method to work better or worse than Sarsa? What other considerations might impact the comparison of this method with Sarsa?

Next Up Previous

Next: 6.6 Actor-Critic Methods (*) **Up:** 6 Temporal Difference Learning **Previous:** 6.4 Sarsa: On-Policy TD

Richard Sutton
Fri May 30 13:53:05 EDT 1997

6.6 Actor-Critic Methods (*)

Actor-critic methods are TD methods that have a separate memory structure to explicitly represent the policy independent of the value function. The policy structure is known as the *actor*, because it is used to select actions, and the estimated value function is known as the *critic*, because it criticizes the actions made by the actor. Learning is always on-policy: the critic must learn about and critique whatever policy is currently being followed by the actor. The critique takes the form of a TD error. This scalar signal is the sole output of the critic and drives all learning in both actor and critic, as suggested by Figure 6.15.

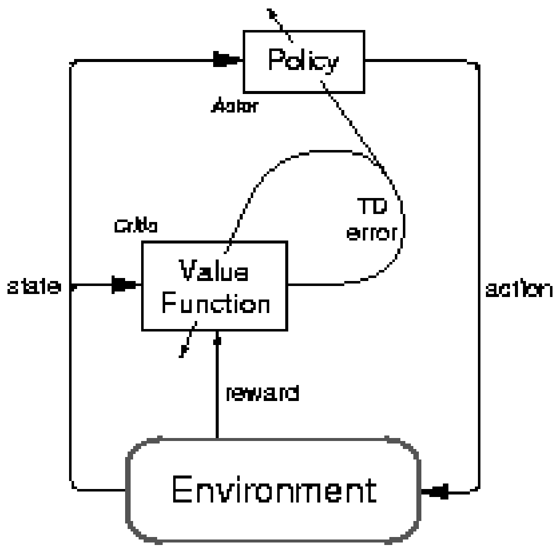


Figure 6.15: The Actor-Critic Architecture.

Actor-critic methods are the natural extension of the idea of reinforcement-comparison methods (Section) to TD learning and to the full reinforcement-learning problem. Typically, the critic is a state-value function. After each action selection, the critic evaluates the new state to determine whether things have gone better or worse than expected. That evaluation is the TD error:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t),$$

where **V** is the current value function implemented by the critic. This TD error can be used to evaluate the action just selected, the action a_t taken in state s_t. If the TD error is positive, it suggests that the tendency to select a_t should be strengthened for the future, whereas if the TD error is negative it suggests the tendency should be weakened. Suppose actions are generated by the Gibbs softmax method:

$$\pi_t(s, a) = Pr\left\{a_t = a | s_t = s\right\} = \frac{e^{p(s,a)}}{\sum_b e^{p(s,b)}},$$

where the $p(s, a)$ are the values at time **t** of the modifiable policy parameters of the actor, indicating the tendency to select (*preference* for) each action **a** when in each state **s**. Then the strengthening or weakening described above can be implemented simply by incrementing or decrementing $p(s_t, a_t)$, e.g., by

$$p(s_t, a_t) \leftarrow p(s_t, a_t) + \beta\delta_t,$$

where β is another positive step-size parameter.

This is just one example of an actor-critic method. Other variations select the actions in different ways, or use eligibility traces like those described in the next chapter. Another common dimension of variation, just as in reinforcement-comparison methods, is to include additional factors varying the amount of credit assigned to the action taken, a_t. For example, one of the most common such factors is inversely related to the probability of selecting a_t, resulting in the update rule:

$$p(s_t, a_t) \leftarrow p(s_t, a_t) + \beta\delta_t\left[1 - \pi_t(s_t, a_t)\right].$$

These issues were explored early on, primarily for the immediate reward case (Williams, 1992; Sutton, 1984) and have not been brought fully up to date.

Many of the earliest reinforcement learning systems that used TD methods were actor-critic methods (Witten, 1977; Barto, Sutton and Anderson, 1983). Since then, more attention has been devoted to methods than learn action-value functions and determine a policy exclusively from the estimated values (such as Sarsa and Q-learning). This divergence may be just historical accident. For example, one could imagine intermediate architectures in which one learns an action-value function and yet still maintains an independent policy. In any event, actor-critic methods are likely to remain of current interest because of two significant apparent advantages:

- They require minimal computation in order to select actions. Consider a case where

there are an infinite number of possible actions---for example, a continuous-valued action. Any method learning just action values must search through this infinite set in order to pick an action. If the policy is explicitly stored, then this extensive computation may not be needed for each action selection.
- They can learn an explicitly stochastic policy; that is, they can learn the optimal probabilities of selecting various actions. This ability turns out to be useful in competitive and nonMarkov cases (e.g., see Singh et al., 1994).

In addition, the separate actor in actor-critic methods make them more appealing to many people as psychological and biological models. In some cases it may also make it easier to impose domain-specific constraints on the set of allowed policies.

Next Up Previous
Next: 6.7 R-Learning for Undiscounted **Up:** 6 Temporal Difference Learning **Previous:** 6.5 Q-learning: Off-Policy TD

Richard Sutton
Fri May 30 13:53:05 EDT 1997

6.7 R-Learning for Undiscounted Continual Tasks (*)

R-learning is an off-policy control method for the advanced version of the reinforcement learning problem in which one neither discounts nor divides experience into distinct episodes with finite returns. In this case one seeks to obtain the maximum reward per-time-step. The value functions for a policy, π, are defined relative to the average expected reward per-time-step under the policy, ρ^π:

$$\rho^\pi = \lim_{n \to \infty} \frac{1}{n} \sum_{t=1}^{n} E_\pi\{r_t\},$$

where here we assume the process is ergodic (non-zero probability of reaching any state from any other under any policy) and thus ρ^π does not depend on the starting state. From any state, in the long run the average reward is the same, but there is a transient. From some states better than average rewards are received for a while, and from others worse than average rewards are received. It is this transient which defines the value of a state:

$$\tilde{V}^\pi(s) = \sum_{k=1}^{\infty} E_\pi\{r_{t+k} - \rho^\pi | s_t = s\},$$

and the value of a state-action pair is similarly the transient difference in reward when starting in that state and taking that action:

$$\tilde{Q}^\pi(s,a) = \sum_{k=1}^{\infty} E_\pi\{r_{t+k} - \rho^\pi | s_t = s, a_t = a\}.$$

We call these *relative values* because they are relative to the average reward under the current policy.

There are several subtle distinctions that need to be drawn between different kinds of optimality in the undiscounted continual case. Nevertheless, for most practical purposes it may be adequate simply to order policies according to their average reward per-time-step, in other words, according to their ρ^π. For now let us consider all policies that attain the maximal value of ρ^π to be optimal.

Other than its use of relative values, R-learning is a standard TD control method based on off-policy GPI, much like Q-learning. It maintains two policies, a behavior policy and an estimation policy, plus an action-value function and an estimated average reward. The behavior policy is used to generate experience; it might be the ϵ -greedy policy with respect to the action-value function. The estimation policy is the one involved in GPI. It is typically the greedy policy with respect to the action-value function. If π is the estimation policy, then the action value function, **Q**, is an approximation of $\tilde{Q}^\pi$ and the average reward, ρ, is an approximation of ρ^π. The complete algorithm is given in Figure 6.16. There has been very little experience with this method and it should be considered experimental.

```
Initialize $\rho$ and $Q(s,a)$, for all s,a, arbitrarily
Repeat Forever:
    $s \leftarrow$ current state
    Choose action a in s using behavior policy, (e.g., $\epsilon$-greedy)
    Take action a, observe r, $s'$
    $Q(s,a) \leftarrow Q(s,a) + \alpha \left[r - \rho + \max_{a'} Q(s',a') - Q(s,a)\right]$
    If $Q(s,a) = \max_a Q(s,a)$, then:
        $\rho \leftarrow \rho + \beta \left[r - \rho + \max_{a'} Q(s',a') - \max_a Q(s,a)\right]$
```

Figure 6.16: R-learning: An off-policy TD control algorithm for the undiscounted, infinite-horizon case. The scalars α and β are step-size parameters.

Example 6.7 *An Access-Control Queuing Task.* This is a decision task involving access control to a set of **n** servers. Customers of four different priorities arrive at a single queue. If given access to a server, the customers pay a reward of 1, 2, 4, or 8, depending on their priority, with higher priority customers paying more. In each time step, the customer at the head of the queue is either accepted (assigned to one of the servers) or rejected (removed from the queue). In either case, on the next time step the next customer in the queue is considered. The queue never empties, and the proportion of (randomly distributed) high-priority customers in the queue is **h**. Of course a customer can only be served if there is a free server. Each busy server becomes free with probability **p** on each time step. Although we have just described them for definiteness, let us assume the statistics of arrivals and departures are unknown. The task is to decide on each step whether to accept or reject the next customer, on the basis of his priority and the number of free servers, so as to maximize long-term reward without discounting Figure 6.17 shows the solution found by R-learning for this task with **n=10**, $h = .5$, and $p = .06$. The R-learning parameters were $\alpha = .1$, $\beta = .01$, and $\epsilon = .1$. The initial action-values and ρ were all zero. $\diamond$

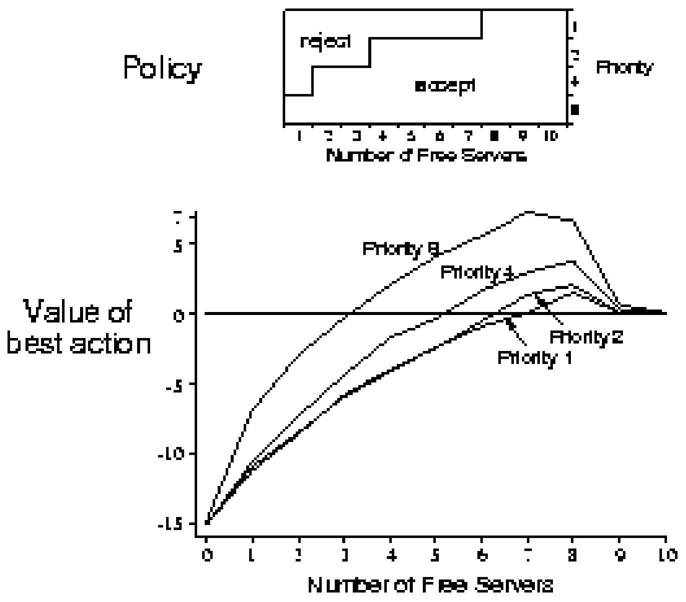


Figure 6.17: The policy and value function found by R-learning on the access-control queuing task after two million steps. The drop on the right of the graph is probably due to insufficient data; many of these states were never experienced. The value learned for ρ was about 2.73.

Exercise 6.11*

Design an on-policy method for undiscounted, continual tasks.

Next Up Previous

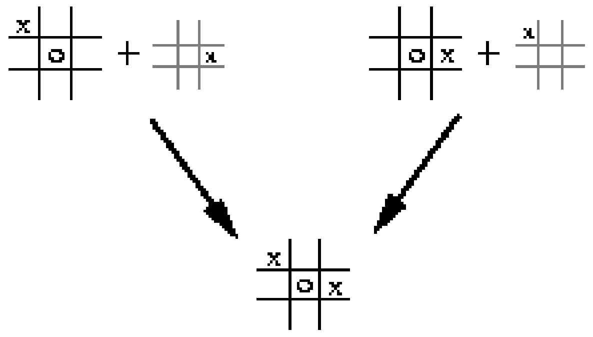
Next: 6.8 GamesAfter States, **Up:** 6 Temporal Difference Learning **Previous:** 6.6 Actor-Critic Methods (*)

Richard Sutton
Fri May 30 13:53:05 EDT 1997

6.8 Games, After States, and other Special Cases

In this book we try to present a uniform approach to a wide class of tasks, but of course there are always exceptional tasks that are better treated in a specialized way. For example, our general approach involves learning an *action*-value function, but in Chapter 1 we presented a TD method for learning to play Tic-Tac-Toe that learned something much more like a *state*-value function. If we look closely at that example, it becomes apparent that the function learned there is neither an action-value function nor a state-value function in the usual sense. A conventional state-value function evaluates states in which the agent has the option of selecting an action, but the state-value function used in Tic-Tac-Toe evaluates board positions *after* the agent has made its move. Let us call these *after states*, and value functions over these *after-state value functions*. After states are useful when we have knowledge of an initial part of the environment's dynamics but not necessarily of the full dynamics. For example, in games we typically know the immediate effects of our moves. We know for each possible chess move what the resulting position will be, but not how our opponent will reply. After-state value functions are a natural way to take advantage of this kind of knowledge and thereby produce a more efficient learning method.

The reason it is more efficient to design algorithms in terms of after states is apparent from the Tic-Tac-Toe example. A conventional action-value function would map from positions *and* moves to an estimate of the value. But many position-move pairs produce the same resulting position, as in this example:



In such cases the position-move pairs are different, but produce the same ``after position'' and thus must have the same value. A conventional action-value function would have to separately assess both pairs, whereas an after-state value function would immediately assess both equally. Any learning about the position-move pair on the left would immediately transfer to the pair on the right.

After states arise in many tasks, not just games. For example, in queuing tasks there are actions such as assigning customers to servers, rejecting customers, or discarding information. In such cases the actions are in fact defined in terms of their immediate effects, which are completely known. For example, in the access-control queuing example described in the previous section, a more efficient learning method could be obtained by breaking the environment's dynamics into the immediate effect of the action, which is deterministic and completely known, and the unknown random processes having to do with the arrival and departure of customers. The after-states would be the number of free servers after the action, but before the random processes had produced the next conventional state. Learning an after-state value function over the after states would enable all actions which produced the same number of free servers to share experience. This should result is a significant reduction in learning time.

It is impossible to describe all the possible kinds of specialized problems and corresponding specialized learning algorithms. However, the principles developed in this book should apply widely. For example, after-state methods are still aptly described in terms of generalized policy iteration, with a policy and (after-state) value function interacting in essentially the same way. In many cases one will still face the choice between on-policy and off-policy methods for managing the need for persistent exploration.

Exercise 6.12

Describe how the task of Jack's Car Rental (Example 4.2) could be reformulated in terms of after states. Why, in terms of this specific task, would such a reformulation be likely to speed convergence?

6.9 Conclusions

In this chapter we have introduced a new kind of learning method, temporal-difference (TD) learning, and showed how it can be applied to the reinforcement learning problem. As usual, we divided the overall problem into a prediction problem and a control problem. TD methods are alternatives to Monte Carlo methods for solving the prediction problem. In both cases, the extension to the control problem is via the idea of generalized policy iteration (GPI) that we abstracted from dynamic programming. This is the idea that approximate policy and value functions should interact in such a way that they both move toward their optimal values.

One of the two processes making up GPI drives the value function to accurately predict returns for the current policy; this is the prediction problem. The other process drives the policy to improve itself locally (e.g., to be ϵ -greedy) with respect to the current value function. When the first process is based on experience, then a complication arises concerning maintaining sufficient exploration. As in Chapter 5, we have grouped the TD control methods according to whether they deal with this complication using an on-policy or off-policy approach. Sarsa and actor-critic methods are on-policy methods, and Q-learning and R-learning are off-policy methods.

The methods presented in this chapter are today the most widely used reinforcement learning methods. This is probably due to their great simplicity: they can be applied online, with a minimal amount of computation, to experience generated from interaction with an environment; they can be expressed nearly completely by single equations that can be implemented with small computer programs. In the next few chapters we extend these algorithms, making them slightly more complicated and significantly more powerful. All the new algorithms will retain the essence of those introduced here: they will be able to process experience online, with relatively little computation, and they will be driven by TD errors. The special cases of TD methods introduced in the present chapter should rightly be called *1-step, tabular, model-free* TD methods. In the next three chapters we extend them to multi-step forms (a link to Monte Carlo methods), forms using function approximators rather than tables (a link to artificial neural networks), and forms that include a model of the environment (a link to planning and dynamic programming).

Finally, in this chapter we have discussed TD methods entirely within the context of reinforcement learning problems, but TD methods are actually more general than this. They are general methods for learning to make long-term predictions about dynamical systems. For example, TD methods may be relevant to predicting financial data, lifespans, election outcomes, weather patterns, animal behavior, demands on power stations, or customer purchases. It was only when TD methods were analyzed as pure prediction methods, independent of their use in reinforcement learning, that their theoretical properties first came to be well understood. Even so, these other potential applications of TD learning methods have not yet been extensively explored.

Next Up Previous

Next: 6.10 Historical and Bibliographical **Up:** 6 Temporal Difference Learning **Previous:** 6.8 GamesAfter States,

---

Richard Sutton
Fri May 30 13:53:05 EDT 1997

6.10 Historical and Bibliographical Remarks

As we outlined in Chapter 1, the idea of TD learning has its early roots in animal learning psychology and artificial intelligence, most notably the work of Samuel (1959) and Klopf (1972). Samuel's work is described as a case study in Section 11 .2 . Also related to TD learning are Holland's (1975, 1976) early ideas about consistency among value predictions. These influenced one of the authors (Barto), who was a graduate student from 1970 to 1975 at the University of Michigan, where Holland was teaching. Holland's ideas led to a number of TD-related systems, including the work of Booker (1982) and the bucket brigade of Holland (1986), which is related to Sarsa as discussed below.

6.1 -- 6.2

Most of the specific material from these sections is from Sutton (1988), including the TD(0) algorithm, the random walk example, and the term ``temporal-difference learning.'' The characterization of the relationship to dynamic programming and Monte Carlo methods was influenced by Watkins (1989), Werbos (1987), and others. The use of backup diagrams here and in other chapters is new to this book. Example 6.4 is due to Sutton, but has not been published before.

Tabular TD(0) was proved to converge in the mean by Sutton (1988) and with probability one by Dayan (1992) based on the work of Watkins and Dayan (1992). These results were then extended and strengthened by Jaakkola, Jordan and Singh (1994) and Tsitsiklis (1994) by using extensions of the powerful existing theory of stochastic approximation. Other extensions and generalizations are covered in the next two chapters.

6.3

The optimality of the TD algorithm under batch training was established by Sutton (1988). The term *certainty equivalence* is from the adaptive control literature (e.g., Goodwin and

Sin, 1984). Illuminating this result is Barnard's (1993) derivation of the TD algorithm as a combination of one step of an incremental method for learning a model of the Markov chain and one step of a method for computing predictions from the model.

6.4

The Sarsa algorithm was first explored by Rummery and Niranjan (1994), who called it *modified Q-learning*. The name ``Sarsa'' was introduced by Sutton (1996). The convergence of 1-step tabular Sarsa (the form treated in this chapter) has been proven by Satinder Singh (personal communication). The ``windy gridworld'' example was suggested by Tom Kalt.

Holland's (1986) bucket-brigade idea evolved into an algorithm very close to Sarsa. The original idea of the bucket brigade involved chains of rules triggering each other; it focused on passing credit back from the current rule to the rules that triggered it. Over time, the bucket brigade came to be more like TD learning in passing credit back to any temporally preceding rule, not just to the ones that triggered the current rule. The modern form of the bucket brigade, when simplified in various natural ways, is very similar to 1-step Sarsa, as detailed by Wilson (1994).

6.5

Q-learning was introduced by Watkins (1989), whose outline of a convergence proof was later made rigorous by Watkins and Dayan (1992). More general convergence results were proved by Jaakkola, Jordan, and Singh (1994) and Tsitsiklis (1994).

6.6

Actor-critic architectures using TD learning were first studied by Witten (1977) and then by Barto, Sutton, and Anderson (1983), who introduced this use of the terms ``actor'' and ``critic.'' Sutton (1984) and Williams (1992) developed the eligibility terms mentioned in this section. Barto (1995) and Houk, Adams, and Barto (1995) presented a model of how an actor-critic architecture might be implemented in the brain.

6.7

R-learning is due to Schwartz (1993). Mahadevan (1996), Tadepalli and Ok (1994), and Bertsekas and Tsitsiklis (1996) have also studied reinforcement learning for undiscounted continual tasks. In the literature, the undiscounted continual case is often called the case of maximizing ``average reward per time step'' or the ``average-reward case.'' The name R-learning is due to Schwartz and probably was meant just to be the alphabetic successor to Q-learning, but we prefer to think of it as a reference to the learning of *relative* values. The access-control queuing example was suggested by the work of Carlström and Nordström (1997).

Next Up Previous

Next: 7 Eligibility Traces **Up:** 6 Temporal Difference Learning **Previous:** 6.9 Conclusions

---

Richard Sutton
Fri May 30 13:53:05 EDT 1997

...journey.

If this were a control problem with the objective of minimizing travel time, then we would of course make the rewards the *negative* of the elapsed time. But since we are concerned here only with prediction (policy evaluation), we can keep things simple by using positive numbers.

Richard Sutton
Fri May 30 13:53:05 EDT 1997

Part III: A Unified View

So far we have discussed three classes of methods for solving the reinforcement learning problem: dynamic programming, Monte Carlo methods, and temporal-difference learning. Although each of these are different, they are not really alternatives in the sense that one must pick one or the other. It is perfectly sensible and often desirable to apply several of the different kinds of methods at once, that is to apply a joint method with parts or aspects of more than one kind. For different tasks or different parts of one task one may want to emphasize one kind of method over another, but these choices can be made smoothly and at the time the methods are used rather that the time at which they are designed. In Part III of the book we present a unified view of the three kinds of elementary solution methods introduced in Part II.

The unifications we present in this part of the book are not rough analogies. We develop specific algorithms that embody the key ideas of one or more of the elementary solution methods. First we present the mechanism of eligibility traces, unifying Monte Carlo and temporal-difference methods. Then we bring in function approximation, enabling generalization across states and actions. Finally we reintroduce models of the environment to obtain the strengths of dynamic programming and heuristic search. All of these can be used synergistically as parts of joint methods.

7 Eligibility Traces

Eligibility traces are one of the basic mechanisms of reinforcement learning. For example, in the popular TD(λ) algorithm, the λ refers to the use of an eligibility trace. Almost any temporal-difference (TD) methods, e.g., Q-learning and Sarsa, can be combined with eligibility traces to obtain a more general method that may learn more efficiently.

There are two ways to view eligibility traces. The more theoretical view, which we emphasize here, is that they are a bridge from TD to Monte Carlo methods. When TD methods are augmented with eligibility traces they produce a family of methods spanning a spectrum that has Monte Carlo methods at one end and 1-step TD methods on the other. In between are intermediate methods that are often better than either extreme method. In this sense eligibility traces unify TD and Monte Carlo methods in a valuable and revealing way.

The other way to view eligibility traces is more mechanistic. From this perspective, an eligibility trace is a temporary record of the occurrence of an event, such as the visiting of a state or the taking of an action. The trace marks the memory parameters associated with the event as eligible for undergoing learning changes. When a TD error occurs, only the eligible states or actions are assigned credit or blame for the error. Thus, eligibility traces help bridge the gap between events and training information. Like TD methods themselves, eligibility traces are a basic mechanism for temporal credit assignment.

For reasons that will become apparent shortly, the more theoretical view of eligibility traces is called the *forward* view, and the more mechanistic view is called the *backward* view. The forward view is most useful for understanding *what* is computed by methods using eligibility traces, whereas the backward view is more appropriate for developing intuition about the algorithms themselves. In this chapter we present both views and then establish the senses in which they are equivalent, that is, in which they describe the same algorithms from two points of view. As usual, we first consider the prediction problem and then the control problem. That is, we first consider how eligibility traces are used to help in predicting returns as a function of state for a fixed policy, i.e., of estimating V^π. Only after exploring the two views of eligibility traces within this prediction setting do we extend the ideas to action values and control methods.

7.1 n-step TD Prediction

What is the space of methods in between Monte Carlo and TD methods? Consider estimating V^π from sample episodes generated using π. Monte Carlo methods perform a backup for each state based on the entire sequence of observed rewards from that state until the end of the episode. The backup of simple TD methods, on the other hand, is based on just the one next reward, using the value of the state one step later as a proxy for the remaining rewards. One kind of intermediate method, then, would perform a backup based on an intermediate number of rewards: more than one, but less than all of them until termination. For example, a 2-step backup would be based on the first two rewards and the estimated value of the state two steps later. Similarly we could have 3-step backups, 4-step backups, etc. Figure 7.1 diagrams the spectrum of *n -step backups* for V^π, with 1-step, simple TD backups on the left and up-until-termination Monte Carlo backups on the right.

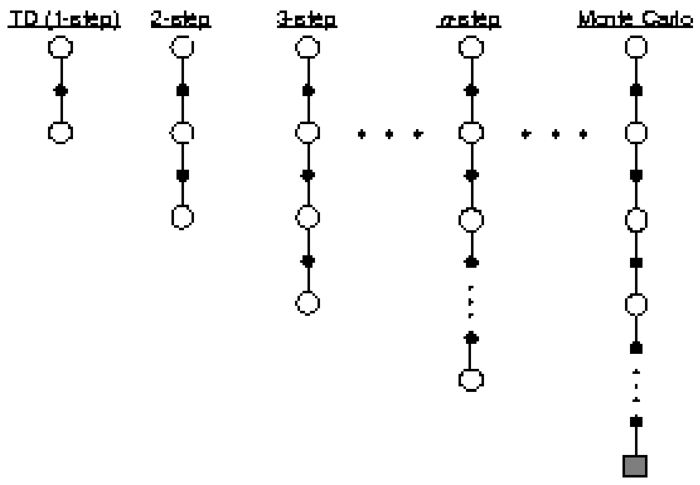


Figure 7.1: The spectrum ranging from the 1-step backups of simple TD methods to the up-until-termination backups of Monte Carlo methods. In between are the n -step backups, based on n steps of real rewards and the estimated value of the n th next state, all appropriately discounted.

The methods that use n -step backups are still TD methods because they still change an earlier estimate based on how it differs from a later estimate. Now the later estimate is not one step later, but n steps later. Methods in which the temporal difference extends over n steps are called *n -step TD methods*. The TD methods introduced in the previous chapter all use 1-step backups and we henceforth call them *1-step TD methods*.

More formally, consider the backup applied to state s_t as a result of the state-reward sequence,

$s_t, r_{t+1}, s_{t+1}, r_{t+2}, \ldots, r_T, s_T$ (omitting the actions for simplicity). We know that in Monte Carlo backups the estimate $V_t(s_t)$ of $V^\pi(s_t)$ is updated in the direction of the complete return:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{T-t-1} r_T,$$

where **T** is the last time step of the episode. Let us call this quantity the *target* of the backup. Whereas in Monte Carlo backups the target is the expected return, in 1-step backups the target is the first reward plus the discounted estimated value of the next state:

$$R_t^{(1)} = r_{t+1} + \gamma V_t(s_{t+1}).$$

This makes sense because $\gamma V_t(s_{t+1})$ takes the place of the remaining terms $\gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{T-t-1} r_T$, as we discussed in the previous chapter. Our point now is that this idea makes just as much sense after two steps as it does after one. The 2-step target is

$$R_t^{(2)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 V_t(s_{t+2}),$$

where now $\gamma^2 V_t(s_{t+2})$ takes the place of the terms $\gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \cdots + \gamma^{T-t-1} r_T$. In general, the **n** -step target is

$$R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V_t(s_{t+n}). \qquad (7.1)$$

This quantity is sometimes called the ``corrected **n** -step truncated return" because it is a return truncated after **n** steps and then approximately corrected for the truncation by adding the estimated value of the **n** th next state. That terminology is very descriptive but a bit long. We instead refer to $R_t^{(n)}$ simply as the ***n*** *-step return* at time **t**.

Of course, if the episode ends in less than **n** steps, then the truncation in an **n** -step return occurs at the episode's end, resulting in the conventional complete return. In other words, if **T-t<n**, then $R_t^{(n)} = R_t^{(T-t)} = R_t$. Thus, the last **n** **n** -step returns of any episode are always complete returns, and an infinite-step return is always a complete return. This definition enables us to simply treat Monte Carlo methods as the special case of infinite-step returns. All of this is consistent with the tricks for treating episodic and continual tasks equivalently that we introduced in Section 3.4. There we chose to treat the terminal state as a state that always transitions to itself with zero reward. Under this trick, all **n** -step returns that last up to or past

termination just happen to have the same value as the complete return.

An ***n*** *-step backup* is defined to be a backup towards the **n** -step return. In the tabular, state-value case, the increment to $V_t(s_t)$, the estimated value of $V^\pi(s_t)$ at time **t**, due to an **n** -step backup of s_t, is defined by

$$\Delta V_t(s_t) = \alpha \left[R_t^{(n)} - V_t(s_t)\right].$$

Of course, the increments to the estimated values of the other states are $\Delta V_t(s) = 0$, for all $s \neq s_t$. We define the **n** -step backup in terms of an increment, rather than as a direct update rule as we did in the previous chapter, in order to distinguish two different ways of making the updates. In *online updating*, the updates are done during the episode, as soon as the increment is computed. In this case we have $V_{t+1}(s) = V_t(s) + \Delta V_t(s)$ for all $s \in \mathcal{S}$. This is the case considered in the previous chapter. In *offline updating*, on the other hand, the increments are accumulated `on the side' and not used to change value estimates until the end of the episode. In this case, $V_t(s)$ is constant within an episode, for all **s**. If its value in this episode is $V(s)$, then its new value in the next episode will be $V(s) + \sum_{t=0}^{T-1} \Delta V_t(s)$.

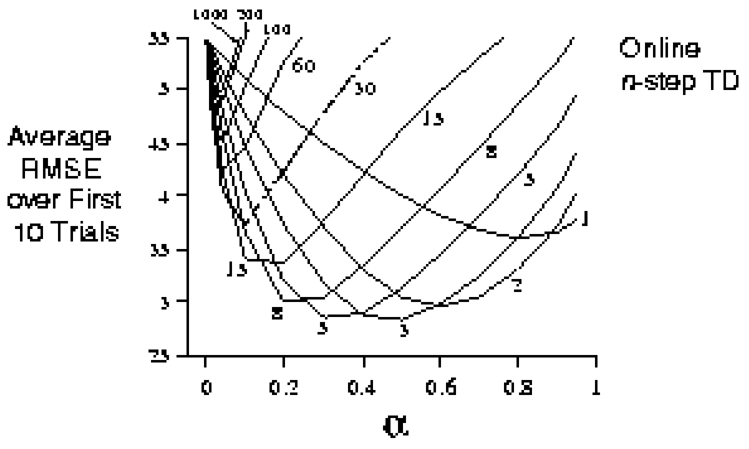
The expected value of all **n** -step returns is guaranteed to improve in a certain way over the current value function as an approximation to the true value function. For any **V**, the expected value of the **n** -step return using **V** is guaranteed to be a better estimate of V^π than **V** is, in a worst-state sense. That is, the worst error under the new estimate is guaranteed to be less than or equal to γ^n times the worst error under **V**:

$$\max_s \left| E_\pi\left\{ R_t^{(n)} \,\middle|\, s_t = s \right\} - V^\pi(s) \right| \;\leq\; \gamma^n \max_s \left| V(s) - V^\pi(s) \right|. \qquad (7.2)$$

This is called the *error-reduction property* of **n** -step returns. Because of the error-correction property, one can show formally that online and offline TD prediction methods using **n** -step backups converge to the correct predictions under appropriate technical conditions. The **n** -step TD methods thus form a family of valid methods, with 1-step TD methods and Monte Carlo methods as extreme members.

Nevertheless, **n** -step TD methods are rarely used because they are inconvenient to implement. Computing **n** -step returns requires waiting **n** steps to observe the resultant rewards and states. For large **n**, this can become problematic, particularly in control applications. The significance of **n** -step TD methods is primarily for theory and for understanding related methods that are more conveniently implemented. In the next few sections we use the idea of **n** -steps TD methods to explain and justify eligibility-trace methods.

Example 7.1 ***n*** *-step TD Methods on the Random Walk*. Consider using **n** -step TD methods on the random-walk task described in Example 6 .2 and shown in Figure 6 .4 . Suppose the first episode progressed directly from the center state, **C**, to the right, through **D** and **E**, and then terminated on the right with a return of 1. Recall that the estimated values of all the states started at an intermediate value, $V_0(s) = 0.5$. As a result of the experience just described, a 1-step method would change only the estimate for the last state, $V(E)$, which would be incremented toward **1**, the observed return. A 2-step method, on the other hand, would increment the values of the two states, **D** and **E**, preceding termination: $V(D)$ and $V(E)$ would both be incremented toward 1. A 3-step method, or any **n** -step method for **n**>**2**, would increment the values of all three of the visited states towards 1, all by the same amount. Which **n** is better? Figure 7.2 shows the results of a simple empirical assessment for a larger random walk, with 19 states (and with a **-1** outcome on the left, all values initialized to **0**). Shown is the root-mean-square error in the predictions at the end of an episode, averaged over states, the first 10 episodes, and 100 repetitions of the whole experiment. Results are shown for online and offline **n** -step TD methods with a range of values for **n** and α . Empirically, online methods with an intermediate value of **n** seem to work best on this task. This illustrates how the generalization of TD and Monte Carlo methods to **n** -step methods can potentially perform better than either of the two extreme methods. ◇



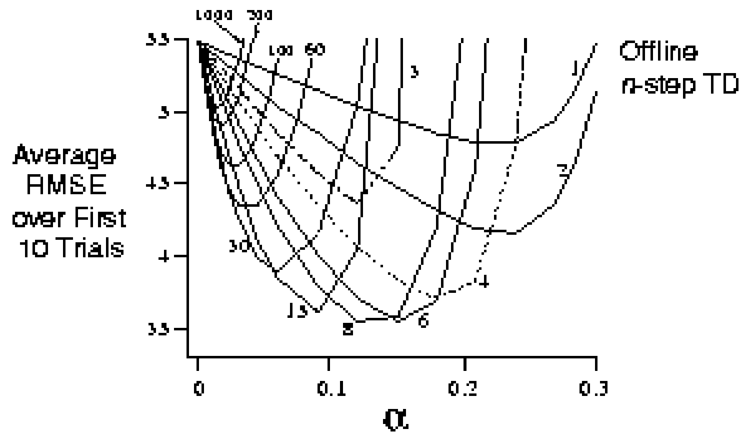


Figure 7.2: Performance of **n** -step TD methods as a function of α , for various values of **n** , on a 19-state random-walk task. Results are averages over 100 different sets of random walks. The same sets of walks were used with all methods.

Exercise 7.1

Why do you think a larger random-walk task (19 states instead of 5) was used in the examples of this chapter? Would a smaller walk have shifted the advantage to a different value of How about the change in left-side outcome from 0 to **-1**? Would that have made any difference in the best value of

Exercise 7.2

Why do you think online methods worked better than offline methods on the example task?

Exercise 7.3*

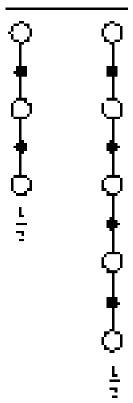
In the lower part of Figure 7.2, notice that the plot for **n=3** is very different than the others, dropping to low performance at a much lower value of α than similar methods. In fact, the same was observed for **n=5**, **n=7**, and **n=9**. Can you explain why this might have been so? In fact, we are not sure ourselves.

Richard Sutton
Fri May 30 15:01:47 EDT 1997

7.2 The Forward View of TD()

Backups can be done not just toward any **n** -step return, but toward any *average* of **n** -step returns. For example, a backup can be done toward a return that is half of a 2-step return and half of a 4-step return: $R_t^{ave} = \frac{1}{2}R_t^{(2)} + \frac{1}{2}R_t^{(4)}$. Any set of returns can be averaged in this way, even an infinite set, as long as the weights on the component returns are positive and sum to one. The overall return possesses an error-correction property similar to that of individual **n** -step returns (7.2) and thus can be used to construct backups with guaranteed convergence properties. Averaging produces a substantial new range of algorithms. For example, one could average 1-step and infinite-step backups to obtain another way of interrelating TD and Monte Carlo methods. In principle, one could even average experience-based backups with DP backups to get a simple combination of learning methods and model-based methods (see Chapter 9).

A backup that averages simpler component backups in this way is called a *complex backup*. The backup diagram for a complex backup consists of the backup diagrams for each of the component backups with a horizontal line above them and the weighting fractions below. For example, the complex backup mentioned above, mixing half of a 2-step backup and half of a 4-step backup, has the diagram:



The TD(λ) algorithm can be understood as one particular way of averaging **n** -step backups. This average contains all the **n** -step backups, each weighted proportional to

λ^{n-1}, where $0 \leq \lambda \leq 1$ (Figure 7.3). A normalization factor of $1-\lambda$ ensures that the weights sum to 1. The resulting backup is toward a return, called the λ *-return*, defined by

$$R_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)}.$$

Figure 7.4 illustrates this weighting sequence. The 1-step return is given the largest weight, $1-\lambda$, the 2-step return the next largest weight, $(1-\lambda)\lambda$, the 3-step return the weight $(1-\lambda)\lambda^2$, and so on, the weight fading by λ with each additional step. After a terminal state has been reached, all subsequent **n** -step returns are equal to R_t. If we want, we can separate these terms from the main sum, yielding

$$R_t^\lambda \;=\; (1-\lambda)\sum_{n=1}^{T-t-1} \lambda^{n-1} R_t^{(n)} \;+\; \lambda^{T-t-1} R_t. \qquad (7.3)$$

This equation makes it clearer what happens when $\lambda = 1$. In this case the main sum goes to zero, and the remaining term reduces to the conventional return, R_t. Thus, for $\lambda = 1$, backing up according to the λ -return is the same as the Monte Carlo algorithm that we called constant-α \ MC (6.1) in the previous chapter. On the other hand, if $\lambda = 0$, then the λ -return reduces to $R_t^{(1)}$, the 1-step return. Thus, for $\lambda = 0$, backing up according to the λ -return is the same as the 1-step TD method, TD(0).

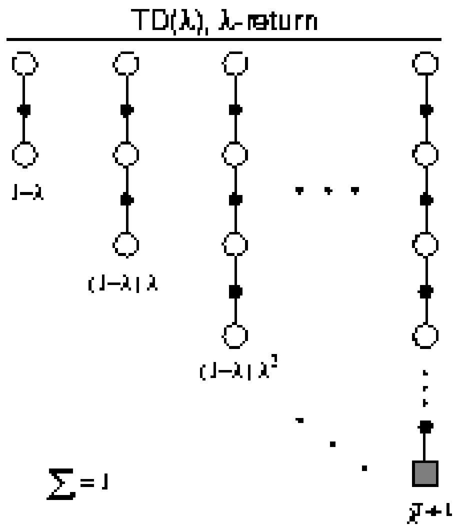


Figure 7.3: The backup digram for TD(λ) . If $\lambda = 0$, then the overall backup reduces to just its first component, the 1-step TD backup, whereas, if $\lambda = 1$, then the overall backup

reduces to just its last component, the Monte Carlo backup.

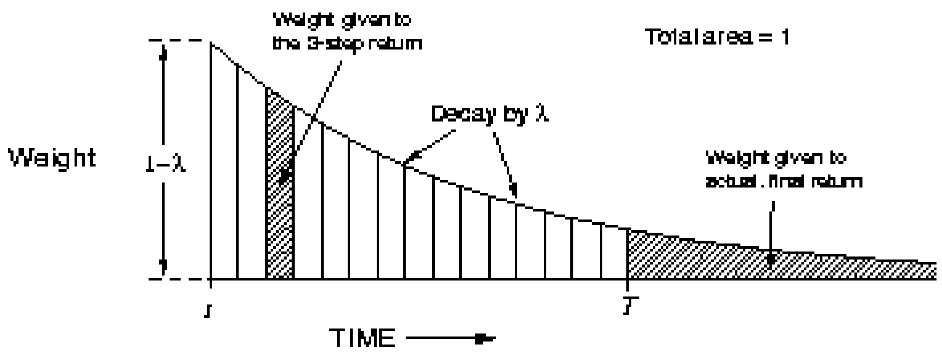


Figure 7.4: Weighting given in the λ -return to each of the **n** -step returns.

We define the λ *-return algorithm* as the algorithm that performs backups using the λ -return. On each step, **t**, it computes an increment, $\Delta V_t(s_t)$, to the value of the state occurring on that step:

$$\Delta V_t(s_t) = \alpha \left[R_t^\lambda - V_t(s_t) \right]. \qquad (7.4)$$

(The increments for other states are of course $\Delta V_t(s) = 0$, for all $s \neq s_t$.) Just as with the **n** -step TD methods, the updating can be either online or offline.

The approach that we have been taking so far is what we call the theoretical, or *forward*, view of a learning algorithm. For each state visited, we look forward in time to all the future rewards and decide how best to combine them. We might imagine ourselves riding the stream of states, looking forward from each state to determine its update, as suggested by Figure 7.5. After looking forward from and updating one state, we move onto the next and never have to work with the preceding state again. Future states, on the other hand, are viewed and processed repeatedly, once from each vantage point preceding them.

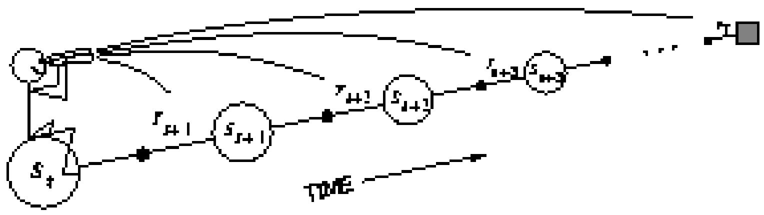


Figure 7.5: The forward or theoretical view. We decide how to update each state by looking forward to future rewards and states.

The λ -return algorithm is the basis for the forward view of eligibility traces as used in the TD(λ) method. In fact, we show in a later section that, in the offline case, the λ -return algorithm *is* the TD(λ) algorithm. The λ -return and TD(λ) methods use the λ parameter to shift from 1-step TD methods to Monte Carlo methods. The specific way this shift is done is interesting, but not obviously better or worse than the way it is done with simple **n** -step methods by varying **n** . Ultimately, the most compelling motivation for the λ way of mixing **n** -step backups is simply that there is a simple algorithm---TD(λ) ---for achieving it. This is a mechanism issue rather than a theoretical one. In the next few sections we develop the mechanistic, or backward, view of eligibility traces as used in TD(λ) .

Example 7.2

λ *-return on the Random Walk*. Figure 7.6 shows the performance of the offline λ -return algorithm on the same 19-state random walk used with the **n** -step methods in Example 7.1. The experiment is just as in the **n** -step case except that here we vary λ instead of **n** . Note that if α is picked appropriately then we get best performance with an intermediate value of λ . $\diamond$

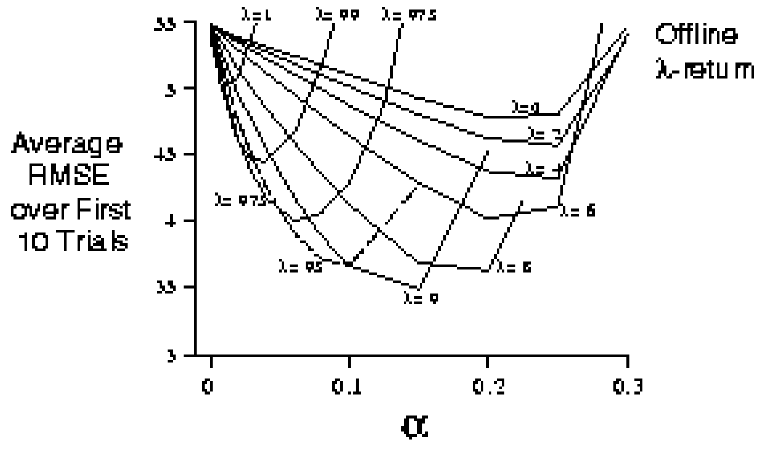


Figure 7.6: Performance of the offline λ -return algorithm on a 19-state random-walk task.

Exercise 7.4

The parameter λ characterizes how fast the exponential weighting in Figure 7.4 falls off, and thus how far into the future the λ -return algorithm looks in determining its backup. But a rate factor such as λ is sometimes an awkward way of characterizing the speed of the decay. For some purposes it is better to specify a time constant, or half-life. What is the formula for converting between λ and the half-life, τ_λ, characterizing the time by which the weighting sequence will have fallen to half of its initial value?

Richard Sutton
Fri May 30 15:01:47 EDT 1997

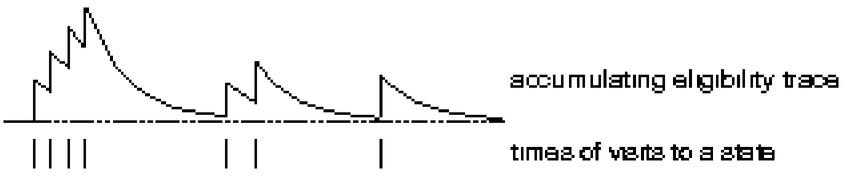
7.3 The Backward View of TD()

In the previous section we presented the forward or theoretical view of the tabular TD(λ) algorithm as a way of mixing backups that parametricaly shifts from a TD method to a Monte Carlo method. In this section we instead define TD(λ) mechanistically, and in the next section we show that this mechanism correctly implements the theoretical result described by the forward view. The mechanistic, or *backward*, view of TD(λ) is useful because it is simple conceptually and computationally. In particular, the forward view itself is not directly implementable because it is *acausal*, using at each step knowledge of what will happen many steps later. The backwards view provides a causal, incremental mechanism for approximating the forward view, and, in the offline case, for exactly achieving it.

