Part 3: Writeup

Lina Ghanim

**B) Matchbox Reinforcement.**

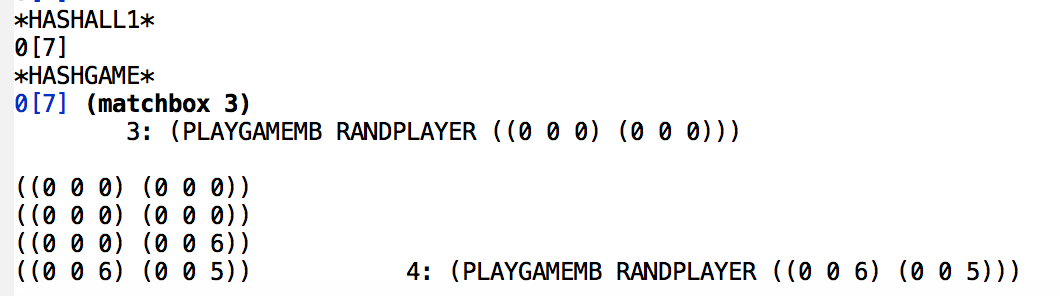
Parameters:

\*hashAll1\*- a big hashtable that stores positions as keys and legal moves as values. It is updated throughout the game and includes duplicates of the winning legal moves which increases the chance of picking them the next time.

\*hashgame\*- a temporary hashtable for the current