In the backward view of TD(λ) , there is an additional memory variable associated with each state, its *eligibility trace*. The eligibility trace for state **s** at time **t** is denoted $e_t(s) \in \Re^+$. On each step, the eligibility traces for all states decay by $\gamma\lambda$, and the eligibility trace for the one state visited on the step is incremented by **1**:

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t; \\ \gamma\lambda e_{t-1}(s) + 1 & \text{if } s = s_t; \end{cases} \tag{7.5}$$

for all nonterminal states **s**, where γ is the discount rate and λ is the parameter introduced in the previous section. Henceforth we refer to λ as the *trace-decay parameter*. This kind of eligibility trace is called an *accumulating* trace because it accumulates each time the state is visited, then fades away gradually when the state is not visited, as illustrated below:



At any time, the traces record which states have recently been visited, where ``recently'' is defined in terms of $\gamma\lambda$. The traces are said to indicate the degree to which each state is *eligible* for undergoing learning changes should a reinforcing event occur. The reinforcing events we are concerned with are the moment-by-moment 1-step TD errors, e.g., for the prediction problem:

$$\delta_t \;=\; r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t). \tag{7.6}$$

In the backward view of TD(λ) , the global TD error signal triggers proportional updates to all recently visited states, as signaled by their non-zero traces:

$$\Delta V_t(s) = \alpha\, \delta_t\, e_t(s), \qquad \text{for all } s \in \mathcal{S}. \tag{7.7}$$

As always, these increments could be done on each step to form an online algorithm, or saved until the end of the episode to produce an offline algorithm. In either case, equations (7.5--7.7) provide the mechanistic definition of the TD(λ) algorithm. A complete algorithm for online TD(λ) is given in Figure 7.7.

```
Initialize $V(s)$ arbitrarily and $e(s) = 0$, for all $s \in \mathcal{S}$
Repeat (for each episode):
   Initialize s
   Repeat (for each step of episode):
      $a \leftarrow$ action given by $\pi$ for $s$
      Take action a, observe reward, r, and next state, $s'$
      $\delta \leftarrow r + \gamma V(s') - V(s)$
      $e(s) \leftarrow e(s) + 1$
      For all s:
          $V(s) \leftarrow V(s) + \alpha\delta e(s)$
          $e(s) \leftarrow \gamma\lambda e(s)$
      $s \leftarrow s'$
   until s is terminal
```

Figure 7.7: Online Tabular TD(λ) for estimating V^π.

The backward view of TD(λ) is oriented toward looking backward in time. At each moment we look at the current TD error and assign it backward to each prior state according to the state's eligibility trace at that time. We might imagine ourselves riding along the stream of states, computing TD errors, and shouting them back to the previously visited states, as suggested by Figure 7.8. Where the TD error and traces come together we get the update given by (7.7).

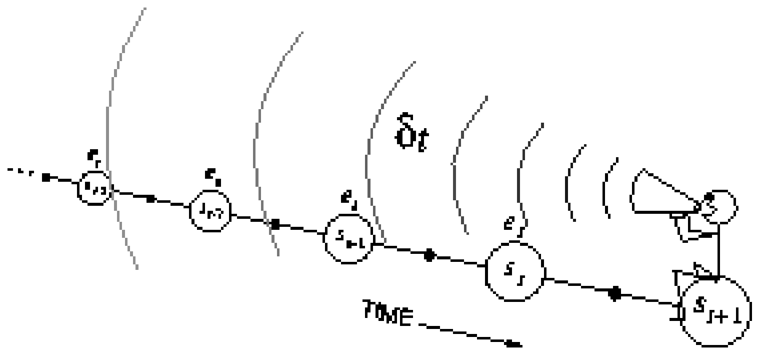


Figure 7.8: The backward or mechanistic view.

To better understand the backward view, consider what happens at various values of λ. If $\lambda = 0$, then by (7.5) all traces are zero at **t** except for the trace corresponding to s_t. Thus the TD(λ) update (7.7) reduces to the simple TD rule (6 .2), which we henceforth call TD(0). In terms of Figure 7.8, TD(0) is the case in which only the one state preceding the current one is changed by the TD error. For larger values of λ, but still $\lambda < 1$, more of the preceding states are changed, but each more temporally distant state is changed less because its eligibility trace is smaller, as suggested in the figure. We say that the earlier states are given less *credit* for the TD error.

If $\lambda = 1$, then the credit given to earlier states falls only by γ per step. This turns out to be just the right thing to do to achieve Monte Carlo behavior. For example, remember that the TD error, δ_t, includes an undiscounted term of r_{t+1}. In passing this back **k** steps it needs to be discounted, just like any reward in a return, by γ^k, which is just what the falling eligibility trace achieves. If both $\lambda = 1$ and $\gamma = 1$, then the eligibility traces do not decay at all with time. This turns out to work just fine, behaving like a Monte Carlo method for an undiscounted, episodic task. If $\lambda = 1$, the algorithm is also known as TD(1).

TD(1) is a way of implementing Monte Carlo algorithms that is more general than those presented earlier and that significantly increases their range of applicability. Whereas the earlier Monte Carlo methods were limited to episodic tasks, TD(1) can be applied to discounted continuing tasks as well. Moreover, TD(1) can be performed incrementally and online. One disadvantage of Monte Carlo methods is that they learn nothing from an episode until it is over. For example, if something a Monte Carlo control method does produces a very poor reward, but does not end the episode, then the agent's tendency to do that will be undiminished during that episode. Online TD(1), on the other hand, learns in an **n** -step TD way from the incomplete ongoing episode, where the **n** \ steps are all the way up to the current step. If something unusually good or bad happens during an episode, control methods based on TD(1) can learn immediately and alter their behavior on that same episode.

Richard Sutton
Fri May 30 15:01:47 EDT 1997

7.4 Equivalence of the Forward and Backward Views

In this section we show that offline TD(λ) , as defined mechanistically above, achieves the same weight updates as the offline λ -return algorithm. In this sense we align the forward (theoretical) and backward (mechanistic) views of TD(λ) . Let $\Delta V_t^\lambda(s_t)$ denote the update at time **t** of $V(s_t)$ according to the λ -return algorithm (7.4), and let $\Delta V_t^{TD}(s)$ denote the update at time **t** of state **s** according to the mechanistic definition of TD(λ) as given by (7.7). Then our goal is to show that, over an episode, the sum of all the updates is the same for the two algorithms:

$$\sum_{t=0}^{T-1} \Delta V_t^{TD}(s) = \sum_{t=0}^{T-1} \Delta V_t^\lambda(s_t) \mathcal{I}_{ss_t}, \qquad \text{for all } s \in \mathcal{S}, \qquad (7.8)$$

where $\mathcal{I}_{ss'}$ is an identity-indicator function, equal to **1** if $s = s'$ and equal to 0 otherwise.

First note that an accumulating eligibility trace can be written explicitly (non-recursively) as

$$e_t(s) = \sum_{k=0}^{t} (\gamma\lambda)^{t-k} \mathcal{I}_{ss_k}.$$

Thus, the lefthand side of (7.8) can be written

$$\begin{aligned}
\sum_{t=0}^{T-1} \Delta V_t^{TD}(s) &= \sum_{t=0}^{T-1} \alpha\delta_t \sum_{k=0}^{t} (\gamma\lambda)^{t-k} \mathcal{I}_{ss_k} \\
&= \sum_{k=0}^{T-1} \alpha\delta_k \sum_{t=0}^{k} (\gamma\lambda)^{k-t} \mathcal{I}_{ss_t} \\
&= \sum_{k=t}^{T-1} \alpha\delta_k \sum_{t=0}^{T-1} (\gamma\lambda)^{k-t} \mathcal{I}_{ss_t} \\
&= \sum_{t=0}^{T-1} \alpha\mathcal{I}_{ss_t} \sum_{k=t}^{T-1} (\gamma\lambda)^{k-t} \delta_k. \qquad (7.9)
\end{aligned}$$

Now we turn to the righthand side of (7.8). Consider an individual update of the λ -return algorithm:

$$\begin{aligned}
\frac{1}{\alpha}\Delta V_t^{\lambda}(s_t) &= R_t^{\lambda} - V_t(s_t) \\
&= -V_t(s_t) + (1-\lambda)\lambda^0[r_{t+1} + \gamma V_t(s_{t+1})] \\
&\qquad + (1-\lambda)\lambda^1[r_{t+1} + \gamma r_{t+2} + \gamma^2 V_t(s_{t+2})] \\
&\qquad + (1-\lambda)\lambda^2[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 V_t(s_{t+3})] \\
&\qquad \qquad \vdots \qquad \vdots \qquad \vdots \qquad \ddots
\end{aligned}$$

Examine the first column inside the brackets---all the r_{t+1}'s with their weighting factors of $1-\lambda$ times powers of λ . It turns out that all the weighting factors sum to 1. Thus we can pull out the first column and get just an unweighted term of r_{t+1}. A similar trick pulls out the second column in brackets, starting from the second row, which sums to $\gamma\lambda r_{t+2}$. Repeating this for each column, we get

$$\begin{aligned}
\frac{1}{\alpha}\Delta V_t^\lambda(s_t) &= -V_t(s_t) \\
&\quad + (\gamma\lambda)^0\left[r_{t+1} + \gamma V_t(s_{t+1}) - \gamma\lambda V_t(s_{t+1})\right] \\
&\quad + (\gamma\lambda)^1\left[r_{t+2} + \gamma V_t(s_{t+2}) - \gamma\lambda V_t(s_{t+2})\right] \\
&\quad + (\gamma\lambda)^2\left[r_{t+3} + \gamma V_t(s_{t+3}) - \gamma\lambda V_t(s_{t+3})\right] \\
&\quad \vdots \\
&= \quad (\gamma\lambda)^0\left[r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)\right] \\
&\quad + (\gamma\lambda)^1\left[r_{t+2} + \gamma V_t(s_{t+2}) - V_t(s_{t+1})\right] \\
&\quad + (\gamma\lambda)^2\left[r_{t+3} + \gamma V_t(s_{t+3}) - V_t(s_{t+2})\right] \\
&\quad \vdots \\
&\approx \sum_{k=t}^{\infty}(\gamma\lambda)^{k-t}\delta_k \\
&\approx \sum_{k=t}^{T-1}(\gamma\lambda)^{k-t}\delta_k.
\end{aligned}$$

The approximation above is exact in the case of offline updating, in which case V_t is the same for all **t**. The last step is exact (not an approximation) because all the δ_k terms omitted are due to fictitious steps ``after'' the terminal state has been entered. All these steps have zero rewards and zero values; thus all their δ's are zero as well. Thus, we have shown that in the offline case the righthand side of (7.8) can be written

$$\sum_{t=0}^{T-1}\Delta V_t^\lambda(s_t)\mathcal{I}_{ss_t} = \sum_{t=0}^{T-1}\alpha\mathcal{I}_{ss_t}\sum_{k=t}^{T-1}(\lambda\gamma)^{k-t}\delta_k,$$

which is the same as (7.9). This proves (7.8).

In the case of online updating, the approximation made above will be close as long as α is small and thus V_t changes little during an episode. Even in the online case we can expect the updates of TD(λ) and of the λ -return algorithm to be similar.

For the moment let us assume that the increments are small enough during an episode that online TD(λ) gives essentially the same update over the course of an episode as does the λ -return algorithm. There still remain interesting questions about what happens *during* an episode. Consider the updating of the value of state s_t in mid-episode, at time **t+k**. Under online TD(λ) , the effect at **t+k** is just as if we had done a λ -return update treating the last observed state as the terminal state of the episode with a nonzero terminal value equal to its current estimated value. This relationship is maintained from step to step as each new state is observed.

Example 7.3

Random Walk with TD(λ) . Because offline TD(λ) is equivalent to the λ -return algorithm, we already have the results for offline TD(λ) on the 19-state random walk task; they are shown in Figure 7.6. The comparable results for online TD(λ) are shown in Figure 7.9. Note that the online algorithm works better over a broader range of parameters. This is often found to be the case for online methods. $\diamond$

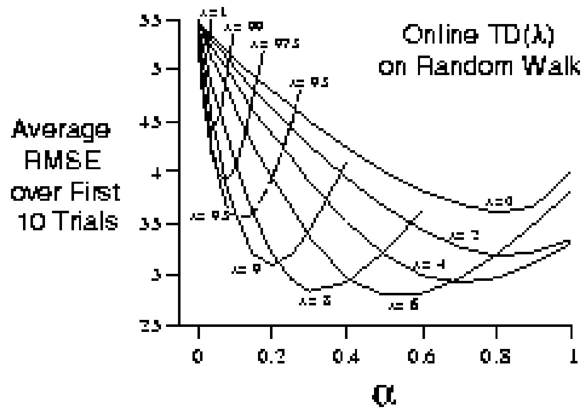


Figure 7.9: Performance of online TD(λ) on the 19-state random-walk task.

Exercise 7.5

As we have noted, when done online, TD(λ) only approximates the λ -return algorithm. We sometimes wonder if there isn't some slightly different TD method which would maintain the equivalence even in the online case. One idea is to define the TD error instead as $\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_{t-1}(s_t)$. Show that in this case the modified TD(λ) algorithm would then achieve exactly

$$\Delta V_t(s_t) = \alpha \left[R_t^\lambda - V_{t-1}(s_t) \right],$$

even in the case of online updating with large α. Would this alternate TD(λ) be better or worse than the one described in the text? Describe an experiment or analysis that would help answer this question.

Richard Sutton
Fri May 30 15:01:47 EDT 1997

7.5 Sarsa()

How can eligibility traces be used not just for prediction, as in TD(λ) , but for control? As usual, the main idea of one popular approach is simply to learn action values, $Q_t(s,a)$, rather than state values, $V_t(s)$. In this section we show how eligibility traces can be combined with Sarsa in a straightforward way to produce an on-policy TD control method. The eligibility-trace version of Sarsa we call *Sarsa* (λ) , and the original version presented in the previous chapter we henceforth call *1-step Sarsa*.

The idea in Sarsa(λ) is to apply the TD(λ) prediction method to state-action pairs rather than to states. Obviously, then, we need a trace not just for each state, but for each state-action pair. Let $e_t(s,a)$ denote the trace for state-action pair **s,a**. Otherwise the method is just like TD(λ) , substituting state-action variables for state variables ($Q(s,a)$ for $V(s)$ and $e_t(s,a)$ for $e_t(s)$):

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha\delta_t e_t(s,a), \qquad \text{for all } s,a$$

where

$$\delta_t = r_{t+1} + \gamma Q_t(s_{t+1},a_{t+1}) - Q_t(s_t,a_t)$$

and

$$e_t(s,a) = \begin{cases} \gamma\lambda e_{t-1}(s,a)+1 & \text{if } s=s_t \text{ and } a=a_t; \\ \gamma\lambda e_{t-1}(s,a) & \text{otherwise.} \end{cases} \qquad \text{for all } s,a \tag{7.10}$$

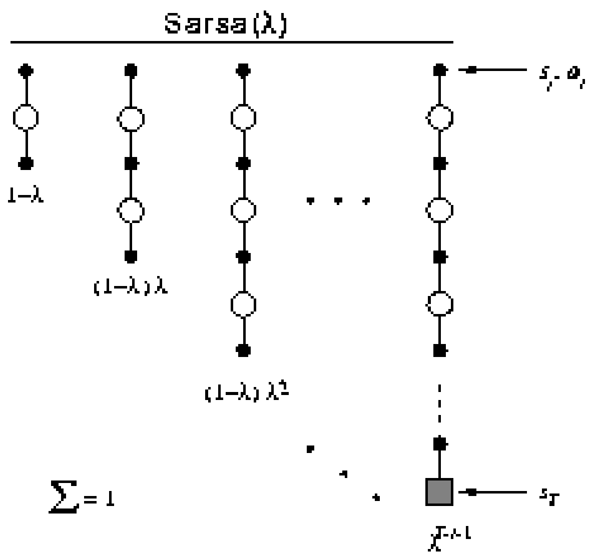


Figure 7.10: Sarsa's backup diagram.

Figure 7.10 shows the backup diagram for Sarsa(λ) . Notice the similarity to the diagram of the TD(λ) algorithm (Figure 7.3). The first backup looks ahead one full step, to the next state-action pair, the second looks ahead two steps, etc. A final backup is based on the complete return. The weighting of each backup is just as in TD(λ) and the λ -return algorithm.

1-step Sarsa and Sarsa(λ) are on-policy algorithms, meaning that they approximate $Q^\pi(s,a)$, the action values for the current policy, π, then improve the policy gradually based on the approximate values for the current policy. The policy improvement can be done in many different ways, as we have seen throughout this book. For example, the simplest approach is to use the ϵ -greedy policy with respect to the current action-value estimates. Figure 7.11 shows the complete Sarsa(λ) algorithm for this case.

```
Initialize $Q(s,a)$ arbitrarily and $e(s,a)=0$, for all $s,a$
Repeat (for each episode):
   Initialize s, a
   Repeat (for each step of episode):
      Take action a, observe r, $s'$
      Choose $a'$ from $s'$ using policy derived from Q
         (e.g., $\epsilon$-greedy)
      $\delta \leftarrow r + \gamma Q(s',a') - Q(s,a)$
      $e(s,a) \leftarrow e(s,a) + 1$
      For all s,a:
         $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
         $e(s,a) \leftarrow \gamma\lambda e(s,a)$
```

```
    $s \leftarrow s'$; $a \leftarrow a'$
until s is terminal
```

Figure 7.11: Tabular Sarsa(λ) .

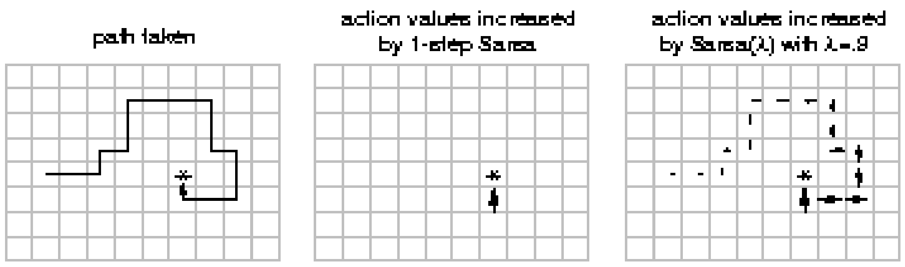

path taken
action values increased by 1-step Sarsa
action values increased by Sarsa(λ) with λ=.9

Figure 7.12: Gridworld example of the speedup of policy learning due to the use of eligibility traces. In one episode, 1-step methods strengthen only the last action leading to an unusually high reward, whereas eligibility trace methods can strengthen the whole sequence of actions.

Example 7.4 *Traces in Gridworld*. The use of eligibility traces can substantially increase the efficiency of control algorithms. The reason for this is illustrated by the gridworld example in Figure 7.12. The first panel shows the path taken by an agent in a single episode, ending at a location of high reward, marked by the `*'. In this example the values were all initially 0, and all rewards were zero except for a positive reward at the `*' location. The arrows in the other two panels show which action values were strengthened as a result of this path by 1-step Sarsa and Sarsa(λ) methods. The 1-step method strengthens only the last action of the sequence of actions that led to the high reward, whereas the trace method strengthens many actions of the sequence. The degree of strengthening (indicated by the size of the arrows) falls off (according to $\gamma\lambda$) with steps from the reward. In this example, $\gamma = 1$ and $\lambda = .9$. $\diamond$

Next Up Previous
Next: 7.6 Q() **Up:** 7 Eligibility Traces **Previous:** 7.4 Equivalence of the

Richard Sutton
Fri May 30 15:01:47 EDT 1997

7.6 Q()

Two different methods have been proposed that combine eligibility traces and Q-learning, which we call *Watkins's Q (λ)* and *Peng's Q (λ)* after the researchers how first proposed them. First we describe Watkins's Q(λ) .

Recall that Q-learning is an off-policy method, meaning that the policy learned about need not be the same as the one used to select actions. In particular, Q-learning learns about the greedy policy while it typically follows a policy involving exploratory actions---occasional selections of actions that are suboptimal according to Q_t. Because of this, special care is required when introducing eligibility traces.

Suppose we are backing up the state-action pair s_t, a_t at time **t**. Suppose that on the successive two time steps the agent selects the greedy action, but on the third, at time **t+3**, the agent selects an exploratory, non-greedy action. In learning about the value of the greedy policy at s_t, a_t we can only use subsequent experience as long as the greedy policy is being followed. Thus, we can use the 1-step and 2-step returns, but not in this case the 3-step return. The **n** -step returns for all $n \geq 3$ no longer have any necessary relationship to the greedy policy.

Thus, unlike TD(λ) or Sarsa(λ) , Watkins's Q(λ) does not look ahead all the way to the end of the episode in its backup. It only looks ahead as far as the next exploratory action. Aside from this difference, however, Watkins's Q(λ) is much like TD(λ) and Sarsa(λ) . Their lookahead stops at episode's end, whereas Q(λ)'s lookahead stops at the first exploratory action, or at episode's end if there are no exploratory actions before that. Actually, to be more precise, 1-step Q-learning and Watkins's Q(λ) both look one action *past* the first exploration, using their knowledge of the action values. For example, suppose the very first action, a_{t+1}, is exploratory. Watkins's Q(λ) would still do the one step update of $Q_t(s_t, a_t)$ toward $r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a)$. In general, if a_{t+n} is the first exploratory action, then the longest backup is toward:

$$r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n \max_a Q_t(s_{t+n}, a),$$

where here we assume offline updating. The backup diagram in Figure 7.13 illustrates the forward view of Watkins's Q(λ), showing all the component backups.

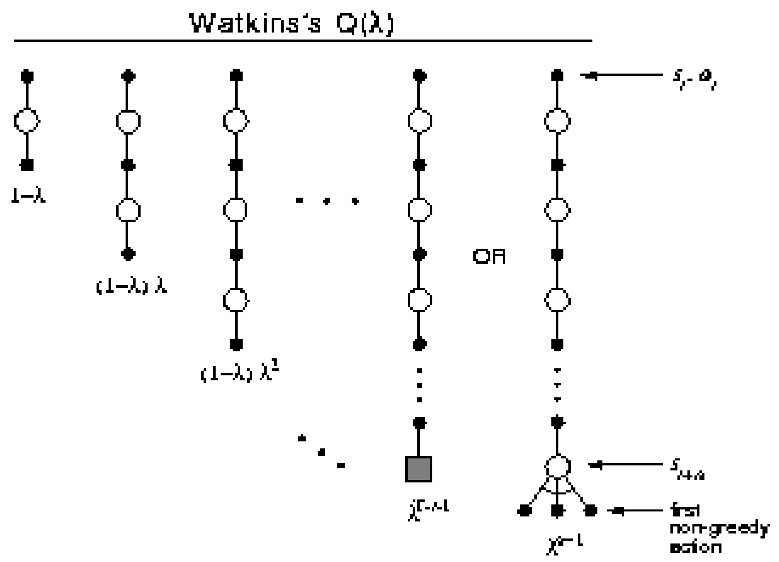


Figure 7.13: The backup diagram for Watkins's Q(λ). The series of component backups ends with either the end of the episode or the first non-greedy action, whichever comes first.

The mechanistic or backward view of Watkins's Q(λ) is also very simple. Eligibility traces are used just as in Sarsa(λ), except that they are set to zero whenever an exploratory (non-greedy) action is taken. The trace update is best thought of as occurring in two steps. First, the traces for all state-action pairs are either decayed by $\gamma\lambda$ or, if an exploratory action was taken, set to **0**. Second, the trace corresponding to the current state and action is incremented by **1**. The overall result is:

$$e_t(s,a) = \mathcal{I}_{ss_t} \cdot \mathcal{I}_{aa_t} + \begin{cases} \gamma\lambda\, e_{t-1}(s,a) & \text{if } Q_{t-1}(s_t,a_t) = \max_a Q_{t-1}(s_t,a); \\ 0 & \text{otherwise,} \end{cases}$$

where, as before, $\mathcal{I}_{xy}$ is an identity-indicator function, equal to **1** if **x=y** and **0** otherwise. The rest of the algorithm is defined by:

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha\,\delta_t\, e_t(s,a),$$

where

$$\delta_t = r_{t+1} + \gamma \max_{a'} Q_t(s_{t+1},a') - Q_t(s_t,a_t).$$

Figure 7.14 shows the complete algorithm in pseudocode.

```
Initialize $Q(s,a)$ arbitrarily and $e(s,a) = 0$, for all s,a
Repeat (for each episode):
   Initialize s, a
   Repeat (for each step of episode):
      Take action a, observe r, $s'$
```

```
        Choose $a'$ from $s'$ using policy derived from Q (e.g., $\epsilon$ -greedy)
        $a^* \leftarrow \arg\max_b Q(s', b)$ (if $a'$ ties for the max, then $a^* \leftarrow a'$)
        $\delta \leftarrow r + \gamma Q(s', a^*) - Q(s, a)$
        $e(s, a) \leftarrow e(s, a) + 1$
        For all s,a:
            $Q(s, a) \leftarrow Q(s, a) + \alpha\delta e(s, a)$
            If $a' = a^*$ then $e(s, a) \leftarrow \gamma\lambda e(s, a)$
                           else $e(s, a) \leftarrow 0$
        $s \leftarrow s'$; $a \leftarrow a'$
    until s is terminal
```

Figure 7.14: Tabular version of Watkins's Q(λ) algorithm.

Unfortunately, cutting off traces every time an exploratory action is taken loses much of the advantage of using eligibility traces. If exploratory actions are frequent, as they often are early in learning, then only rarely will backups of more than one or two steps be done, and learning may be little faster than 1-step Q-learning. Peng's Q(λ) is an alternate version of Q(λ) meant to remedy this. Peng's Q(λ) can be thought of as a hybrid of Sarsa(λ) and Watkins's Q(λ) .

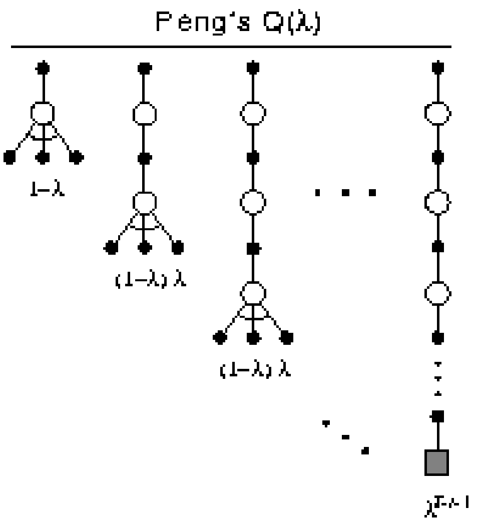


Figure 7.15: The backup diagram for Peng's Q(λ) .

Conceptually, Peng's Q(λ) uses the mixture of backups shown in Figure 7.15. Unlike Q-learning, there is no distinction between exploratory and greedy actions. Each component backup is over many steps of actual experiences, and all but the last is capped by a final maximization over actions. The component backups then are neither on-policy nor off-policy. The earlier transitions of each are on-policy whereas the last (fictitious) transition uses the greedy policy. As a consequence, for a fixed non-greedy policy, Q_t converges to neither Q^π nor Q^* under Peng's Q(λ) algorithm, but to some hybrid of the two. However, if the policy is gradually made more greedy then the method may still converge to Q^*. As of this writing this has not yet been proved. Nevertheless, the method performs well empirically. Most studies have shown it performing significantly better than Watkins's Q(λ) and almost as well as Sarsa(λ) .

On the other hand, Peng's Q(λ) can not be implemented as simply as Watkins's Q(λ) . For a complete description of the needed implementation, see Peng and Williams (1994, 1996). One could imagine yet a third version of Q(λ) , let us call it *naive Q (λ)* , that is just like Watkins's Q(λ) except that the traces are not set to zero on exploratory actions. This method might have some of the same advantages of Peng's Q(λ) , but without the complex implementation. We know of no experience with this method, but perhaps it is not as naive as one might at first suppose.

Next Up Previous
Next: 7.7 Eligibility Traces for **Up:** 7 Eligibility Traces **Previous:** 7.5 Sarsa()

Richard Sutton
Fri May 30 15:01:47 EDT 1997

7.7 Eligibility Traces for Actor-Critic Methods (*)

In this section we describe how to extend the actor-critic methods introduced in Section 6 .6 to use eligibility traces. This is fairly straightforward. The critic part of an actor-critic method is simply on-policy learning of V^π. The TD(λ) algorithm can be used for that, with one eligibility trace for each state. The actor part needs to use an eligibility trace for each state-action pair. Thus, an actor-critic method needs two sets of traces, one for each state and one for each state-action pair.

Recall that the 1-step actor-critic method updates the actor by

$$p_{t+1}(s,a) = \begin{cases} p_t(s,a) + \alpha\,\delta_t & \text{if } a = a_t \text{ and } s = s_t \\ p_t(s,a) & \text{otherwise,} \end{cases}$$

where δ_t is the TD(λ) error (7.6), and $p_t(s,a)$ is the preference for taking action **a** at time **t** if in state **s**. The preferences determine the policy via, for example, a softmax method (Section 2 .3). We generalize the above equation to use eligibility traces as follows:

$$p_{t+1}(s,a) = p_t(s,a) + \alpha\,\delta_t\, e_t(s,a), \tag{7.11}$$

where $e_t(s,a)$ denotes the trace at time **t** for state-action pair **s,a**. For the simplest case mentioned above, the trace can be updated just as in Sarsa(λ) .

In Section 6 .6 we also discussed a more sophisticated actor-critic method that uses the update

$$p_{t+1}(s,a) = \begin{cases} p_t(s,a) + \alpha\,\delta_t[1 - \pi_t(s,a)] & \text{if } a = a_t \text{ and } s = s_t \\ p_t(s,a) & \text{otherwise.} \end{cases}$$

To generalize this equation to eligibility traces we can use the same update (7.11) with a slightly different trace. Rather than incrementing the trace by 1 each time a state-action pair occurs, it is updated by $1 - \pi_t(s_t,a_t)$:

$$e_t(s,a) = \begin{cases} \gamma\lambda e_{t-1}(s,a) + 1 - \pi_t(s_t,a_t) & \text{if } s = s_t \text{ and } a = a_t; \\ \gamma\lambda e_{t-1}(s,a) & \text{otherwise,} \end{cases} \tag{7.12}$$

for all **s,a**.

Richard Sutton
Fri May 30 15:01:47 EDT 1997

7.8 Replacing Traces

In some cases significantly better performance can be obtained by using a slightly modified kind of trace known as a *replacing trace*. Suppose a state is visited and then revisited before the trace due to the first visit has fully decayed to zero. With accumulating traces (7.5), the revisit causes a further increment in the trace, driving it greater than **1**, whereas with replacing traces, the trace is reset to **1**. Figure 7.16 contrasts these two kinds of traces. Formally, a replacing trace for a discrete state **s** is defined by

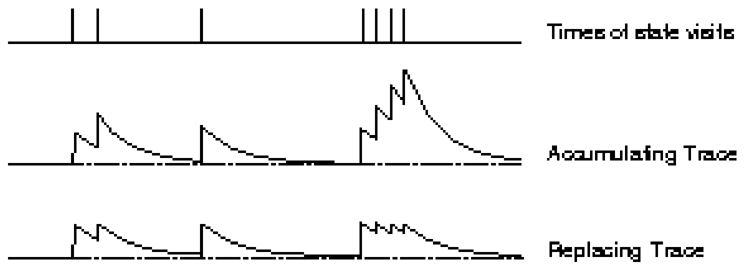


Figure 7.16: Accumulating and replacing traces.

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t; \\ 1 & \text{if } s = s_t. \end{cases} \qquad (7.13)$$

Prediction or control algorithms using replacing traces are often called *replace-trace* methods. Although replacing traces are only slightly different from accumulating traces, they can produce a significant improvement in learning rate. Figure 7.17 compares the performance of conventional and replace-trace versions of TD(λ) on the 19-state random-walk prediction task. Other examples for a slightly more general case are given in Figure 8 .10 in the next chapter.

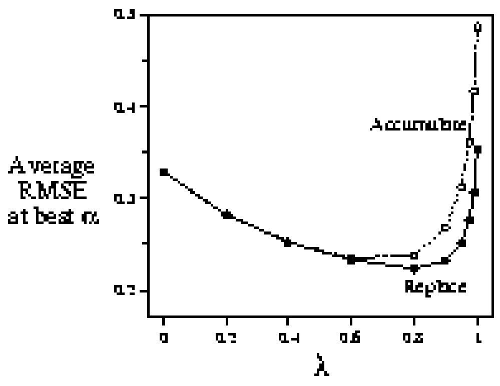


Figure 7.17: Error as a function of λ on a 19-state random walk process. These data are using the best value of α for each value of λ.

Example 7.5

Figure 7.18 shows an example of the kind of task that is difficult for control methods using accumulating eligibility traces. All rewards are zero except on entering the terminal state for a reward of +1. From each state, selecting the `right` action brings the agent one step closer to the terminal reward, whereas the `wrong` (upper) action leaves it in the same state to try again. The full sequence of states is long enough that one would like to use long traces to get the fastest learning. However, problems occur if long accumulating traces are used. Suppose, on the first episode, at some state, **s**, the agent by chance takes the `wrong` action a few times before taking the `right` action. As the agent continues, the trace $e(s, \mathtt{wrong})$ is likely to be larger than the trace $e(s, \mathtt{right})$. The `right` action was more recent, but the `wrong` action was selected more times. When reward is finally received, then, the value for the wrong action is likely to go up more than for the right action. On the next episode the agent will be even more likely to go the `wrong` way many times before going `right`, making it even more likely that the `wrong` action will have the larger trace. Eventually, all of this will be corrected, but learning is significantly slowed. With replacing traces, on the other hand, this problem never occurs. No matter how many times the `wrong` action is taken, its eligibility trace is always less than that for the `right` action after the `right` action has been taken. $\diamond$

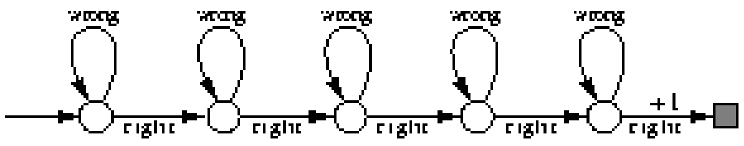


Figure 7.18: A simple task that causes problems for control methods using accumulating traces.

There is an interesting relationship between the replace-trace methods and Monte Carlo methods in the undiscounted case. Just as conventional TD(1) is related to the every-visit MC algorithm, replace-trace TD(1) is related to the first-visit MC algorithm. In particular, the offline version of replace-trace TD(1) is formally identical to first-visit MC (Singh and Sutton, 1996). How, or even whether, these methods and results extend to the discounted case is unknown.

There are several possible ways to generalize replacing eligibility traces for use in control methods. Obviously, when a state is re-visited and a new action selected, the trace for that action should be reset to 1. But what of the traces for the other actions for that state? The approach recommended by Singh and Sutton (1996) is to set the traces of all the other actions from the re-visited state to 0. In this case, the state-action traces are updated by the following instead of (7.10):

$$e_t(s,a) = \begin{cases} 1+\gamma\lambda e_{t-1}(s,a) & \text{if } s=s_t \text{ and } a=a_t; \\ 0 & \text{if } s=s_t \text{ and } a \neq a_t; \\ \gamma\lambda e_{t-1}(s,a) & \text{if } s \neq s_t. \end{cases} \qquad \text{for all } s,a \tag{7.14}$$

Note that this variant of replacing traces works out even better than the original replacing traces in the example task. Once the `right` action has been selected, the `wrong` action is left with no trace at all. The results shown in Figure 8 .10 were all obtained using this kind of replacing trace.

Exercise 7.6

In Example 7.5, suppose from state **s** the `wrong` action is taken twice before the `right` action is taken. If accumulating traces are used, then how big must the trace parameter λ be in order for the wrong action to end up with a larger eligibility trace than the right action?

Exercise 7.7 (programming)

Program Example 7.5 and compare accumulate-trace and replace-trace versions of Sarsa(λ) on it, for $\lambda = .9$ and a range of α values. Can you empirically demonstrate the claimed advantage of replacing traces on this example?

Exercise 7.8*

Draw a backup diagram for Sarsa(λ) with replacing traces.

Richard Sutton
Fri May 30 15:01:47 EDT 1997

7.9 Implementation Issues

It might at first appear that methods using eligibility traces are much more complex than 1-step methods. A naive implementation would require every state (or state-action pair) to update both its value estimate and its eligibility trace on every time step. This would not be a problem for implementations on common Single-Instruction-Multiple-Data parallel computers or in plausible neural implementations, but it is for implementations on conventional serial computers. Fortunately, for typical values of λ and γ the eligibility traces of almost all states are always very near zero; only those that have recently been visited will have traces significantly greater than zero. Only these few states really need to be updated because the updates at the others will have essentially no effect.

In practice, then, implementations on conventional computers keep track of and update only the few states with non-zero traces. Using this trick, the computational expense of using traces is typically a few times that of a 1-step method. The exact multiple of course depends on λ and γ and on the expense of the other computations. Cichosz (1995) has demonstrated a further implementation technique which reduces complexity to a constant independent of λ and γ . Finally, it should be noted that the tabular case is in some sense a worst case for the computational complexity of traces. When function approximators are used (Chapter 8) the computational advantages of not using traces generally decrease. For example, if artificial neural networks and backpropagation are used, then traces generally cause only a doubling of the required memory and computation per step.

Exercise 7.9

Write pseudocode for an implementation of TD(λ) that updates only value estimates for states whose traces are greater than ϵ .

Richard Sutton

Fri May 30 15:01:47 EDT 1997

7.10 Variable (*)

The λ -return can be significantly generalized beyond what we have described so far by allowing λ to vary from step to step, i.e., by redefining the trace update as

$$e_t(s) = \begin{cases} \gamma \lambda_t e_{t-1}(s) & \text{if } s \neq s_t; \\ \gamma \lambda_t e_{t-1}(s) + 1 & \text{if } s = s_t; \end{cases}$$

where λ_t denotes the value of λ at time **t**. This is an advanced topic because the added generality has never been used in practical applications, but it is interesting theoretically and may yet prove useful. For example, one idea is to vary λ as a function of state, i.e., $\lambda_t = \lambda(s_t)$. If a state's value estimate is believed to be known with high certainty, then it makes sense to use that estimate fully, ignoring whatever states and rewards are received after it. This corresponds to cutting off all the traces once this state has been reached, i.e., to choosing the λ for the certain state to be zero or very small. Similarly, states whose value estimates are highly uncertain, perhaps because even the state estimate is unreliable, can be given λs near 1. This causes their estimated values to have little effect on any updates. They are ``skipped over'' until a state that is known better is encountered. Some of these ideas were explored formally by Sutton and Singh (1994).

The eligibility-trace equation above is the backward view of variable λs. The corresponding forward view is a more general definition of the λ -return:

$$\begin{aligned} R_t^\lambda &= \sum_{n=1}^{\infty} R_t^{(n)} (1 - \lambda_{t+n}) \prod_{i=t+1}^{t+n-1} \lambda_i \\ &= \sum_{k=t+1}^{T-1} R_t^{(k-t)} (1 - \lambda_k) \prod_{i=t+1}^{k-1} \lambda_i \quad + \quad R_t \prod_{i=t+1}^{T-1} \lambda_i . \end{aligned}$$

Exercise 7.10*

Prove that the forward and backward views of offline TD(λ) remain equivalent under their new definitions with variable λ given in this section. Follow the example of the proof in Section 7.4.

Richard Sutton
Fri May 30 15:01:47 EDT 1997

7.11 Conclusions

Eligibility traces in conjunction with TD errors provide an efficient, incremental way of shifting and choosing between Monte Carlo and TD methods. Traces can be used without TD errors to achieve a similar effect, but only awkwardly. A method such as TD(λ) enables this to be done from partial experiences and with little memory and little non-meaningful variation in predictions.

As we discussed in Chapter 5 , Monte Carlo methods have advantages in tasks in which the state is not completely known and which thus appear to be non-Markov. Because eligibility traces make TD methods more like Monte Carlo methods, they also have advantages in these cases. If one wants to use TD methods because of their other advantages, but the task is at least partially non-Markov, then the use of an eligibility-trace method is indicated. Eligibility traces are the first line of defense against both long delayed rewards and non-Markov tasks.

By adjusting λ , we can place eligibility-trace methods anywhere along a continuum from Monte Carlo and TD methods. Where shall we place them? We do not yet have a good theoretical answer to this question, but a clear empirical answer appears to be emerging. On tasks with many steps per episode, or many steps within the half-life of discounting, then it appears significantly better to use eligibility traces than not to (e.g., see Figure 8 .10). On the other hand, if the traces are so long as to produce a pure Monte Carlo method, or nearly so, then performance again degrades sharply. An intermediate mixture appears the best choice. Eligibility traces should be used to bring us toward Monte Carlo methods, but not all the way there. In the future it may be possible to vary the tradeoff between TD and Monte Carlo methods more finely using variable λ , but at present it is not clear how this can be done reliably and usefully.

Methods using eligibility traces require more computation than 1-step methods, but in return they offer significantly faster learning, particularly when rewards are delayed by many steps. Thus it often makes sense to use eligibility traces when data is scarce and can not be repeatedly processed, as is often the case in online applications. On the other hand, in offline applications in which data can be generated cheaply, perhaps from an inexpensive simulation, then it often does not pay to use eligibility traces. In these cases the objective is not to get more out of a limited amount of data, but simply to process as much data as quickly as possible. In these cases the speedup per datum due to traces is typically not worth their computational cost, and 1-step methods are favored.

Next Up Previous

Next: 7.12 Bibliographical and Historical **Up:** 7 Eligibility Traces **Previous:** 7.10 Variable (*)

Richard Sutton
Fri May 30 15:01:47 EDT 1997

7.12 Bibliographical and Historical Remarks

7.1 and 7 .2

The forward view of eligibility traces in terms of **n** -step returns and the λ -return is due to Watkins (1989), who also first discussed the error-correction property of **n** -step returns. Our presentation is based on the slightly modified treatment by Jaakkola, Jordan, and Singh (1994). The results in the random-walk examples were made for this text based on the prior work by Sutton (1988) and Singh and Sutton (1996). The use of backup diagrams to describe these and other algorithms in this chapter is new, as are the terms ``forward view'' and ``backward view.''

TD(λ) was proved to converge in the mean by Dayan (1992), and with probability one by many researchers, including Peng (1993), Dayan and Sejnowski (1994), and Tsitsiklis (1994). Jaakkola, Jordan and Singh (1994) in addition first proved convergence of TD(λ) under online updating. Gurvits, Lin, and Hanson (1994) proved convergence of a more general class of eligibility-trace methods.

7.3

The idea that stimuli produce aftereffects in the nervous system that are important for learning is very old. Animal learning psychologists at least as far back as Pavlov (1927) and Hull (1943, 1952) included such ideas in their theories. However, stimulus traces in these theories are more like transient state representations than what we are calling eligibility traces: they could be associated with actions, whereas an eligibility trace is used only for credit assignment. The idea of a stimulus trace serving exclusively for credit assignment is apparently due to Klopf (1972), who proposed that a neuron's synapses become ``eligible'' under certain conditions for modification if, and when, reinforcement arrives later at the neuron. Our subsequent use of eligibility traces was based on Klopf's work (Sutton, 1978; Barto and Sutton, 1981; Sutton and Barto, 1981; Barto, Sutton, and

Anderson, 1983; Sutton, 1984). The TD(λ) algorithm itself is due to Sutton (1988).

7.4

The equivalence of forward and backwards views, and the relationships to Monte Carlo methods, were developed by Sutton (1988) for undiscounted episodic tasks, then extended by Watkins to the general case. The idea in exercise 7.5 is new.

7.5

Sarsa(λ) was first explored as a control method by Rummery and Niranjan (1994) and Rummery (1995).

7.6

Watkins's Q(λ) is due to Watkins (1989). Peng's Q(λ) is due to Peng and Williams (Peng, 1993; Peng and Williams, 1994, 1996). Rummery (1995) made extensive comparative studies of these algorithms.

Convergence has not been proved for any control method for $0 < \lambda < 1$.

7.7

Actor-critic methods were among the first to use eligibility traces (Barto, Sutton and Anderson, 1983; Sutton, 1984). The specific algorithm discussed in this chapter has never been tried before.

7.8

Replacing traces are due to Singh and Sutton (1996). The results in Figure 7.17 are from their paper. The task in the example of this section was first used to show the weakness of accumulating traces by Sutton (1984). The relationship of both kinds of traces to specific Monte Carlo methods were developed by Singh and Sutton (1996).

7.9 and 7 .10

The ideas in these two sections were generally known for many years, but, beyond what is in the sources cited in the sections themselves, this text may be the first place they have been described. Perhaps the first published discussion of variable λ was by Watkins (1989), who pointed out that the cutting off of the backup sequence (Figure 7.13) in his Q(

λ) when a non-greedy action was selected could be implemented by temporarily setting λ to 0.

Richard Sutton
Fri May 30 15:01:47 EDT 1997

8 Generalization and Function Approximation

We have so far assumed that our estimates of value functions are represented as a table with one entry for each state or for each state-action pair. This is a particularly clear and instructive case, but of course it is limited to tasks with small numbers of states and actions. The problem is not just the memory needed for large tables, but the time and data needed to accurately fill them. In other words, the key issue is that of *generalization*. How can experience with a limited subset of the state space be usefully generalized to produce a good approximation over a much larger subset?

This is a severe problem. In many tasks to which we would like to apply reinforcement learning, most states encountered will never have been experienced exactly before. This will almost always be the case when the state or action spaces include continuous variables or large number of sensors, such as a visual image. The only way to learn anything at all on these tasks is to generalize from previously experienced states to ones that have never been seen.

Fortunately, generalization from examples has already been extensively studied, and we do not need to invent totally new methods for use in reinforcement learning. To a large extent we need only combine reinforcement learning methods with existing generalization methods. The kind of generalization we require is often called *function approximation* because it takes examples from a desired function (e.g., a value function) and attempts to generalize from them to construct an approximation of the entire function. Function approximation is an instance of *supervised learning*, the primary topic studied in machine learning, artificial neural networks, pattern recognition, and statistical curve fitting. In principle, any of the methods studied in these fields can be used in reinforcement learning as we describe in this chapter.

Next Up Previous
Next: 8.1 Value Prediction with **Up:** Contents **Previous:** 7 Eligibility Traces

Richard Sutton
Fri May 30 15:50:45 EDT 1997

8.1 Value Prediction with Function Approximation

As usual, we begin with the prediction problem, that of estimating the state-value function V^π from experience generated using policy π. The novelty in this chapter is that the approximate value function at time **t**, V_t, is represented not as a table, but as a parameterized functional form with parameter vector $\vec{\theta}_t$. This means that the value function V_t depends totally on $\vec{\theta}_t$, varying from time step to time step only as $\vec{\theta}_t$ varies. For example, V_t might be the function computed by an artificial neural network, with $\vec{\theta}_t$ the vector of connection weights. By adjusting the weights, any of a wide range of different functions V_t can be implemented by the network. Or V_t might be the function computed by a decision tree, where $\vec{\theta}_t$ is all the parameters defining the split points and leaf values of the tree. Typically, the number of parameters (the number of components of $\vec{\theta}_t$) is much less than the number of states, and changing one parameter changes the estimated value of many states. Consequently, when a single state is backed up, the change generalizes from that state to affect the values of many other states.

All of the prediction methods covered in this book have been described as backups, that is, as updates to an estimated value function that shift its value at particular states toward a ``backed-up value" for that state. Let us refer to an individual backup by the notation $s \longrightarrow v$, where **s** is the state backed up and **v** is the backed-up value that **s**'s estimated value is shifted toward. For example, the DP backup for value prediction is $s \longrightarrow E_\pi\{r_{t+1} + \gamma V_t(s_{t+1}) | s_t = s\}$, the Monte Carlo backup is $s_t \longrightarrow R_t$, the

TD(0) backup is $s_t \longrightarrow r_{t+1} + \gamma V_t(s_{t+1})$, and the general TD($\lambda$) backup is $s_t \longrightarrow R_t^\lambda$. In the DP case, an arbitrary state **s** is backed up, whereas in the the other cases the state, s_t, encountered in (possibly simulated) experience is backed up.

It is natural to interpret each backup as specifying an example of the desired input-output behavior of the estimated value function. In a sense, the backup $s \longrightarrow v$ means that the estimated value for state **s** should be more like **v**. Up to now, the actual update implementing the backup has been trivial: the table entry for **s**'s estimated value has simply been shifted a fraction of the way towards **v**. Now we permit arbitrarily complex and sophisticated function approximation methods to implement the backup. The normal inputs to these methods are examples of the desired input-output behavior of the function they are trying to approximate. We use these methods for value prediction simply by passing to them the $s \longrightarrow v$ of each backup as a training example. The approximate function they produce we then interpret as an estimated value function.

Viewing each backup as a conventional training example in this way enables us to use any of a wide range of existing function approximation methods for value prediction. In principle, we can use any method for supervised learning from examples, including artificial neural networks, decision trees, and various kinds of multivariate regression. However, not all function-approximation methods are equally well suited for use in reinforcement learning. The most sophisticated neural-network and statistical methods all assume a static training set over which multiple passes are made. In reinforcement learning, however, it is important that learning be able to occur online, while interacting with the environment or with a model of the environment. To do this requires methods that are able to learn efficiently from incrementally-acquired data. In addition, reinforcement learning generally requires function approximators able to handle nonstationary target functions (functions that change over time). For example, in GPI (Generalized Policy Iteration) control methods we often seek to learn Q^π while π changes. Even if the policy remains the same, the target values of training examples are nonstationary if they are generated by bootstrapping methods (DP and TD). Methods that cannot easily handle such nonstationarity are less suitable for reinforcement learning.

What performance measures are appropriate for evaluating function-approximation methods? Most supervised-learning methods seek to minimize the mean squared error over some distribution, **P**, of the inputs. In our value-prediction problem, the inputs are states and the target function is the true value function V^π, so the mean squared error (MSE) for an approximation V_t, using parameter $\vec{\theta}_t$, is

$$MSE(\vec{\theta}_t) = \sum_{s \in S} P(s) \left(V^{\pi}(s) - V_t(s) \right)^2, \qquad (8.1)$$

where **P** is a distribution weighting the errors of different states. This distribution is important because it is usually not possible to reduce the error to zero at all states. After all, there are generally far more states than there are components to $\vec{\theta}_t$. The flexibility of the function approximator is thus a scarce resource. Better approximation at some states can be gained generally only at the expense of worse approximation at other states. The distribution, **P**, specifies how these tradeoffs should be made.

The distribution **P** is also usually the distribution from which the states in the training examples are drawn, and thus the distribution of states at which backups are done. If we wish to minimize error over a certain distribution of states, then it makes sense to train the function approximator with examples from that same distribution. For example, if you want a uniform level of error over the entire state set, then it makes sense to train with backups distributed uniformly over the entire state set, such as in the exhaustive sweeps of some DP methods. Henceforth, let us assume that the distribution of states at which backups are done and the distribution that weights errors, **P**, are the same.

A distribution of particular interest is the one describing the frequency with which states are encountered while the agent is interacting with the environment and selecting actions according to π, the policy whose value function we are approximating. We call this the *on-policy distribution*, in part because it is the distribution of backups in on-policy control methods. Minimizing error over the on-policy distribution focuses function-approximation resources on the states that actually occur while following the policy, ignoring those that never occur. The on-policy distribution is also the one for which it is easiest to get training examples using Monte Carlo or TD methods. These methods generate backups from sample experience using the policy π. Because a backup is generated for each state encountered in the experience, the training examples available are naturally distributed according to the on-policy distribution. Stronger convergence results are available for the on-policy distribution than for other distributions, as we discuss later.

It is not completely clear that we should care about minimizing the MSE. Our goal in value prediction is potentially different because our ultimate purpose is to use the predictions to aid in finding a better policy. The best predictions for that purpose are not necessarily the best for minimizing MSE. However, it is not yet clear what a more useful alternative goal for value prediction might be. For now, we continue to focus on MSE.

An ideal goal in terms of MSE would be to find a *global optimum*, a parameter vector $\vec{\theta}^*$

for which $MSE(\vec{\theta}^*) \leq MSE(\vec{\theta})$ for all possible $\vec{\theta}$. Reaching this goal is sometimes possible for simple function approximators such as linear ones, but is rarely possible for complex function approximators such as artificial neural networks and decision trees. Short of this, complex function approximators may seek to converge instead to a *local optimum*, a parameter $\vec{\theta}^*$ for which $MSE(\vec{\theta}^*) \leq MSE(\vec{\theta})$ for all $\vec{\theta}$ in some neighborhood of $\vec{\theta}^*$. Although this guarantee is only slightly reassuring, it is typically the best that can be said for nonlinear function approximators. For many cases of interest in reinforcement learning, convergence to an optimum, or even true convergence, does not occur. Nevertheless, a MSE that is within a small bound of an optimum may still be achieved with some methods. Others may in fact diverge, with their MSE approaching infinity in the limit.

In this section we have outlined a framework for combining a wide range of reinforcement learning methods for value prediction with a wide range of function-approximation methods, using the backups of the former to generate training examples for the latter. We have also outlined a range of MSE performance measures to which these methods may aspire. The range of possible methods is far too large to cover all, and anyway there is too little known about most of them to make a reliable evaluation or recommendation. Of necessity, we consider only a few possibilities. In the rest of this chapter we focus on function-approximation methods based on gradient principles, and on linear gradient-descent methods in particular. We focus of these methods in part because we consider them to be particularly promising and because they reveal key theoretical issues, but also just because they are simple and our space is limited. If we had another chapter devoted to function approximation, we would also cover at least memory-based methods and decision-tree methods.

Richard Sutton
Sat May 31 15:08:20 EDT 1997

8.2 Gradient-Descent Methods

We now develop in detail one class of learning methods for function approximation in value prediction, those based on gradient descent. Gradient-descent methods are among the most widely used of all function approximation methods and are particularly well-suited to reinforcement learning.

In gradient-descent methods, the parameter vector is a column vector with a fixed number of real valued components, $\vec{\theta}_t = (\theta_t(1), \theta_t(2), \ldots, \theta_t(n))^T$ (the `T' here denotes transpose), and $V_t(s)$ is a smooth differentiable function of $\vec{\theta}_t$ for all $s \in \mathcal{S}$. For now, let us assume that on each step, **t**, we observe a new example $s_t \rightarrow V^\pi(s_t)$. These states might be successive states from an interaction with the environment, but for now we do *not* assume so. Even though we are given the exact, correct values, $V^\pi(s_t)$ for each s_t, there is still a difficult problem because our function approximator has limited resources and thus limited resolution. In particular, there is generally no $\vec{\theta}$ that gets all the states, or even all the examples, exactly correct. In addition, we must generalize to all the other states that have not appeared in examples.

We assume that states appear in examples with the same distribution, **P**, over which we are trying to minimize the MSE as given by (8.1). A good strategy in this case is to try to minimize error on the observed examples. Gradient-descent methods do this by adjusting the parameter vector after each example by a small amount in the direction that would most reduce the error on that example:

$$\begin{aligned} \vec{\theta}_{t+1} &= \vec{\theta}_t - \frac{1}{2}\alpha\nabla_{\vec{\theta}_t}\Big(V^\pi(s_t) - V_t(s_t)\Big)^2 \\ &= \vec{\theta}_t + \alpha\Big(V^\pi(s_t) - V_t(s_t)\Big)\nabla_{\vec{\theta}_t}V_t(s_t), \end{aligned} \qquad (8.2)$$

where α is a positive step-size parameter, and $\nabla_{\vec{\theta}_t} f(\vec{\theta}_t)$, for any function, **f**, denotes the vector of partial derivatives, $(\frac{\partial f(\vec{\theta}_t)}{\partial \theta_t(1)}, \frac{\partial f(\vec{\theta}_t)}{\partial \theta_t(2)}, \ldots, \frac{\partial f(\vec{\theta}_t)}{\partial \theta_t(2)})^T$. This derivative vector is the *gradient* of **f** with respect to $\vec{\theta}_t$. This kind of method is called *gradient descent* because the overall step in $\vec{\theta}_t$ is proportional to the negative gradient of the example's squared error. This is the direction in which the error falls most rapidly.

It may not be immediately apparent why only a small step is taken in the direction of the gradient. Could we not move all the way in this direction and completely eliminate the error on the example? In many cases this could be done, but usually it is not desirable. Remember that we do not seek or expect to find a value function that has zero error on all states, but only an approximation that balances the errors in different states. If we completely corrected each example in one step, then we would not find such a balance. In fact, the convergence results for gradient methods assume that the step-size parameter decreases over time. If it decreases in such a way as to satisfy the standard stochastic-approximation conditions (2 .7), then the gradient-descent method (8.2) is guaranteed to converge to a local optimum.

We turn now to the case in which the output, v_t, of the **t**th training example, $s_t \longrightarrow v_t$, is not the true value, $V^\pi(s_t)$, but some approximation of it. For example, v_t might be a noise-corrupted version of $V^\pi(s_t)$, or it might be one of the backed-up values mentioned in the previous section. In such cases we cannot perform the exact update (8.2) because $V^\pi(s_t)$ is unknown, but we can approximate it by substituting v_t in place of $V^\pi(s_t)$. This yields the general gradient-descent method for state-value prediction:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha\Big(v_t - V_t(s_t)\Big)\nabla_{\vec{\theta}_t} V_t(s_t). \qquad (8.3)$$

If v_t is an *unbiased* estimate, i.e., if $E\{v_t\} = V^\pi(s_t)$, for each **t**, then $\vec{\theta}_t$ is guaranteed to converge to a local optimum under the usual stochastic approximation conditions (2 .7) for decreasing the step-size parameter, α .

For example, suppose the states in the examples are the states generated by interaction (or simulated interaction) with the environment using policy π. Let R_t denote the return following each state, s_t. Because the true value of a state is the expected value of the return following it, the Monte Carlo target $v_t = R_t$ is by definition an unbiased estimate of $V^\pi(s_t)$. With this choice, the general gradient-descent method (8.3) converges to a locally-optimal approximation to $V^\pi(s_t)$. Thus, the gradient-descent version of Monte Carlo state-value prediction is guaranteed to find a locally-optimal solution.

Similarly, we can use **n** -step TD returns and their averages for v_t. For example, the gradient-descent form of TD(λ) uses the λ -return, $v_t = R_t^\lambda$, as its approximation to $V^\pi(s_t)$, yielding the forward-view update:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha \left(R_t^\lambda - V_t(s_t) \right) \nabla_{\vec{\theta}_t} V_t(s_t). \tag{8.4}$$

Unfortunately, for $\lambda < 1$, R_t^λ is not an unbiased estimate of $V^\pi(s_t)$, and thus this method does not converge to a local optimum. The situation is the same when DP targets are used such as $v_t = E_\pi\{r_{t+1} + \gamma V_t(s_{t+1}) \mid s_t\}$. Nevertheless, such bootstrapping methods can be quite effective, and other performance guarantees are available for important special cases, as we discuss later in this chapter. For now we emphasize the relationship of these methods to the general gradient-descent form (8.3). Although, increments as in (8.4) are not themselves gradients, it is useful to view this method as a gradient-descent method (8.3) with a bootstrapping approximation in place of the desired output, $V^\pi(s_t)$.

Just as (8.4) provides the forward view of gradient-descent TD(λ) , the backwards view is provided by

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha\, \delta_t\, \vec{e}_t, \tag{8.5}$$

where δ_t is the usual TD error,

$$\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t), \tag{8.6}$$

and $\vec{e}_t$ is a column vector of eligibility traces, one for each component of $\vec{\theta}_t$, updated by

$$\vec{e}_t = \gamma\lambda\vec{e}_{t-1} + \nabla_{\vec{\theta}_t} V_t(s_t), \tag{8.7}$$

with $\vec{e}_0 = \vec{0}$. A complete algorithm for online gradient-descent TD(λ) is given in Figure 8.1.

```
Initialize $\vec{\theta}$ arbitrarily and $\vec{e} = 0$
Repeat (for each episode):
   $s \leftarrow$ initial state of episode
   Repeat (for each step of episode):
      $a \leftarrow$ action given by $\pi$ for $s$
      Take action a, observe reward, r, and next state, $s'$
      $\delta \leftarrow r + \gamma V(s') - V(s)$
      $\vec{e} \leftarrow \gamma\lambda\vec{e} + \nabla_{\vec{\theta}} V(s)$
      $\vec{\theta} \leftarrow \vec{\theta} + \alpha\,\delta\,\vec{e}$
      $s \leftarrow s'$
   until s is terminal
```

Figure 8.1: Online gradient-descent TD(λ) for estimating V^π. The approximate value function, **V**, is implicitly a function of $\vec{\theta}$.

Two forms of gradient-based function approximators have been used widely in reinforcement learning. One is multi-layer artificial neural networks using the error backpropagation algorithm. This maps immediately onto the equations and algorithms just given, where the backpropagation process is the way of computing the gradients. The second popular form is the linear form, which we discuss extensively in the next section.

Exercise 8.1

Show that table-lookup TD(λ) is a special case of general TD(λ) as given by equations (8.5--8.7).

Exercise 8.2

State aggregation is a simple form of generalizing function approximator in which states are grouped together, with one table entry (value estimate) used for each group. Whenever a state in a group is encountered, the group's entry is used to determine the state's value, and when the state is updated, the group's entry is updated. Show that this kind of state aggregation is a special case of a gradient method such as (8.4).

Exercise 8.3

The equations given in this section are for the online version of gradient-descent TD(λ) . What are the equations for the *offline* version? Give a complete description specifying the new approximate value function at the end of an episode, V', in terms of the approximate value function used during the episode, **V**. Start by modifying a forward-view equation for TD(λ) , such as (8.4).

Exercise 8.4

For offline updating, show that equations (8.5--8.7) produce updates identical to (8.4).

Richard Sutton
Sat May 31 15:08:20 EDT 1997

8.3 Linear Methods

One of the most important special cases in gradient-descent function approximation is that in which the approximate function, V_t, is a linear function of the parameter vector, $\vec{\theta}_t$. Corresponding to every state, **s**, there is a column vector of features, $\vec{\phi}_s = (\phi_s(1), \phi_s(2), \ldots, \phi_s(n))^T$, with the same number of components as $\vec{\theta}_t$. The features may be constructed from the states in many different ways; we cover a few possibilities below. However the features are constructed, the approximate state-value function is given by

$$V_t(s) = \vec{\theta}_t^T \vec{\phi}_s = \sum_{i=1}^{n} \theta_t(i)\phi_s(i). \qquad (8.8)$$

In this case the approximate value function is said to be *linear in the parameters*, or simply *linear*.

It is natural to use gradient-descent updates with linear function approximators. The gradient of the approximate value function with respect to $\vec{\theta}_t$ in this case is just

$$\nabla_{\vec{\theta}_t} V_t(s) = \vec{\phi}_s.$$

Thus, the general gradient-descent update (8.3) reduces to a particularly simple form in the linear case. In addition, in the linear case there is only one optimum $\vec{\theta}^*$ (or, in degenerate cases, one set of equally good optima). Thus, any method guaranteed to converges to or near a local optimum is automatically guaranteed to converge to or near the global optimum. Because they are simple in these ways, linear gradient-descent function approximators are the most favorable for mathematical analysis. Almost all useful convergence results for learning systems of all kinds are for linear (or simpler) function approximators.

In particular, the gradient-descent TD(λ) algorithm discussed in the previous section (Figure 8.1) has been proven to converge in the linear case if the step-size parameter is reduced over time according to the usual conditions (2 .7). Convergence is not to the minimum-error parameter vector, $\vec{\theta}^*$, but to a nearby parameter vector, $\vec{\theta}_\infty$, whose error is bounded according to:

$$MSE(\vec{\theta}_\infty) \leq \frac{1-\gamma\lambda}{1-\gamma} MSE(\vec{\theta}^*). \qquad (8.9)$$

That is, the asymptotic error is no more than $\frac{1-\gamma\lambda}{1-\gamma}$ times the smallest possible error. As λ approaches 1, the bound approaches the minimum error. An analogous bound applies to other on-policy bootstrapping methods. For example, linear gradient-descent DP backups, using (8.3), with the on-policy distribution, will converge to the same result as TD(0). Technically, this bound applies only to discounted continual tasks, but a related result presumably holds for episodic tasks. There are also a few technical conditions on the rewards, features, and decrease in the step-size parameter, which we are omitting here. The full details can be found in the original paper (Tsitsiklis and Van Roy, 1997).

Critical to the above result is that states are backed up according to the on-policy distribution. For other backup distributions, bootstrapping methods using function approximation may actually diverge to infinity. Examples of this and a discussion of possible solution methods are given in Section 8.5

Beyond these theoretical results, linear learning methods are also of interest because in practice they can be very efficient both in terms of data and computation. Whether or not this is so depends critically on how the states are represented in terms of the features. Choosing features appropriate to the task is an important way of adding prior domain knowledge to reinforcement learning systems. Intuitively, the features should correspond to the natural features of the task, those along which generalization is most appropriate. If we are valuing geometric objects, for example, we might want to have features for each possible shape, color, size, or function. If we are valuing states of a mobile robot, then we might want to have features for locations, degrees of remaining battery power, recent sonar readings, etc.

In general, we also need features for combinations of these natural qualities. This is because the linear form prohibits the representation of interactions between features, such as the presence of feature **i** being good only in the absence of feature **j**. For example, in the pole-balancing task (Example 3.4), a high angular velocity may be either good or bad depending on the angular position. If the angle is high, then high angular velocity means an imminent danger of falling, a bad state, whereas if the angle is low, then high angular velocity means the pole is righting itself, a good state. In cases with such interactions one needs to introduce features for particular conjunctions of feature values when using a linear function approximator. We next consider some general ways of doing this.

Exercise 8.5

How could we reproduce the tabular case within the linear framework?

Exercise 8.6

How could we reproduce the state aggregation case (see Exercise 8.4) within the linear framework?

Richard Sutton
Sat May 31 15:08:20 EDT 1997

8.3.1 Coarse Coding

Consider a task in which the state set is continuous and two-dimensional. A state in this case consists of a point in two-space, say a vector with two real components. One kind of feature for this case are those corresponding to *circles* in state space, as shown in Figure 8.2. If the state is inside a circle, then the corresponding feature has the value **1** and is said to be *present*; otherwise the feature is **0** and is said to be *absent*. This kind of 1-0 valued feature is called a *binary feature*. Given a state, which binary features are present indicate within which circles the state lies, and thus coarsely code for its location. Representing a state with features that overlap in this way (although they need not be circles or binary) is known as *coarse coding*.

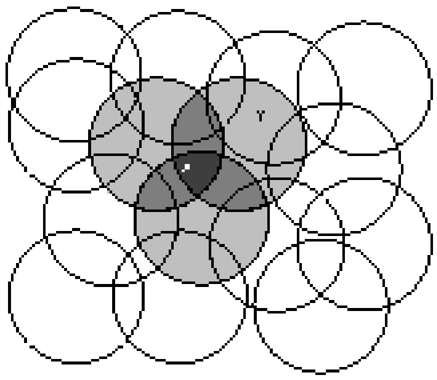


Figure 8.2: Coarse coding. Generalization from state **X** to state **Y** depends on the number of their features whose receptive fields (in this case, circles) overlap. These states have one feature in common, so there will be slight generalization between them.

Assuming a linear gradient-descent function approximator, consider the effect of the size and density of the circles. Corresponding to each circle is a single parameter (a component of $\vec{\theta}_t$) that is affected by learning. If we train at one point (state), **X**, then the parameters of all circles intersecting **X** will be affected. Thus, by (8.8), the approximate value function will be affected at all points within the union of the circles, with a greater effect the more circles a point has ``in common'' with **X**, as shown in Figure 8.2. If the circles are small, then the generalization will be over a short distance, as in Figure 8.3a, whereas if they are large, it will be over a large distance, as in Figure 8.3b. Moreover, the shape of the features will determine the nature of the generalization. For example, if they are not strictly circular, but are elongated in one direction, then generalization will be similarly affected, as in Figure 8.3c.

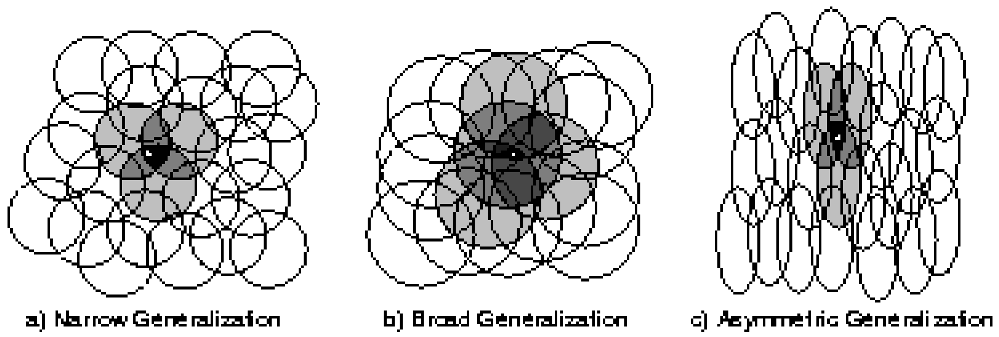


Figure 8.3: Generalization in linear function approximators is determined by the sizes and shapes of the features' receptive fields. All three of these cases have roughly the same number and density of features.

Features with large receptive fields give broad generalization, but might also seem to limit the learned function to a coarse approximation, unable to make discriminations much finer than the width of the receptive fields. Happily, this is not the case. Initial generalization from one point to another is indeed controlled by the size and shape of the receptive fields, but acuity, the finest discrimination ultimately possible, is controlled more by the total number of features.

Example 8.1 *The Coarseness of Coarse Coding*. This example illustrates the effect on learning of the size of the receptive fields in coarse coding. A linear function approximator based on coarse coding and (8.3) was used to learn a one-dimensional square-wave function (shown at the top of Figure 8.4). The values of this function were used as the targets, v_t. With just one dimension, the receptive fields were intervals rather than circles. Learning was repeated with three different sizes of the intervals: narrow, medium, and broad, as shown at the bottom of the figure. All three cases had the same density of features, about 50 over the extent of the function being learned. Training examples were generated uniformly at random over this extent. The step-size parameter was $\alpha = \frac{0.2}{m}$, where **m** is the number of features that were present at one time. Figure 8.4 shows the functions learned in all three cases over the course of learning. Note that the width of the features had a strong effect early in learning. With broad features, the generalization tended to be broad; with narrow features, only the close neighbors of each trained point were changed, causing the function learned to be more bumpy. However, the final function learned was affected only slightly by the width of the features. Receptive field shape tends to have a strong effect on generalization, but little effect on asymptotic solution quality. $\diamond$

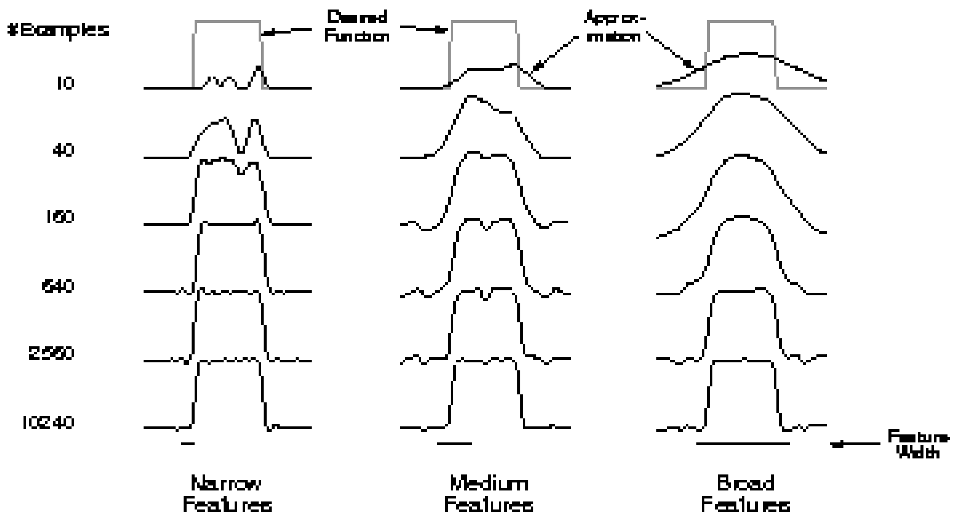


Figure 8.4: Example of feature width's strong effect on initial generalization (first row) and weak effect on asymptotic accuracy (last row).

Richard Sutton
Sat May 31 15:08:20 EDT 1997

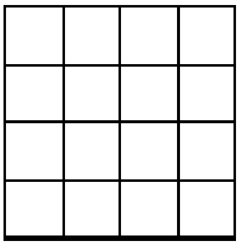
8.3.2 Tile Coding

Tile coding is a form of coarse coding that is particularly well suited for use on sequential digital computers and for efficient online learning. In tile coding the receptive fields of the features are grouped into exhaustive partitions of input space. Each such partition is called a *tiling*, and each element of the partition is called a *tile*. Each tile is a the receptive field for one binary feature.

An immediate advantage of tile coding is that the overall number of features that are present at one time is strictly controlled and independent of the input state. Exactly one feature is present in each tiling, thus the total number of features present is always the same as the number of tilings. Together with the level of response being set at **1**, this allows the step-size parameter, α , to be set in an easy, intuitive way. For example, choosing $\alpha = \frac{1}{m}$, where **m** is the number of tilings, results in exact one-trial learning. If the example $s_t \rightarrow v_t$ is received, then whatever the prior value, $V_t(s_t)$, the new value will be $V_{t+1}(s_t) = v_t$. Usually one wishes to change more slowly than this, to allow for generalization and stochastic variation in target outputs. For example, one might choose $\alpha = \frac{1}{10m}$, in which case one would move one-tenth of the way to the target in one update.

Because tile coding uses exclusively binary (0-1 valued) features, the weighted sum making up the approximate value function (8.8) is almost trivial to compute. Rather than performing **n** multiplications and additions, one simply computes the indices of the **m<<n** present features and then adds up the **m** corresponding components of the parameter vector. The eligibility-trace computation (8.7) is also simplified because the components of the gradient, $\nabla_{\vec{\theta}} V_t(s_t)$, are also usually **0**, and otherwise **1**.

The computation of the indices of the present features is particularly easy if grid-like tilings are used. The ideas and techniques here are best illustrated by examples. Suppose we address a task with two continuous state variables. Then the simplest way to tile the space is with a uniform two-dimensional grid:



Given the **x** and **y** coordinates of a point in the space, it is computationally easy to determine the index of the tile it is in. When multiple tilings are used. each is offset by a different amount, so that each cuts the space in a different way. In the example shown in Figure 8.5, an extra row and column of tiles have been added to the grid so that no points are left uncovered. The two tiles highlighted are those that present in the state indicated by the ``X." The different tilings may be offset by random amounts, or by cleverly designed deterministic strategies (simply offsetting each dimension by the same increment is known not to be a good idea). The effects on generalization and asymptotic accuracy illustrated in Figures 8.3 and 8.4 apply here as well. The width and shape of the tiles should be chosen to match the width of generalization that one expects to be appropriate. The number of tilings should be chosen to influence the density of tiles. The denser the tiling, the finer and more accurately the desired function can be approximated, but the greater the computational costs.

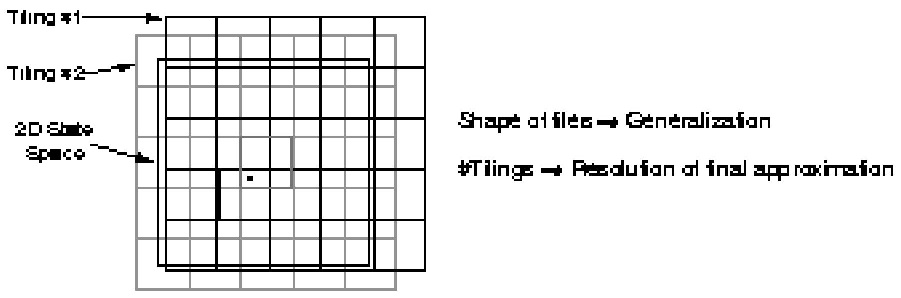

Tiling #1
Tiling #2
2D State Space
Shape of tiles → Generalization
#Tilings → Resolution of final approximation

Figure 8.5: Multiple, overlapping grid-tilings.

It is important to note that the tilings can be arbitrary and need not be uniform grids. Not only can the tiles be strangely shaped, as in Figure 8.6a, but they can be shaped and distributed to give particular kinds of generalization. For example, the stripe tiling in Figure 8.6b will promote generalization along the vertical dimension and discrimination along the horizontal dimension, particularly on the left. The diagonal stripe tiling in Figure 8.6c will promote generalization along one diagonal. In higher dimensions, axis-aligned stripes correspond to ignoring some of the dimensions in some of the tilings, i.e., to hyperplanar slices.

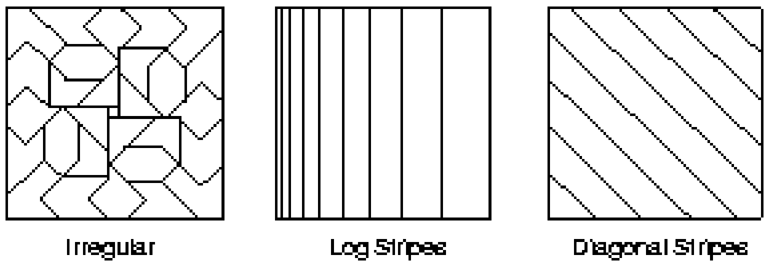
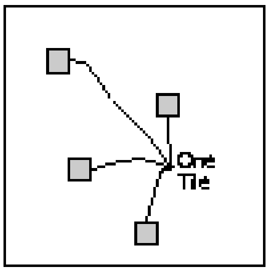


Figure 8.6: A variety of tilings.

Another important trick for reducing memory requirements is *hashing*---a consistent pseudo-random collapsing of a large tiling into a much smaller set of tiles. Hashing produces tiles consisting of non-contiguous, disjoint regions randomly spread throughout the state space, but which still form an exhaustive tiling. For example, one tile might consist of the four sub-tiles shown below:



Through hashing, memory requirements are often reduced by large factors with little loss of performance. This is possible because high resolution is needed in only a small fraction of the state space. Hashing frees us from the curse of dimensionality in the sense that memory requirements need not be exponential in the number of dimensions, but need merely match the real demands of the task. Good public-domain implementations of tile coding, including hashing, are widely available.

Exercise 8.7

Suppose we believe that one of two state dimensions is more likely to have an effect on the value function than the other, that generalization should be primarily across this dimension rather than along it. What kind of tilings could be used to take advantage of this prior knowledge?

Richard Sutton
Sat May 31 15:08:20 EDT 1997

8.3.3 Radial Basis Functions

Radial basis functions (RBFs) are the natural generalization of coarse coding to continuous-valued features. Rather than each feature being either 0 or 1, it can be anything in the interval $[0, 1]$, reflecting various *degrees* to which the feature is present. A typical RBF feature, **i**, has a gaussian (bell-shaped) response, $\phi_s(i)$, dependent only on the distance between the state, **s**, and the feature's prototypical or center state, c_i, and relative to the feature's width, σ_i:

$$\phi_s(i) = \exp\left(-\frac{||s - c_i||^2}{2\sigma_i^2}\right).$$

The norm or distance metric of course can be chosen in whatever way seems most appropriate to the states and task at hand. Figure 8.7 shows a 1-dimensional example with a euclidean distance metric.

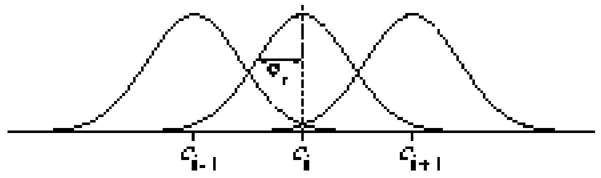


Figure 8.7: One-dimensional radial basis functions.

An *RBF network* is simply a linear function approximator using RBFs for its features. Learning is defined by equations, (8.3) and (8.8) in exactly the same way it is for other linear function approximators. The primary advantage of RBFs over binary features is that they produce approximate functions that vary smoothly and are differentiable. In addition, some learning methods for RBF networks change the centers and widths of the features as well. Such nonlinear methods may be able to fit the target function much more precisely. The downside to RBF networks, and to nonlinear RBF networks especially, is greater computational complexity and, often, more manual tuning before learning is robust and efficient.

Richard Sutton
Sat May 31 15:08:20 EDT 1997

8.3.4 Kanerva Coding

On tasks with very high dimensionality, say *hundreds* of dimensions, tile coding and RBF networks become impractical. If we take either method at face value, its computational complexity increases exponentially with the number of dimensions. There are a number of tricks that can reduce this growth (such as hashing), but even these become impractical after a few tens of dimensions.

On the other hand, some of the general ideas underlying these methods can be practical for high-dimensional tasks. In particular, the idea of representing situations by a list of the features present and then mapping those features linearly to an approximation may scale well to large tasks. The key is to keep the number of features from scaling explosively. Is there any reason to think this might be possible?

First we need to establish some realistic expectations. Roughly speaking, a function approximator of a given complexity can only accurately approximate target functions of comparable complexity. But as dimensionality increases, the size of the state space inherently increases exponentially. It is reasonable to assume that in the worst case the complexity of the target function scales like that of the state space. Thus, if we focus the worst case, then there is no solution, no way to get good approximations for high-dimensional tasks without using resources exponential in the dimension.

A more useful way to think about the problem is to focus on the complexity of the target function as separate and distinct from the size and dimensionality of the state space. The size of the state space may give an upper bound on complexity, but short of that high bound, complexity and dimension can be unrelated. For example, one might have a 1000-dimensional task where only one of the dimensions happens to matter. Given a certain level of complexity, we then seek to be able to accurately approximate any target function of that complexity or less. As the target level of complexity increases, we would like to get by with a proportionate increase in computational resources.

From this point of view, the real source of the problem is the complexity of the target function or of a reasonable approximation of it, not the dimensionality of the state space.

Thus, adding dimensions, such as new sensors or new features, to a task should be almost without consequence if the complexity of the needed approximations remain the same. The new dimensions may even make things easier if the target function can be simply expressed in terms of them. Methods like tile and RBF coding, however, do not work this way. Their complexity increases exponentially with dimensionality even if the complexity of the target function does not. For these methods, dimensionality itself is still a problem. We need methods whose complexity is unaffected by dimensionality per se, methods that are limited only by, and scale well with, the complexity of what they approximate.

One simple approach that meets these criteria, which we call *Kanerva coding*, is to choose binary features that correspond to particular *prototype states*. For definiteness, let us say that the prototypes are randomly selected from the entire state space. The receptive field of such a feature is all states sufficiently close to the prototype. Kanerva coding uses a different kind of distance metrics than in tile coding and RBFs. For definiteness, consider a *binary* state space and the *hamming distance*, the number of bits at which two states differ. States are considered similar if they agree on enough dimensions, even if they are totally different on others.

The strength of Kanerva coding is that the complexity of the functions that can be learned depends entirely on the number of features, which bears no necessary relationship to the dimensionality of the task. The number of features can be more or less than the number of dimensions. Only in the worst case must it be exponential in the number of dimensions. Dimensionality itself is thus no longer a problem. Complex functions are still a problem, as they have to be. To handle more complex tasks, a Kanerva coding approach simply needs more features. There is not a great deal of experience with such systems, but what there is suggests the abilities of the networks increase proportionately with their computational resources. This is an area of current research, and significant improvements in existing methods can still easily be found.

Next Up Previous

Next: 8.4 Control with Function **Up:** 8.3 Linear Methods **Previous:** 8.3.3 Radial Basis Functions

---

Richard Sutton

Sat May 31 15:08:20 EDT 1997

8.4 Control with Function Approximation

We now extend value-prediction methods using function approximation to control methods, following the pattern of GPI. First we extend the the state-value prediction methods to action-value prediction methods, then we combine them with policy-improvement and action-selection techniques. As usual, the problem of ensuring exploration is solved by pursuing either an on-policy or an off-policy approach.

The extension to action-value prediction is straightforward. In this case it is the action-value function, $Q_t \approx Q^\pi$, that is represented as a parameterized functional form with parameter vector $\vec{\theta}_t$. Whereas before we considered training examples of the form $s_t \longrightarrow v_t$, now we consider examples of the form $s_t, a_t \longrightarrow v_t$. The example output, v_t, can be any approximation of $Q^\pi(s_t, a_t)$, including the usual backed-up values such as the full Monte-Carlo return, R_t, or the 1-step Sarsa-style return, $r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1})$. The general gradient-descent update for action-value prediction is

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha \Big(v_t - Q_t(s_t, a_t) \Big) \nabla_{\vec{\theta}_t} Q_t(s_t, a_t).$$

For example, the backward view of the action-value method analogous to TD(λ) is

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha\, \delta_t\, \vec{e}_t,$$

where

$$\delta_t = r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t),$$

and

$$\vec{e}_t = \gamma\lambda \vec{e}_{t-1} + \nabla_{\vec{\theta}_t} Q_t(s_t, a_t),$$

with $\vec{e}_0 = \vec{0}$. We call this method *gradient-descent Sarsa(λ)* , particularly when it is elaborated to form a full control method. For a constant policy, this method converges in the same way that TD(λ) does, with the same kind of error bound (8.9).

To form control methods, we need to couple such action-value prediction methods with techniques for policy improvement and action selection. Suitable techniques applicable to continuous actions, or to actions from large discrete sets, are a topic of ongoing research with as yet no clear resolution. On the other hand, if the action set is discrete and not too large, then we can use the techniques already developed in previous chapters. That is, for each possible action, **a**, available in the current state, s_t, we can compute $Q_t(s_t, a)$ and then find the greedy action $a_t^* = \arg\max_a Q_t(s_t, a)$. Policy improvement is done by changing the estimation policy to the greedy policy (in off-policy methods) or to a soft approximation of the greedy policy such as the ϵ -greedy policy (in on-policy methods). Actions are selected according to this same policy in on-policy methods, or by an arbitrary policy in off-policy methods.

```
Initialize $\vec{\theta}$ arbitrarily and $\vec{e} = \vec{0}$
Repeat (for each episode):
   $s \leftarrow$ initial state of episode
   For all $a \in \mathcal{A}(s)$:
      $\mathcal{F}_a \leftarrow$ set of features present in $s, a$
      $Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta(i)$
   $a \leftarrow \arg\max_a Q_a$
   With probability $\epsilon$: $a \leftarrow$ a random action $\in \mathcal{A}(s)$
   Repeat (for each step of episode):
      $\vec{e} \leftarrow \gamma\lambda\vec{e}$
      For all $\bar{a} \neq a$:              (optional block for replacing traces)
          For all $i \in \mathcal{F}_{\bar{a}}$:
              $e(i) \leftarrow 0$
      For all $i \in \mathcal{F}_a$:
          $e(i) \leftarrow e(i) + 1$                         (accumulating traces)
          or $e(i) \leftarrow 1$                            (replacing traces)
      Take action **a**, observe reward, **r**, and next state, $s'$
      $\delta \leftarrow r - Q_a$
      For all $a \in \mathcal{A}(s)$:
            $\mathcal{F}_a \leftarrow$ set of features present in $s', a$
            $Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta(i)$
        $a' \leftarrow \arg\max_a Q_a$
        With probability $\epsilon$: $a' \leftarrow$ a random action $\in \mathcal{A}(s)$
        $\delta \leftarrow \delta + \gamma Q_{a'}$
        $\vec{\theta} \leftarrow \vec{\theta} + \alpha \delta \vec{e}$
        $a \leftarrow a'$
    until $s'$ is terminal
```

Figure 8.8: Linear, gradient-descent Sarsa(λ) with binary features and ϵ -greedy policy. Updates for both accumulating and replacing traces are specified, including the option (when using replacing traces) of clearing the traces of non-selected actions.

```
Initialize $\vec{\theta}$ arbitrarily and $\vec{e} = \vec{0}$
Repeat (for each episode):
    $s \leftarrow$ initial state of episode
    For all $a \in \mathcal{A}(s)$:
        $\mathcal{F}_a \leftarrow$ set of features present in $s, a$
        $Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta(i)$
    Repeat (for each step of episode):
        With probability $1 - \epsilon$:
            $a \leftarrow \arg\max_a Q_a$
            $\vec{e} \leftarrow \gamma \lambda \vec{e}$
        else
            $a \leftarrow$ a random action $\in \mathcal{A}(s)$
            $\vec{e} \leftarrow 0$
        For all $i \in \mathcal{F}_a$: $e(i) \leftarrow e(i) + 1$
        Take action a, observe reward, r, and next state, $s'$
        $\delta \leftarrow r - Q_a$
        For all $a \in \mathcal{A}(s)$:
            $\mathcal{F}_a \leftarrow$ set of features present in $s', a$
        $Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta(i)$
    $a' \leftarrow \arg\max_a Q_a$
    $\delta \leftarrow \delta + \gamma Q_{a'}$
    $\vec{\theta} \leftarrow \vec{\theta} + \alpha \delta \vec{e}$
until $s'$ is terminal
```

Figure 8.9: A linear, gradient-descent version of Watkins's Q(λ) with binary features, ϵ -greedy policy, and accumulating traces.

Figures 8.8 and 8.9 show examples of on-policy (Sarsa(λ)) and off-policy (Watkins's Q(λ)) control methods using function approximation. Both methods use linear, gradient-descent function approximation with binary features, such as in tile coding and Kanerva coding. Both methods use an ϵ -greedy policy for action selection, and the Sarsa method uses it for GPI as well. Both compute the sets of present features, $\mathcal{F}_a$, corresponding to the current state and all possible actions, **a**. If the value function for each action is a separate linear function of the same features (a common case), then the indices of all the $\mathcal{F}_a$ may be the same except for an offset, simplifying the computation significantly.

All the methods we have discussed above have used *accumulating* eligibility traces. Although replacing traces (Section 7.8) are known to have advantages in tabular methods, they do not directly extend to the use of function approximation. Recall that the idea of replacing traces is to reset a state's trace to **1** each time it is visited instead of incrementing it by **1**. But with function approximation there is no single trace corresponding to a state, just a trace for each component of $\vec{\theta}_t$, each of which corresponds to many states. One approach that seems to work well for linear, gradient-descent function approximation with binary features is to treat the features as if they were states for the purposes of replacing traces. That is, each time a state is encountered that has feature **i**, the trace for feature **i** is set to **1** rather than being incremented by **1** as it would be with accumulating traces.

When working with state-action traces, it may also be useful to clear (set to zero) the traces of all non-selected actions in the states encountered (see Section 7.8). This idea can also be extended to the case of linear function approximation with binary features. For each state encountered, we first clear the traces of all features for the state and the actions not selected, then we set to **1** the traces of the features for the state and the action that was selected. As we noted for the tabular case, this may or may not be the best way to proceed when using replacing traces. A procedural specification of both kinds of traces, including the optional clearing for non-selected actions, is given for the Sarsa algorithm in Figure 8.8.

Example 8.2 *The Mountain-Car Task*. Consider the task of driving an underpowered car up a steep mountain road, as suggested by the diagram in the upper left of Figure 8.10. The difficulty is that gravity is stronger than the car's engine, and even at full throttle it cannot accelerate up the steep slope. The only solution is to first move away from the goal and up the opposite slope on the left.

Then, by applying full throttle the car can build up enough inertia to carry it up the steep slope even though it is slowing down the whole way. This is a simple example of a continuous control task where things have to get worse in a sense (farther from the goal) before they can get better. Many control methodologies have great difficulties with tasks of this kind unless explicitly aided by a human designer.

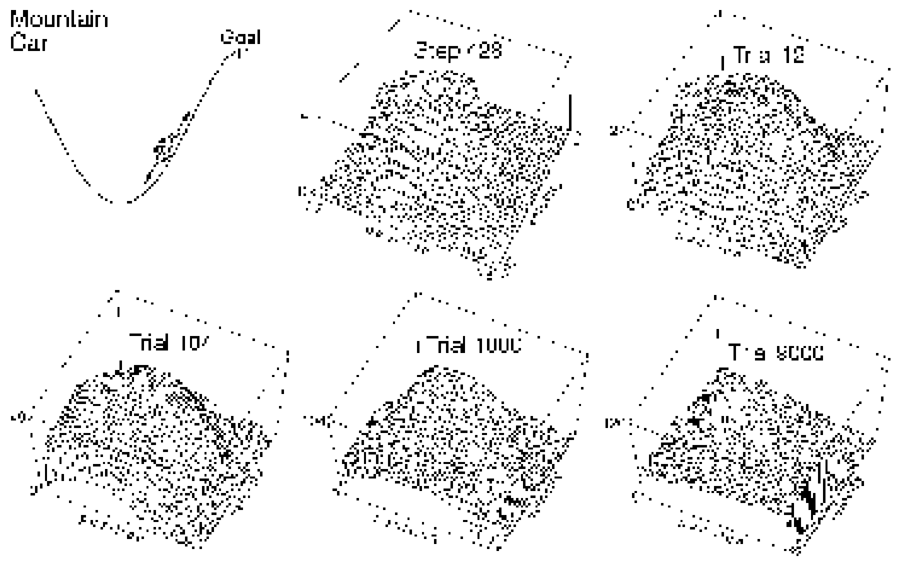


Figure 8.10: The mountain-car task (upper left panel) and the cost-to-go function ($-\max_a Q_t(s,a)$) learned during one run.

The reward in this problem is **-1** on all time steps until the car has moved past its goal position at the top of the mountain, which ends the episode. There are three possible actions: full throttle forward (**+1**), full throttle reverse (**-1**), and zero throttle (**0**). The car moves according to a simplified physics. It's position, x_t, and velocity, $\dot{x}_t$, are given by

$$x_{t+1} = bound\left[x_t + \dot{x}_{t+1}\right]$$

$$\dot{x}_{t+1} = bound\left[\dot{x}_t + 0.001a_t + -0.0025\cos(3x_t)\right],$$

where the **bound** operation enforces $-1.2 \leq x_{t+1} \leq 0.5$ and $-0.07 \leq \dot{x}_{t+1} \leq 0.07$. When x_{t+1} reached the left bound, then $\dot{x}_{t+1}$ was reset to zero. When it reached right bound, then the goal was reached and the episode was terminated. Each episode started from a random position and velocity uniformly chosen from these ranges. To convert the two continuous state variables to binary features, we used grid-tilings exactly as in Figure 8.5. We used ten 9×9 tilings, each offset by a random fraction of a tile width.

The Sarsa algorithm in Figure 8.8 (using replace traces and the optional clearing) readily solved this task, learning a near optimal policy within 100 episodes. Figure 8.10 shows the negative of the value function (the *cost-to-go* function) learned on one run, using the parameters $\lambda = .9$, $\epsilon = 0$, and $\alpha = .05\ (\frac{0.1}{m})$. The initial action values were all zero, which was optimistic (all true values are negative in this task), causing extensive exploration to occur even though the exploration parameter, ϵ

, was **0**. This can be seen in the middle-top panel of the figure, labeled ``Step 428.'' At this time not even one episode has been completed, but the car has oscillated back and forth in the valley, following circular trajectories in state space. All the states visited frequently are valued worse than unexplored states, because the actual rewards have been worse than what was (unrealistically) expected. This continually drives the agent away from wherever it has been to explore new states, until a solution is found. Figure 8.11 shows the results of a detailed study of the effect of the parameters α and λ, and of the kind of traces, on the rate of learning on this task. $\diamond$

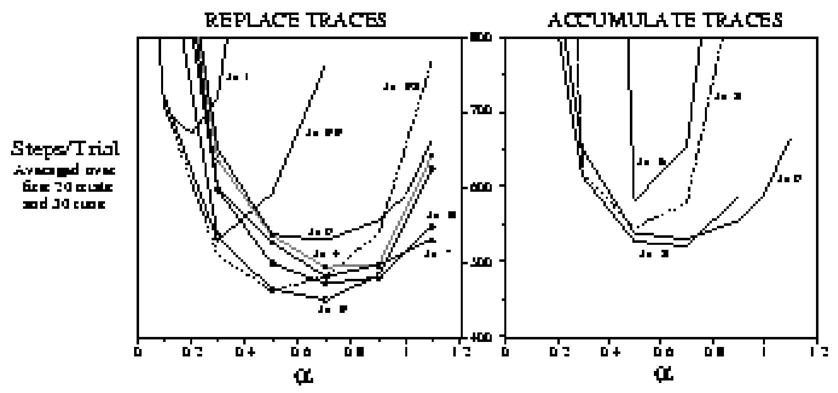


Figure 8.11: The effect of α, λ, and the kind of traces on early performance on the Mountain-Car task. This study used five 9×9 tilings.

Exercise 8.8*

Describe how the actor-critic control method could be combined with gradient-descent function approximation.

Richard Sutton
Sat May 31 15:08:20 EDT 1997

8.5 Off-Policy Bootstrapping

We return now to the prediction case to take a closer look at the interaction between bootstrapping, function approximation, and the on-policy distribution. By bootstrapping we mean the updating of a value estimate on the basis of other value estimates. TD methods involve bootstrapping, as do DP methods, whereas Monte Carlo methods do not. TD(λ) is a bootstrapping method for $\lambda < 1$, and by convention we consider it to *not* be a bootstrapping method for $\lambda = 1$. Although TD(1) involves bootstrapping within an episode, the net effect over a complete episode is the same as a non-bootstrapping Monte Carlo update.

Bootstrapping methods are more difficult to combine with function approximation methods than non-bootstrapping methods. For example, consider the case of value value prediction with linear, gradient-descent function approximation. In this case, non-bootstrapping methods find minimal MSE (8.1) solutions for any distribution of training examples, **P**, whereas bootstrapping methods find only near minimal MSE (8.9) solutions, and only for the on-policy distribution. Moreover, the quality of the MSE bound for TD(λ) gets worse the farther λ strays from 1, i.e., the farther the method moves from its non-bootstrapping form.

The restriction of the convergence results for bootstrapping methods to the on-policy distribution is of greatest concern. This is not a problem for on-policy methods such as Sarsa and actor-critic methods, but it is for off-policy methods such as Q-learning and DP methods. Off-policy control methods do not backup states (or state-action pairs) with exactly the same distribution with which the states would be encountered following the estimation policy (the policy whose value function they are estimating). Many DP methods, for example, backup all states uniformly. Q-learning may backup states according to an arbitrary distribution, but typically it backs them up according to the distribution generated by interacting with the environment and following a soft policy close to a greedy estimation policy. We use the term *off-policy bootstrapping* for any kind of bootstrapping using a distribution of backups different from the on-policy distribution. Surprisingly, off-policy bootstrapping combined with function approximation can lead to divergence and infinite MSE.

Example 8.3 *Baird's Counterexample*. Consider the six-state, episodic Markov process shown in Figure 8.12. Episodes begin in one of the five upper states, proceed immediately to the lower state, and then cycle there for some number of steps before terminating. The reward is zero on all transitions, so the true value function is $V^\pi(s) = 0$, for all **s**. The form of the approximate value function is shown by the equations inset in each state. Note that the overall function is linear and that there are fewer states than components of $\vec{\theta}_t$. Moreover, the set of feature vectors, $\{\vec{\phi}_s \,:\, s \in \mathcal{S}\}$ corresponding to this function form a linearly independent set, and the true value function is easily formed by setting $\vec{\theta}_t = \vec{0}$. In all ways, this task seems a favorable case for linear function approximation.

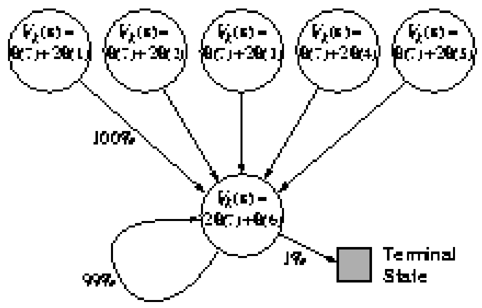

$V_k(s) = \theta(7)+2\theta(1)$
$V_k(s) = \theta(7)+2\theta(2)$
$V_k(s) = \theta(7)+2\theta(3)$
$V_k(s) = \theta(7)+2\theta(4)$
$V_k(s) = \theta(7)+2\theta(5)$
100%
$V_k(s) = 2\theta(7)+\theta(6)$
99%
1%
Terminal State

Figure 8.12: Baird's counterexample. The approximate value function for this Markov process is of the form shown by the linear expressions inside each state. The reward is always zero.

The prediction method we apply to this task is a linear, gradient-descent form of DP policy evaluation. The parameter vector, $\vec{\theta}_k$, is updated in sweeps through the state space, performing a synchronous, gradient-descent backup at every state, **s**, using the DP (full backup) target:

$$\vec{\theta}_{k+1} = \vec{\theta}_k + \alpha \sum_s \Big(E\left\{ r_{t+1} + \gamma \right.$$

$$\left. V_t(s_{t+1}) \middle| s_t = s - V_k(s) \right) \nabla_{\vec{\theta}_k} V_k(s).$$

Like most DP methods, this one uses a uniform backup distribution, one of the simplest off-policy distributions. Otherwise this is an ideal case. There is no randomness and no asynchrony. Each state is updated exactly once per sweep according to a classical DP backup. The method is entirely conventional except in its use of a gradient-descent function approximator. Yet for some initial values of the parameters, the system becomes unstable, as shown computationally in Figure 8.13. ◇

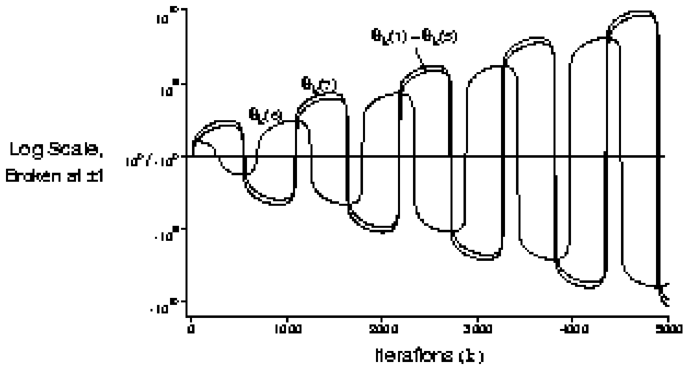


Figure 8.13: Computational demonstration of the instability of DP value prediction with a linear function approximator on Baird's counterexample. The parameters were $\gamma = .99$, $\alpha = 0.01$, and $\vec{\theta}_0 = (1, 1, 1, 1, 1, 10, 1)^T$.

If we alter just the distribution of DP backups in Baird's counterexample, from the uniform distribution to the on-policy distribution (which generally requires asynchronous updating), then convergence is guaranteed to a solution with error bounded by (8.9) for $\lambda = 0$. This example is striking because the DP method used is arguably the simplest and best understood bootstrapping method, and the linear, gradient-descent method used is arguably the simplest and best understood function approximator. The example shows that even the simplest combination of bootstrapping and function approximation can be unstable if the backups are not done according to the on-policy distribution.

Counterexamples similar to Baird's also exist showing divergence for Q-learning. This is cause for concern because otherwise Q-learning has the best convergence guarantees of all control methods. Considerable effort has gone into trying to find a remedy to this problem or to obtain some weaker, but still workable, guarantee. For example, it may be possible to guarantee convergence of Q-learning as long as the behavior policy (the policy used to select actions) is sufficiently close to the estimation policy (the policy used in GPI), e.g., when it is the ϵ -greedy policy. To the best of our knowledge, Q-learning has never been found to diverge in this case, but there has been no theoretical analysis. In the rest of this section we present several of the other ideas that have been explored.

Suppose that instead of taking just a step toward the expected one-step return on each iteration, as in Baird's counterexample, we actually change the value function all the way to the best, least-squares approximation. Would this solve the instability problem? Of course it would if the feature vectors, $\{\vec{\phi}_s : s \in S\}$, formed a linearly independent set, as they do in Baird's counterexample, because then exact approximation is possible on each iteration and the method reduces to standard tabular DP. But of course the point here is to consider the case when an exact solution is *not* possible. In this case stability is not guaranteed even when forming the best approximation at each iteration, as shown by the following example.

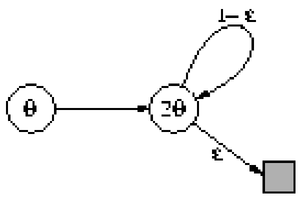


Figure 8.14: Tsitsiklis and Van Roy's counterexample to DP policy evaluation with least-squares linear function approximation.

Example 8.4 *Tsitsiklis and Van Roy's Counterexample*. The simplest counterexample to linear least-squares DP is shown in Figure 8.14. There are just two nonterminal states, and the modifiable parameter θ_k is a scalar. The estimated value of the first state is just θ_k, and the estimated value of the second state is $2\theta_k$. The reward is zero on all transitions, so the true values are zero at both states, which is exactly representable with $\theta_k = 0$. If we set θ_{k+1} at each step so as to minimize the MSE between the estimated value and the expected one-step return, then we have

$$\theta_{k+1} \quad = \quad \arg\min_{\theta\in\Re} \sum_{s\in S} \Big(V_\theta(s) - E_\pi\{r_{t+1} + \gamma$$

$$V_{\theta_k}(s_{t+1}) | s_t = s\Big)^2$$
$$= \arg\min_{\theta\in\Re}(\theta - \gamma 2\theta_k)^2 + (2\theta - (1-\epsilon)\gamma 2\theta_k)^2$$
$$= 6\text{-}4\epsilon \quad \overline{5\gamma\theta_k, (8.10)}$$

where V_θ denotes the value function given θ. The sequence $\{\theta_k\}$ diverges when $\gamma > \frac{5}{6-4\epsilon}$ and $\theta_0 \neq 0$. ◇

One way to try to prevent instability is to use special kinds of function approximators. In particular, stability is guaranteed for function approximators that do not extrapolate from the observed targets. These methods, called *averagers*, include nearest neighbor methods and local weighted regression, but not popular methods such as tile coding and backpropagation.

Another approach is to attempt to minimize not the squared error from the true value function, but the squared error from the expected one step return. It is natural to call this error measure the mean squared Bellman error:

$$\sum_s P(s)\Big(E_\pi\{r_{t+1} + \gamma V_t(s_{t+1}) | s_t = s\} - V_t(s)\Big)^2. \qquad (8.11)$$

This suggests the gradient-descent procedure:

$$\begin{aligned}\vec{\theta}_{t+1} &= \vec{\theta}_t - \frac{1}{2}\alpha\nabla_{\vec{\theta}_t}\Big(E_\pi\{r_{t+1} + \gamma V_t(s_{t+1})\} - V_t(s_t)\Big)^2 \\ &= \vec{\theta}_t + \alpha\Big(E_\pi\{r_{t+1} + \gamma V_t(s_{t+1})\} - V_t(s_t)\Big)\Big(\nabla_{\vec{\theta}_t} V_t(s_t) - E_\pi\Big\{\nabla_{\vec{\theta}_t} V_t(s_{t+1})\Big\}\Big),\end{aligned}$$

where the expected values are implicitly conditional on s_t. This update is guaranteed to converge to minimize the expected Bellman error. However, this method is feasible only for deterministic systems or when a model is available. The problem is that the update above involves the next state, s_{t+1}, appearing in *two* expected values that are multiplied together. To get an unbiased sample of the product one needs two independent samples of the next state, but during normal interaction with the environment only one is obtained. Because of this, the method is probably limited in practice to cases in which a model is available (to produce a second sample). In practice, this method is also sometimes slow to converge. To handle that problem, Baird (1995) has proposed combining this method parametrically with conventional TD methods.

Exercise 8.9 (programming)

Look up the paper by Baird (1995) on the internet and obtain his counterexample for Q-learning. Implement it and demonstrated the divergence.

Richard Sutton
Sat May 31 15:08:20 EDT 1997

8.6 Should We Bootstrap?

At this point you may be wondering why we bother with bootstrapping methods at all. Non-bootstrapping methods can be used with function approximation more reliably and over a broader range of conditions than bootstrapping methods. Non-bootstrapping methods achieve a lower asymptotic error than bootstrapping methods, even when backups are done according to the on-policy distribution. By using eligibility traces and $\lambda = 1$, it is even possible to implement non-bootstrapping methods online, in a step-by-step incremental manner. Despite all this, in practice bootstrapping methods are usually the methods of choice.

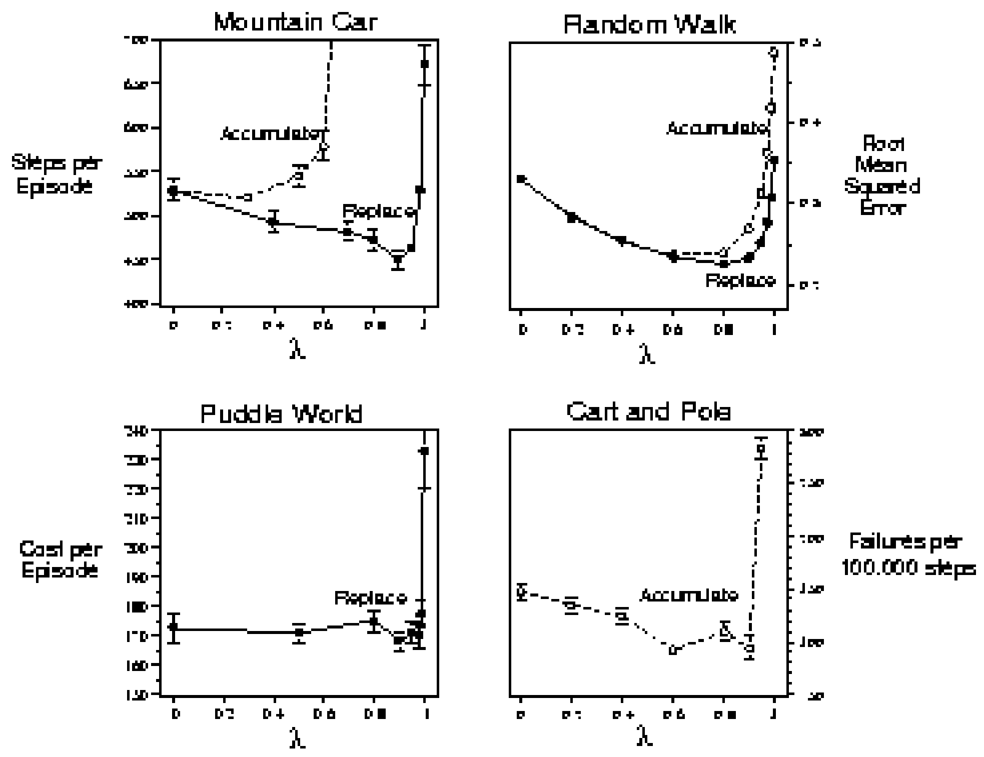


Figure: The effect of λ on reinforcement learning performance. In all cases, the better the performance, the *lower* the curve. The two left panels are applications to simple continuous-state control tasks using the Sarsa(λ) algorithm and tile coding, with either replacing or accumulating traces (Sutton, 1996). The upper right panel is for policy evaluation on a random-walk task using TD(λ) (Singh and

Sutton, 1996). The lower right panel is unpublished data for the pole-balancing task (Example 3.4) from an earlier study (Sutton, 1984).

In empirical comparisons, bootstrapping methods usually perform much better than non-bootstrapping methods. A convenient way to make such comparisons is to use a TD method with eligibility traces and vary λ from 0 (pure bootstrapping) to 1 (pure non-bootstrapping). Figure 8.15 summarizes a collection of such results. In all cases, performance became much worse as λ \ approached **1**, the non-bootstrapping case. The example in the upper right of the figure is particularly significant in this regard. This is a policy-evaluation task and the performance measure used is root MSE (at the end of each trial, averaged over the first 20 trials). Asymptotically, the $\lambda = 1$ case must be best according to this measure, but here, short of the asymptote, we see it performing much worse.

At this time it is unclear why methods that involve some bootstrapping perform so much better than pure non-bootstrapping methods. It could be just that bootstrapping methods learn faster, or it could be that they actually learn something better than non-bootstrapping methods. The available results indicate that non-bootstrapping methods are better than bootstrapping methods at reducing MSE from the true value function, but reducing MSE is not necessarily the most important goal. For example, if you add 1000 to the true action-value function at all state-action pairs, then it will have very poor MSE, but you will still get the optimal policy. Nothing quite that simple is going on with bootstrapping methods, but they do seem to do something right. We expect the understanding of these issues to improve as research continues.

Next Up Previous

Next: 8.7 Summary **Up:** 8 Generalization and Function **Previous:** 8.5 Off-Policy Bootstrapping

Richard Sutton
Sat May 31 15:08:20 EDT 1997

8.7 Summary

Reinforcement learning systems must be capable of *generalization* if they are to be applicable to artificial intelligence or to large engineering applications in general. To achieve this, any of a broad range of existing methods for *supervised-learning function approximation* can be used simply by treating each backup as a training example. *Gradient-descent methods*, in particular, allow a natural extension to function approximation of all the techniques developed in previous chapters, including eligibility traces. *Linear* gradient-descent methods are particularly appealing theoretically and work well in practice when provided with appropriate features. Choosing the features is one of the most important ways of adding prior domain knowledge to reinforcement learning systems. Linear methods include radial basis functions, tile coding (CMAC), and Kanerva coding. Backpropagation methods for multi-layer neural networks are instances of *nonlinear* gradient-descent function approximators.

For the most part, the extension of reinforcement learning prediction and control methods to gradient-descent forms is straightforward. However, there is an interesting interaction between function approximation, bootstrapping, and the on-policy/off-policy distinction. Bootstrapping methods, such as DP and TD(λ) for $\lambda < 1$, work reliably in conjunction with function approximation over a narrower range of conditions than non-bootstrapping methods. As the control case has not yet yielded to theoretical analysis, research has focused on the value prediction problem. In this case, on-policy bootstrapping methods converge reliably with linear gradient-descent function approximators to a solution with mean square error bounded by $\frac{1-\gamma\lambda}{1-\gamma}$ times the minimum possible error. Off-policy methods, on the other hand, may diverge to infinite error. Several approaches have been explored to making off-policy bootstrapping methods work with function approximation, but this is still an open research issue. Bootstrapping methods are of persistent interest in reinforcement learning, despite their limited theoretical guarantees, because in practice they usually work significantly better than non-bootstrapping methods.

8.8 Bibliographical and Historical Remarks

Despite our treatment of generalization and function approximation late in the book, they have always been an integral part of reinforcement learning. It is only in the last decade or less that the field has focused on the tabular case, as we have here for the first seven chapters. Bertsekas and Tsitsiklis (1996) present the state of the art in function approximation in reinforcement learning, and the collection of papers by Boyan, Moore, and Sutton (1995) is also useful. Some of the early work with function approximation in reinforcement learning is discussed at the end of this section.

8.2

Gradient-descent methods for the minimizing mean-squared-error in supervised learning are well known. Widrow and Hoff (1960) introduced the Least-Mean-Square (LMS) algorithm, which is the prototypical incremental gradient-descent algorithm. Details of this and related algorithms are provided in many texts (e.g., Widrow and Stearns, 1985; Bishop, 1995; Duda and Hart, 1973).

Gradient-descent analyses of TD learning date back at least to Sutton (1988). Methods more sophisticated than the simple gradient-descent methods covered in this section have also been studied in the context of reinforcement learning, such as quasi-Newton methods (see Werbos, 1990) and recursive-least squares methods (Bradtke, 1993, 1994; Bradtke and Barto, 1996; Bradtke, Barto, and Ydstie, 1994). Bertsekas and Tsitsiklis (1996) provide a good discussion of these methods.

The earliest use of state aggregation in reinforcement learning may have been Michie and Chambers's (1968) BOXES system. The theory of state aggregation in reinforcement learning has been developed by Singh, Jaakkola, and Jordan (1995) and Tsitsiklis and Van Roy (1996).

8.3

TD(λ) with linear gradient-descent function approximation was first explored by Sutton (1988, 1984), who proved convergence of TD(0) in the mean to the minimal MSE solution for the case in which the feature vectors, $\{\vec{\phi}_s : s \in \mathcal{S}\}$, were linearly independent.

Convergence with probability one for general λ was proved by several researchers all at about the same time (Peng, 1993; Dayan and Sejnowski, 1994; Tsitsiklis, 1994; Gurvitz, Lin, and Hanson, 1994). Jaakkola, Jordan, and Singh (1994), in addition, proved convergence under online updating. All of these results assumed linearly independent features vectors, which implies at least as many component to $\vec{\theta}_t$ as there are states.

Convergence of linear TD(λ) for the more interesting case of general (dependent) feature vectors was first shown by Dayan (1992). A significant generalization and strengthening of Dayan's result was proved by Tsitsiklis and Van Roy (1997). They proved the main presented in Section 8.2, the bound on the asymptotic error of TD(λ) and other bootstrapping methods.

Our presentation of the range of possibilities for linear function approximators is based on that by Barto (1990). The term *coarse coding* is due to Hinton (1984), and our Figure 8.2 is based on one of his figures. Waltz and Fu (1965) provide an early example of this type of function approximation in a reinforcement learning system.

Tile coding, including hashing, was introduced by Albus (1971, 1981). He described it in terms of his ``cerebellar model articulator controller,'' or *CMAC*, as tile coding is known in the literature. The term ``tile coding'' is new to this book, though the idea of describing CMAC in these terms is taken from Watkins (1989). Tile coding has been used in many reinforcement learning systems (e.g., Shewchuk and Dean, 1990; Lin and Kim, 1991; Miller, 1994; Sofge and White, 1992; Tham, 1994; Sutton, 1996; Watkins, 1989) as well as in other types of learning control systems (e.g., Kraft and Campagna, 1990; Kraft, Miller, and Dietz, 1992).

Function approximation using radial basis functions (RBFs) has received wide attention ever since being related to neural networks by Broomhead and Lowe (1988). Powell (1987) reviewed earlier uses of RBFs, and Poggio and Girosi (1989, 1990) extensively developed and applied this approach.

What we call ``Kanerva coding'' was introduced by Kanerva (1988), as part of his more general idea of *sparse distributed memory*. A good review of this and related memory models is provided by Kanerva (1993). This approach has been pursued by Gallant (1993) and by Sutton and Whitehead (1993), among others.

8.4

Q(λ) with function approximation was first explored by Watkins (1989). Sarsa(λ) with function approximation was first explored by Rummery and Niranjan (1994). The mountain-car example is based on a similar task studied by Moore (1990). The results on it presented here are from Sutton (1995) and Singh and Sutton (1996).

Convergence of the control methods presented in this section has not been proven (and seems unlikely for Q(λ) given the results presented in the next section). Convergence results for control methods with state-aggregation and other special-case function approximators are proved by Tsitsiklis and Van Roy (1996), Singh, Jaakkola and Jordan (1995), and Gordon (1995).

8.5

Baird's counterexample is due to Baird (1995). Tsitsiklis and Van Roy's counterexample is due to Tsitsiklis and Van Roy (1997). Averaging function approximators are developed by Gordon (1995, 1996). Gradient-descent methods for minimizing the Bellman error are due to Baird, who calls them *residual-gradient methods*. Other examples of instability with off-policy DP methods and more complex function approximators are given by Boyan and Moore (1995). Bradtke (1993) gives an example in which Q-learning using linear function approximation in a linear quadratic regulation problem can converge to a destabilizing policy.

The use of function approximation in reinforcement learning goes back to the early neural networks of Farley and Clark (1954; Clark and Farley, 1955), who used reinforcement learning to adjust the parameters of linear threshold functions representing policies. The earliest example we know of in which function approximation methods were used for learning value functions was Samuel's checkers player (1959, 1967). Samuel followed Shannon's (1950b) suggestion that a value function did not have to be exact to be a useful guide to selecting moves in a game and that it might be approximated by linear combination of features. In addition to linear function approximation, Samuel also experimented with lookup tables and hierarchical lookup tables called signature tables (Griffith, 1966, 1974; Page, 1977; Biermann, Fairfield, and Beres, 1982).

At about the same time as Samuel's work, Bellman and Dreyfus (1959) proposed using function approximation methods with DP. (It is tempting to think that Bellman and Samuel had some influence on one another, but we know of no reference to the other in the work of either.) There is now a fairly extensive literature on function approximation methods and DP, such as multigrid methods and methods using splines and orthogonal polynomials (e.g., Bellman and Dreyfus, 1959; Bellman, Kalaba, and Kotkin, 1973; Daniel, 1976; Whitt, 1978; Reetz, 1977; Schweitzer and Seidmann, 1985; Chow and Tsitsiklis, 1991; Kushner and Dupuis, 1992; Rust, 1996).

Holland's (1986) classifier system used a selective feature match technique to generalize evaluation information across state-action pairs. Each classifier matched a subset of states having specified values for a subset of features, with the remaining features having arbitrary values (``wild cards"). These subsets were then used in a conventional state-aggregation approach to function approximation. Holland's idea was to use a genetic algorithm to evolve a set of classifiers that collectively would implement a useful action-value function. Holland's ideas influenced the early research of the authors on reinforcement learning, but we focused on different approaches to function approximation. As function approximators, classifiers are limited in several ways. First, they are state-aggregation methods, with concomitant limitations in scaling and in representing smooth functions efficiently. In addition, the matching rules of classifiers can implement only aggregation boundaries that are parallel to the feature axes. Perhaps the most important limitation of conventional classifier systems is that the classifiers are learned via the genetic algorithm, an evolutionary method. As we discussed in Chapter 1, there is available during learning much more detailed information about how to learn than can be used by evolutionary methods. This perspective led us to instead adapt supervised-learning methods for use in reinforcement learning, specifically gradient-descent and neural-network methods. These differences between Holland's approach and ours are not surprising because Holland's ideas were developed during a period when neural networks were generally regarded as being too weak in computational power to be useful, whereas our work was at the beginning of the period that saw widespread questioning of that conventional wisdom. There remain many opportunities for combining aspects of these different approaches.

A number of reinforcement learning studies using function approximation methods that we have not covered previously should be mentioned. Barto, Sutton and Brouwer (1981) and Barto and Sutton (1981) extended the idea of an associative memory network (e.g., Kohonen, 1977; Anderson, Silverstein, Ritz, and Jones, 1977) to reinforcement learning. Hampson (1983, 1989) was an early proponent of multi-layer neural networks for learning value functions. Anderson (1986, 1987) coupled a TD algorithm with the error backpropagation algorithm to learn a value function. Barto and Anandan (1985) introduced a stochastic version of Widrow, Gupta, and Maitra's (1973) *selective bootstrap algorithm*, which they called the *associative reward-penalty* (A_{R-P}) *algorithm*. Williams (1986, 1987, 1988, 1992) extended this type of algorithm to a general class of ``REINFORCE" algorithms, showing that they perform stochastic gradient ascent on the expected reinforcement. Gullapalli (1990) and Williams devised algorithms for learning generalizing policies for the case of continuous actions. Phansalkar and Thathachar (1995) proved both local and global convergence theorems for modified versions of REINFORCE algorithms. Christensen and Korf (1986) experimented with regression methods for modifying coefficients of linear value function approximations in the game of chess. Chapman and Kaelbling (1991) and Tan (1991) adapted decision-tree methods for learning value functions. Explanation-based learning methods have also been adapted for learning value functions, yielding compact representations (Yee, Saxena, Utgoff, and Barto, 1990; Dietterich and Flann, 1995).

Richard Sutton
Sat May 31 15:08:20 EDT 1997

9 Planning and Learning

In this chapter we develop a unified view of methods that require a model of the environment, such as dynamic programming and heuristic search, and methods that can be used without a model, such as Monte Carlo and temporal-difference methods. We think of the former as *planning* methods and of the latter as *learning* methods. Although there are real differences between these two kinds of methods, there are also great similarities. In particular, the heart of both kinds of methods is the computation of value functions. Moreover, all the methods are based on looking ahead to future events, computing a backed-up value, and then using it to update an approximate value function. Earlier in this book we presented Monte Carlo and temporal-difference methods as distinct alternatives, then showed how they can be seamlessly integrated using eligibility traces in methods such as TD(λ) . Our goal in this chapter is a similar integration of planning and learning methods. Having established these as distinct in earlier chapters, we now explore the extent to which they can be blended together.

Richard Sutton
Fri May 30 17:11:38 EDT 1997

9.1 Models and Planning

By a *model* of the environment we mean anything that an agent can use to predict how the environment will respond to its actions. Given a state and an action, a model produces a prediction of the resultant next state and next reward. If the model is stochastic, then there are several possible next states and next rewards, each with some probability of occurring. Some models produce a description of all possibilities and their probabilities; these we call *distribution models*. Other models produce just one of the possibilities, sampled according to the probabilities; these we call *sample models*. For example, consider modeling the sum of a dozen dice. A distribution model would produce all possible sums and their probabilities of occurring, whereas a sample model would produce an individual sum drawn according to this probability distribution. The kind of model assumed in dynamic programming---estimates of the state transition probabilities and expected rewards, $P^a_{ss'}$ and $R^a_{ss'}$---is a distribution model. The kind of model used in the blackjack example in Chapter 5 is a sample model. Distribution models are stronger than sample models in that they can always be used to produce samples. However, in surprisingly many applications it is much easier to obtain sample models than distribution models.

Models can be used to mimic or simulate experience. Given a starting state and action, a sample model produces a possible transition, and a distribution model generates all possible transitions weighted by their probabilities of occurring. Given a starting state and a policy, a sample model could produce an entire episode, and a distribution model could generate all possible episodes and their probabilities. In either case, we say the model is used to *simulate* the environment and produce *simulated experience*.

The word ``planning" is used in several different ways in different fields. We use the term to refer to any computational process that takes a model as input and produces or improves a policy for interacting with the modeled environment:

$$\begin{tikzcd}\text{model} \arrow[r, "\text{planning}"] & \text{policy}\end{tikzcd}$$

Within artificial intelligence, there are two distinct approaches to planning according to our definition. In *state-space planning*, which includes the approach we take in this book, planning is viewed primarily as a search through the state space for an optimal policy or path to a goal. Actions cause transitions from state to state, and value functions are computed over states. In what we call *plan-space planning*, planning is instead viewed as a search through the space of plans. Operators transform one complete plan into another, and value functions, if any, are defined over the space of complete plans. Plan-space planning includes evolutionary methods and *partial-order planning*, a popular kind of planning in artificial intelligence in which the ordering of steps is not completely determined at all stages of planning. Plan-space methods are difficult to apply efficiently to the stochastic optimal control problems that are the focus in reinforcement learning, and we do not consider them further (but see Section 11 .6 for one application of reinforcement learning within plan-space planning).

The unified view we present in this chapter is that all state-space planning methods share a common structure, a structure that is also present in the learning methods presented in this book. It takes the rest of the chapter to develop this view, but there are two basic ideas: 1) all state-space planning methods involve computing value functions as a key intermediate step toward improving the policy, and 2) they compute their value functions by backup operations applied to simulated experience. This common structure can be diagrammed as follows:

$$\begin{tikzcd}\text{model} \arrow[r] & \begin{array}{c}\text{simulated}\\ \text{experience}\end{array} \arrow[r, "\text{backups}"] & \text{values} \arrow[r] & \text{policy}\end{tikzcd}$$

Dynamic programming methods clearly fit this structure: they make sweeps through the space of states, generating for each the distribution of possible transitions. Each distribution is then used to compute a backed-up value and update the state's estimated value. In this chapter we argue that a variety of other state-space planning methods also fit this structure, with individual methods differing only in the kinds of backups they do and the order in which they do them.

Viewing planning methods in this way emphasizes their relationship to the learning methods that we have described in this book. The heart of both learning and planning methods is the estimation of value functions by backup operations. The difference is that whereas planning uses simulated experience generated by a model, learning methods use real experience generated by the environment. Of course this difference leads to a number of other differences, for example, in how performance is assessed and in how flexibly experience can be generated. But the common structure means that many ideas and algorithms can be transferred between planning and learning. In particular, in many cases, a learning algorithm can be substituted for the key backup step of a planning method. Learning methods require only experience as input, and in many cases they can be applied to simulated experience just as well as to real experience. Figure 9.1 shows a simple example of a planning method based on 1-step tabular Q-learning and on random samples from a sample model. This method, which we call *random-sample 1-step tabular Q-*

planning, converges to the optimal policy for the model under the same conditions that 1-step tabular Q-learning converges to the optimal policy for the real environment (each state-action pair must be selected an infinite number of times in Step 1, and α \ must decrease appropriately over time).

```
Do Forever:
1. Select a state, $s \in \mathcal{S}$, and an action, $a \in \mathcal{A}(s)$, at random
2. Send s,a to a sample model, and obtain
     a sample next state, $s'$, and a sample next reward, $r$
3. Apply 1-step tabular Q-learning to $s, a, s', r$:
     $Q(s,a) \leftarrow Q(s,a) + \alpha [r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
```

Figure 9.1: Random-sample 1-step tabular Q-planning

In addition to the unified view of planning and learning methods, a second theme in this chapter is the benefits of planning in small, incremental steps. This enables planning to be interrupted or re-directed at any time with little wasted computation, which appears to be a key requirement for efficiently intermixing planning with acting and with learning of the model. More surprisingly, later in this chapter we present evidence that planning in very small steps may be the most efficient approach even on pure planning problems if the problem is too large to be solved exactly.

Next Up Previous

Next: 9.2 Integrating PlanningActing, **Up:** 9 Planning and Learning **Previous:** 9 Planning and Learning

Richard Sutton
Fri May 30 17:11:38 EDT 1997

9.2 Integrating Planning, Acting, and Learning

When planning is done online, while interacting with the environment, a number of interesting issues arise. New information gained from the interaction may change the model and thereby interact with planning. It may be desirable to customize the planning process in some way to the states or decisions currently under consideration, or expected in the near future. If decision making and model learning are both computation-intensive processes, then the available computational resources may need to be divided between them. To begin exploring these issues, in this section we present Dyna-Q, a simple architecture integrating the major functions needed in an online planning agent. Each function appears in Dyna-Q in a simple, almost trivial, form. In subsequent sections we elaborate some of the alternate ways of achieving each function and the tradeoffs between them. For now, we seek merely to illustrate the ideas and stimulate your intuition.

Within a planning agent, there are at least two roles for real experience: it can be used to improve the model (to make it more accurately match the real environment) and it can be used to directly improve the value function and policy using the kinds of reinforcement learning methods we have discussed in previous chapters. The former we call *model learning*, and the latter we call *direct reinforcement learning* (direct RL). The possible relationships between experience, model, values and policy are summarized in Figure 9.2. Each arrow shows a relationship of influence and presumed improvement. Note how experience can improve value and policy functions either directly or indirectly via the model. It is the latter, which is sometimes called *indirect reinforcement learning*, that is involved in planning.

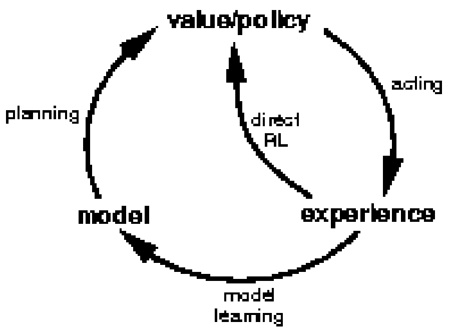


Figure 9.2: Relationships among learning, planning, and acting.

Both direct and indirect methods have advantages and disadvantages. Indirect methods often make fuller use of a limited amount of experience and thus achieve a better policy with fewer environmental interactions. On the other hand, direct methods are much simpler and are not affected by biases in the design of the model. Some have argued that indirect methods are always superior to direct ones, while others have argued that direct methods are responsible for most human and animal learning. Related debates in psychology and AI concern the relative importance of cognition as opposed to trial-and-error learning, and of deliberative planning as opposed to reactive decision making. Our view is that the contrast between the alternatives in all these debates has been exaggerated, that more insight can be gained by recognizing the similarities between these two sides than by opposing them. For example, in this book we have emphasized the deep similarities between dynamic programming and temporal-difference methods, even though one was designed for planning and the other for model-free learning.

Dyna-Q includes all of the processes shown in Figure 9.2---planning, acting, model learning, and direct RL---all occurring continually. The planning method is the random-sample 1-step tabular Q-planning method given in Figure 9.1. The direct RL method is 1-step tabular Q-learning. The model learning method is also table-based and assumes the world is deterministic. After each transition $s_t, a_t \rightsquigarrow s_{t+1}, r_{t+1}$, the model records in its table entry for s_t, a_t the prediction that s_{t+1}, r_{t+1} will deterministically follow. Thus, if the model is queried with a state-action pair that has been experienced before, it simply returns the last-observed next state and next reward as its prediction. During planning, the Q-planning algorithm randomly samples only from state-action pairs that have previously been experienced (in Step 1), so the model is never queried with a pair about which it has no information.

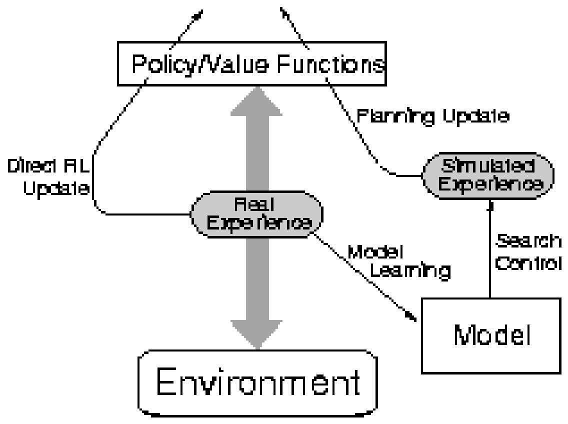


Figure 9.3: The general Dyna Architecture

The overall architecture of Dyna agents, of which the Dyna-Q algorithm is one example, is shown in Figure 9.3. The central column represents the basic interaction between agent and environment, giving rise to a trajectory of real experience. The arrow on the left of the figure represents direct reinforcement learning operating on real experience to improve the value function and the policy. On the right are model-based processes. The model is learned on the basis of real experience and gives rise to simulated experience. We use the term *search control* to refer to the process that selects the starting states and actions for the simulated experiences generated by the model. Finally, planning is achieved by applying reinforcement learning methods to the simulated experiences just as if they had really happened. Typically, as in Dyna-Q, the same reinforcement learning method is used both for learning from real experience and for planning from simulated experience. The reinforcement learning method is thus the ``final common path'' for both learning and planning. Learning and planning are deeply integrated in they sense that they share almost all the same machinery, differing only in the source of their experience.

Conceptually, planning, acting, model learning, and direct RL occur simultaneously and in parallel in Dyna agents. For concreteness and implementation on a serial computer, however, we fully specify the order in which they occur within a time step. In Dyna-Q, the acting, model-learning, and direct-RL processes require little computation, and we assume they consume just a fraction of the time. The remaining time in each step can be devoted to the planning process, which is inherently computation intensive. Let us assume that there is time in each step, after acting, model learning and direct RL, to complete **N** iterations (Steps 1-3) of the Q-planning algorithm. Figure 9.4 shows the complete algorithm for Dyna-Q.

```
Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Do Forever:
```

```
(a) $s \leftarrow$ current (non-terminal) state
(b) $a \leftarrow \epsilon$-greedy$(s, Q)$
(c) Execute action a; observe resultant state, $s'$, and reward, r
(d) $Q(s,a) \leftarrow Q(s,a) + \alpha\,[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
(e) $Model(s,a) \leftarrow s', r$          (assuming deterministic environment)
(f) Repeat N times:
     $s \leftarrow$ random previously observed state
     $a \leftarrow$ random action previously taken in $s$
     $s', r \leftarrow Model(s,a)$
     $Q(s,a) \leftarrow Q(s,a) + \alpha\,[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
```

Figure 9.4: Dyna-Q Algorithm. $Model(s,a)$ denotes the contents of the model (predicted next state and reward) for state-action pair **s,a**. Direct reinforcement learning, model learning, and planning are implemented by steps (d), (e), and (f), respectively. If (e) and (f) were omitted, the remaining algorithm would be just 1-step tabular Q-learning.

Example 9.1 *Dyna Maze*. Consider the simple maze shown inset in Figure 9.5. In each of the 47 states there are four actions, `up`, `down`, `right`, and `left`, which take the agent deterministically to the corresponding neighboring states, except when movement is blocked by an obstacle or the edge of the maze, in which case the agent remains where it is. Reward is zero on all transitions, except those into the goal state, on which it is **+1**. After reaching the goal state (G), the agent returns to the start state (S) to begin a new episode. This is a discounted, episodic task with $\gamma = .95$.

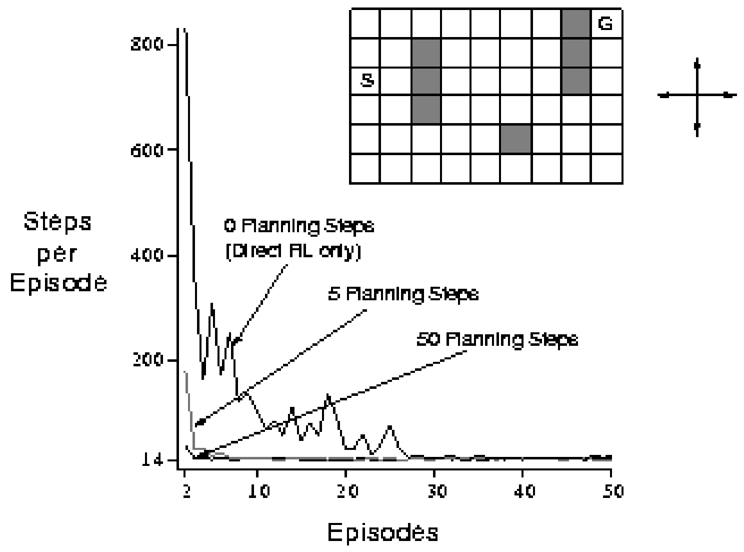


Figure 9.5: A simple maze (inset) and the average learning curves for Dyna-Q agents varying in their number of planning steps per real step. The task is to travel from `S' to `G' as quickly as possible.

The main part of Figure 9.5 shows average learning curves from an experiment in which Dyna-Q agents were applied to the maze task. The initial action values were zero, the step size was $\alpha = 0.1$, and the exploration parameter was $\epsilon = 0.1$. When selecting greedily among actions, ties were broken randomly. The agents varied in the number of planning steps, **N**, they performed per real step. For each **N**, the curves show the number of steps taken by the agent in each episode, averaged over 30 repetitions of the experiment. In each repetition, the initial seed for the random number generator was held constant across algorithms. Because of this, the first episode was exactly the same (about 1700 steps) for all values of **N**, and its data are not shown in the figure. After the first episode, performance improved for all values of **N**, but much more rapidly for larger values. Recall that the **N=0** agent is a non-planning agent, utilizing only direct reinforcement learning (1-step tabular Q-learning). This was by far the slowest agent on this problem, despite the fact that the parameter values (α and ϵ) were optimized for it. The non-planning agent took about 25 episodes to reach (ϵ -)optimal performance, whereas the **N=5** agent took about 5 episodes, and the **N=50** agent took only 3 episodes.

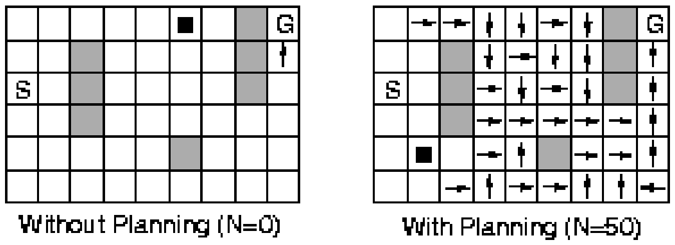


Figure 9.6: Policies found by planning and non-planning Dyna-Q agents halfway through the second episode. The arrows indicate the greedy action in each state; no arrow is shown for a state if all of its action values are equal. The black square indicates the location of the agent.

Figure 9.6 shows why the planning agents found the solution so much faster than the non-planning agent. Shown are the policies found by the **N=0** and **N=50** agents halfway through the second episode. Without planning (**N=0**), each episode adds only one additional step to the policy, and so only one step (the last) has been learned so far. With planning, again only one step is learned during the first episode, but here during the second episode an extensive policy has been developed that by the episode's end will reach almost back to the start state. This policy is built by the planning process while the agent is still wandering near the start state. By the end of the third episode a complete optimal policy will have been found and perfect performance attained. $\diamond$

In Dyna-Q, learning and planning are accomplished by exactly the same algorithm, operating on real experience for learning and on simulated experience for planning. Because planning proceeds incrementally, it is trivial to intermix planning and acting. Both proceed as fast as they can. The agent is always reactive and always deliberative, responding instantly to the latest sensory information and yet always planning in the background. Also ongoing in the background is the model learning process. As new information is gained, the model is updated to better match reality. As the model changes, the ongoing planning process will gradually compute a different way of behaving to match the new model.

Exercise 9.1

The non-planning method looks particularly poor in Figure 9.6 because it is a 1-step method; a method using eligibility traces would do better. Do you think an eligibility trace method could do as well as the Dyna method? Explain why or why not.

Richard Sutton
Fri May 30 17:11:38 EDT 1997

9.3 When the Model is Wrong

In the maze example presented in the previous section, the changes in the model were relatively modest. The model started out empty, and was then filled only with exactly correct information. In general, we can not expect to be so fortunate. Models may be incorrect because the environment is stochastic and only a limited number of samples have been observed, because the model was learned using function approximation that has generalized imperfectly, or simply because the environment has changed and its new behavior has not yet been observed. When the model is incorrect the planning process will compute a suboptimal policy.

In some cases, the suboptimal policy computed by planning quickly leads to the discovery and correction of the modeling error. This tends to happen when the model is optimistic in the sense of predicting greater reward or better state transitions than are actually possible. The planned policy attempts to exploit these opportunities and in so doing discovers that they do not exist.

Example 9.2 *Blocking Maze*. A maze example illustrating this relatively minor kind of modeling error and recovery from it is shown in Figure 9.7. Initially, there is a short path from start to goal, to the right of the barrier, as shown in the upper left of the figure. After 1000 time steps, the short path is ``blocked,'' and a longer path is opened up along the left-hand side of the barrier, as shown in upper right of the figure. The graph shows average cumulative reward for Dyna-Q and two other Dyna agents. The first part of the graph shows that all three Dyna agents found the short path within 1000 steps. When the environment changed, the graphs become flat, indicating a period during which the agents obtained no reward because they were wandering around behind the barrier. After a while, however, they were able to find the new opening and the new optimal behavior. $\diamond$

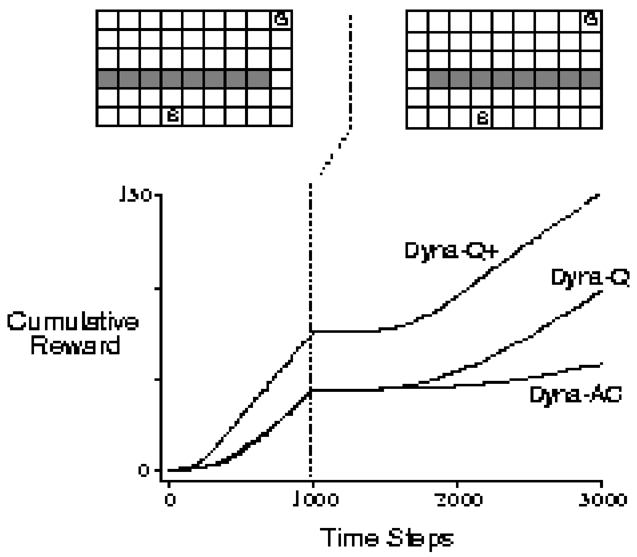


Figure 9.7: Average performance of Dyna agents on a Blocking task. The left environment was used for the first 1000 steps, the right environment for the rest. Dyna-Q+ is Dyna-Q with an exploration bonus that encourages exploration. Dyna-AC is a Dyna agent that uses an actor-critic learning method instead of Q-learning.

Greater difficulties arise when the environment changes to become *better* than it was before, and yet the formerly correct policy does not reveal the improvement. In these cases the modeling error may not be detected for a very long time, if ever, as we see in the next example.

Example 9.3 *Shortcut Maze*. The problem caused by this kind of environmental change is illustrated by the maze example shown in Figure 9.8. Initially, the optimal path is to go around the left side of the barrier (upper left). After 3000 steps, however, a shorter path is opened up along the right side, without disturbing the longer path (upper right). The graph shows that two of the three Dyna agents never switch to the shortcut. In fact, they never realize that it exists. Their models say there is no shortcut, so the more they plan, the less likely they are to step to the right and discover it. Even with an ϵ -greedy policy, it is very unlikely that an agent will take so many exploratory actions that the shortcut will be discovered. ◇

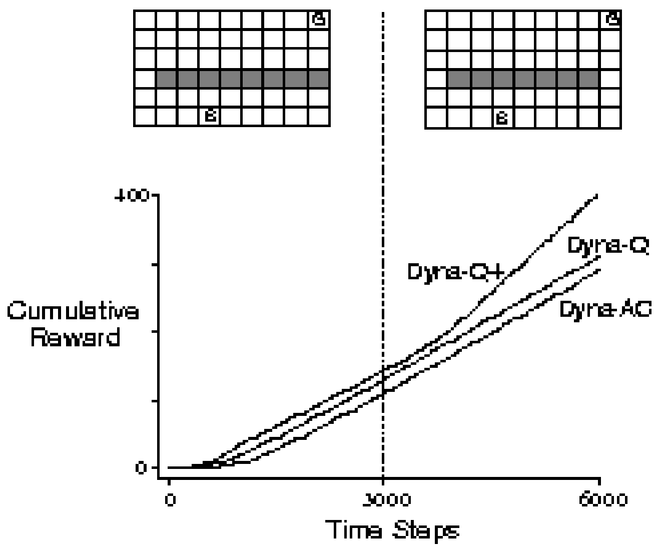


Figure 9.8: Average performance of Dyna agents on a Shortcut task. The left environment was used for the first 3000 steps, the right environment for the rest.

The general problem here is another version of the conflict between exploration and exploitation. In a planning context, exploration means trying actions that improve the model, whereas exploitation means behaving in the optimal way given the current model. We want the agent to explore to find changes in the environment, but not so much that performance is greatly degraded. Just as in the earlier exploitation/exploration conflict, there probably is no solution that is both perfect and practical, but simple heuristics are often effective.

The Dyna-Q+ agent that did solve the shortcut problem used one such heuristic. This agent kept track for each state-action pair of how many time steps had elapsed since the pair had last been tried in a real interaction with the environment. The more time that has elapsed, the greater (we might presume) the chance that the dynamics of this pair has changed and that the model of it is incorrect. To encourage behavior that tests long-untried actions, a special ``bonus reward'' was given on simulated experiences involving these actions. In particular, if the modeled reward for a transition was **r**, and the transition had not been tried in **n** time steps, then planning backups were done as if that transition produced a reward of $r + \kappa\sqrt{n}$, for some small κ. This encouraged the agent to keep testing all accessible state transitions and even to plan long sequences of actions in order to carry out such tests. Of course all this testing has its cost, but in many cases, as here, this kind of computational curiosity is well worth the extra exploration.

Exercise 9.2

Why does the Dyna agent with exploration bonus, Dyna-Q+, perform better in the first phase as well as in the second phase of the blocking and shortcut experiments?

Exercise 9.3

Careful inspection of Figure 9.8 reveals that the difference between Dyna-Q+ and Dyna-Q is narrowing slightly over the first part of the experiment. What is the reason for this?

Exercise 9.4 (programming)

The exploration bonus described above actually changes the estimated values of states and actions. Is this necessary? Suppose the bonus $\kappa\sqrt{n}$ was used not in backups, but solely in action selection. That is, suppose the action selected was always that for which $Q(s,a) + \kappa\sqrt{n_{sa}}$ was maximal. Carry out a gridworld experiment that tests and illustrates the strengths and weaknesses of this alternate approach.

Richard Sutton
Fri May 30 17:11:38 EDT 1997

9.4 Prioritized Sweeping

In the Dyna agents presented in the preceding sections, simulated transitions were started in state-action pairs selected uniformly at random from all previously experienced pairs. But a uniform selection is usually not the best; planning can be much more efficient if simulated transitions and backups are focused on particular state-action pairs. For example, consider what happens during the second episode of the first maze example (Figure 9.6). At the beginning of this episode, only the state-action pair leading directly into the goal has a positive value; the values of all other pairs are still zero. This means that it is pointless to back up along almost all transitions, because they take the agent from one zero-valued state to another, and thus the backups would have no effect. Only a backup along a transition into the state just prior to the goal, or from it into the goal, will change any values. If simulated transitions are generated uniformly, then many wasteful backups will be made before stumbling on one of the two useful ones. As planning progresses, the region of useful backups grows, but planning is still far less efficient than it would be if focused where it would do the most good. In the much larger problems that are our real objective, the number of states is so large that an unfocused search would be extremely inefficient.

This example suggests that search could be usefully focused by working *backward* from goal states. Of course, we do not really want to use any methods specific to the idea of ``goal state.'' We want methods that work for general reward functions. Goal states are just a special case, convenient for stimulating intuition. In general, we want to work back not just from goal states, but from any state whose value has changed. Assume that the values are initially correct given the model, as they were in the maze example prior to discovering the goal. Suppose now that the agent discovers a change in the environment and changes its estimated value of one state. Typically, this will imply that the values of many other states should also be changed, but the only useful 1-step backups are those of actions that lead directly into the one state whose value has already been changed. If the values of these actions are updated, then the values of the predecessor states may change in turn. If so, then actions leading into them need to be backed up, and then *their* predecessor states may have changed. In this way one can work backward from arbitrary states that have changed in value, either performing useful backups or terminating the propagation.

As the frontier of useful backups propagates backward, it often grows rapidly, producing many state-action pairs that could usefully be backed up. But these will not all be equally useful. The value of some states may have changed a lot, others a little. The predecessor pairs of those that have changed a lot are more likely to also change a lot. In a stochastic environment, variations in estimated transition probabilities also contribute to variations in the sizes of changes and in the urgency with which pairs need to be backed up. It is natural to prioritize the backups according to a measure of their urgency, and perform them in order of priority. This is the idea behind *prioritized* *sweeping*. A queue is maintained of every state-action pair that would change nontrivially if backed up, prioritized by the size of the change. When the top pair in the queue is backed up, the effect on each of its predecessor pairs is computed. If the effect is greater than some small threshold, then the pair is inserted in the queue with the new priority (if there is a previous entry of the pair in the queue, then insertion results in only the higher priority entry remaining in the queue). In this way the effects of changes are efficiently propagated backwards until quiescence. The full algorithm for the case of deterministic environments is given in Figure 9.9.

```
Initialize $Q(s,a)$, $Model(s,a)$, for all s,a, and PQueue to empty
Do Forever:
    (a) $s \leftarrow$ current (nonterminal) state
    (b) $a \leftarrow policy(s,Q)$
    (c) Execute action a; observe resultant state, $s'$, and reward, r
    (d) $Model(s,a) \leftarrow s', r$
    (e) $p \leftarrow |r + \gamma \max_{a'} Q(s',a') - Q(s,a)|$.
    (f) if $p > \theta$ then insert s,a into PQueue with priority p
    (g) Repeat N times, while PQueue is not empty:
          $s, a \leftarrow first(PQueue)$
          $s', r \leftarrow Model(s,a)$
          $Q(s,a) \leftarrow Q(s,a) + \alpha [r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
          Repeat, for all $\bar{s}, \bar{a}$ predicted to lead to s:
            $\bar{r} \leftarrow$ predicted reward
            $p \leftarrow |\bar{r} + \gamma \max_a Q(s,a) - Q(\bar{s},\bar{a})|$.
            if $p > \theta$ then insert $\bar{s}, \bar{a}$ into PQueue with priority p
```

Figure 9.9: The prioritized sweeping algorithm for a deterministic environment.

Example 9.4 *Prioritized Sweeping on Mazes*. Prioritized sweeping has been found to dramatically increase the speed at which optimal solutions are found in maze tasks, often by a factor of 5 to 10. A typical example is shown in Figure 9.10. These data are for a sequence of maze tasks of exactly the same structure as the one shown in Figure 9.5, except varying in the grid resolution. Prioritized sweeping maintained a decisive advantage over unprioritized Dyna-Q. Both systems made at most **N=5** backups per environmental interaction. $\diamond$

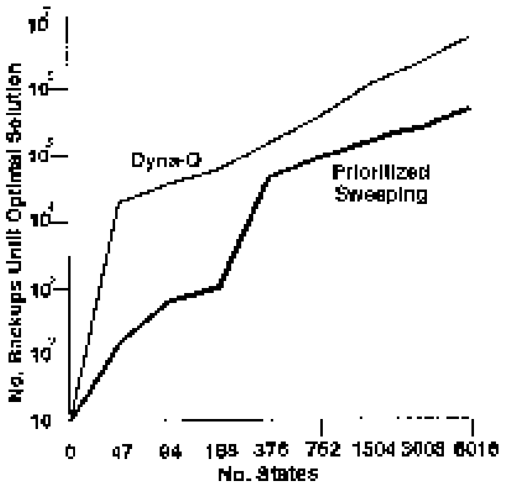


Figure 9.10: Prioritized sweeping significantly shortens learning time on the Dyna maze task for a wide range of grid resolutions. Reprinted from Peng and Williams (1993).

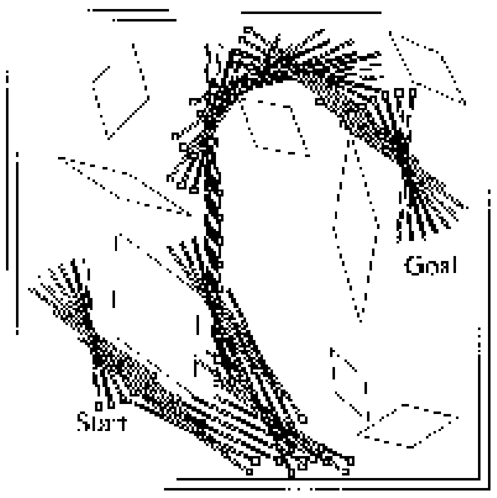


Figure 9.11: A rod maneuvering task and its solution by prioritized sweeping. The rod must be translated and rotated around the obstacles from start to goal in minimum number of steps. Reprinted from Moore and Atkeson (1993).

Example 9.5 *Rod Maneuvering*. The objective in this task is to maneuver a rod around some awkwardly placed obstacles to a goal position in the fewest number of steps (Figure 9.11). The rod can be translated along its long axis, perpendicular to that axis, or it can be rotated in either direction around its center. The distance of each movement was approximately 1/20th of the workspace, and the rotation increment was 10 degrees. Translations were deterministic and quantized to one of 20×20 positions. The figure shows the obstacles and the shortest solution from start to goal, found by prioritized sweeping. This problem is still deterministic, but has 4 actions and 14,400 potential states (some of these are unreachable because of the obstacles). This problem is probably too large to be solved with unprioritized methods. $\diamondsuit$

Prioritized sweeping is clearly a powerful idea, but the algorithms that have been developed so far appear not to extend easily to more interesting cases. The greatest problem is that the algorithms appear to rely on the assumption of discrete states. When a change occurs at one state, these methods perform a computation on all the predecessor states that may have been affected. If a function approximator is used to learn the model or the value function, then a single backup could influence a great many other states. It is not apparent how these states could be identified or processed efficiently. On the other hand, the general idea of focusing search on the states believed to have changed in value, and then on their predecessors, seems intuitively to be valid in general. Additional research may produce more general versions of prioritized sweeping.

Extensions of prioritized sweeping to stochastic environments are relatively straightforward. The model is maintained by keeping counts of the number of times each state-action pair has been experienced and of what the next states were. It is natural then to backup each pair not with a sample backup, as we have been using so far, but with a full backup, taking into account all possible next states and their probability of occurring.

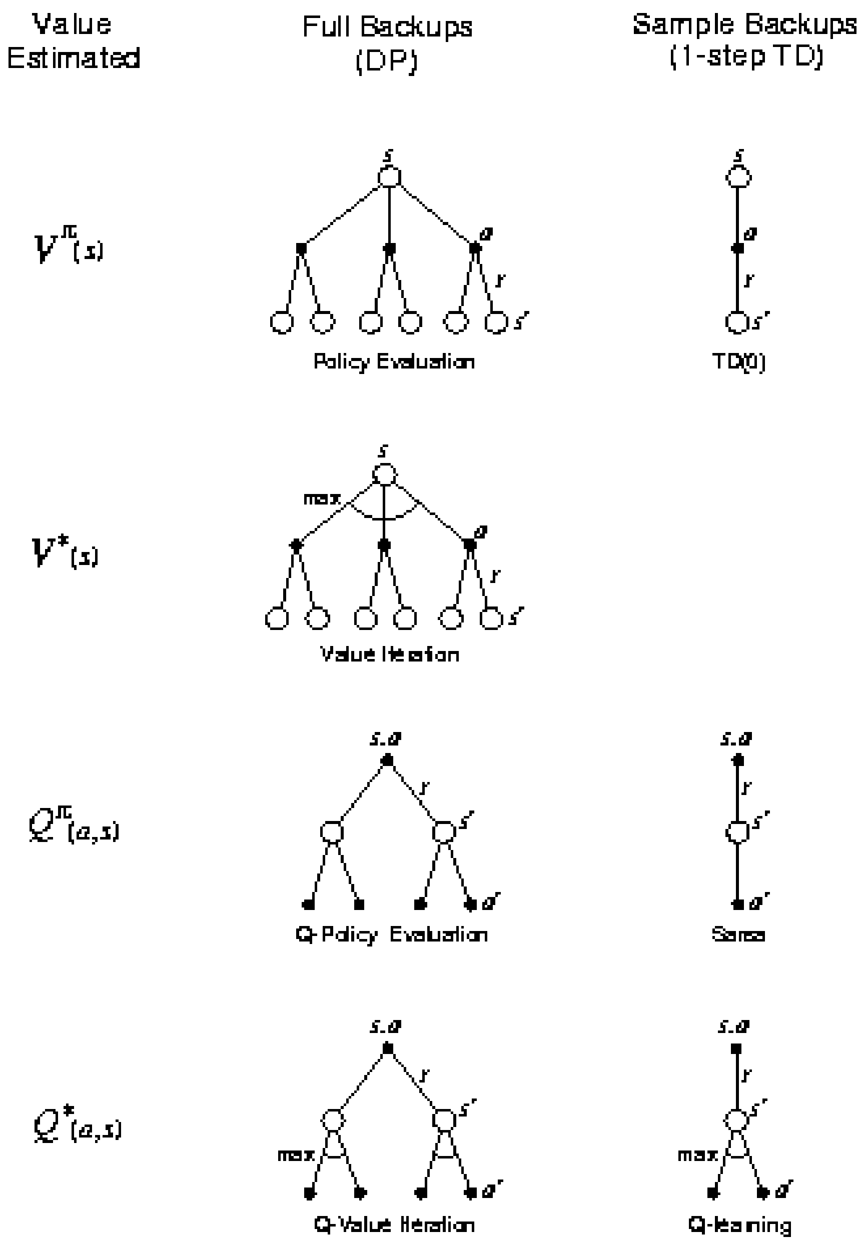
Next Up Previous
Next: 9.5 Full vs. Sample **Up:** 9 Planning and Learning **Previous:** 9.3 When the Model

---

Richard Sutton
Fri May 30 17:11:38 EDT 1997

9.5 Full vs. Sample Backups

The examples in the previous sections give some idea of the range of possibilities in combining methods for learning and for planning. In the rest of this chapter, we analyze some of the component ideas involved, starting with the relative advantages of full and sample backups.



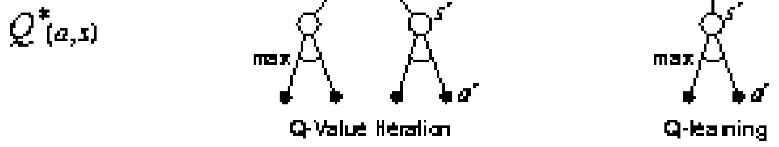


Figure 9.12: The 1-step backups.

Much of this book has been about different kinds of backups, and we have considered a great many varieties. Focusing for the moment on 1-step backups, they vary primarily along three binary dimensions. The first two dimensions are whether they back up state values or action values and whether they estimate the value for the optimal policy or for an arbitrary given policy. These two dimensions give rise to four classes of backups for approximating the four value functions, Q^*, V^*, Q^π, and V^π. The other binary dimension is whether the backups are *full* backups, considering all possible events that might happen, or *sample* backups, considering a single sample of what might happen. These three binary dimensions given rise to eight cases, seven of which correspond to specific algorithms, as shown in Figure 9.12. (The eighth case does not seem to correspond to any useful backup.) Any of these 1-step backups can be used in planning methods. The Dyna-Q agents discussed earlier use Q^* sample backups, but they could just as well use Q^* full backups, or either full or sample Q^π backups. The Dyna-AC system uses V^π sample backups together with a learning policy structure. For stochastic problems, prioritized sweeping is always done using one of the full backups.

When we introduced 1-step sample backups in Chapter 6 , we presented them as a substitute for full backups. In the absence of a distribution model, full backups are not possible, but sample backups can be done using sample transitions from the environment or a sample model. Implicit in that point of view is that full backups, if possible, are preferable to sample backups. But are they? Full backups certainly yield a better estimate because they are uncorrupted by sampling error, but they also require more computation, and computation is often the limiting resource in planning. To properly assess the relative merits of full and sample backups for planning we must control for their different computational requirements.

For concreteness, consider the full and sample backups for approximating Q^*, and the special case of discrete states and actions, a table-lookup representation of the approximate value function, **Q**, and a model in the form of estimated state-transition probabilities, $\hat{P}^a_{ss'}$, and expected rewards, $\hat{R}^a_{ss'}$. The full backup for a state-action pair, **s**,**a**, is:

$$Q(s,a) \leftarrow \sum_{s'} \hat{P}^a_{ss'} \left[\hat{R}^a_{ss'} + \gamma \max_{a'} Q(s',a') \right]. \qquad (9.1)$$

The corresponding sample backup for **s**,**a**, given a sample next state, s', is the Q-learning-like update:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\left[\hat{R}^a_{ss'} + \gamma \max_{a'} Q(s',a')\right], \qquad (9.2)$$

where α is the usual positive step-size parameter and the model's expected-value of the reward, $\hat{R}^a_{ss'}$ is used in place of the sample reward that is used in applying Q-learning without a model.

The difference between these full and sample backups is significant to the extent that the environment is stochastic, specifically, to the extent that, given a state and action, many possible next states may occur with various probabilities. If only one next state is possible, then the full and sample backups given above are identical (taking $\alpha = 1$). If there are many possible next states, then there may be significant differences. In favor of the full backup is that it is an exact computation, resulting in a new $Q(s,a)$ whose correctness is limited only by the correctness of the $Q(s',a')$ at successor states. The sample backup is in addition affected by sampling error. On the other hand, the sample backup is cheaper computationally because it considers only one next state, not all possible next states. In practice, the computation required by backup operations is usually dominated by the number of state-action pairs at which **Q** is evaluated. For a particular starting pair, **s,a**, let **b** be the *branching factor*, the number of possible next states, s', for which $\hat{P}^a_{ss'} > 0$. Then a full backup of this pair requires roughly **b** times as much computation as a sample backup.

If there is enough time to complete a full backup, then the resulting estimate is generally better than that of **b** sample backups because of the absence of sampling error. But if there is insufficient time to complete a full backup, then sample backups are always preferable because they at least make some improvement in the value estimate with fewer than **b** backups. In a large problem with many state-action pairs, we are often in the latter situation. With so many state-action pairs, full backups of all of them would take a very long time. Before that we may be much better off with a few sample backups at many state-action pairs than with full backups at a few pairs. Given a unit of computational effort, is it better devoted to a few full backups or to **b**-times as many sample backups?

Figure 9.13 shows the results of an analysis that suggests an answer to this question. It shows the estimation error as a function of computation time for full and sample backups for a variety of branching factors, **b**. The case considered is that in which all **b** successor states are equally likely and in which the error in the initial estimate is 1. The values at the next states are assumed correct, so the full backup reduces the error to zero upon its completion. In this case, sample backups reduce the error according to $\frac{1}{\sqrt{t}}\frac{b-1}{b}$ where **t** is the number of sample backups that have been performed (assuming sample averages, i.e., $\alpha = 1/t$). The key observation is that for moderately large **b** the error falls dramatically with a tiny fraction of **b** backups. For these cases, many state-action pairs could have their values improved dramatically, to within a few percent of the effect of a full backup, in the same time that one state-action pair could be backed up fully.

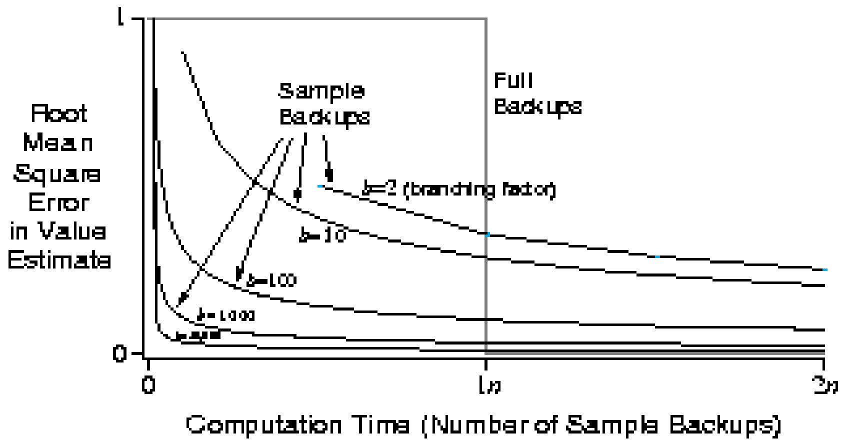


Figure 9.13: Comparison of efficiency of full and sample backups. This analysis is for backing up a single state-action pair with **b** equally likely successors, all of whose values are presumed correct.

The advantage of sample backups shown in Figure 9.13 is probably an underestimate of the real effect. In a real problem, the values of the successor states would themselves be estimates updated by backups. By causing estimates to be more accurate sooner, sample backups will have a second advantage in that the values backed up from the successor states will be more accurate. These results suggest that sample backups are likely to be superior to full backups on problems with large stochastic branching factors and too many states to be solved exactly.

Exercise 9.5

The analysis above assumed that all of the **b** possible next states were equally likely to occur. Suppose instead that the distribution was highly skewed, that some of the **b** states were much more likely to occur than most. Would this strengthen or weaken the case for sample backups over full backups. Support your answer.

Next Up Previous
Next: 9.6 Trajectory Sampling **Up:** 9 Planning and Learning **Previous:** 9.4 Prioritized Sweeping

Richard Sutton
Fri May 30 17:11:38 EDT 1997

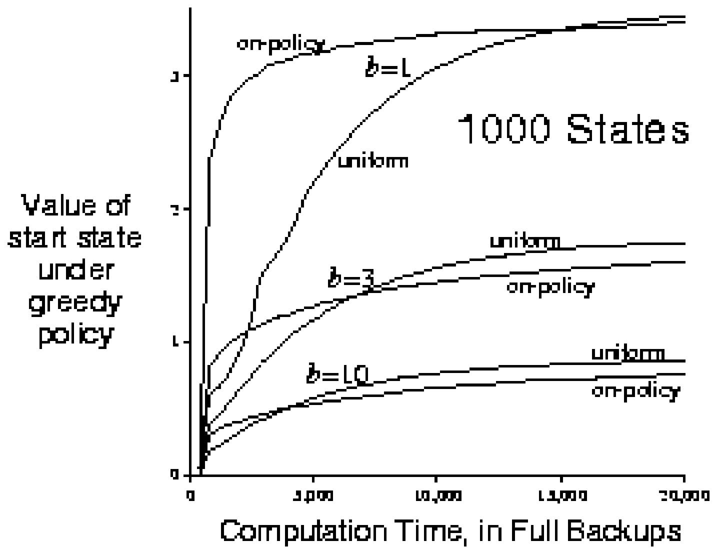
9.6 Trajectory Sampling

In this section we compare two ways of distributing backups. The classical approach from dynamic programming is to perform sweeps through the entire state (or state-action) space, backing up each state (or state-action pair) once per sweep. This is problematic on large tasks because there may not be time to complete even one sweep. In many tasks the vast majority of the states are irrelevant because they are visited only under very poor policies or with very low probability. Exhaustive sweeps implicitly devote equal time to all parts of the state space rather than focusing it where it is needed. As we discussed in Chapter 4, exhaustive sweeps and the equal treatment of all states that they imply are not necessary properties of dynamic programming. In principle, backups can be distributed any way one likes (to assure convergence, all states or state-action pairs must be visited in the limit an infinite number of times), but in practice exhaustive sweeps are often used.

The second approach is to sample from the state or state-action space according to some distribution. One could sample uniformly, as in the Dyna-Q agent, but this would suffer from some of the same problems as exhaustive sweeps. More appealing is to distribute backups according to the on-policy distribution, that is, according to the distribution observed when following the current policy. One advantage of this distribution is that it is easily generated; one simply interacts with the model, following the current policy. In an episodic task, one starts in the start state (or according to the starting-state distribution) and simulates until the terminal state. In a continual task, one starts anywhere and just keeps simulating. In either case, sample state transitions and rewards are given by the model, and sample actions are given by the current policy. In other words, one simulates explicit individual trajectories and performs backups at the state or state-action pairs encountered along the way. We call this way of generating experience and backups *trajectory sampling*.

It is hard to imagine any efficient way of distributing backups according to the on-policy distribution other than by trajectory sampling. If one had an explicit representation of the on-policy distribution, then one could sweep through all states, weighting the backup of each according to the on-policy distribution, but this leaves us again with all the computational costs of exhaustive sweeps. Possibly one could sample and update individual state-action pairs from the distribution, but, even if this could be done efficiently, what benefit would this provide over simulating trajectories? Even knowing the on-policy distribution in an explicit form is unlikely. The distribution changes whenever the policy changes, and computing the distribution requires computation comparable to a complete policy evaluation. Consideration of such other possibilities makes trajectory sampling seem both efficient and elegant.

Is the on-policy distribution of backups a good one? Intuitively it seems like a good choice, at least better than the uniform distribution. For example, if you are learning to play chess, you study positions that might arise in real games, not random positions of chess pieces. The latter may be valid states, but to be able to accurately value them is a very different skill from evaluating positions in real games. We also know from the previous chapter that the on-policy distribution has significant advantages when function approximation is used. At the current time this is the only distribution for which we can guarantee convergence with general linear function approximation. Whether or not function approximation is used, one might expect on-policy focusing to significantly improve the speed of planning.



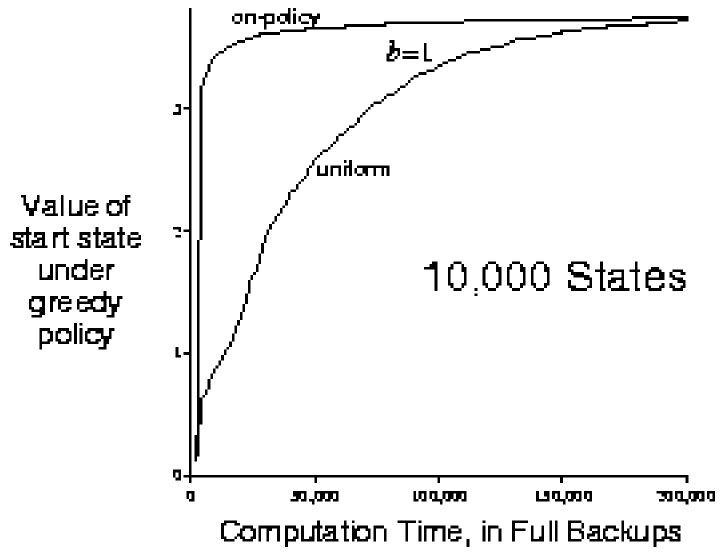


Figure 9.14: Relative efficiency of backups distributed uniformly across the state space versus focused on simulated on-policy trajectories. Results are for randomly generated tasks of two sizes and various branching factors, **b**.

Focusing on the on-policy distribution could be beneficial because it causes vast, uninteresting parts of the space to be ignored, or it could be detrimental because it causes the same old parts of the space to be backed up over and over. We conducted a small experiment to assess the effect empirically. To isolate the effect of the backup distribution, we used entirely 1-step full tabular backups, as defined by (9.1). In the *uniform* case, we cycled through all state-action pairs, backing up each in place, and in the *on-policy* case we simulated episodes, backing up each state-action pair that occurred under the current ϵ-greedy policy ($\epsilon = .1$). The tasks were undiscounted episodic tasks, generated randomly as follows. From each of the $|S|$ states, two actions were possible, each of which resulted in one of **b** next states, all equally likely, with a different random selection of **b** states for each state-action pair. The branching factor, **b**, was the same for all state-action pairs. In addition, on all transitions there was a 0.1 probability of transition to the terminal state, ending the episode. We used episodic tasks to get a clear measure of the quality of the current policy. At any point in the planning process one can stop and exhaustively compute $V^{\tilde{\pi}}(s_0)$, the true value of the start state under the greedy policy, $\tilde{\pi}$, given the current action-value function, **Q**, as an indication how well the agent would do on a new episode on which it acted greedily (all the while assuming the model is correct).

The upper part of Figure 9.14 shows results averaged over 200 sample tasks with 1000 states and branching factors of 1, 3, and 10. The quality of the policies found is plotted as a function of the number of full backups completed. In all cases, sampling according to the on-policy distribution resulted in faster planning initially and retarded planning in the long run. The effect was stronger, and the initial period of faster planning longer, at smaller branching factors. In other experiments, we found that these effects also became stronger as the number of states increased. For example, the lower part of Figure 9.14 shows results for a branching factor of 1 for tasks with 10,000 states. In this case the advantage of on-policy focusing is very large and long-lasting.

All of these results make sense. In the short term, sampling according to the on-policy distribution helps by focusing on states that are near descendants of the start-state. If there are many states and a small branching factor, this effect will be large and long lasting. In the long run, focusing on the on-policy distribution may hurt because the commonly occurring states all already have the correct values. Sampling them is useless, whereas sampling other states may actually perform some useful work. This presumably is why the exhaustive, unfocused approach does better in the long run, at least for small problems. These results are not conclusive because they are only for problems generated in a particular, random way, but they do suggest that sampling according to the on-policy distribution can be a great advantage for large problems, in particular, for problems in which a small subset of the state-action space is visited under the on-policy distribution.

Exercise 9.6

Some of the graphs in Figure 9.14 seem to be scalloped in their early portions, particularly the upper graph for **b=1** and the uniform distribution. Why do you think this is? What aspects of the data shown support your hypothesis?

Exercise 9.7 (programming)

If you have access to a moderately large computer, try replicating the experiment whose results are shown in the lower part of Figure 9.14. Then try the same experiment but with **b=3**. Discuss the meaning of your results.

Richard Sutton
Fri May 30 17:11:38 EDT 1997

9.7 Heuristic Search

The predominant state-space planning methods in artificial intelligence are collectively known as *heuristic search*. Although superficially different from the planning methods we have discussed so far in this chapter, heuristic search and some of its component ideas can be combined with these methods in useful ways. Unlike these methods, heuristic search is not concerned with changing the approximate, or ``heuristic,'' value function, but only with making improved action selections given the current value function. In other words, heuristic search is planning as part of a policy computation.

In heuristic search, for each state encountered, a large tree of possible continuations is considered. The approximate value function is applied to the leaf nodes and then backed up toward the current state at the root. The backing up within the search tree is just the same as in the max-backups (those for V^* and Q^*) discussed throughout this book. The backing up stops at the state-action nodes for the current state. Once the backed up values of these nodes are computed, the best of them is chosen as the current action, and then all backed up values are discarded.

In conventional heuristic search no effort is made to save the backed up values by changing the approximate value function. In fact, the value function is generally designed by people and never changed as a result of search. However, it is natural to consider allowing the value function to be improved over time, using either the backed-up values computed during heuristic search or any of the other methods presented throughout this book. In a sense we have taken this approach all along. Our greedy and ϵ-greedy action-selection methods are not unlike heuristic search, albeit on a smaller scale. For example, to compute the greedy action given a model and a state-value function, we must look ahead from each possible action to each possible next state, backup the rewards and estimated values, and then pick the best action. Just as in conventional heuristic search, this process computes backed-up values of the possible actions, but does not attempt to save them. Thus, heuristic search can be viewed as an extension of the idea of a greedy policy beyond a single step.

The point of searching deeper than one step is to obtain better action selections. If one has a perfect model and an imperfect action-value function, then in fact deeper search will usually yield better policies. Certainly, if the search is all the way to the end of the episode, then the effect of the imperfect value function is eliminated, and the action determined in this way must be optimal. If the search is of sufficient depth **k** such that γ^k is very small, then the actions will be correspondingly near optimal. On the other hand, the deeper the search, the more computation is required, usually resulting in a slower response time. A good example is provided by Tesauro's grandmaster-level backgammon player, TD-Gammon (Section 11 .1). This system used TD(λ) to learn an after-state value function through many games of self-play, using a form of heuristic search to make its moves. As a model, TD-Gammon used a priori knowledge of the probabilities of dice rolls and the assumption that the opponent always selected the actions that TD-Gammon rated as best for it. Tesauro found that the deeper the heuristic search, the better the moves made by TD-Gammon, but the longer it took to make each move. Backgammon has a large branching factor, yet moves must be made within a few seconds. It was only feasible to search ahead selectively a few steps, but even so the search resulted in significantly better action selections.

So far we have emphasized heuristic search as an action-selection technique, but this may not be its most important aspect. Heuristic search also suggests ways of selectively distributing backups that may lead to better and faster approximation of the optimal value function. A great deal of research on heuristic search has been devoted to making the search as efficient as possible. The search tree is grown very selectively, deeper along some lines and shallower along others. For example, the search tree is often deeper for the actions that seem most likely to be best, shallower for those that the agent will probably not want to take anyway. Can we use a similar idea to improve the distribution of backups? Perhaps it can be done by preferentially, updating state-action pairs whose values appears to be close to the maximum available from the state. To our knowledge, this and other possibilities for distributing backups based on ideas borrowed from heuristic search have not yet been explored.

We should not overlook the most obvious way in which heuristic search focuses backups: on the current state. Much of the effectiveness of heuristic search is due to its search tree being tightly focused on the states and actions that might immediately follow the current state. You may spend more of your life playing chess than checkers, but when you play checkers it pays to think about checkers and about your particular checkers position, your likely next moves, and successor positions. However you select actions, it is these states and actions that are of highest priority for backups and where you most urgently want your approximate value function to be accurate. Not only should your computation be preferentially devoted to imminent events, but so should your limited memory resources. In chess, for example, there are far too many possible positions to store distinct value estimates for each of them, but chess programs based on heuristic search can easily store distinct estimates for the millions of positions they encounter looking ahead from a single position. This great focusing of memory and computational resources on the current decision is presumably the reason why heuristic search can be so effective.

The distribution of backups can be altered in similar ways to focus on the current state and its likely successors. As a limiting case we might use exactly the methods of heuristic search to construct a search tree, and then perform the individual, 1-step backups from bottom up, as suggested by Figure 9.15. If the backups are ordered in this way and a table-lookup representation is used, then exactly the same backup would be achieved as in heuristic search. Any state-space search can be viewed in this way as the piecing together of a large number of individual 1-step backups. Thus, the performance improvement observed with deeper searches is not due to the use of multi-step backups as such. Instead, it is due to the focus and concentration of backups on states and actions immediately downstream from the current state. By devoting a large amount of computation specifically relevant to the candidate actions, a much better decision can be made than by relying on unfocused backups.

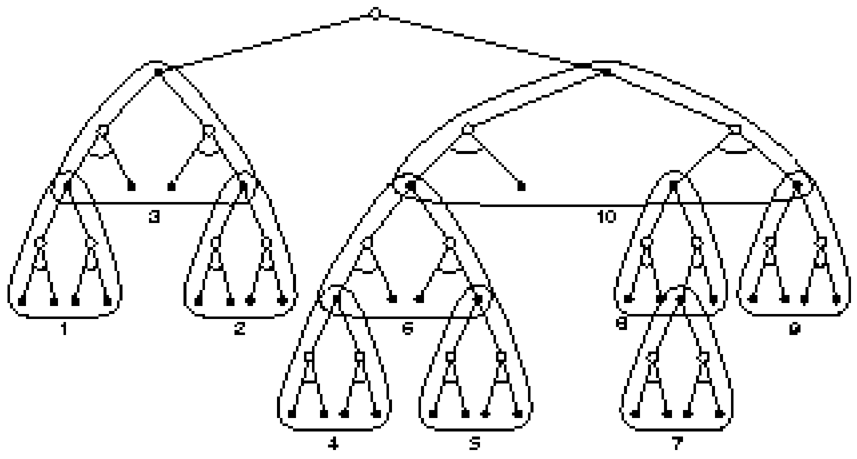


Figure 9.15: The deep backups of heuristic search can be implemented as a sequence of 1-step backups (shown here outlined). The ordering shown is for a selective depth-first search.

Next Up Previous

Next: 9.8 Summary **Up:** 9 Planning and Learning **Previous:** 9.6 Trajectory Sampling

Richard Sutton

Fri May 30 17:11:38 EDT 1997

9.8 Summary

We have presented a perspective emphasizing the surprisingly close relationships between planning optimal behavior and learning optimal behavior. Both involve estimating the same value functions, and in both cases it is natural to update the estimates incrementally, in a long series of small backup operations. This makes it straightforward to integrate learning and planning processes simply by allowing both to update the same estimated value function. In addition, any of the learning methods can be converted into planning methods simply by applying them to simulated (model generated) experience rather than to real experience. In this case learning and planning become even more similar; they are possibly identical algorithms operating on two different sources of experience.

It is straightforward to integrate incremental planning methods with acting and model learning. Planning, acting, and model learning interact in a circular fashion (Figure 9.2), each producing what the other needs to improve, but with no other interaction among them either required or prohibited. The most natural approach is for all processes to proceed asynchronously and in parallel. If the processes must share computational resources, then the division can be handled almost arbitrarily---by whatever organization is most convenient and efficient for the task at hand.

In this chapter we have touched upon a number of dimensions of variation among state-space planning methods. One of the most important of these is the distribution of backups i.e., of the focus of search. Prioritized sweeping focuses on the predecessors of states whose values have recently changed. Heuristic search applied to reinforcement learning focuses on the successors of the current state, and in other ways. Trajectory sampling is a convenient way of focusing on the on-policy distribution. All of these approaches can significantly speed planning and are current topics of research.

Another interesting dimension of variation is the size of backups. Among the smallest backups are 1-step sample backups. The smaller the backups, the more incremental the planning methods can be. We presented one study suggesting that sample backups may be preferable on very large problems. A related issue is the depth of backups. In many cases deep backups can be implemented as sequences of shallow backups.

Next Up Previous

Next: 9.9 Historical and Bibliographical **Up:** 9 Planning and Learning **Previous:** 9.7 Heuristic Search

Richard Sutton
Fri May 30 17:11:38 EDT 1997

9.9 Historical and Bibliographical Remarks

9.1

The overall view of planning and learning presented here has gradually developed over a number of years, in part by the authors (Sutton, 1990, 1991a, 1991b; Barto, Bradtke, and Singh, 1991, 1995; Sutton and Pinette, 1985; Barto and Sutton, 1981b), but also strongly influenced by Agre and Chapman (1990; Agre 1988), Bertsekas and Tsitsiklis (1989), Singh (1994), and others. The authors were also strongly influenced by psychological studies of latent learning (Tolman, 1932) and by psychological views of the nature of thought (e.g., Galanter and Gerstenhaber, 1956; Craik, 1943; Campbell, 1959; Dennett, 1978).

9.2 and 9 .3

The terms *direct* and *indirect*, which we use to describe different kinds of reinforcement learning, are from the adaptive control literature (e.g., Goodwin and Sin, 1984), where they are used to make the same kind of distinction. The term *system identification* is used in adaptive control for what we call *model learning* (e.g., Goodwin and Sin, 1984; Ljung and Söderstrom, 1983; Young, 1984). The Dyna architecture is due to Sutton (1990), and the results in these sections are based on results reported there.

9.4

Prioritized sweeping was developed simultaneously and independently by Moore and Atkeson (1993) and Peng and Williams (1993). The results in Figure 9.10 are due to Peng and Williams (1993). The results in Figure 9.11 are due to Moore and Akteson.

9.5

This section was strongly influenced by the experiments of Singh (1994).

9.7

For further reading on heuristic search, the reader is encouraged to consult texts and surveys such as those by Russell and Norvig (1995) and Korf (1988). Peng and Williams (1993) explored a forward focusing of backups much as is suggested in this section.

Richard Sutton
Fri May 30 17:11:38 EDT 1997

...policies.

There are interesting exceptions to this. See, e.g., Pearl (1984).

Richard Sutton
Fri May 30 17:11:38 EDT 1997

10 Dimensions

In this book we have tried to present reinforcement learning not as a collection of individual methods, but as a coherent set of ideas cutting across methods. Each idea can be viewed as a dimension along which methods vary. The set of such dimensions spans a very large space of possible methods. By exploring this space at the level of dimensions we hope to obtain the broadest and most lasting understanding. In this chapter we use the concept of dimensions in method space to recapitulate the view of reinforcement learning we have developed in this book and to identify some of the more important gaps in our coverage of the field.

10.1 The Unified View

All of the reinforcement learning methods we have explored in this book have three key ideas in common. First, the objective of all of them is the estimation of value functions. Second, all operate by backing up values along actual or possible state trajectories. Third, all follow the general strategy of generalized policy iteration (GPI), meaning that they maintain an approximate value function and an approximate policy, and they continually try to improve each on the basis of the other. These three ideas that the our methods have in common circumscribe the subject covered in this book. We suggest that value functions, backups, and GPI are powerful organizing principles potentially relevant to any model of intelligence.

Two of the most important dimensions along which the methods vary are shown in Figure 10.1. These dimensions have to do with the kind of backup used to improve the value function. The vertical dimension is whether they are sample backups (based on a sample trajectory) or full backups (based on a distribution of possible trajectories). Full backups of course require a model, whereas sample backups can be done either with or without a model (another dimension of variation). The horizontal dimension corresponds to the depth of backups, i.e., to the degree of bootstrapping. At three of the four corners of the space are the three primary methods for estimating values: DP, TD, and Monte Carlo methods. Along the lower edge of the space are the sample-backup methods, ranging from 1-step TD backups to full-return Monte Carlo backups. Between these is a spectrum including methods based on n -step backups and mixtures of n -step backups such as the λ -backups implemented by eligibility traces.

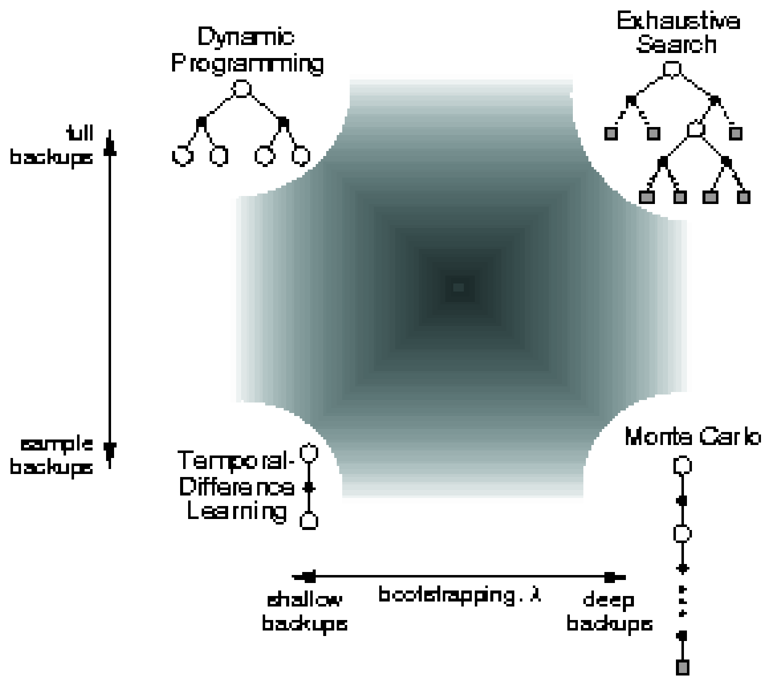


Figure 10.1: A slice of the space of reinforcement learning methods.

DP methods are shown in the extreme upper-left corner of the space because they involve 1-step full backups. The upper-right corner is the extreme case of full backups so deep that they run all the way to terminal states (or, in a continual problem, until discounting has reduced the contribution of any further rewards to a negligible level). This is the case of exhaustive search. Intermediate methods along this dimension include heuristic search and related methods that search and backup up to a limited depth, perhaps selectively. There are also methods that are intermediate along the vertical dimension. These include methods that mix full and sample backups, such as Dyna, as well as the possibility of methods that mix samples and distributions within a single backup. The interior of the square is filled in to represent the space of all such intermediate methods.

A third important dimension is that of function approximation. Function approximation can be viewed as an orthogonal spectrum of possibilities ranging from tabular methods at one extreme through state aggregation, a variety of linear methods, and then a diverse set of nonlinear methods. This third dimension might be visualized as perpendicular to the plane of the page in Figure 10.1.

Another dimension that we heavily emphasized in this book is the binary distinction between on-policy and off-policy methods. In the former case, the agent learns the value function for the policy it is currently following, whereas in the latter case it learns the value function for the policy that it currently thinks is best. These two policies are often different because of the need to explore. The interaction between this dimension and the bootstrapping and function approximation dimension discussed in Chapter 8 illustrates the advantages of analyzing the space of methods in terms of dimensions. Even though this did involve an interaction between three dimensions, many other dimensions were found to be irrelevant, greatly simplifying the analysis and increasing its significance.

In addition to the four dimensions just discussed, we have identified a number of others throughout the book, including:

definition of return
: Is the task episodic or continual, discounted or undiscounted?

action values vs. state values
: What kind of values should be estimated? If only state values are estimated, then either a model or a separate policy (as in actor-critic methods) is required for action selection.

action selection/exploration
: How are actions selected to ensure a suitable tradeoff between exploration and exploitation? We have considered only the simplest ways to do this: ϵ -greedy and softmax action selection, and optimistic initialization of values.

synchronous vs. asynchronous
: Are the backups for all states performed simultaneously or one-by-one in some order?

replacing vs. accumulating traces
: If eligibility traces are used, which kind is most appropriate?

real vs. simulated
: Should one backup real experience or simulated experience? If both, how much of each?

location of backups
: What states or state-actions pairs should be backed up? Model-free methods can choose only among the states and state-action pairs actually encountered, but model-based methods can choose arbitrarily. There are many potent possibilities here.

timing of backups
: Should backups be done as part of selecting actions, or only afterward?

Of course, these dimensions are neither exhaustive nor mutually exclusive. Individual algorithms differ in many other ways as well, and many algorithms lie in several places along several dimensions. For example, Dyna methods use both real and simulated experience to affect the same value function. It is also perfectly sensible to maintain multiple value functions computed in different ways or over different state and action representations. These dimensions do, however, constitute a coherent set of ideas for describing and exploring a wide space of possible algorithms.

Richard Sutton
Fri May 30 18:01:42 EDT 1997

Next Up Previous
Next: 11 Case Studies **Up:** 10 Dimensions **Previous:** 10.1 The Unified View

10.2 Other Frontier Dimensions

Much research remains to be done within this space of reinforcement learning methods. For example, even for the tabular case no control method using multi-step backups has been proved to converge to an optimal policy. Among planning methods, basic ideas such as trajectory sampling and selective backups are almost completely unexplored. On closer inspection, parts of the space will undoubtedly turn out to have far greater complexity and internal structure than is now apparent. There are also other dimensions along which reinforcement learning can be extended that we have not yet mentioned, leading to a much larger space of methods. Here we identify some of these dimensions and note some of the open questions and frontiers that have been left out of the preceding chapters.

One of the most important extensions of reinforcement learning beyond what we have treated in this book is to eliminate the requirement that the state representation has the Markov property. There are a number of interesting approaches to the non-Markov case. Most strive to construct from the given state signal and its past values a new signal that is Markov, or more nearly Markov. For example, one approach is based on the theory of partially-observable MDPs (POMDPs). POMDPs are finite MDPs in which the state is not observable, but another ``sensation'' signal stochastically related to the state is observable. The theory of POMDPs has been extensively studied for the case of complete knowledge of the dynamics of the POMDP. In this case, Bayesian methods can be used to compute at each time step the probability of the environment being in each state of the underlying MDP. This probability distribution can then be used as a new state signal for the original problem. The downside for the Bayesian POMDP approach is its computational expense and its strong reliance on complete environment models. Some of the recent work pursuing this approach is by Littman, Cassandra and Kaelbling (1995), Parr and Russell (1995), and Chrisman (1992).

If we are not willing to assume a complete model of a POMDP's dynamics, then existing theory seems to offer little guidance. Nevertheless, one can still attempt to construct a Markov state signal from the sequence of sensations. A variety of statistical and ad hoc methods along these lines have been explored (e.g., McCallum, 1992, 1993, 1995; Lin and Mitchell, 1992; Chapman and Kaelbling, 1991; Moore, 1994; Rivest and Schapire, 1987; Colombetti and Dorigo, 1994; Whitehead and Ballard, 1991; Hochreiter and Schmidhuber, 1997).

All of the above methods involve constructing an improved state representation from the non-Markov one provided by the environment. Another approach is to leave the state representation unchanged and use methods that are not too adversely affected by its being non-Markov (e.g., Singh, Jaakkola and Jordan, 1994, 1995; Jaakkola, Singh and

Jordan, 1995). In fact, most function approximation methods can be viewed in this way. For example, state aggregation methods for function approximation are in effect equivalent to a non-Markov representation in which all members of a set of states are mapped into a common sensation. There are other parallels between the issues of function approximation and non-Markov representations. In both cases the overall problem divides into two parts: constructing an improved representation, and making do with the current representation. In both cases the ``making do'' part is relatively well understood, whereas the constructive part is unclear and wide open. At this point we can only guess as to whether or not these parallels point to any common solution methods for the two problems.

Another important direction for extending reinforcement learning beyond what we have covered in this book is to incorporate ideas of modularity and hierarchy. Introductory reinforcement learning is about learning value functions and one-step models of the dynamics of the environment. But much of what people learn does not seem to fall exactly into either of these categories. For example, consider what we know about tying our shows, making a phone call, or traveling to London. Having learned how to do such things, we are then able to choose and plan among them as if they were primitive actions. What we have learned in order to do this are not conventional value functions or one-step models. We are able to plan and learn at a variety of levels and flexibly interrelate them. Much of our learning appears not to be about learning values directly, but about preparing us to quickly estimate values later in response to new situations or new information. Considerable reinforcement learning research has been directed at capturing such abilities (e.g., Watkins, 1989; Dayan and Hinton, 1993; Singh, 1992a,b; Ring, 1994, Kaelbling, 1993; Sutton, 1995).

Researchers have also explored ways of using the structure of particular tasks to advantage. For example, many problems have state representations that are naturally lists of variables, like the readings of multiple sensors, or actions that are lists of component actions. The independence or near independence of some variables from others can sometimes be exploited to obtain more efficient special forms of various reinforcement learning algorithms. It might even be possible to decompose a problem into several independent subproblems that can be solved by separate learning agents. An abstract reinforcement learning problem can usually be structured in many different ways, some reflecting natural aspects of the problem, such as the existence of physical sensors, and others being the result of explicit attempts to decompose the problem into simpler subproblems. Possibilities for exploiting structure in reinforcement learning and related planning problems have been studied by many researchers (e.g., Boutilier, Dearden and Goldszmidt, 1995; Dean and Lin, 1995). There are also related studies of multi-agent or distributed reinforcement learning (e.g., Littman, 1994; Markey, 1994; Crites and Barto, 1996; Tan, 1993).

Finally, we want to emphasize that reinforcement learning is meant to be a *general* approach to learning from interaction. It is general enough not to require special purpose teachers and domain knowledge, but also general enough to utilize such things if they are available. For example, it is often possible to accelerate reinforcement learning by giving advice or hints to the agent (Clouse and Utgoff, 1992; Maclin and Shavlik, 1994) or by demonstrating instructive behavioral trajectories (Lin, 1992). Another way to make learning easier, related to ``shaping'' in psychology, is to give the learning agent a series of relatively easy problems building up to the harder problem of ultimate interest (e.g., Selfridge, Sutton, and Barto, 1985). These methods, and others not yet developed, have the potential to give the machine learning terms ``training'' and ``teaching'' new meanings that are closer to their meanings for animal and human learning.

Next Up Previous
Next: 11 Case Studies **Up:** 10 Dimensions **Previous:** 10.1 The Unified View

---

Richard Sutton
Fri May 30 18:01:42 EDT 1997

11 Case Studies

In this final chapter we briefly present a few case studies of reinforcement learning. Several of these are substantial applications of potential economic significance. One, Samuel's checkers player, is primarily of historical interest. Our presentations are intended to illustrate some of the tradeoffs and issues that arise in real applications. For example, we emphasize how domain knowledge is incorporated into the formulation and solution of the problem. We also highlight the representation issues that are so often critical to successful applications. The algorithms used in some of these case studies are also substantially more complex than those we have presented in the rest of the book. Applications of reinforcement learning are still far from routine and typically require as much art as science. Making applications easier and more straightforward is one of the goals of current research in reinforcement learning.

Richard Sutton
Fri May 30 18:04:45 EDT 1997

11.1 TD-Gammon

One of the most impressive applications of reinforcement learning to date is that by Gerry Tesauro's to the game of backgammon (Tesauro, 1992, 1994, 1995). Tesauro's program, *TD-Gammon*, required little backgammon knowledge and yet learned to play extremely well, near the level of the world's strongest grandmasters. The learning algorithm in TD-Gammon was a straightforward combination of the TD(λ) algorithm and nonlinear function approximation using a multi-layer neural network trained by backpropagating TD errors.

Backgammon is a major game in the sense that it is played throughout the world, with numerous tournaments and regular world championship matches. It is in part a game of chance, and it is a popular vehicle for waging significant sums of money. There are probably more professional backgammon players than there are professional chess players. The game is played with 15 white and 15 black pieces on a board of 24 locations, called *points*. Figure 11.1 shows a typical position early in the game, seen from the perspective of the white player.

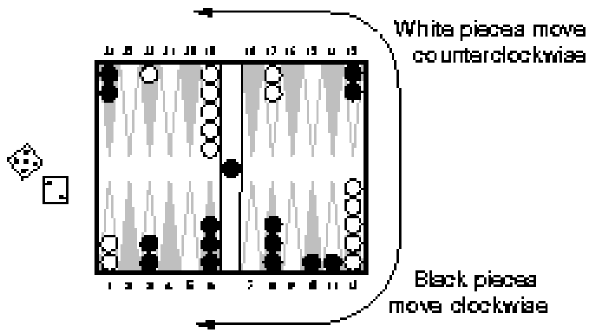

White pieces move counterclockwise
Black pieces move clockwise

Figure 11.1: A Backgammon Position

In this figure, white has just rolled the dice and obtained a 5 and a 2. This means that he can move one of his pieces 5 steps and one (possibly the same piece) 2 steps. For example, he could move two pieces from the 12 point, one to the 17 point, and one to the 14 point. White's objective is to advance all of his pieces into the last quadrant (points 19-24) and then off the board. The first player to remove all his pieces wins. One complication is that the pieces interact as they pass each other going in different directions. For example, if it were black's move in Figure 11.1, he could use the dice roll of 2 to move a piece from the 24 point to the 22 point, ``hitting'' the white piece there. Pieces that have been hit are placed on the ``bar'' in the middle of the board (where we already see one previously hit black piece), from whence they re-enter the race from the start. However, if there are two pieces on a point, then the opponent cannot move to that point; the pieces are protected from being hit. Thus, white cannot use his 5-2 dice roll to move either of his pieces on the 1 point, because their possible resulting points are occupied by groups of black pieces. Forming contiguous blocks of occupied points to block the opponent is one of the elementary strategies of the game.

Backgammon involves several further complications, but the above description gives the basic idea. With 30 pieces and 24 possible locations (26, counting the bar and off-the-board) it should be clear that the number of possible backgammon positions is enormous, far more than the number of memory elements one could have in any physically realizable computer. The number of moves possible from each position is also very large. For a typical dice roll there might be 20 different ways of playing. In considering future moves, such as the response of the opponent, one must consider the possible dice rolls as well. The result is that the game tree has an effective branching factor of about 400. This is far too large to permit effective use of the conventional heuristic search methods that have proved so effective in games like chess and checkers.

On the other hand, the game is a good match to the capabilities of TD learning methods. Although the game is highly stochastic, a complete description of the game's state is available at all times. The game evolves over a sequence of moves and positions until finally ending in a win for one player or the other, ending the game. The outcome can be interpreted as a final reward to be predicted. On the other hand, the theoretical results we have described so far cannot be usefully applied to this task. The number of states is so large that a lookup table cannot be used, and the opponent is a source of uncertainty and time variation.

TD-Gammon used a nonlinear form of TD(λ) . The estimated value, $V_t(s)$, of any state (board position) **s** was meant to estimate of the probability of winning starting from state **s**. To achieve this, rewards were defined as zero for all time steps except those on which the game is won. To implement the value function, TD-Gammon used a standard multi-layer neural network, much as shown in Figure 11.2. (The real network actually had two additional units in its final layer to estimate the probability of each player winning in a special way called a ``gammon'' or ``backgammon.'') The network consisted of a layer of input units, a layer of hidden units, and a final output unit. The input to the network was a representation of a backgammon position, and the output was an estimate of the value of that position.

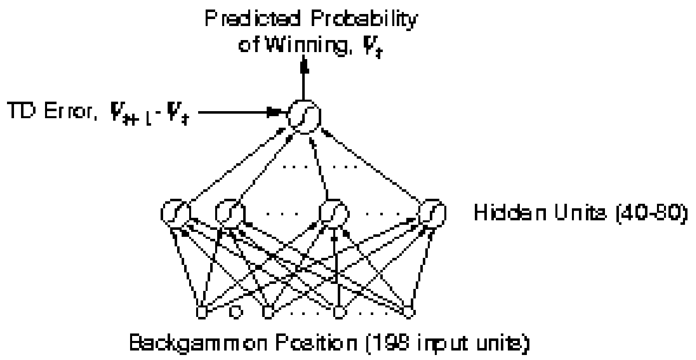

Predicted Probability of Winning, V_t
TD Error, $V_{t+1} - V_t$
Hidden Units (40-80)
Backgammon Position (198 input units)

Figure 11.2: The neural network used in TD-Gammon

In the first version of TD-Gammon, TD-Gammon 0.0, backgammon positions were represented to the network in a relatively direct way that involved little backgammon knowledge. It did, however, involve substantial knowledge of how neural networks work and how information is best presented to them. It is instructive to note the exact representation Tesauro chose. There were a total of 198 input units to the network. For each point on the backgammon board, 4 units indicated the number of white pieces on the point. If there were no white pieces, then all 4 units took on the value zero. If there was one piece, then the first unit took on the value 1. If there were two pieces, then both the first and the second unit were 1. If there were three or more pieces on the point, then all of the first three units were 1. If there were more than three pieces, the fourth unit also came on, to a degree indicating the number of additional pieces beyond three. Letting **n** denote the total number of pieces on the point, if **n > 3**, then the fourth unit took on the value $(n-3)/2$. With four units for white and four for black at each of the 24 points, that made a total of 192 units. Two additional units encoded the number of white and black pieces on the bar (each took on the value $n/2$, where **n** is the number of pieces on the bar) and two more encoded the number of black and white pieces already successfully removed from the board (these took on the value $n/15$, where **n** is the number of pieces already borne off). Finally, two units indicated in a binary fashion whether it was white's or black's turn to move. The general logic behind these choices should be clear. Basically, Tesauro tried to represent the position in a very straightforward way, making little attempt to minimize the number of units. He provided one unit for each conceptually distinct possibility that seemed likely to be relevant, and he scaled them to roughly the same range, in this case between 0 and 1.

Given a representation of a backgammon position, the network computed its estimated value in the standard way. Corresponding to each connection from an input unit to a hidden unit was a real-valued weight. Signals from each input unit were multiplied by their corresponding weights and summed at the hidden unit. The output, $h(j)$, of hidden unit **j** was a nonlinear ``sigmoid" function of the weighted sum:

$$h(j) = \sigma\left(\sum_i w_{ij}\phi(i)\right) = \frac{1}{1+e^{-\sum_i w_{ij}\phi(i)}},$$

where $\phi(i)$ is the value of the **i**th input unit and w_{ij} is the weight of its connection to the **j**th hidden unit. The output of the sigmoid is always between 0 and 1, and has a natural interpretation as a probability based on a summation of evidence. The computation from hidden units to the output unit was entirely analogous. Each connection from a hidden unit to the output unit had a separate weight. The output unit formed the weighted sum and then passed it through the same sigmoid nonlinearity.

TD-Gammon used the gradient-descent form of the TD(λ) algorithm described in Section 8.2, with the gradients computed by the error backpropagation algorithm (Rumelhart, Hinton, and Williams, 1986). Recall that the general update rule for this case is

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha\Big[r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)\Big]\vec{e}_t, \qquad (11.1)$$

where $\vec{\theta}_t$ is the vector of all modifiable parameters (in this case, the weights of the network) and $\vec{e}_t$ is a vector of eligibility traces, one for each component of $\vec{\theta}_t$, updated by

$$\vec{e}_t = \gamma\lambda\vec{e}_{t-1} + \nabla_{\vec{\theta}_t} V_t(s_t),$$

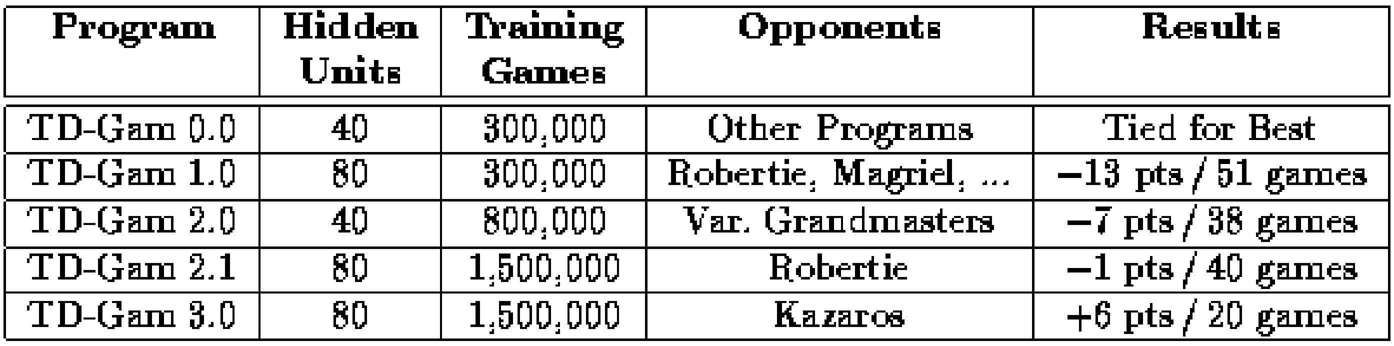
with $\vec{e}_0 = \vec{0}$. The gradient in this equation can be computed efficiently by the backpropagation procedure. For the backgammon application, in which $\gamma = 1$ and the reward is always zero except upon winning, the TD-error portion of the learning rule is usually just $V_t(s_{t+1}) - V_t(s_t)$, as suggested in Figure 11.2.

To apply the learning rule we need a source of backgammon games. Tesauro obtained an unending sequence of games by playing his learning backgammon player against itself. To choose its moves, TD-Gammon considered each of the 20 or so ways it could play its dice role and the corresponding positions that would result. The resulting positions are *after state* as discussed Section 6 .8 . The network was consulted to estimate each of their values. The move was then selected that would lead to the position with the highest estimated value. Continuing in this way, with TD-Gammon making the moves for both sides, it was possible to easily generate large numbers of backgammon games. Each game was treated as an episode, with the sequence of positions acting as the states, $s_0, s_1, s_2, \ldots$. Tesauro applied the nonlinear TD rule (11.1) fully incrementally, that is, after each individual move.

The weights of the network were set initially to small random values. The initial evaluations were thus entirely arbitrary. Since the moves were selected on the basis of these evaluations, the initial moves were inevitably poor, and the initial games often lasted hundreds or thousands of moves before one side or the other won, almost by accident. After a few dozen games however, performance improved rapidly.

After playing about 300,000 games against itself, TD-Gammon 0.0 as described above learned to play approximately as well as the best previous backgammon computer programs. This was a striking result because all the previous high-performance computer programs had used extensive backgammon knowledge. For example, the reigning champion program at the time was, arguably, *Neurogammon*, another program written by Tesauro that used a neural network but not TD learning. Neurogammon's network was trained on a large training corpus of exemplary moves provided by backgammon experts, and, in addition, started with a set of features specially crafted for backgammon. Neurogammon was a highly tuned, highly effective backgammon program that decisively won the World Backgammon Olympiad in 1989. TD-Gammon 0.0, on the other hand, was constructed with essentially zero backgammon knowledge. That it was able to do as well as Neurogammon and all other approaches is striking testimony to the potential of self-play learning methods.

The tournament success of TD-Gammon 0.0 with zero backgammon knowledge suggested an obvious modification: add the specialized backgammon features but keep the self-play TD learning method. This produced TD-Gammon 1.0. TD-Gammon 1.0 was clearly substantially better than all previous backgammon programs and only found serious competition among human experts. More recent versions of the program, TD-Gammon 2.0 (40 hidden units) and TD-Gammon 2.1 (80 hidden units), were also augmented with a selective two-ply search procedure. To select moves, these programs looked ahead not just to the positions that would immediately result, but also to the opponent's possible dice rolls and moves. Assuming the opponent always took the move that appeared immediately best for him, the expected value of each candidate move was computed and the best was selected. To save computer time, the second ply of search was conducted only for candidate moves that were ranked highly after the first ply, about 4 or 5 moves on average. Two-ply search affected only the moves selected; the learning process proceeded exactly as before. The most recent version of the program, TD-Gammon 3.0, uses 160 hidden units and a selective three-ply search. TD-Gammon illustrates the combination of learned value functions and decide-time search as in heuristic search methods. In more recent work, Tesauro and Galperin (1997) have begun exploring trajectory sampling methods as an alternative to search.



Program	Hidden Units	Training Games	Opponents	Results
TD-Gam 0.0	40	300,000	Other Programs	Tied for Best
TD-Gam 1.0	80	300,000	Robertie, Magriel, ...	−13 pts / 51 games
TD-Gam 2.0	40	800,000	Var. Grandmasters	−7 pts / 38 games
TD-Gam 2.1	80	1,500,000	Robertie	−1 pts / 40 games
TD-Gam 3.0	80	1,500,000	Kazaros	+6 pts / 20 games

Table 11.1: Summary of TD-Gammon Results

Tesauro was able to play his programs in a significant number of games against world-class human players. A summary of the results is given in Table 11.1. Based on these results and analyses by backgammon grandmasters (Robertie, 1992; see Tesauro, 1995) TD-Gammon 3.0 appears to be at, or very near, the playing strength of the best human players in the world. It may already be the world champion. These programs have also already changed the way the best human players play the game. For example, TD-Gammon learned to play certain opening positions differently than was the convention among the best human players. Based on TD-Gammon's success and further analysis, the best human players now play these positions as TD-Gammon does (Tesauro, 1995).

Next Up Previous

Next: 11.2 Samuel's Checkers Player **Up:** 11 Case Studies **Previous:** 11 Case Studies

Richard Sutton
Fri May 30 18:04:45 EDT 1997

11.2 Samuel's Checkers Player

An important precursor to Tesauro's TD-Gammon was the seminal work of Arthur Samuel in constructing programs for learning to play checkers. Samuel was one of the first to make effective use of heuristic search methods and of what we would now call temporal-difference learning. His checkers players are instructive case studies in addition to being of historical interest. We emphasize the relationship of Samuel's methods to modern reinforcement learning methods and try to convey some of Samuel's motivation for using them.

Samuel first wrote a checkers-playing program for the IBM 701 in 1952. His first *learning* program was completed in 1955 and demonstrated on television in 1956. Later versions of the program achieved good, though not expert, playing skill. Samuel was attracted to game playing as a domain for studying machine learning because games are less complicated than problems ``taken from life'' while still allowing fruitful study of how heuristic procedures and learning can be used together. He chose to study checkers instead of chess because its relative simplicity made it possible to focus more strongly on learning.

Samuel's programs played by performing a look-ahead search from each current position. They used what we now call heuristic search methods to determine how to expand the search tree and when to stop searching. The terminal board positions of each search were evaluated, or ``scored,'' by a value function, or ``scoring polynomial,'' using linear function approximation. In this and other respects Samuel's work seems to have been inspired by the suggestions of Shannon (1950). In particular, Samuel's program was based on Shannon's minimax procedure to find the best move from the current position. Working backwards through the search tree from the scored terminal positions, each position was given the score of the position that would result from the best move, assuming that the machine would always try to maximize the score, while the opponent would always try to minimize it. Samuel called this the *backed-up score* of the position. When the minimax procedure reached the search tree's root---the current position---it yielded the best move under the assumption that the opponent would be using the same evaluation criterion, shifted to its point of view. Some versions of Samuel's programs used sophisticated search control methods analogous to what are known as ``alpha-beta'' cutoffs (e.g., see Pearl, 1984).

Samuel used two main learning methods, the simplest of which he called *rote learning*. It consisted simply of saving a description of each board position encountered during play together with its backed-up value determined by the minimax procedure. The result was that if a position that had already been encountered were to occur again as a terminal position of a search tree, the depth of the search was effectively amplified since's this position's stored value cached the results of one or more searches conducted earlier. One initial problem was that the program was not encouraged to move along the most direct path to a win. Samuel gave it a ``a sense of direction'' by decreasing a position's value a small amount each time it was backed up a level (called a ply) during the mini-max analysis. ``If the program is now faced with a choice of board positions whose scores differ only by the ply number, it will automatically make the most advantageous choice, choosing a low-ply alternative if winning and a high-ply alternative if losing'' (Samuel, 1959, p. 80). Samuel found this discounting-like technique essential to successful learning. He found that rote learning produced slow but continuous improvement that was most effective for opening and end-game play. His program became a ``better-than-average novice'' after learning from many games against itself, a variety of human opponents, and from book games in a supervised learning mode.

Rote learning and other aspects of Samuel's work strongly suggest the essential idea of temporal-difference learning---that the value of a state should equal the value of likely later states starting from that state. Samuel came closest to this idea in his second learning method, his ``learning by generalization'' procedure for modifying the parameters of the value function. Samuel's method was the same in concept as that used much later by Tesauro in TD-Gammon. He played his program many games against another version of itself and performed a backup operation after each move. The idea of Samuel's backup is suggested by the diagram in Figure 11.3. Each open circle represents a position where the program moves next, an *on-move* position, and each solid circle represents a position where the opponent moves next. A backup was made to the value of each on-move position after a move by each side, resulting in a second on-move position. The backup was toward the minimax value of a search launched from the second on-move position. Thus, the overall effect was that of a backup consisting of one full move of real events and then a search over possible events, as suggested by Figure 11.3. Samuel's actual algorithm was significantly more complex than this for computational reasons, but this was the basic idea.

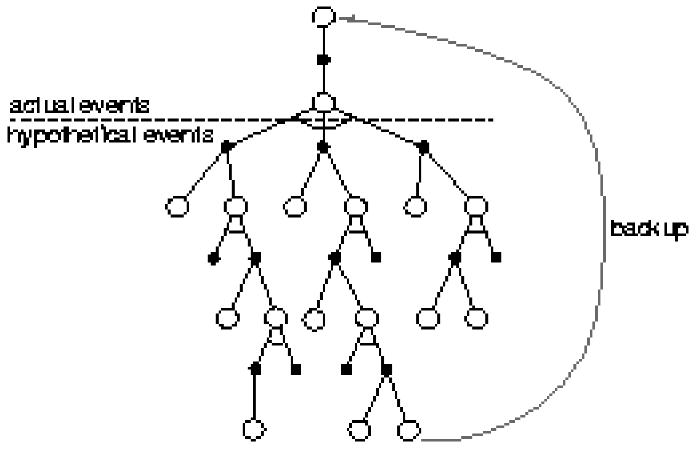


Figure 11.3: The backup diagram for Samuel's checkers player. Open circles represent positions with the program to move, dark circles positions with the opponent to move.

Samuel did not include explicit rewards. Instead, he fixed the weight of the most important feature, the *piece advantage* feature, which measured the number of pieces the program had relative to how many its opponent had, giving higher weight to kings, and including refinements so that it was better to trade pieces when winning than when losing. Thus, the goal of Samuel's program was to improve its piece advantage, which in checkers is highly correlated with winning.

However, Samuel's learning method may have been missing an essential part of a sound temporal-difference algorithm. Temporal-difference learning can be viewed as a way of making a value function consistent with itself, and this we can clearly see in Samuel's method. But also needed is a way of tying the value function to the true value of the states. We have enforced this via rewards and by discounting or giving a fixed value to the terminal state. But Samuel's method included no rewards and no special treatment of the terminal positions of games. As Samuel himself pointed out, his value function could have become consistent merely by giving a constant value to all positions. Samuel hoped to discourage such solutions by giving his piece advantage term a large, non-modifiable weight. But although this may decrease the likelihood of finding useless evaluation functions, it does not prohibit them. For example, a constant function could still be attained by setting the modifiable weights so as to cancel the effect of the non-modifiable one.

If Samuel's learning procedure was not constrained to find useful evaluation functions, then it should have been possible for it to become worse with experience. In fact, Samuel reported observing this during extensive self-play training sessions. To get the program improving again, Samuel had to intervene and set the weight with the largest absolute value back to zero. His interpretation was that this drastic intervention jarred the program out of local optima, but another possibility is that it jarred the program out of evaluation functions that were consistent but had little to do with winning or losing the game.

Despite these potential problems, Samuel's checkers player using the generalization learning method approached ``better-than-average'' play. Fairly good amateur opponents characterized it as ``tricky but beatable'' (Samuel, 1959). In contrast to the rote-learning version, this version was able to develop a good middle game but remained weak in opening and end-game play. This program also included an ability to search through sets of features to find those that were most useful in forming the value function. A later version (Samuel, 1967) included refinements in its search procedure, such as alpha-beta pruning, extensive use of a supervised learning mode called ``book learning,'' and hierarchical lookup tables called signature tables (Griffith, 1966) to represent the value function instead of linear function approximation. This version learned to play much better than the 1959 program, though still not at a master level. Samuel's checkers-playing program was widely recognized as a significant achievement in artificial intelligence and machine learning (see, e.g., Gardner, 1981).

Next Up Previous
Next: 11.3 The Acrobot **Up:** 11 Case Studies **Previous:** 11.1 TD-Gammon

Richard Sutton
Fri May 30 18:04:45 EDT 1997

11.3 The Acrobot

Reinforcement learning has been applied to a wide variety of physical control tasks, both real and simulated. One such task is the *acrobot*, a two-link, under-actuated robot roughly analogous to a gymnast swinging on a highbar (Figure 11.4). The first joint (corresponding to the gymnast's hands on the bar) cannot exert torque, but the second joint (corresponding to the gymnast bending at the waist) can. The system has four continuous state variables: two joint positions and two joint velocities. The equations of motion are given in Figure 11.5. This system has been widely studied by control engineers (e.g., Spong, 1994) and machine learning researchers (e.g., Dejong and Spong, 1994; Boone, 1997).

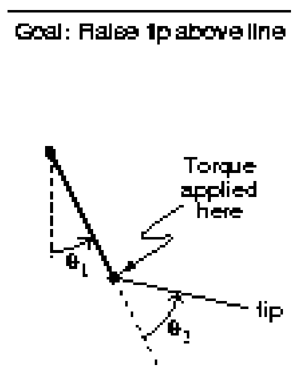


Figure 11.4: The acrobot.

$$
\begin{aligned}
\ddot{\theta}_1 &= -d_1^{-1}(d_2\ddot{\theta}_2 + \phi_1) \\
\ddot{\theta}_2 &= \left(m_2 l_{c2}^2 + I_2 - \frac{d_2^2}{d_1}\right)^{-1}\left(\tau + \frac{d_2}{d_1}\phi_1 - \phi_2\right) \\
d_1 &= m_1 l_{c1}^2 + m_2(l_1^2 + l_{c2}^2 + 2l_1 l_{c2}\cos\theta_2) + I_1 + I_2 \\
d_2 &= m_2(l_{c2}^2 + l_1 l_{c2}\cos\theta_2) + I_2 \\
\phi_1 &= -m_2 l_1 l_{c2}\dot{\theta}_2^2 \sin\theta_2 - 2m_2 l_1 l_{c2}\dot{\theta}_2\dot{\theta}_1 \sin\theta_2 \\
&\quad + (m_1 l_{c1} + m_2 l_1) g \cos(\theta_1 - \pi/2) + \phi_2 \\
\phi_2 &= m_2 l_{c2} g \cos(\theta_1 + \theta_2 - \pi/2)
\end{aligned}
$$

Figure 11.5: The equations of motions of the simulated acrobot. A time step of 0.05 seconds was used in the simulation, with actions chosen after every four time steps. The

torque applied at the second joint is denoted by $\tau \in \{+1, -1, 0\}$. There were no constraints on the joint positions, but the angular velocities were limited to $\dot{\theta}_1 \in [-4\pi, 4\pi]$ and $\dot{\theta}_2 \in [-9\pi, 9\pi]$. The constants were $m_1 = m_2 = 1$ (masses of the links), $l_1 = l_2 = 1$ (lengths of links), $l_{c1} = l_{c2} = 0.5$ (lengths to center of mass of links), $I_1 = I_2 = 1$ (moments of inertia of links), and $g = 9.8$ (gravity).

One objective for controlling the acrobot is to swing the tip (the ``feet") up above the first joint by an amount equal to one of the links in minimum time. In this task, the torque applied at the second joint is limited to three choices, positive torque of a fixed magnitude, negative torque of the same magnitude, or no torque. A reward of **-1** is given on all time steps until the goal is reached, which ends the episode. No discounting is used ($\gamma = 1$). Thus, the optimal value, $V^*(s)$, of any state, **s**, is the minimum time to reach the goal (an integer number of steps) starting from **s**.

Sutton (1996) addressed the acrobot swing-up task in an online, model-free context. Although the acrobot was simulated, the simulator was not available for use by the agent/controller in any way. The training and interaction was just as if a real, physical acrobot had been used. Each episode began with both links of the acrobot hanging straight down and at rest. Torques were applied by the reinforcement learning agent until the goal was reached, which always eventually happened. Then the acrobot was restored to its initial rest position and a new episode was begun.

The learning algorithm used was Sarsa(λ) with linear function approximation, tile coding, and replacing traces as in Figure 8 .8 . With a small, discrete action set, it is natural to use a separate set of tilings for each action. The next choice is of the continuous variables with which to represent the state. A clever designer would probably represent the state in terms of the angular position and velocity of the center of mass and of the second link, which might make the solution simpler and consistent with broad generalization. But since this was just a test problem, a more naive, direct representation was used in terms of the positions and velocities of the links: $\theta_1, \dot{\theta}_1, \theta_2$, and $\dot{\theta}_2$. The two angles are restricted to a limited range by the physics of the acrobot (see Figure 11.5) and the two angles are naturally restricted to $[0, 2\pi]$. Thus, the state space in this task is a bounded rectangular region in four dimensions.

This leaves the question of what tilings to use. There are many possibilities, as discussed in Section 8 .3.2 . One is to use a complete grid, slicing the four-dimensional space along all dimensions, and thus into many small four-dimensional tiles. Alternatively, one could slice along just one of the dimensions, making hyper-planar stripes. In this case one has to pick which dimension to slice along. And of course in all cases one has to pick the width of the slices, the number of tilings of each kind, and, if there are multiple tilings, how to offset them. One could also slice along pairs or triplets of dimensions to get other tilings. For example, if one expected the velocities of the two links to strongly interact in their effect on value, one might make many tilings that sliced along both of these dimensions. If one thought the region around zero velocity was particularly critical, the slices could be more closely spaced there.

Sutton used tilings that sliced in a variety of simple ways. Each of the four dimensions was divided into six equal intervals. A seventh interval was added to the angular velocities so that tilings could be offset by a random fraction of an interval in all dimensions (see Section 8 .3.2). Of the total of 48 tilings, 12 sliced along all four dimensions as discussed above, dividing the space into $6 \times 7 \times 6 \times 7 = 1764$ tiles each. Another 12 tilings sliced along three dimensions (3 randomly-offset tilings each for each of the 4 sets of three dimensions), and another 12 sliced along two dimensions (2 tilings for each of the 6 sets of two dimensions. Finally, a set of 12 tilings depended each on only one dimension (3 tilings for each of the 4 dimensions). This resulted in a total of approximately **25,000** tiles for each action. This number is small enough that hashing was not necessary. All tilings were offset by a random fraction of an interval in all relevant dimensions.

The remaining parameters of the learning algorithm were $\alpha = 0.2/48$, $\lambda = 0.9$, $\epsilon = 0$, and $Q_0 = 0$. The use of a greedy policy ($\epsilon = 0$) seemed preferable on this task because long sequences of correct actions are needed to do well. One exploratory action could spoil a whole sequence of good actions. Exploration was ensured instead by starting the action values optimistically, at the low value of 0. As discussed in Section 2.7 and Example 8.2, this makes the agent continually disappointed with whatever rewards it initially experiences, driving it to keep trying new things.

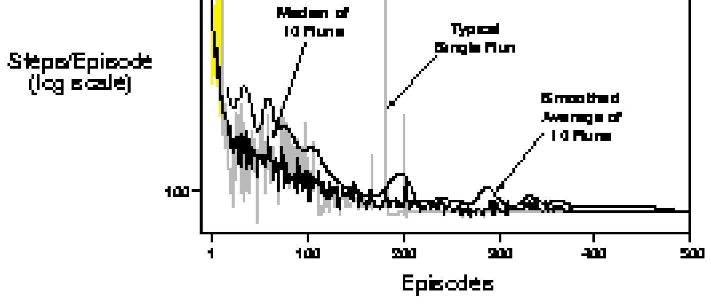


Figure 11.6: Learning curves for Sarsa(λ) on the Acrobot task.

Figure 11.6 shows learning curves for the acrobot task and the learning algorithm described above. Note from the single-run curve that single episodes were sometimes extremely long. On these episodes, the acrobot was usually spinning repeatedly at the second joint while the first joint changed only slightly from vertical down. Although this often happened for many time steps, it always eventually ended as the action values were driven lower. All runs eventually ended with an efficient policy for solving the problem, usually lasting about 75 steps. A typical final solution is shown in Figure 11.7. First the acrobot pumps back and forth several times symmetrically, with the second link always down. Then, once enough energy has been added to the system, the second link is swung upright and stabbed to the goal height.

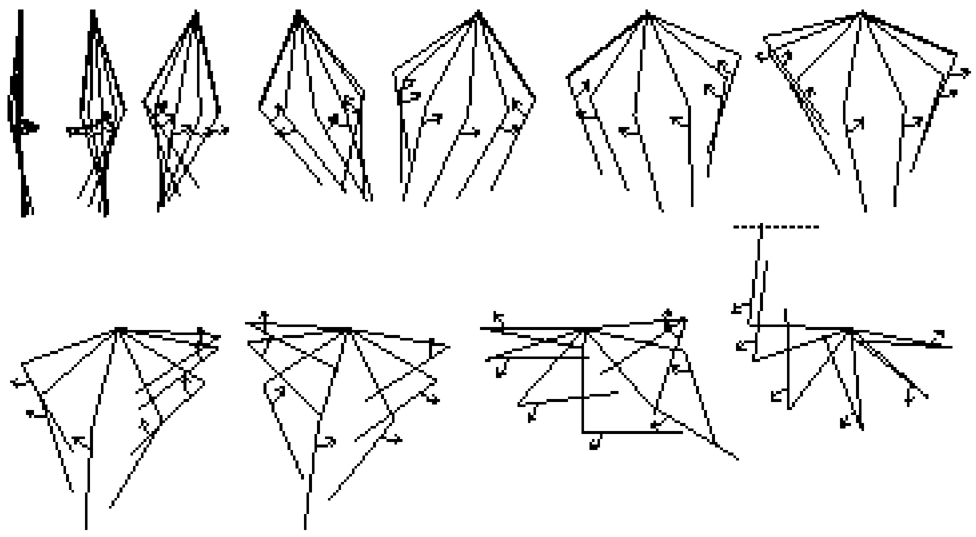


Figure 11.7: A typical learned behavior of the acrobot. Each group is a series of consecutive positions, the thicker line being the first. The arrow indicates the torque applied at the second joint.

Richard Sutton
Fri May 30 18:04:45 EDT 1997

11.4 Elevator Dispatching

Waiting for an elevator is a situation with which we are all familiar. We press a button and then wait for an elevator to arrive traveling in the right direction. We may have to wait a long time if there are too many passengers or not enough elevators. Just how long we wait depends on the dispatching strategy the elevators use to decide where to go. For example, if passengers on several floors have requested pickups, which should be served first? If there are no pickup requests, how should the elevators distribute themselves to await the next one? Elevator dispatching is a good example of a stochastic optimal control problem of economic importance that is too large to solve by classical techniques such as dynamic programming.

Crites and Barto (1995; Crites, 1996) studied the application of reinforcement learning techniques to the four-elevator, ten-floor system shown Figure 11.8. Along the righthand side are pickup requests and an indication of how long each has been waiting. Each elevator has a position, direction, and speed, plus a set of buttons of its own to indicate where passengers want to get off. Roughly quantizing the continuous variables, Crites and Barto estimated that the system has over 10^{22} states. This large state set rules out classical dynamic programming methods such as value iteration. Even if one state could be backed up every microsecond it would still require over 1000 years to complete just one sweep through the state space.

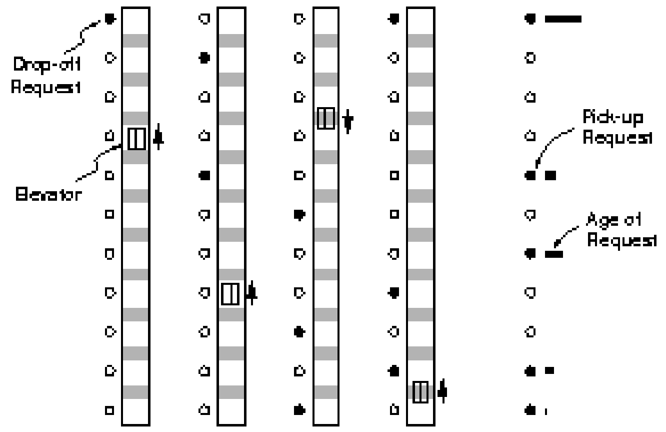


Figure 11.8: Four Elevators in a Ten-Story Building.

In practice, modern elevator dispatchers are designed heuristically and evaluated on simulated buildings. The simulators are quite sophisticated and detailed. The physics of each elevator car is modeled in continuous time with continuous state variables. Passenger arrivals are modeled as discrete, stochastic events, with arrival rates varying frequently over the course of a simulated day. Not surprisingly, the times of greatest traffic and greatest challenge to the dispatching algorithm are the morning and evening rush hours. Dispatchers are generally designed primarily for these difficult periods.

The performance of elevator dispatchers is measured in several different ways, all with respect to an average passenger entering the system. The average *waiting time* is how long the passenger waits before getting on an elevator, and the average *system time* is how long the passenger waits before being dropped off at the destination floor. Another frequently encountered statistic is the percentage of passengers whose waiting time exceeds 60 seconds. The objective that Crites and Barto focused on is the average *squared waiting time*. This objective is commonly used because it tends to keep the waiting times low while also encouraging fairness in serving all the passengers.

Crites and Barto applied a version of 1-step Q-learning augmented in several ways to take advantage of special features of the problem. The most important of these concerned the formulation of the actions. First, each elevator made its own decisions independently of the others. Second, a number of constraints were placed on the decisions. An elevator carrying passengers could not pass by a floor if any of its passengers wanted to get off there, nor could it reverse direction until all of its passengers wanting to go in its current direction had reached their floors. In addition, a car was not allowed to stop at a floor unless someone wanted to get on or off there, and it could not stop to pick up passengers at a floor if another elevator was already stopped there. Finally, given a choice between moving up or down, the elevator was constrained always to move up (otherwise evening rush hour traffic would tend to push all the elevators down to the lobby). These last three constraints were explicitly provided to provide some prior knowledge and make the problem easier. The net result of all these constraints was that each elevator had to make few and simple decisions. The only decision that had to be made was whether or not to stop at a floor that was being approached and that has passengers waiting to be picked up. At all other times, no choices needed to be made.

That each elevator made choices only at infrequent times permitted a second simplification of the problem. As far as the learning agent was concerned, the system made discrete jumps from one time at which it had to make a decision to the next. When a continuous-time decision problem is treated as a discrete-time system in this way it is known as a *semi-Markov* decision process. To a large extent, such processes can be treated just like any other Markov decision process by taking the reward on each discrete transition as the integral of the reward over the corresponding continuous-time interval. The notion of return generalizes naturally from a discounted sum of future rewards to a discounted *integral* of future rewards:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \qquad \text{becomes} \qquad R_t = \int_0^{\infty} e^{-\beta\tau} r_{t+\tau}\, d\tau,$$

where r_t on the left is the usual immediate reward in discrete time and $r_{t+\tau}$ on the right is the instantaneous reward at continuous time $t+\tau$. In the elevator problem the continuous-time reward is the negative of the sum of the squared waiting times of all waiting passengers. The parameter $\beta > 0$ plays a role similar to that of the discount-rate parameter $\gamma \in [0,1)$.

The basic idea of the extension of Q-learning to semi-Markov decision problems can now be explained. Suppose the system is in state **s** and takes action **a** at time t_1, and then the next decision is required at time t_2 in state s'. After this discrete-event transition, the semi-Markov Q-learning backup for a tabular action-value function, **Q**, would be:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[\int_{t_1}^{t_2} e^{-\beta(\tau - t_1)} r_\tau \, d\tau + e^{-\beta(t_2 - t_1)} \min_{a'} Q(s',a') - Q(s,a) \right].$$

Note how $e^{-\beta(t_2-t_1)}$ acts as a variable discount factor that depends on the amount of time between events. This method is due to Bradtke and Duff (1995).

One complication is that the reward as defined---the negative sum of the squared waiting times---is not something that would normally be known while an actual elevator was running. This is because in a real elevator system one does not know how many people are waiting at a floor, only how long the button requesting a pickup on that floor has been pressed. Of course this information is known in a simulator, and Crites and Barto used it to obtain their best results. They also experimented with another technique that used only information that would be known in an online learning situation with a real set of elevators. In this case one can use how long each button has been pushed together with an estimate of the arrival rate to compute an *expected* summed squared waiting time for each floor. Using this in the reward measure proved nearly as effective as using the actual summed squared waiting time.

For function approximation, a nonlinear neural network trained by backpropagation was used to represent the action-value function. Crites and Barto experimented with a wide variety of ways of representing states to the network. After much exploration, their best results were obtained using networks with 47 input units, 20 hidden units, and 2 output units, one for each action. The way the state was encoded by the input units was found to be critical to the effectiveness of the learning. The 47 input units were as follows:

- 18 units: Two units encoded information about each of the nine down hall buttons. A real-valued unit encoded the elapsed time if the button had been pushed, and a binary unit was on if the button had not been pushed.
- 16 units: A unit for each possible location and direction for the car whose decision was required. Exactly one of these units was on at any given time.
- 10 units: The location of the other elevators superimposed over the 10 floors. Each elevator had a ``footprint" that depended on its direction and speed. For example, a stopped elevator caused activation only on the unit corresponding to its current floor, but a moving elevator caused activation on several units corresponding to the floors it was approaching, with the highest activations on the closest floors. No information was provided about which one of the other cars was at a particular location.
- 1 unit: This unit was on if the elevator whose decision was required was at the highest floor with a passenger waiting.
- 1 unit: This unit was on if the elevator whose decision was required was at the floor with the passenger who had been waiting for the longest amount of time.
- 1 unit: Bias unit was always on.

Two architectures were used. In ``RL1," each elevator was given its own action-value function and its own neural network. In ``RL2," there was only one network and one action-value function, with the experiences of all four elevators contributing to learning in the one network. In both cases, each elevator made its decisions independently of the other elevators, but shared a single reward signal with them. This introduced additional stochasticity as far as each elevator was concerned because its reward depended in part on the actions of the other elevators, which it could not control. In the architecture in which each elevator had its own action-value function, it was possible for different elevators to learn different specialized strategies (although in fact they tended to learn the same strategy). On the other hand, the architecture with a common action-value function could learn faster because it learned simultaneously from the experiences of all elevators. Training time was an issue here, even though the system was trained in simulation. The reinforcement learning methods were trained for about four days of computer time on a 100 mips processor (corresponding to about 60,000 hours of simulated time). While this is a considerable amount of computation, it is negligible compared to what would be required by any conventional dynamic programming algorithm.

The networks were trained by simulating a great many evening rush hours while making dispatching decisions using the developing, learned action-value functions. Crites and Barto used the Gibbs softmax procedure to select actions as described in Section 2 .3 , reducing the ``temperature" gradually over training. A temperature of zero was used during test runs on which the performance of the learned dispatchers was assessed.

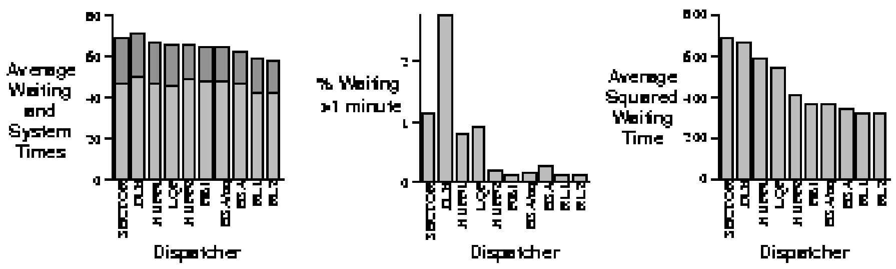


Figure 11.9: Comparison of elevator dispatchers. The ``SECTOR'' dispatcher is similar to what is used in many actual elevator systems. The ``RL1'' and ``RL2'' dispatchers were constructed through reinforcement learning.

Figure 11.9 shows the performance of several dispatchers during a simulated evening rush hour, what researchers call *down-peak* traffic. The dispatchers include methods similar to those commonly used in the industry, a variety of heuristic methods, sophisticated research algorithms that repeatedly run complex optimization algorithms on-line (Bao et al., 1994), and dispatchers learned using the two reinforcement learning architectures. By all of the performance measures, the reinforcement learning dispatchers compare very favorably with the others. Although the optimal policy for this problem is unknown, and the state-of-the-art is difficult to pin down because details of commercial dispatching strategies are proprietary, these learned dispatchers appeared to perform very well.

Next Up Previous
Next: 11.5 Dynamic Channel Allocation **Up:** 11 Case Studies **Previous:** 11.3 The Acrobot

---

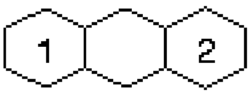
Richard Sutton
Fri May 30 18:04:45 EDT 1997

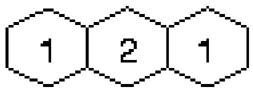
11.5 Dynamic Channel Allocation

An important problem in the operation of a cellular telephone system is how to efficiently use the available bandwidth to provide good service to as many customers as possible. This problem is becoming critical with the rapid growth in the use of cellular telephones. Here we describe a study due to Singh and Bertsekas (1996) in which they applied reinforcement learning to this problem.

Mobile telephone systems take advantage of the fact that a communication channel---a band of frequencies---can be used simultaneously by many callers if these callers are spaced physically far enough apart that their calls do not interfere with one another. The minimum distance at which there is no interference is called the *channel reuse constraint*. In a cellular telephone system, the service area is divided into a number of regions called cells. In each cell is a base station that handles all the calls made within the cell. The total available bandwidth is divided permanently into a number of channels. Channels must then be allocated to cells and to calls made within cells without violating the channel reuse constraint. There are a great many ways to do this, some of which are better than others in terms of how reliably they make channels available to new calls, or to calls that are ``handed off'' from one cell to another as the caller crosses a cell boundary. If no channel is available for a new or a handed-off call, the call is lost, or *blocked*. Singh and Bertsekas considered the problem of allocating channels so that the number of blocked calls is minimized.

A simple example provides some intuition about the nature of the problem. Imagine a situation with three cells sharing two channels. The three cells are arranged in a line where no two adjacent cells can use the same channel without violating the channel reuse constraint. If the left cell is serving a call on channel 1 while the right cell is serving another call on channel 2, as in the left diagram below, then any new call arriving in the middle cell must be blocked.





Obviously, it would be better for both the left and the right cells to use channel 1 for their calls. Then a new call in the middle cell could be assigned channel 2, as in the right diagram, without violating the channel reuse constraint. Such interactions and possible optimizations are typical of the channel assignment problem. In larger and more realistic cases with many cells, channels, and calls, and uncertainty about when and where new calls will arrive or existing calls will have to be handed off, the problem of allocating channels to minimize blocking can become extremely complex.

The simplest approach is to permanently assign channels to cells in such a way that the channel reuse constraint can never be violated even if all channels of all cells are used simultaneously. This is called a *fixed assignment* (FA) method. In a *dynamic assignment* method, in contrast, all channels are potentially available to all cells and are assigned to cells dynamically as calls arrive. If this is done right, it can take advantage of temporary changes in the spatial and temporal distribution of calls in order to serve more users. For example, when calls are concentrated in a few cells, these cells can be assigned more channels without increasing the blocking rate in the lightly used cells.

The channel assignment problem can be formulated as a semi-Markov decision process much as the elevator dispatching problem was in the previous section. A state in the semi-MDP formulation has two components. The first is the configuration of the entire cellular system giving for each cell the usage state (occupied or unoccupied) of each channel for that cell. A typical cellular system with 49 cells and 70 channels has a staggering 70^{49} configurations, ruling out the use of conventional dynamic programming methods. The other state component is an indicator of what kind of event caused a state transition: arrival, departure, or handoff. This state component determines what kinds of actions are possible. When a call arrives, the possible actions are to assign it a free channel or to block it if no channels are available. When a call departs, that is, when a caller hangs up, the system is allowed to reassign the channels in use in that cell in an attempt to create a better configuration. At time **t** the immediate reward, r_t, is the number of calls taking place at that time, and the return is

$$R_t = \int_0^\infty e^{-\beta\tau} r_{t+\tau}\, d\tau,$$

where $\beta > 0$ plays a role similar to that of the discount-rate parameter γ. Maximizing the expectation of this return is the same as minimizing the expected (discounted) number of calls blocked over an infinite horizon.

This is another problem greatly simplified if treated in terms of after states (Section 6 .8). For each state and action, the immediate result is a new configuration, an after state. A value function is learned over just these configurations. To select among the possible actions, the resulting configuration was determined and evaluated. The action was then selected that would lead to the configuration of highest estimated value. For example, when a new call arrived at a cell, it could be assigned to any of the free channels, if there were any; otherwise, it had to be blocked. The new configuration that would result from each assignment was easy to compute because it was always a simple deterministic consequence of the assignment. When a call terminated, the newly released channel became available for reassigning to any of the ongoing calls. In this case, the actions of reassigning each ongoing call in the cell to the newly released channel were considered. An action was then selected leading to the configuration with the highest estimated value.

Linear function approximation was used for the value function: the estimated value of a configuration was a weighted sum of features. Configurations were represented by two sets of features: an availability feature for each cell and a packing feature for each cell-channel pair. For any configuration, the availability feature for a cell gave the number of additional calls it could accept without conflict if the rest of the cells were frozen in the current configuration. For any given configuration, the packing feature for a channel-cell pair gave the number of times that channel was being used in that configuration within a four-cell radius of that cell. All of these features were normalized to lie between **-1** and 1. A semi-Markov version of linear TD(0) was used to update the weights.

Singh and Bertsekas compared three channel-allocation methods using a simulation of a 7 by 7 cellular array with 70 channels. The channel reuse constraint was that calls had to be 3 cells apart to be allowed to use the same channel. Calls arrived at cells randomly according to Poisson distributions possibly having different means for different cells, and call durations were determined randomly by an exponential distribution with a mean of 3 minutes. The methods compared were a fixed allocation method (FA), a dynamic allocation method called ``Borrowing with Directional Channel Locking'' (BDCL) due to Zhang and Yum (1989), and the reinforcement learning method (RL). BDCL was the best dynamic channel allocation method they found in the literature. It is a heuristic method that assigns channels to cells as in FA, but channels can be borrowed from neighboring cells when needed. It orders the channels in each cell and uses this ordering to determine which channels to borrow and how calls are dynamically reassigned channels within a cell.

Figure 11.10 shows the blocking probabilities of these methods for mean arrival rates of 150, 200, 300 calls/hour as well as for a case in which different cells had different mean arrival rates. The reinforcement learning method learned online. The data shown are for its asymptotic performance, but in fact learning was very rapid. The RL method blocked calls less frequently than did the other methods for all arrival rates and very soon after starting to learn. Note that the differences between the methods decreased as the call-arrival rate increased. This is to be expected because as the system gets saturated with calls there are fewer opportunities for a dynamic allocation method to set up favorable usage patterns. In practice, however, it is the performance of the unsaturated system that is most important. For marketing reasons, cellular telephone systems are built with enough capacity that more than 10% blocking is very rare.

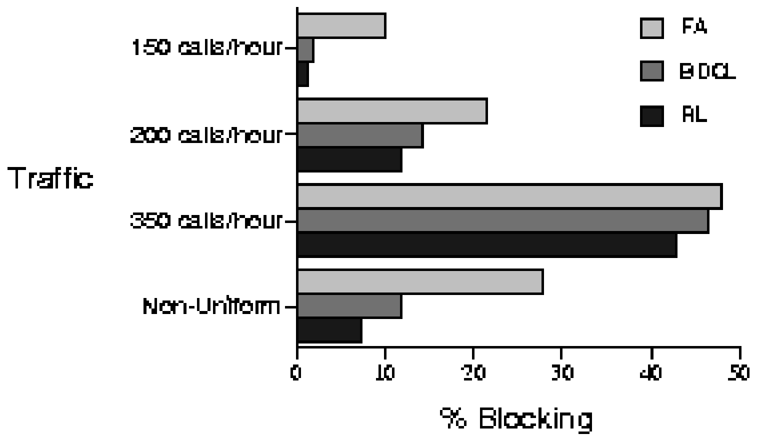


Figure 11.10: Performance of FA, BDCL, and RL channel allocation methods for different mean call-arrival rates.

Nie and Haykin (1996) also studied the application of reinforcement learning to dynamic channel allocation. They formulated the problem somewhat differently than Singh and Bertsekas did. Instead of trying to directly minimize the probability of blocking a call, their system tried to minimize a more indirect measure of system performance. Cost was assigned to patterns of channel use depending on the distances between calls using the same channels. Patterns in which channels were being used by multiple calls that were close to each other were favored over patterns in which channel-sharing calls were far apart. Nie and Haykin compared their system with a method called MAXAVAIL (Sivarajan *et al.*, 1990), considered to be one of the best dynamic channel allocation methods. For each new call, it selects the channel that maximizes the total number of channels available in the entire system. Nie and Haykin showed that the blocking probability achieved by their reinforcement learning system was closely comparable to that of MAXAVAIL under a variety of conditions in a 49 cell, 70 channel simulation. A key point, however, is that the allocation policy produced by reinforcement learning can be implemented on-line much more efficiently than MAXAVAIL, which requires so much on-line computation that it is not feasible for large systems.

The studies we described in this section are so recent that the many questions they raise have not yet been answered. We can see, though, that there can be different ways to apply reinforcement learning to the same real-world problem. In the near future, we expect to see many refinements of these applications, as well as many new applications of reinforcement learning to problems arising in communication systems.

Next Up Previous

Next: 11.6 Job-Shop Scheduling **Up:** 11 Case Studies **Previous:** 11.4 Elevator

Dispatching

Richard Sutton
Fri May 30 18:04:45 EDT 1997

11.6 Job-Shop Scheduling

Many jobs in industry and elsewhere require completing a collection of tasks while satisfying temporal and resource constraints. Temporal constraints say that some tasks have to be finished before others can be started; resource constraints say that two tasks requiring the same resource cannot be done simultaneously (e.g., the same machine cannot do two tasks as once). The objective is to create a schedule specifying when each task is to begin and what resources it will use that satisfies all the constraints while taking as little overall time as possible. This is the job-shop scheduling problem. In its general form, it is NP-complete, meaning that there is probably no efficient procedure for exactly finding shortest schedules for arbitrary instances of the problem. Job-shop scheduling is usually done using heuristic algorithms that take advantage of special properties of each specific instance.

Zhang and Dietterich (1995, 1996; Zhang, 1996) were motivated to apply reinforcement learning to job-shop scheduling because the design of domain-specific, heuristic algorithms can be very expensive and time-consuming. Their goal was to show how reinforcement learning can be used to learn how to quickly find constraint-satisfying schedules of short duration in specific domains, thereby reducing the amount of hand engineering required. They addressed the NASA space-shuttle payload processing problem (SSPP), which requires scheduling the tasks required for installation and testing of shuttle cargo bay payloads. An SSPP typically requires scheduling for 2 to 6 shuttle missions, each requiring between 34 and 164 tasks. An example of a task is MISSION-SEQUENCE-TEST, which has a duration of 7200 time units and requires the following resources: two quality control officers, two technicians, one ATE, one SPCDS, and one HITS. Some resources are divided into pools, and if a task needs more than one resource of a specific type, they must belong to the same pool, and the pool has to be the right one. For example, if a task needs two quality control officers, they both have to be in the pool of quality control officers working on the same shift at the right site. It is not too hard to find a conflict-free schedule for a job, one that meets all the temporal and resource constraints, but the objective is to find a conflict-free schedule with the shortest possible total duration, which is much more difficult.

How can you do this using reinforcement learning? Job-shop scheduling is usually formulated as a search in the space of schedules, what is called a discrete, or combinatorial, optimization problem. A typical solution method would sequentially generate schedules, attempting to improve each over its predecessor in terms of constraint violations and duration (a hillclimbing, or local search, method). You could think of this as a non-associative reinforcement learning problem of the type we discussed in Chapter 2 with a very large number of possible actions: all the possible schedules! But aside from the problem of having so many actions, any solution obtained this way would just be a *single* schedule for a *single* job instance. In contrast, what Zhang and Dietterich wanted their learning system to end up with was a *policy* that could quickly find good schedules for *any* SSPP. They wanted it to learn a skill for job-shop scheduling in this specific domain.

For clues about how to do this, they looked to an existing optimization approach to SSPP, in fact, the one actually in use by NASA at the time of their research: the iterative repair method developed by Zweben *et al.*\ (1994). The starting point for the search is a *critical path schedule*, a schedule that meets the temporal constraints but ignores the resource constraints. This schedule can be constructed efficiently by scheduling each task prior to launch as late as the temporal constraints permit, and each task after landing as early as these constraints permit. Resource pools are assigned randomly. Two types of operators are used to modify schedules. They can be applied to any task that violates a resource constraint. A REASSIGN-POOL operator changes the pool assigned to one of the task's resources. This type of operator only applies if it can reassign a pool so that the resource requirement is satisfied. A MOVE operator moves a task to the first earlier or later time at which its resource needs can be satisfied and uses the critical path method to reschedule all of the task's temporal dependents.

At each step of the iterative repair search, one operator is applied to the current schedule, selected according to the following rules. The earliest task with a resource constraint violation is found, and a REASSIGN-POOL operator is applied to this task if possible. If more than one applies, that is, if several different pool reassignments are possible, one is selected at random. If no REASSIGN-POOL operator applies, then a MOVE operator is selected at random based on a heuristic that prefers short-distance moves of tasks having few temporal dependents and whose resource requirements are a close to the task's over-allocation. After an operator is applied, the number of constraint violations of the resulting schedule is determined. A simulated annealing procedure is used decide whether to ``accept'' or ``reject'' this new schedule. If ΔV denotes the number of constraint violations removed by the repair, then the new schedule is accepted with probability $\exp(-\Delta V/T)$, where **T** is the current computational temperature that is gradually decreased throughout the search. If accepted, the new schedule becomes the current schedule for the next iteration; otherwise, the algorithm attempts to repair the old schedule again, which will usually produce different results due to the random decisions involved. Search stops when all constraint are satisfied. Short schedules are obtained by running the algorithm several times and selecting the shortest of the resulting conflict-free schedules.

Zhang and Dietterich's treated entire schedules as states in the sense of reinforcement learning. The actions were the applicable REASSIGN-POOL and MOVE operators, typically numbering about 20. The problem was treated as episodic, each episode starting with the same critical path schedule that the iterative repair algorithm would start with and ending when a schedule was found that did not violate any constraint. The initial state---a critical path schedule---is denoted s_0. The rewards were designed to promote the quick construction of conflict-free schedules of short duration. The system received a small negative reward (-0.001) on each step that resulted in a schedule that still violated a constraint. This encouraged the agent to find conflict-free schedules quickly, that is, with a small number of repairs to s_0. Encouraging the system to find short schedules is more difficult because what it means for a schedule to be short depends on the specific SSPP instance. The shortest schedule for a difficult instance, one with a lot of tasks and constraints, will be longer than the shortest schedule for a simpler instance. Zhang and Dietterich devised a formula for a *resource dilation factor* (RDF), intended to be an instance-independent measure of a schedule's duration. To account for an instance's intrinsic difficulty, the formula makes use of a measure of the resource overallocation of s_0. Since longer schedules tend to produce larger RDFs, the negative of the RDF of the final conflict-free schedule was used as a reward at the end of each episode. With this reward function, if it takes **N** repairs starting from a schedule **s** to obtain a final conflict-free schedule, s_f, the return from **s** is $-RDF(s_f) - 0.001(N-1)$.

This reward function was designed to try to make a system learn to satisfy the two goals of finding conflict-free schedules of short duration and finding conflict-free schedules quickly. But the reinforcement learning system really has only one goal---maximizing expected return---so the particular reward values determine how a learning system will tend to trade off these two goals. Setting the immediate reward to the small value of -0.001 means that the learning system will regard one repair, one step in the scheduling process, as being worth 0.001 units of RDF. So, for example, if from some schedule it is possible to produce a conflict-free schedule with one repair or with two, an optimal policy will take extra repair only if it promises a reduction in final RDF of more than 0.001.

Zhang and Dietterich's used TD(λ) to learn the value function. Function approximation was by a multi-layer neural network trained by backpropagating TD errors. Actions were selected by an ϵ -greedy policy, with ϵ decreasing during learning. One-step lookahead search was used to find the greedy action. Their knowledge of the problem made it easy to predict the schedules that would result from each repair operation. They experimented with a number of modifications to this basic procedure to improve its performance. One was to use the TD(λ) algorithm *backwards* after each episode, with the eligibility trace extending to future rather than to past states. Their results suggested that this was more accurate and efficient than forward learning. In updating the weights of the network, they also sometimes performed multiple weight updates when the TD error was large. This is apparently equivalent to dynamically varying the step-size parameter in an error-dependent way during learning. They also tried an *experience replay* technique due to Lin (1992). At any point in learning, the agent remembered the best trial up to that point. After every four trials, it re-played this remembered trial, learning from it as if it were a new trial. At the start of training, they similarly allowed the system to learn from trials generated by a good scheduler, and these could also be replayed later in learning. To make the lookahead search faster for large-scale problems, which typically had a branching factor of about 20, they used a variant they called *random sample greedy search* that estimated the greedy action by considering only random samples of actions, increasing the sample size until a preset confidence was reached that the greedy action of the sample was the true greedy action. Finally, having discovered that learning could be slowed considerably by excessive looping in the scheduling process, they made their system explicitly check for loops and alter action selections when a loop was detected. Although all of these techniques could improve the efficiency of learning, it is not clear how crucial all of them were for the success of the system.

Zhang and Dietterich experimented with two different network architectures. In the first version of their system, each schedule was represented using a set of 20 hand-crafted features. To define these features, they studied small scheduling problems to find features that had some ability to predict RDF. For example, experience with small problems showed that only four of the resource pools tended to cause allocation problems. The mean and standard deviation of each of these pools' unused portions over the entire schedule were computed, resulting in 10 real-valued features. Two other features were the RDF of the current schedule and the percentage of its duration during which it violated resource constraints. The network had 20 input units, one for each feature, a hidden layer of 40 sigmoidal units, and an output layer of 8 sigmoidal units. The output units coded the value of a schedule using a code in which, roughly, the location of the activity peak over the 8 units represented the value. Using the appropriate TD error, the network weights were updated using error backpropagation, with the multiple weight update scheme mentioned above.

The second version of the system (Zhang and Dietterich, 1995) used a more complicated time-delay neural network (TDNN) borrowed from the field of speech recognition (Lang *et al*., 1990). This version divided each schedule into a sequence of blocks (maximal time intervals during which tasks and resource assignments did not change) and represented each block by a set of features similar to those used in the first program. It then scanned a set of ``kernel'' networks across the blocks to create a set of more abstract features. Since different schedules had different numbers of blocks, another layer averaged these abstract features over each third of the blocks. Then a final layer of 8 sigmoidal output units represented the schedule's value using the same code used in the first version of the system. In all, this network had 1123 adjustable weights.

A set of 100 artificial scheduling problems was constructed and divided into subsets used for training, determining when to stop training (a validation set), and for final testing. During training they tested the system on the validation set after every 100 trials and stopped training when performance on the validation set stopped changing, which generally took about 10,000 trials. They trained networks with several different values of λ (0.2 and 0.7), with three different training sets, and they saved both the final set of weights and the set of weights producing the best performance on the validation set. Counting each set of weights as a different network, this produced 12 networks, each of which corresponded to a different scheduling algorithm.

Figure 11.11 shows how the mean performance of the 12 TDNN networks (labeled G12TDN) compared to the performances of two versions of Zweben *et al*.'s iterative repair algorithm, one using the number of constraint violations as the function to be minimized by simulated annealing (IR-V) and the other using the RDF measure (IR-RDF). The figure also shows the performance of the first version of their system that did not use a TDNN (G12N). The mean RDF of the best schedule found by repeatedly running an algorithm is plotted against the total number of schedule repairs (using a log scale). These results show that the learning system produced scheduling algorithms that needed many fewer repairs to find conflict-free schedules of the same quality as those found by the iterative repair algorithms. Figure 11.12 compares the computer time required by each scheduling algorithm to find schedules of various RDFs. According to this measure of performance, the best tradeoff between computer time and schedule quality is produced by the non-TDNN algorithm (G12N). The TDNN algorithm (G12TDN) suffered due to the time it took to apply the kernel scanning process, but Zhang and Dietterich point out that there are many ways to make it run faster.

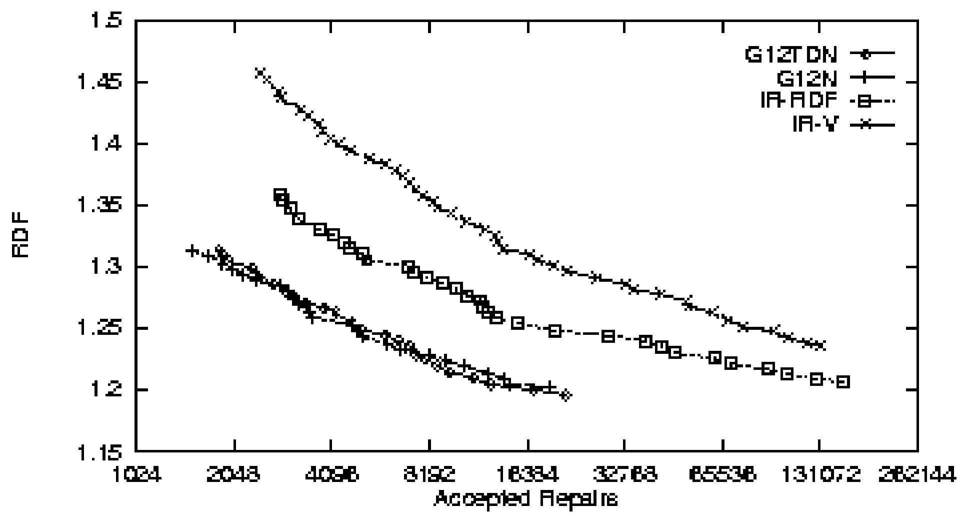


Figure 11.11: Comparison of accepted schedule repairs. (Reprinted with permission from Zhang and Dietterich, 1996.)

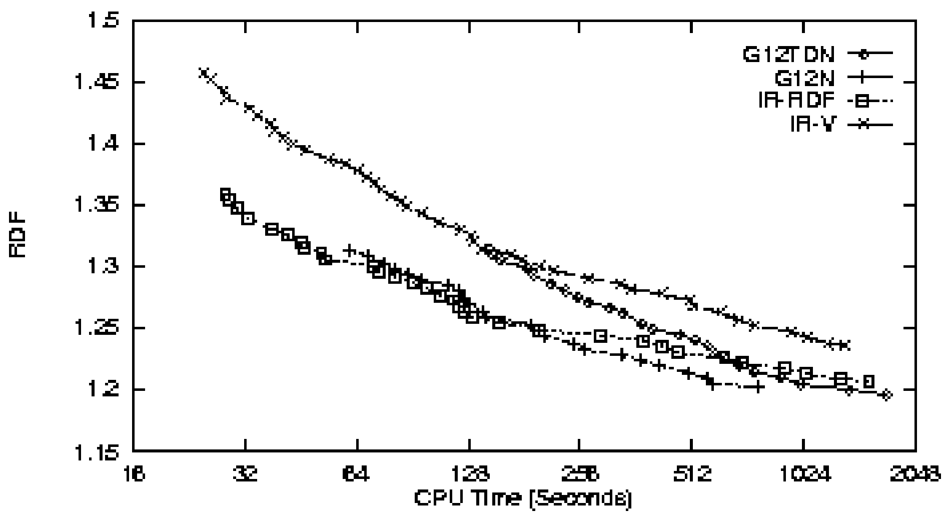


Figure 11.12: Comparison of CPU time. (Reprinted with permission from Zhang and Dietterich, 1996.)

These results do not unequivocally establish the utility of reinforcement learning for job-shop scheduling, or for other difficult search problems. But they do suggest that it is possible to use reinforcement learning methods to learn how to improve the efficiency of search. Zhang and Dietterich's job-shop scheduling system is the first successful instance of which we are aware in which reinforcement learning was applied in *plan-space*, that is, in which states are complete plans (job-shop schedules in this case), and actions are plan modifications. This is a more abstract application of reinforcement learning than we are used to thinking about. Note that in this application the system learned not just to efficiently create *one* good schedule, a skill that would not be particularly useful; it learned how to quickly find good schedules for a class of related scheduling problems. It is clear that Zhang and Dietterich went through a lot of trial-and-error learning of their own in developing this example. But remember that this was a ground-breaking exploration of a new aspect of reinforcement learning. We expect that future applications of this kind and complexity will become more routine as experience accumulates.

Next Up Previous

Next: References **Up:** 11 Case Studies **Previous:** 11.5 Dynamic Channel Allocation

Richard Sutton
Fri May 30 18:04:45 EDT 1997

References

Agre, 1988
Agre, P. E. (1988). *The Dynamic Structure of Everyday Life*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA. AI-TR 1085, MIT Artificial Intelligence Laboratory.

Agre and Chapman, 1990
Agre, P. E. and Chapman, D. (1990). What are plans for? *Robotics and Autonomous Systems*, 6:17--34.

Albus, 1971
Albus, J. S. (1971). A theory of cerebellar function. *Mathematical Biosciences*, 10:25--61.

Albus, 1981
Albus, J. S. (1981). *Brain, Behavior, and Robotics*. Byte Books.

Anderson, 1986
Anderson, C. W. (1986). *Learning and Problem Solving with Multilayer Connectionist Systems*. PhD thesis, University of Massachusetts, Amherst, MA.

Anderson, 1987
Anderson, C. W. (1987). Strategy learning with multilayer connectionist representations. Technical Report TR87-509.3, GTE Laboratories, Incorporated, Waltham, MA. (This is a corrected version of the report published in *Proceedings of the Fourth International Workshop on Machine Learning*,103--114, 1987, San Mateo, CA: Morgan Kaufmann.).

Anderson et al., 1977
Anderson, J. A., Silverstein, J. W., Ritz, S. A., and Jones, R. S. (1977). Distinctive features, categorical perception, and probability learning: Some applications of a neural model. *Psychological Review*, 84:413--451.

Andreae, 1963
Andreae, J. H. (1963). STELLA: A scheme for a learning machine. In *Proceedings of the 2nd IFAC Congress, Basle*, pages 497--502, London. Butterworths.

Andreae, 1969a
Andreae, J. H. (1969a). A learning machine with monologue. *International Journal of Man-Machine Studies*, 1:1--20.

Andreae, 1969b
Andreae, J. H. (1969b). Learning machines---a unified view. In Meetham, A. R. and Hudson, R. A., editors, *Encyclopedia of Information, Linguistics, and Control*, pages 261--270. Pergamon, Oxford.

Andreae, 1977
Andreae, J. H. (1977). *Thinking with the Teachable Machine*. Academic Press, London.

Baird, 1995
Baird, L. C. (1995). Residual algorithms: Reinforcement learning with function approximation. In Prieditis, A. and Russell, S., editors, *Proceedings of the Twelfth International Conference on Machine Learning*, pages 30--37, San Francisco, CA. Morgan Kaufmann.

Bao et al., 1994
Bao, G., Cassandras, C. G., Djaferis, T. E., Gandhi, A. D., and Looze, D. P. (1994). Elevator dispatchers for down peak traffic. Technical report, ECE Department, University of Massachusetts.

Barnard, 1993
Barnard, E. (1993). Temporal-difference methods and Markov models. *IEEE Transactions on Systems, Man, and Cybernetics*, 23:357--365.

Barto, 1985
Barto, A. G. (1985). Learning by statistical cooperation of self-interested neuron-like computing elements. *Human Neurobiology*, 4:229--256.

Barto, 1986
Barto, A. G. (1986). Game-theoretic cooperativity in networks of self-interested units. In Denker, J. S., editor, *Neural Networks for Computing*, pages 41--46. American Institute of Physics, New York.

Barto, 1990
Barto, A. G. (1990). Connectionist learning for control: An overview. In Miller, T.,

Sutton, R. S., and Werbos, P. J., editors, *Neural Networks for Control*, pages 5--58. MIT Press, Cambridge, MA.

Barto, 1991
Barto, A. G. (1991). Some learning tasks from a control perspective. In Nadel, L. and Stein, D. L., editors, *1990 Lectures in Complex Systems*, pages 195--223. Addison-Wesley Publishing Company, The Advanced Book Program, Redwood City, CA.

Barto, 1992
Barto, A. G. (1992). Reinforcement learning and adaptive critic methods. In White, D. A. and Sofge, D. A., editors, *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*, pages 469--491. Van Nostrand Reinhold, New York.

Barto, 1995a
Barto, A. G. (1995a). Adaptive critics and the basal ganglia. In Houk, J. C., Davis, J. L., and Beiser, D. G., editors, *Models of Information Processing in the Basal Ganglia*, pages 215--232. MIT Press, Cambridge, MA.

Barto, 1995b
Barto, A. G. (1995b). Reinforcement learning. In Arbib, M. A., editor, *Handbook of Brain Theory and Neural Networks*, pages 804--809. The MIT Press, Cambridge, MA.

Barto and Anandan, 1985
Barto, A. G. and Anandan, P. (1985). Pattern recognizing stochastic learning automata. *IEEE Transactions on Systems, Man, and Cybernetics*, 15:360--375.

Barto and Anderson, 1985
Barto, A. G. and Anderson, C. W. (1985). Structural learning in connectionist systems. In *Program of the Seventh Annual Conference of the Cognitive Science Society*, pages 43--54, Irvine, CA.

Barto et al., 1982
Barto, A. G., Anderson, C. W., and Sutton, R. S. (1982). Synthesis of nonlinear control surfaces by a layered associative search network. *Biological Cybernetics*, 43:175--185.

Barto et al., 1991
Barto, A. G., Bradtke, S. J., and Singh, S. P. (1991). Real-time learning and control using asynchronous dynamic programming. Technical Report 91-57, Department of Computer and Information Science, University of Massachusetts, Amherst, MA.

Barto et al., 1995

Barto, A. G., Bradtke, S. J., and Singh, S. P. (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72:81--138.

Barto and Duff, 1994

Barto, A. G. and Duff, M. (1994). Monte carlo matrix inversion and reinforcement learning. In Cohen, J. D., Tesauro, G., and Alspector, J., editors, *Advances in Neural Information Processing Systems: Proceedings of the 1993 Conference*, pages 687--694, San Francisco, CA. Morgan Kaufmann.

Barto and Jordan, 1987

Barto, A. G. and Jordan, M. I. (1987). Gradient following without back-propagation in layered networks. In Caudill, M. and Butler, C., editors, *Proceedings of the IEEE First Annual Conference on Neural Networks*, pages II629--II636, San Diego, CA.

Barto and Sutton, 1981a

Barto, A. G. and Sutton, R. S. (1981a). Goal seeking components for adaptive intelligence: An initial assessment. Technical Report AFWAL-TR-81-1070, Air Force Wright Aeronautical Laboratories/Avionics Laboratory, Wright-Patterson AFB, OH.

Barto and Sutton, 1981b

Barto, A. G. and Sutton, R. S. (1981b). Landmark learning: An illustration of associative search. *Biological Cybernetics*, 42:1--8.

Barto and Sutton, 1982

Barto, A. G. and Sutton, R. S. (1982). Simulation of anticipatory responses in classical conditioning by a neuron-like adaptive element. *Behavioural Brain Research*, 4:221--235.

Barto et al., 1983

Barto, A. G., Sutton, R. S., and Anderson, C. W. (1983). Neuronlike elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:835--846. Reprinted in J. A. Anderson and E. Rosenfeld, *Neurocomputing: Foundations of Research*, MIT Press, Cambridge, MA, 1988.

Barto et al., 1981

Barto, A. G., Sutton, R. S., and Brouwer, P. S. (1981). Associative search network: A reinforcement learning associative memory. *IEEE Transactions on Systems, Man, and Cybernetics*, 40:201--211.

Bellman and Dreyfus, 1959

Bellman, R. and Dreyfus, S. E. (1959). Functional approximations and dynamic programming. *Math Tables and Other Aides to Computation*, 13:247--251.

Bellman et al., 1973
Bellman, R., Kalaba, R., and Kotkin, B. (1973). Polynomial approximation---A new computational technique in dynamic programming: Allocation processes. *Mathematical Computation*, 17:155--161.

Bellman, 1956
Bellman, R. E. (1956). A problem in the sequential design of experiments. *Sankhya,* 16:221--229.

Bellman, 1957a
Bellman, R. E. (1957a). *Dynamic Programming*. Princeton University Press, Princeton, NJ.

Bellman, 1957b
Bellman, R. E. (1957b). A Markov decision process. *Journal of Mathematical Mech.,* 6:679--684.

Berry and Fristedt, 1985
Berry, D. A. and Fristedt, B. (1985). *Bandit Problems*. Chapman and Hall, London.

Bertsekas, 1982
Bertsekas, D. P. (1982). Distributed dynamic programming. *IEEE Transactions on Automatic Control,* 27:610--616.

Bertsekas, 1983
Bertsekas, D. P. (1983). Distributed asynchronous computation of fixed points. *Mathematical Programming,* 27:107--120.

Bertsekas, 1987
Bertsekas, D. P. (1987). *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall, Englewood Cliffs, NJ.

Bertsekas, 1995
Bertsekas, D. P. (1995). *Dynamic Programming and Optimal Control.* Athena, Belmont, MA.

Bertsekas and Tsitsiklis, 1989
Bertsekas, D. P. and Tsitsiklis, J. N. (1989). *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Englewood Cliffs, NJ.

Bertsekas and Tsitsiklis, 1996

Bertsekas, D. P. and Tsitsiklis, J. N. (1996). *Neural Dynamic Programming*. Athena Scientific, Belmont, MA.

Biermann et al., 1982
Biermann, A. W., Fairfield, J. R. C., and Beres, T. R. (1982). Signature table systems and learning. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-12:635--648.

Bishop, 1995
Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*. Clarendon, Oxford.

Booker, 1982
Booker, L. B. (1982). *Intelligent Behavior as an Adaptation to the Task Environment*. PhD thesis, University of Michigan, Ann Arbor, MI.

Boone, 1997
Boone, G. (1997). Minimum-time control of the acrobot. In *1997 International Conference on Robotics and Automation*, Albuquerque, NM.

Boutilier et al., 1995
Boutilier, C., Dearden, R., and Goldszmidt, M. (1995). Exploiting structure in policy construction. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*.

Boyan and Moore, 1995
Boyan, J. A. and Moore, A. W. (1995). Generalization in reinforcement learning: Safely approximating the value functions. In G. Tesauro, D. Touretzky, T. L., editor, *Advances in Neural Information Processing Systems: Proceedings of the 1994 Conference*, pages 369--376, San Mateo, CA. Morgan Kaufmann.

Boyan et al., 1995
Boyan, J. A., Moore, A. W., and Sutton, R. S., editors (1995). *Proceedings of the Workshop on Value Function Approximation. Machine Learning Conference 1995*, Pittsburgh, PA. School of Computer Science, Carnegie Mellon University. Technical Report CMU-CS-95-206.

Bradtke, 1993
Bradtke, S. J. (1993). Reinforcement learning applied to linear quadratic regulation. In S. J. Hanson, J. D. Cowan, C. L. G., editor, *Advances in Neural Information Processing Systems: Proceedings of the 1992 Conference*, pages 295--302, San Mateo, CA. Morgan Kaufmann.

Bradtke, 1994

Bradtke, S. J. (1994). *Incremental Dynamic Programming for On-Line Adaptive Optimal Control*. PhD thesis, University of Massachusetts, Amherst. Appeared as CMPSCI Technical Report 94-62.

Bradtke and Barto, 1996

Bradtke, S. J. and Barto, A. G. (1996). Linear least--squares algorithms for temporal difference learning. *Machine Learning*, 22:33--57.

Bradtke and Duff, 1995

Bradtke, S. J. and Duff, M. O. (1995). Reinforcement learning methods for continuous-time Markov decision problems. In G. Tesauro, D. Touretzky, T. L., editor, *Advances in Neural Information Processing Systems: Proceedings of the 1994 Conference,* pages 393--400, San Mateo, CA. Morgan Kaufmann.

Bridle, 1990

Bridle, J. S. (1990). Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimates of parameters. In Touretzky, D. S., editor, *Advances in Neural Information Processing Systems 2*, pages 211--217, San Mateo, CA. Morgan Kaufmann.

Broomhead and Lowe, 1988

Broomhead, D. S. and Lowe, D. (1988). Multivariable functional interpolation and adaptive networks. *Complex Systems*, 2:321--355.

Bryson, 1996

Bryson, Jr., A. E. (1996). Optimal control---1950 to 1985. *IEEE Control Systems*, 13(3):26--33.

Bush and Mosteller, 1955

Bush, R. R. and Mosteller, F. (1955). *Stochastic Models for Learning*. Wiley, New York.

Byrne et al., 1990

Byrne, J. H., Gingrich, K. J., and Baxter, D. A. (1990). Computational capabilities of single neurons: Relationship to simple forms of associative and nonassociative learning in *aplysia*. In Hawkins, R. D. and Bower, G. H., editors, *Computational Models of Learning,* pages 31--63. Academic Press, New York.

Campbell, 1959

Campbell, D. T. (1959). Blind variation and selective survival as a general strategy in knowledge-processes. In Yovits, M. C. and Cameron, S., editors, *Self-Organizing Systems*, pages 205--231. Pergamon.

Carlström and Nordström, 1997
Carlström, J. and Nordström, E. (1997). Control of self-similar atm call traffic by reinforcement learning. In *Proceedings of the International Workshop on Applications of Neural Networks to Telecommunications 3 (IWANNT*97),* Hillsdale NJ. Lawrence Erlbaum.

Chapman and Kaelbling, 1991
Chapman, D. and Kaelbling, L. P. (1991). Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings of the 1991 International Joint Conference on Artificial Intelligence*.

Chow and Tsitsiklis, 1991
Chow, C.-S. and Tsitsiklis, J. N. (1991). An optimal one-way multigrid algorithm for discrete-time stochastic control. *IEEE Transactions on Automatic Control,* 36:898--914.

Chrisman, 1992
Chrisman, L. (1992). Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Proceedings of the Tenth National Conference on Artificial Intelligence,* pages 183--188, Menlo Park, CA. AAAI Press/MIT Press.

Christensen and Korf, 1986
Christensen, J. and Korf, R. E. (1986). A unified theory of heuristic evaluation functions and its application to learning. In *Proceedings of the Fifth National Conference on Artificial Intelligence AAAI-86,* pages 148--152, San Mateo, CA. Morgan Kaufmann.

Cichosz, 1995
Cichosz, P. (1995). Truncating temporal differences: On the efficient implementation of TD(lambda) for reinforcement learning. *Journal of Artificial Intelligence Research*, 2:287--318.

Clark and Farley, 1955
Clark, W. A. and Farley, B. G. (1955). Generalization of pattern recognition in a self-organizing system. In *Proceedings of the 1955 Western Joint Computer Conference,* pages 86--91.

Clouse, 1997
Clouse, J. (1997). *On Integrating Apprentice Learning and Reinforcement Learning TITLE2*. PhD thesis, University of Massachusetts, Amherst. Appeared as CMPSCI Technical Report 96-026.

Clouse and Utgoff, 1992
Clouse, J. and Utgoff, P. (1992). A teaching method for reinforcement learning systems. In *Proceedings of the Ninth International Machine Learning Conference*, pages 92--101.

Colombetti and Dorigo, 1994
Colombetti, M. and Dorigo, M. (1994). Training agent to perform sequential behavior. *Adaptive Behavior*, 2(3):247--275.

Connell, 1989
Connell, J. (1989). A colony architecture for an artificial creature. Technical Report Technical Report AI-TR-1151, MIT Artificial Intelligence Laboratory, Cambridge, MA.

Craik, 1943
Craik, K. J. W. (1943). *The Nature of Explanation*. Cambridge University Press, Cambridge.

Crites, 1996
Crites, R. H. (1996). *Large-Scale Dynamic Optimization Using Teams of Reinforcement Learning Agents*. PhD thesis, University of Massachusetts, Amherst, MA.

Crites and Barto, 1996
Crites, R. H. and Barto, A. G. (1996). Improving elevator performance using reinforcement learning. In D. S. Touretzky, M. C. Mozer, M. E. H., editor, *Advances in Neural Information Processing Systems: Proceedings of the 1995 Conference*, pages 1017--1023, Cambridge, MA. MIT Press.

Curtiss, 1954
Curtiss, J. H. (1954). A theoretical comparison of the efficiencies of two classical methods and a monte carlo method for computing one component of the solution of a set of linear algebraic equations. In Meyer, H. A., editor, *Symposium on Monte Carlo Methods*, pages 191--233. Wiley, New York.

Cziko, 1995
Cziko, G. (1995). *Without Miracles. Universal Selection Theory and the Second Darvinian Revolution*. The MIT Press.

Daniel, 1976
Daniel, J. W. (1976). Splines and efficiency in dynamic programming. *Journal of Mathematical Analysis and Applications*, 54:402--407.

Dayan, 1991

Dayan, P. (1991). Reinforcement comparison. In Touretzky, D. S., Elman, J. L., Sejnowski, T. J., and Hinton, G. E., editors, *Connectionist Models: Proceedings of the 1990 Summer School*, pages 45--51. Morgan Kaufmann, San Mateo, CA.

Dayan, 1992

Dayan, P. (1992). The convergence of TD(λ) for general λ. *Machine Learning*, 8:341--362.

Dayan and Hinton, 1993

Dayan, P. and Hinton, G. E. (1993). Feudal reinforcement learning. In Hanson, S. J., Cohen, J. D., and Giles, C. L., editors, *Advances in Neural Information Processing Systems: Proceedings of the 1992 Conference*, pages 271--278, San Mateo, CA. Morgan Kaufmann.

Dayan and Sejnowski, 1994

Dayan, P. and Sejnowski, T. (1994). TD(λ) converges with probability 1. *Machine Learning*, 14:295--301.

Dean and Lin, 1995

Dean, T. and Lin, S.-H. (1995). Decomposition techniques for planning in stochastic domains. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*.

DeJong and Spong, 1994

DeJong, G. and Spong, M. W. (1994). Swinging up the acrobot: An example of intelligent control. In *Proceedings of the American Control Conference*, pages 2158--2162.

Denardo, 1967

Denardo, E. V. (1967). Contraction mappings in the theory underlying dynamic programming. *SIAM Review*, 9:165--177.

Dennett, 1978

Dennett, D. C. (1978). *Brainstorms*, chapter Why the Law-of-Effect Will Not Go Away, pages 71--89. Bradford/MIT Press, Cambridge, MA.

Dietterich and Flann, 1995

Dietterich, T. G. and Flann, N. S. (1995). Explanation-based learning and reinforcement learning: A unified view. In Prieditis, A. and Russell, S., editors, *Proceedings of the Twelfth International Conference on Machine Learning*, pages 176--184, San Francisco, CA. Morgan Kaufmann.

Doya, 1996

Doya, K. (1996). Temporal difference learing in continuous time and space. In Touretzky, D. S., Mozer, M. C., and Hasselmo, M. E., editors, *Advances in Neural Information Processing Systems: Proceedings of the 1995 Conference*, pages 1073--1079, Cambridge, MA. MIT Press.

Doyle and Snell, 1984

Doyle, P. G. and Snell, J. L. (1984). *Random Walks and Electric Networks*. The Mathematical Association of America. Carus Mathematical Monograph 22.

Dreyfus and Law, 1977

Dreyfus, S. E. and Law, A. M. (1977). *The Art and Theory of Dynamic Programming*. Academic Press, New York.

Duda and Hart, 1973

Duda, R. O. and Hart, P. E. (1973). *Pattern Classification and Scene Analysis*. Wiley, New York.

Duff, 1995

Duff, M. O. (1995). Q-learning for bandit problems. In Prieditis, A. and Russell, S., editors, *Proceedings of the Twelfth International Conference on Machine Learning*, pages 209--217, San Francisco, CA. Morgan Kaufmann.

Estes, 1950

Estes, W. K. (1950). Toward a statistical theory of learning. *Psychololgical Review*, 57:94--107.

Farley and Clark, 1954

Farley, B. G. and Clark, W. A. (1954). Simulation of self-organizing systems by digital computer. *IRE Transactions on Information Theory*, 4:76--84.

Feldbaum, 1960

Feldbaum, A. A. (1960). *Optimal Control Theory*. Academic Press, New York.

Friston et al., 1994

Friston, K. J., Tononi, G., Reeke, G. N., Sporns, O., and Edelman, G. M. (1994). Value-dependent selection in the brain: Simulation in a synthetic neural model. *Neuroscience*, 59:229--243.

Fu, 1970

Fu, K. S. (1970). Learning control systems---Review and outlook. *IEEE Transactions on Automatic Control*, pages 210--221.

Galanter and Gerstenhaber, 1956

Galanter, E. and Gerstenhaber, M. (1956). On thought: The extrinsic theory. *Psychological Review*, 63:218--227.

Gällmo and Asplund, 1995

Gällmo, O. and Asplund, H. (1995). Reinforcement learning by construction of hypothetical targets. In Alspector, J., Goodman, R., and Brown, T. X., editors, *Proceedings of the International Workshop on Applications of Neural Networks to Telecommunications 2 (IWANNT-2),* pages 300--307. Stockholm, Sweden.

Gardner, 1981

Gardner (1981). Samuel's checkers player. In Barr, A. and Feigenbaum, E. A., editors, *The Handbook of Artificial Intelligence, I,* pages 84--108. William Kaufmann, Los Altos, CA.

Gardner, 1973

Gardner, M. (1973). Mathematical games. *Scientific American*, 228:108.

Gelperin et al., 1985

Gelperin, A., Hopfield, J. J., and Tank, D. W. (1985). The logic of *limax* learning. In Selverston, A., editor, *Model Neural Networks and Behavior*. Plenum Press, New York.

Gittins and Jones, 1974

Gittins, J. C. and Jones, D. M. (1974). A dynamic allocation index for the sequential design of experiments. *Progress in Statistics,* pages 241--266.

Goldberg, 1989

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA.

Goldstein, 1957

Goldstein, H. (1957). *Classical Mechanics*. Addison-Wesley, Rreadin, MA.

Goodwin and Sin, 1984

Goodwin, G. C. and Sin, K. S. (1984). *Adaptive Filtering Prediction and Control*. Prentice-Hall, Englewood Cliffs, N.J.

Gordon, 1995

Gordon, G. J. (1995). Stable function approximation in dynamic programming. In Prieditis, A. and Russell, S., editors, *Proceedings of the Twelfth International Conference on Machine Learning*, pages 261--268, San Francisco, CA. Morgan Kaufmann. An expanded version was published as Technical Report CMU-CS-95-

103, Carnegie Mellon University, Pittsburgh, PA, 1995.

Gordon, 1996
Gordon, G. J. (1996). Stable fitted reinforcement learning. In D. S. Touretzky, M. C. Mozer, M. E. H., editor, *Advances in Neural Information Processing Systems: Proceedings of the 1995 Conference,* pages 1052--1058, Cambridge, MA. MIT Press.

Griffith, 1966
Griffith, A. K. (1966). A new machine learning technique applied to the game of checkers. Technical Report Project MAC Artificial Intelligence Memo 94, Massachusetts Institute of Technology.

Griffith, 1974
Griffith, A. K. (1974). A comparison and evaluation of three machine learning procedures as applied to the game of checkers. *Artificial Intelligence,* 5:137--148.

Gullapalli, 1990
Gullapalli, V. (1990). A stochastic reinforcement algorithm for learning real-valued functions. *Neural Networks,* 3:671--692.

Gurvits et al., 1994
Gurvits, L., Lin, L.-J., and Hanson, S. J. (1994). Incremental learning of evaluation functions for absorbing Markov chains: New methods and theorems. Preprint.

Hampson, 1983
Hampson, S. E. (1983). *A Neural Model of Adaptive Behavior*. PhD thesis, University of California, Irvine, CA.

Hampson, 1989
Hampson, S. E. (1989). *Connectionist Problem Solving: Computational Aspects of Biological Learning*. Birkhauser, Boston.

Hawkins and Kandel, 1984
Hawkins, R. D. and Kandel, E. R. (1984). Is there a cell-biological alphabet for simple forms of learning? *Psychological Review*, 91:375--391.

Hersh and Griego, 1969
Hersh, R. and Griego, R. J. (1969). Brownian motion and potential theory. *Scientific American*, pages 66--74.

Hilgard and Bower, 1975
Hilgard, E. R. and Bower, G. H. (1975). *Theories of Learning*. Prentice-Hall,

Englewood Cliffs, NJ.

Hinton, 1984
Hinton, G. E. (1984). Distributed representations. Technical Report CMU-CS-84-157, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA.

Hochreiter and Schmidhuber, 1997
Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*.

Holland, 1975
Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor.

Holland, 1976
Holland, J. H. (1976). Adaptation. In Rosen, R. and Snell, F. M., editors, *Progress in Theoretical Biology*, volume 4, pages 263--293. Academic Press, NY.

Holland, 1986
Holland, J. H. (1986). Escaping brittleness: The possibility of general-purpose learning algorithms applied to rule-based systems. In Michalski, R. S., Carbonell, J. G., and Mitchell, T. M., editors, *Machine Learning: An Artificial Intelligence Approach, Volume II*, pages 593--623. Morgan Kaufmann, San Mateo, CA.

Houk et al., 1995
Houk, J. C., Adams, J. L., and Barto, A. G. (1995). A model of how the basal ganglia generates and uses neural signals that predict reinforcement. In Houk, J. C., Davis, J. L., and Beiser, D. G., editors, *Models of Information Processing in the Basal Ganglia*, pages 249--270. MIT Press, Cambridge, MA.

Howard, 1960
Howard, R. (1960). *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, MA.

Hull, 1943
Hull, C. L. (1943). *Principles of Behavior*. D. Appleton-Century, NY.

Hull, 1952
Hull, C. L. (1952). *A Behavior System*. Wiley, NY.

Jaakkola et al., 1994
Jaakkola, T., Jordan, M. I., and Singh, S. P. (1994). On the convergence of

stochastic iterative dynamic programming algorithms. *Neural Computation*, 6.

Jaakkola et al., 1995
Jaakkola, T., Singh, S. P., and Jordan, M. I. (1995). Reinforcement learning algorithm for partially observable Markov decision problems. In G. Tesauro, D. Touretzky, T. L., editor, *Advances in Neural Information Processing Systems: Proceedings of the 1994 Conference*, pages 345--352, San Mateo, CA. Morgan Kaufmann.

Kaelbling, 1996
Kaelbling (1996). A special issue of machine learning on reinforcement learning. 22.

Kaelbling, 1993a
Kaelbling, L. (1993a). Hierarchical learning in stochastic domains: Preliminary results. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 167--173. Morgan Kaufmann.

Kaelbling, 1993b
Kaelbling, L. P. (1993b). *Learning in Embedded Systems*. MIT Press, Cambridge MA.

Kaelbling et al., 1996
Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4.

Kakutani, 1945
Kakutani, S. (1945). Markov processes and the dirichlet problem. *Proc. Jap. Acad.*, 21:227--233.

Kalos and Whitlock, 1986
Kalos, M. H. and Whitlock, P. A. (1986). *Monte Carlo Methods*. Wiley, NY.

Kanerva, 1988
Kanerva, P. (1988). *Sparse Distributed Memory*. MIT Press, Cambridge, MA.

Kanerva, 1993
Kanerva, P. (1993). Sparse distributed memory and related models. In Hassoun, M. H., editor, *Associative Neural Memories: Theory and Implementation*, pages 50--76. Oxford University Press, NY.

Kashyap et al., 1970
Kashyap, R. L., Blaydon, C. C., and Fu, K. S. (1970). Stochastic approximation. In

Mendel, J. M. and Fu, K. S., editors, *Adaptive, Learning, and Pattern Recognition Systems: Theory and Applications*. Academic Press, New York.

Keerthi and Ravindran, 1997
Keerthi, S. S. and Ravindran, B. (1997). Reinforcement learning. In Fiesler, E. and Beale, R., editors, *Handbook of Neural Computation*. Oxford University Press, USA.

Kimble, 1961
Kimble, G. A. (1961). *Hilgard and Marquis' Contitioning and Learning*. Appleton-Century-Crofts, Inc., New York.

Kimble, 1967
Kimble, G. A. (1967). *Foundations of Conditioning and Learning*. Appleton-Century-Crofts.

Kirkpatrick et al., 1983
Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, 220:671--680.

Klopf, 1972
Klopf, A. H. (1972). Brain function and adaptive systems---A heterostatic theory. Technical Report AFCRL-72-0164, Air Force Cambridge Research Laboratories, Bedford, MA. A summary appears in *Proceedings of the International Conference on Systems, Man, and Cybernetics*, 1974, IEEE Systems, Man, and Cybernetics Society, Dallas, TX.

Klopf, 1975
Klopf, A. H. (1975). A comparison of natural and artificial intelligence. *SIGART Newsletter*, 53:11--13.

Klopf, 1982
Klopf, A. H. (1982). *The Hedonistic Neuron: A Theory of Memory, Learning, and Intelligence*. Hemisphere, Washington, D.C.

Klopf, 1988
Klopf, A. H. (1988). A neuronal model of classical conditioning. *Psychobiology*, 16:85--125.

Kohonen, 1977
Kohonen, T. (1977). *Associative Memory: A System Theoretic Approach*. Springer-Varlag, Berlin.

Korf, 1988

Korf, R. E. (1988). Optimal path finding algorithms. In Kanal, L. N. and Kumar, V., editors, *Search in Artificial Intelligence*, pages 223--267. Springer Verlag, Berlin.

Kraft and Campagna, 1990

Kraft, L. G. and Campagna, D. P. (1990). A summary comparison of CMAC neural network and traditional adaptive control systems. In Miller, T., Sutton, R. S., and Werbos, P. J., editors, *Neural Networks for Control*, pages 143--169. MIT Press, Cambridge, MA.

Kraft et al., 1992

Kraft, L. G., Miller, W. T., and Dietz, D. (1992). Development and application of CMAC neural network-based control. In White, D. A. and Sofge, D. A., editors, *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*, pages 215--232. Van Nostrand Reinhold, New York.

Kuman and Varaiya, 1986

Kuman, P. R. and Varaiya, P. (1986). *Stochastic Systems: Estimation, Identification, and Adaptive Control*. Prentice-Hall, Englewood Cliffs, NJ.

Kumar, 1985

Kumar, P. R. (1985). A survey of some results in stochastic adaptive control. *SIAM Journal of Control and Optimization*, 23:329--380.

Kumar and Kanal, 1988

Kumar, V. and Kanal, L. N. (1988). The CDP: A unifying formulation for heuristic search, dynamic programming, and branch-and-bound. In Kanal, L. N. and Kumar, V., editors, *Search in Artificial Intelligence*, pages 1--37. Springer-Verlag.

Kushner and Dupuis, 1992

Kushner, H. J. and Dupuis, P. (1992). *Numerical Methods for Stochastic Control Problems in Continuous Time*. Springer-Verlag, New York.

Lai, 1987

Lai, T. L. (1987). Adaptive treatment allocation and the multi-armed bandit problem. *The Annals of Statistics*, 15(3):1091--1114.

Lang et al., 1990

Lang, K. J., Waibel, A. H., and Hinton, G. E. (1990). A time-delay neural network architecture for isolated word recognition. *Neural Networks*, 3:33--43.

Lin and Kim, 1991

Lin, C.-S. and Kim, H. (1991). Cmac-based adaptive critic self-learning control.

IEEE Transactions on Neural Networks, 2:530--533.

Lin, 1992
Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8:293--321.

Lin and Mitchell, 1992
Lin, L.-J. and Mitchell, T. (1992). Reinforcement learning with hidden states. In *Proceedings of the Second International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, pages 271--280. MIT Press.

Littman, 1994
Littman, M. L. (1994). Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the Eleventh International Conference on Machine Learning*, pages 157--163, San Francisco, CA. Morgan Kaufmann.

Littman et al., 1995a
Littman, M. L., Cassandra, A. R., and Kaelbling, L. P. (1995a). Learning policies for partially observable environments: Scaling up. In Prieditis, A. and Russell, S., editors, *Proceedings of the Twelfth International Conference on Machine Learning*, pages 362--370, San Francisco, CA. Morgan Kaufmann.

Littman et al., 1995b
Littman, M. L., Dean, T. L., and Kaelbling, L. P. (1995b). On the complexity of solving Markov decision processes. In *Proceedings of the Eleventh International Conference on Uncertainty in Artificial Intelligence*.

Ljung and Söderstrom, 1983
Ljung, L. and Söderstrom, T. (1983). *Theory and Practice of Recursive Identification*. MIT Press, Cambridge, MA.

Lovejoy, 1991
Lovejoy, W. S. (1991). A survey of algorithmic methods for partially observed Markov decision processes. *Annals of Operations Research*, 28:47--66.

Luce, 1959
Luce, D. (1959). *Individual Choice Behavior*. Wiley, NY.

M. Zweben, 1994
M. Zweben, B. Daun, M. D. (1994). Scheduling and rescheduling with iterative repair. In Zweben, M. and Fox, M. S., editors, *Intelligent Scheduling*, pages 241--255. Morgan Kaufmann, San Francisco, CA.

Maclin and Shavlik, 1994

Maclin, R. and Shavlik, J. W. (1994). Incorporating advice into agents that learn from reinforcements. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94).*

Mahadevan, 1996

Mahadevan, S. (1996). Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine Learning*, 22:159--196.

Markey, 1994

Markey, K. L. (1994). Efficient learning of multiple degree-of-freedom control problems with quasi-independent q-agents. In Mozer, M. C., Smolensky, P., Touretzky, D. S., Elman, J. L., and Weigend, A. S., editors, *Proceedings of the 1009 Connectionist Models Summer School*, Hillsdale, NJ. Erlbaum.

Mazur, 1994

Mazur, J. E. (1994). *Learning and Behavior, Third Edition*. Prentice-Hall, Englewood Cliffs, NJ.

McCallum, 1992

McCallum, A. K. (1992). Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 183--188, Menlo Park, CA. AAAI Press/MIT Press.

McCallum, 1993

McCallum, A. K. (1993). Overcoming incomplete perception with utile distinction memory. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 190--196. Morgan Kaufmann.

McCallum, 1995

McCallum, A. K. (1995). *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis, University of Rochester, Rochester.

Mendel, 1966

Mendel, J. M. (1966). Applications of artificial intelligence techniques to a spacecraft control problem. Technical Report NASA CR-755, National Aeronautics and Space Administration.

Mendel and McLaren, 1970

Mendel, J. M. and McLaren, R. W. (1970). Reinforcement learning control and pattern recognition systems. In Mendel, J. M. and Fu, K. S., editors, *Adaptive, Learning and Pattern Recognition Systems: Theory and Applications*, pages 287--318. Academic Press, New York.

Michie, 1961
Michie, D. (1961). Trial and error. In Barnett, S. A. and McLaren, A., editors, *Science Survey, Part 2*, pages 129--145, Harmondsworth. Penguin.

Michie, 1963
Michie, D. (1963). Experiments on the mechanisation of game learning. 1. characterization of the model and its parameters. *Computer Journal*, 1:232--263.

Michie, 1974
Michie, D. (1974). *On Machine Intelligence*. Edinburgh University Press.

Michie and Chambers, 1968
Michie, D. and Chambers, R. A. (1968). BOXES: An experiment in adaptive control. In Dale, E. and Michie, D., editors, *Machine Intelligence 2*, pages 137--152. Oliver and Boyd.

Miller and Williams, 1992
Miller, S. and Williams, R. J. (1992). Learning to control a bioreactor using a neural net dyna-q system. In *Proceedings of the Seventh Yale Workshop on Adaptive and Learning Systems*, pages 167--172, Center for Systems Science, Dunham Laboratory, Yale University.

Miller et al., 1994
Miller, W. T., Scalera, S. M., and Kim, A. (1994). Neural network control of dynamic balance for a biped walking robot. In *Proceedings of the Eighth Yale Workshop on Adaptive and Learning Systems*, pages 156--161, Dunham Laboratory, Yale University. Center for Systems Science.

Minsky, 1954
Minsky, M. L. (1954). *Theory of Neural-Analog Reinforcement Systems and its Application to the Brain-Model Problem*. PhD thesis, Princeton University.

Minsky, 1961
Minsky, M. L. (1961). Steps toward artificial intelligence. *Proceedings of the Institute of Radio Engineers*, 49:8--30. Reprinted in E. A. Feigenbaum and J. Feldman, editors, *Computers and Thought*. McGraw-Hill, New York, 406--450, 1963.

Minsky, 1967
Minsky, M. L. (1967). *Computation: Finite and Infinite Machines*. Prentice Hall, Englewood Cliffs, NJ.

Montague et al., 1996
Montague, P. R., Dayan, P., and Sejnowski, T. J. (1996). A framework for mesencephalic dopamine systems based on predictive hebbian learning. *Journal of Neuroscience*, 16:1936--1947.

Moore, 1990
Moore, A. W. (1990). *Efficient Memory-Based Learning for Robot Control*. PhD thesis, University of Cambridge, Cambridge, UK.

Moore, 1994
Moore, A. W. (1994). The parti-game algorithm for variable resolution reinforcement learning in multidimensional spaces. In Cohen, J. D., Tesauro, G., and Alspector, J., editors, *Advances in Neural Information Processing Systems: Proceedings of the 1993 Conference*, pages 711--718, San Francisco, CA. Morgan Kaufmann.

Moore and Atkeson, 1993
Moore, A. W. and Atkeson, C. G. (1993). Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13:103--130.

Moore et al., 1986
Moore, J. W., Desmond, J. E., Berthier, N. E., Blazis, E. J., Sutton, R. S., and Barto, A. G. (1986). Simulation of the classically conditioned nictitating membrane response by a neuron-like adaptive element: I. Response topography, neuronal firing, and interstimulus intervals. *Behavioural Brain Research*, 21:143--154.

Narendra and Thathachar, 1989
Narendra, K. and Thathachar, M. A. L. (1989). *Learning Automata: An Introduction*. Prentice Hall, Englewood Cliffs, NJ.

Narendra and Thathachar, 1974
Narendra, K. S. and Thathachar, M. A. L. (1974). Learning automata---A survey. *IEEE Transactions on Systems, Man, and Cybernetics*, 4:323--334.

Nie and Haykin, 1996
Nie, J. and Haykin, S. (1996). A dynamic channel assignment policy through q-learning. CRL Report 334, Hamilton, Ontario, Canada L8S 4K1.

Page, 1977
Page, C. V. (1977). Heuristics for signature table analysis as a pattern recognition technique. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-7:77--86.

Parr and Russell, 1995

Parr, R. and Russell, S. (1995). Approximating optimal policies for partially observable stochastic domains. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*.

Pavlov, 1927
Pavlov, P. I. (1927). *Conditioned Reflexes*. Oxford, London.

Pearl, 1984
Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Peng, 1993
Peng, J. (1993). *Efficient Dynamic Programming-Based Learning for Control*. PhD thesis, Northeastern University, Boston, MA.

Peng and Williams, 1993
Peng, J. and Williams, R. J. (1993). Efficient learning and planning within the Dyna framework. *Adaptive Behavior*, 1(4).

Peng and Williams, 1994
Peng, J. and Williams, R. J. (1994). Incremental multi-step q-learning. In Cohen, W. W. and Hirsh, H., editors, *Proceedings of the Eleventh International Conference on Machine Learning*, pages 226--232.

Peng and Williams, 1996
Peng, J. and Williams, R. J. (1996). Incremental multi-step q-learning. *Machine Learning*, 22(1/2/3).

Phansalkar and Thathachar, 1995
Phansalkar, V. V. and Thathachar, M. A. L. (1995). Local and global optimization algorithms for generalized learning automata. *Neural Computation*, 7:950--973.

Poggio and Girosi, 1989
Poggio, T. and Girosi, F. (1989). A theory of networks for approximation and learning. A.I. Memo 1140, Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

Poggio and Girosi, 1990
Poggio, T. and Girosi, F. (1990). Regularization algorithms for learning that are equivalent to multilayer networks. *Science*, 247:978--982.

Powell, 1987
Powell, M. J. D. (1987). Radial basis functions for multivariate interpolation: A

review. In Mason, J. C. and Cox, M. G., editors, *Algorithms for Approximation*. Clarendon Press, Oxford.

Puterman, 1994
Puterman, M. L. (1994). *Markov Decision Problems*. Wiley, NY.

Puterman and Shin, 1978
Puterman, M. L. and Shin, M. C. (1978). Modified policy iteration algorithms for discounted Markov decision problems. *Management Science*, 24:1127--1137.

Reetz, 1977
Reetz, D. (1977). Approximate solutions of a discounted Markovian decision process. *Bonner Mathematische Schriften, vol 98: Dynamische Optimierung*, pages 77--92.

Ring, 1994
Ring, M. B. (1994). *Continual Learning in Reinforcement Environments*. PhD thesis, University of Texas at Austin, Austin, Texas 78712.

Rivest and Schapire, 1987
Rivest, R. L. and Schapire, R. E. (1987). Diversity-based inference of finite automata. In *Proceedings of the Twenty-Eighth Annual Symposium on Foundations of Computer Science*, pages 78--87.

Robbins, 1952
Robbins, H. (1952). Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58:527--535.

Robertie, 1992
Robertie, B. (1992). Carbon versus silicon: Matching wits with TD-gammon. *Inside Backgammon*, 2(2):14--22.

Rosenblatt, 1961
Rosenblatt, F. (1961). *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books, 6411 Chillum Place N.W., Washington, D.C.

Ross, 1983
Ross, S. (1983). *Introduction to Stochastic Dynamic Programming*. Academic Press, New York.

Rubinstein, 1981
Rubinstein, R. Y. (1981). *Simulation and the Monte Carlo Method*. Wiley, NY.

Rumelhart et al., 1986

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representations by error propagation. In Rumelhart, D. E. and McClelland, J. L., editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, vol.1: Foundations*. Bradford Books/MIT Press, Cambridge, MA.

Rummery, 1995

Rummery, G. A. (1995). *Problem Solving with Reinforcement Learning*. PhD thesis, Cambridge University.

Rummery and Niranjan, 1994

Rummery, G. A. and Niranjan, M. (1994). On-line q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR 166, Cambridge University Engineering Department.

Russell and Norvig, 1995

Russell, S. and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, NJ.

Rust, 1996

Rust, J. (1996). Numerical dynamic programming in economics. In Amman, H., Kendrick, D., and Rust, J., editors, *Handbook of Computational Economics*, pages 614--722. Elsevier, Amsterdam.

S. J. Bradtke, 1994

S. J. Bradtke, B. E. Ydstie, A. G. B. (1994). Adaptive linear quadratic control using policy iteration. In *Proceedings of the American Control Conference*.

Samuel, 1959

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, pages 210--229. Reprinted in E. A. Feigenbaum and J. Feldman, editors, *Computers and Thought*, McGraw-Hill, New York, 1963.

Samuel, 1967

Samuel, A. L. (1967). Some studies in machine learning using the game of checkers. II---Recent progress. *IBM Journal on Research and Development*, pages 601--617.

Schultz and Melsa, 1967

Schultz, D. G. and Melsa, J. L. (1967). *State Functions and Linear Control Systems*. McGraw-Hill, New York.

Schultz et al., 1997
Schultz, W., Dayan, P., and Montague, P. R. (1997). A neural substrate of prediction and reward. *Science*, 275:1593--1598.

Schwartz, 1993
Schwartz, A. (1993). A reinforcement learning method for maximizing undiscounted rewards. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 298--305. Morgan Kaufmann.

Schweitzer and Seidmann, 1985
Schweitzer, P. J. and Seidmann, A. (1985). Generalized polynomial approximations in Markovian decision processes. *Journal of Mathematical Analysis and Applications*, 110:568--582.

Selfridge et al., 1985
Selfridge, O. J., Sutton, R. S., and Barto, A. G. (1985). Training and tracking in robotics. In Joshi, A., editor, *Proceedings of the Ninth International Joint Conference of Artificial Intelligence*, pages 670--672, San Mateo, CA. Morgan Kaufmann.

Shannon, 1950a
Shannon, C. E. (1950a). A chess-playing machine. *Scientific American*, 182:48--51.

Shannon, 1950b
Shannon, C. E. (1950b). Programming a computer for playing chess. *Philosophical Magazine*, 41:256--275.

Shewchuk and Dean, 1990
Shewchuk, J. and Dean, T. (1990). Towards learning time-varying functions with high input dimensionality. In *Proceedings of the Fifth IEEE International Symposium on Intelligent Control*, pages 383--388. IEEE.

Singh, 1992a
Singh, S. P. (1992a). Reinforcement learning with a hierarchy of abstract models. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 202--207, Menlo Park, CA. AAAI Press/MIT Press.

Singh, 1992b
Singh, S. P. (1992b). Scaling reinforcement learning algorithms by learning variable temporal resolution models. In *Proceedings of the Ninth International Machine Learning Conference*, pages 406--415, San Mateo, CA. Morgan Kaufmann.

Singh, 1993

Singh, S. P. (1993). *Learning to Solve Markovian Decision Processes*. PhD thesis, University of Massachusetts, Amherst. Appeared as CMPSCI Technical Report 93-77.

Singh and Bertsekas, 1997

Singh, S. P. and Bertsekas, D. (1997). Reinforcement learning for dynamic channel allocation in cellular telephone systems. In *Advances in Neural Information Processing Systems: Proceedings of the 1996 Conference*, Cambridge, MA. MIT Press.

Singh et al., 1994

Singh, S. P., Jaakkola, T., and Jordan, M. I. (1994). Learning without state-estimation in partially observable Markovian decision problems. In Cohen, W. W. and Hirsch, H., editors, *Proceedings of the Eleventh International Conference on Machine Learning,* pages 284--292, San Francisco, CA. Morgan Kaufmann.

Singh et al., 1995

Singh, S. P., Jaakkola, T., and Jordan, M. I. (1995). Reinforcement learing with soft state aggregation. In G. Tesauro, D. Touretzky, T. L., editor, *Advances in Neural Information Processing Systems: Proceedings of the 1994 Conference,* pages 359--368, Cambridge, MA. MIT Press.

Singh and Sutton, 1996

Singh, S. P. and Sutton, R. S. (1996). Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22:123--158.

Sivarajan et al., 1990

Sivarajan, K. N., McEliece, R. J., and Ketchum, J. W. (1990). Dynamic channel assignment in cellular radio. In *Proceedings of the 40th Vehicular Technology Conference*, pages 631--637.

Skinner, 1938

Skinner, B. F. (1938). *The Behavior of Organisms*. Appleton-Century, NY.

Sofge and White, 1992

Sofge, D. A. and White, D. A. (1992). Applied learning: Optimal control for manufacturing. In White, D. A. and Sofge, D. A., editors, *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches,* pages 259--281. Van Nostrand Reinhold, New York.

Spong, 1994

Spong, M. W. (1994). Swing up control of the acrobot. In *Proceedings of the 1994*

IEEE Conference on Robotics and Automation, San Diego, CA.

Staddon, 1983
Staddon, J. E. R. (1983). *Adaptive Behavior and Learning*. Cambridge University Press, Cambridge.

Sutton, 1978a
Sutton, R. S. (1978a). Learning theory support for a single channel theory of the brain.

Sutton, 1978b
Sutton, R. S. (1978b). Single channel theory: A neuronal theory of learning. *Brain Theory Newsletter*, 4:72--75.

Sutton, 1978c
Sutton, R. S. (1978c). A unified theory of expectation in classical and instrumental conditioning.

Sutton, 1984
Sutton, R. S. (1984). *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, University of Massachusetts, Amherst, MA.

Sutton, 1988
Sutton, R. S. (1988). Learning to predict by the method of temporal differences. *Machine Learning*, 3:9--44.

Sutton, 1990
Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 216--224, San Mateo, CA. Morgan Kaufmann.

Sutton, 1991a
Sutton, R. S. (1991a). Dyna, an integrated architecture for learning, planning, and reacting. *SIGART Bulletin*, 2:160--163. Also appeared in *Working Notes of the 1991 AAAI Spring Symposium*, pages 151--155.

Sutton, 1991b
Sutton, R. S. (1991b). Planning by incremental dynamic programming. In Birnbaum, L. A. and Collins, G. C., editors, *Proceedings of the Eighth International Workshop on Machine Learning*, pages 353--357, San Mateo, CA. Morgan Kaufmann.

Sutton, 1992

Sutton, R. S., editor (1992). *A Special Issue of Machine Learning on Reinforcement Learning*, volume 8. *Machine Learning*. Also published as *Reinforcement Learnng*, Kluwer Academic Press, Boston, MA 1992.

Sutton, 1995

Sutton, R. S. (1995). TD models: Modeling the world at a mixture of time scales. In Prieditis, A. and Russell, S., editors, *Proceedings of the Twelfth International Conference on Machine Learning*, pages 531--539, San Francisco, CA. Morgan Kaufmann.

Sutton, 1996

Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. In Touretzky, D. S., Mozer, M. C., and Hasselmo, M. E., editors, *Advances in Neural Information Processing Systems: Proceedings of the 1995 Conference*, pages 1038--1044, Cambridge, MA. MIT Press.

Sutton and Barto, 1981a

Sutton, R. S. and Barto, A. G. (1981a). An adaptive network that constructs and uses an internal model of its world. *Cognition and Brain Theory*, 3:217--246.

Sutton and Barto, 1981b

Sutton, R. S. and Barto, A. G. (1981b). Toward a modern theory of adaptive networks: Expectation and prediction. *Psychological Review*, 88:135--170.

Sutton and Barto, 1987

Sutton, R. S. and Barto, A. G. (1987). A temporal-difference model of classical conditioning. In *Proceedings of the Ninth Annual Conference of the Cognitive Science Society*, Hillsdale, NJ. Erlbaum.

Sutton and Barto, 1990

Sutton, R. S. and Barto, A. G. (1990). Time-derivative models of pavlovian reinforcement. In Gabriel, M. and Moore, J., editors, *Learning and Computational Neuroscience: Foundations of Adaptive Networks*, pages 497--537. MIT Press, Cambridge, MA.

Sutton and Pinette, 1985

Sutton, R. S. and Pinette, B. (1985). The learning of world models by connectionist networks. In *Proceedings of the Seventh Annual Conference of the Cognitive Science Society*, Irvine, CA.

Sutton and Singh, 1994

Sutton, R. S. and Singh, S. (1994). On bias and step size in temporal-difference learning. In *Proceedings of the Eighth Yale Workshop on Adaptive and Learning Systems*, pages 91--96, New Haven, CT. Yale University.

Tadepally and Ok, 1994
Tadepally, P. and Ok, D. (1994). H-learning: A reinforcement learning method to optimize undiscounted average reward. Technical Report 94-30-01, Oregon State University.

Tan, 1991
Tan, M. (1991). Learning a cost-sensitive internal representation for reinforcement learning. In Birnbaum, L. A. and Collins, G. C., editors, *Proceedings of the Eighth International Workshop on Machine Learning*, pages 358--362, San Mateo, CA. Morgan Kaufmann.

Tan, 1993
Tan, M. (1993). Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 330--337. Morgan Kaufmann.

Tesauro, 1986
Tesauro, G. J. (1986). Simple neural models of classical conditioning. *Biological Cybernetics*, 55:187--200.

Tesauro, 1992
Tesauro, G. J. (1992). Practical issues in temporal difference learning. *Machine Learning*, 8:257--277.

Tesauro, 1994
Tesauro, G. J. (1994). TD--gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215--219.

Tesauro, 1995
Tesauro, G. J. (1995). Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38:58--68.

Tesauro and Galperin, 1997
Tesauro, G. J. and Galperin, G. R. (1997). On-line policy improvement using monte-carlo search. In *Advances in Neural Information Processing Systems: Proceedings of the 1996 Conference*, Cambridge, MA. MIT Press.

Tham, 1994
Tham, C. K. (1994). *Modular On-Line Function Approximation for Scaling up Reinforcement Learning*. PhD thesis, Cambridge University.

Thathachar and Sastry, 1986

Thathachar, M. A. L. and Sastry, P. S. (1986). Estimator algorithms for learning automata. In *Proceedings of the Platinum Jubilee Conference on Systems and Signal Processing*, Bengalore, India.

Thathachar and Sastry, 1995

Thathachar, M. A. L. and Sastry, P. S. (1995). A new approach to the design of reinforcement schemes for learning automata. *IEEE Transactions on Systems, Man, and Cybernetics*, 15:168--175.

Thompson, 1933

Thompson, W. R. (1933). On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25:285--294.

Thompson, 1934

Thompson, W. R. (1934). On the theory of apportionment. *American Journal of Mathematics*, 57:450--457.

Thorndike, 1911

Thorndike, E. L. (1911). *Animal Intelligence*. Hafner, Darien, Conn.

Thorp, 1966

Thorp, E. O. (1966). *Beat the Dealer: A Winning Strategy for the Game of Twenty-One*. Random House, New York.

Tolman, 1932

Tolman, E. C. (1932). *Purposive Behavior in Animals and Men*. Century, New York.

Tsetlin, 1973

Tsetlin, M. L. (1973). *Automaton Theory and Modeling of Biological Systems*. Academic Press, New York.

Tsitsiklis, 1994

Tsitsiklis, J. N. (1994). Asynchronous stochastic approximation and q-learning. *Machine Learning*, 16:185--202.

Tsitsiklis and Van Roy, 1996

Tsitsiklis, J. N. and Van Roy, B. (1996). Feature-based methods for large scale dynamic programming. *Machine Learning*, 22:59--94.

Tsitsiklis and Van Roy, 1997

Tsitsiklis, J. N. and Van Roy, B. (1997). An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*.

Ungar, 1990

Ungar, L. H. (1990). A bioreactor benchmark for adaptive network-based process control. In Miller, W. T., Sutton, R. S., and Werbos, P. J., editors, *Neural Networks for Control*, pages 387--402. MIT Press, Cambridge, MA.

Varga, 1962

Varga, R. S. (1962). *Matrix Iterative Analysis*. Prentice-Hall, Englewood Cliffs, NJ.

Waltz and Fu, 1965

Waltz, M. D. and Fu, K. S. (1965). A heuristic approach to reinforcment learning control systems. *IEEE Transactions on Automatic Control*, 10:390--398.

Watkins, 1989

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. PhD thesis, Cambridge University, Cambridge, England.

Watkins and Dayan, 1992

Watkins, C. J. C. H. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8:279--292.

Werbos, 1992

Werbos, P. (1992). Approximate dynamic programming for real-time control and neural modeling. In White, D. A. and Sofge, D. A., editors, *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*, pages 493--525. Van Nostrand Reinhold, New York.

Werbos, 1977

Werbos, P. J. (1977). Advanced forecasting methods for global crisis warning and models of intelligence. *General Systems Yearbook*, 22:25--38.

Werbos, 1982

Werbos, P. J. (1982). Applications of advances in nonlinear sensitivity analysis. In Drenick, R. F. and Kosin, F., editors, *System Modeling an Optimization*. Springer-Verlag. Proceedings of the Tenth IFIP Conference, New York, 1981.

Werbos, 1987

Werbos, P. J. (1987). Building and understanding adaptive systems: A statistical/numerical approach to factory automation and brain research. *IEEE Transactions on Systems, Man, and Cybernetics*, pages 7--20.

Werbos, 1988

Werbos, P. J. (1988). Generalization of back propagation with applications to a recurrent gas market model. *Neural Networks*, 1:339--356.

Werbos, 1989

Werbos, P. J. (1989). Neural networks for control and system identification. In *Proceedings of the 28th Conference on Decision and Control*, pages 260--265, Tampa, Florida.

Werbos, 1990

Werbos, P. J. (1990). Consistency of HDP applied to simple reinforcement learning problem. *Neural Networks*, 3:179--189.

White, 1969

White, D. J. (1969). *Dynamic Programming*. Holden-Day, San Francisco.

White, 1985

White, D. J. (1985). Real applications of Markov decision processes. *Interfaces*, 15:73--83.

White, 1988

White, D. J. (1988). Further real applications of Markov decision processes. *Interfaces*, 18:55--61.

White, 1993

White, D. J. (1993). A survey of applications of Markov decision processes. *Journal of the Operational Research Society*, 44:1073--1096.

Whitehead and Ballard, 1991

Whitehead, S. D. and Ballard, D. H. (1991). Learning to perceive and act by trial and error. *Machine Learning*, 7(1):45--83.

Whitt, 1978

Whitt, W. (1978). Approximations of dynamic programs I. *Mathematics of Operations Research*, 3:231--243.

Whittle, 1982

Whittle, P. (1982). *Optimization over Time*, volume 1. Wiley, NY.

Whittle, 1983

Whittle, P. (1983). *Optimization over Time*, volume 2. Wiley, NY.

Widrow et al., 1973

Widrow, B., Gupta, N. K., and Maitra, S. (1973). Punish/reward: Learning with a critic in adaptive threshold systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 5:455--465.

Widrow and Hoff, 1960
Widrow, B. and Hoff, M. E. (1960). Adaptive switching circuits. In *1960 WESCON Convention Record Part IV*, pages 96--104. Reprinted in J. A. Anderson and E. Rosenfeld, *Neurocomputing: Foundations of Research*, MIT Press, Cambridge, MA, 1988.

Widrow and Smith, 1964
Widrow, B. and Smith, F. W. (1964). Pattern-recognizing control systems. In *Computer and Information Sciences (COINS) Proceedings*, Washington, D.C. Spartan.

Widrow and Stearns, 1985
Widrow, B. and Stearns, S. D. (1985). *Adaptive Signal Processing*. Prentice-Hall, Inc., Englewood Cliffs, N.J.

Williams, 1986
Williams, R. J. (1986). Reinforcement learning in connectionist networks: A mathematical analysis. Technical Report ICS 8605, Institute for Cognitive Science, University of California at San Diego, La Jolla, CA.

Williams, 1987
Williams, R. J. (1987). Reinforcement-learning connectionist systems. Technical Report NU-CCS-87-3, College of Computer Science, Northeastern University, Boston, MA.

Williams, 1988
Williams, R. J. (1988). On the use of backpropagation in associative reinforcement learning. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 263--270, San Diego, CA.

Williams, 1992
Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229--256.

Williams and Baird, 1990
Williams, R. J. and Baird, L. C. (1990). A mathematical analysis of actor-critic architectures for learning optimal controls through incremental dynamic programming. In *Proceedings of the Sixth Yale Workshop on Adaptive and Learning Systems*, pages 96--101, New Haven, CT.

Wilson, 1994
Wilson, S. W. (1994). ZCS: A zeroth order classifier system. *Evolutionary Compuation*, 2:1--18.

Witten, 1976
Witten, I. H. (1976). The apparent conflict between estimation and control---A survey of the two-armed problem. *Journal of the Franklin Institute,* 301:161--189.

Witten, 1977
Witten, I. H. (1977). An adaptive optimal controller for discrete-time Markov environments. *Information and Control*, 34:286--295.

Witten and Corbin, 1973
Witten, I. H. and Corbin, M. J. (1973). Human operators and automatic adaptive controllers: A comparative study on a particular control task. *International Journal of Man-Machine Studies*, 5:75--104.

Yee et al., 1990
Yee, R. C., Saxena, S., Utgoff, P. E., and Barto, A. G. (1990). Explaining temporal differences to create useful concepts for evaluating states. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 882--888, Cambridge, MA.

Young, 1984
Young, P. (1984). *Recursive Estimation and Time-Series Analysis*. Springer-Verlag.

Zhang and Yum, 1989
Zhang, M. and Yum, T. P. (1989). Comparisons of channel-assignment strategies in cellular mobile telephone systems. *IEEE Transactions on Vehicular Technology,* 38.

Zhang, 1996
Zhang, W. (1996). *Reinforcement Learning for Job-shop Scheduling*. PhD thesis, Oregon State University. Tech Report CS-96-30-1.

Zhang and Dietterich, 1995
Zhang, W. and Dietterich, T. G. (1995). A reinforcement learning approach to job-shop scheduling. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1114--1120.

Zhang and Dietterich, 1996
Zhang, W. and Dietterich, T. G. (1996). High-performance job-shop scheduling

with a time--delay TD(λ) network. In D. S. Touretzky, M. C. Mozer, M. E. H., editor, *Advances in Neural Information Processing Systems: Proceedings of the 1995 Conference*, pages 1024--1030, Cambridge, MA. MIT Press.

Richard Sutton
Fri May 30 21:03:54 EDT 1997

Next Up Previous
Next: About this document **Up:** Contents **Previous:** References

Summary of Notation

t	discrete time step
T	final time step of an episode
s_t	state at t
a_t	action at t
r_t	reward at t, dependent, like s_t, on a_{t-1} and s_{t-1}
R_t	return (cumulative discounted reward) following t
$R_t^{(n)}$	n-step return (Section 7.1)
R_t^λ	λ-return (Section 7.2)
π	policy
$\pi(s)$	action taken in state s under *deterministic* policy π
$\pi(s,a)$	probability of taking action a in state s under stochastic policy π
$\mathcal{S}$	set of all nonterminal states
$\mathcal{S}^+$	set of all states, includng the terminal state
$\mathcal{A}(s)$	set of actions possible in state s
$\mathcal{P}_{ss'}^a$	probability of transition from state s to state s' under action a

$\mathcal{R}^a_{ss'}$	expected immediate reward on transition $s \rightsquigarrow s'$ under action **a**
$V^\pi(s)$	value of state **s** under policy π (expected return)
$V^*(s)$	value of state **s** under the optimal policy
V, V_t	estimates of V^π or V^*
$Q^\pi(s, a)$	value of taking action **a** in state **s** under policy π
$Q^*(s, a)$	value of taking action **a** in state **s** under the optimal policy
Q, Q_t	estimates of Q^π or Q^*
δ_t	temporal-difference error at **t**
$e_t(s)$	eligibility trace for state **s** at **t**
$e_t(s, a)$	eligibility trace for a state-action pair
γ	discount-rate parameter
α, β	step-size parameters
λ	decay-rate parameterfor eligibility traces

Richard Sutton
Fri May 30 21:25:06 EDT 1997

About this document ...

This document was generated using the **LaTeX**2`HTML` translator Version 95.1 (Fri Jan 20 1995) Copyright © 1993, 1994, Nikos Drakos, Computer Based Learning Unit, University of Leeds.

The command line arguments were:
latex2html `-show_section_numbers book.tex`.

The translation was initiated by Richard Sutton on Fri May 30 21:25:06 EDT 1997

Richard Sutton
Fri May 30 21:25:06 EDT 1997

Next Up Previous
Next: Summary of Notation

Richard Sutton
Fri May 30 21:25:06 EDT 1997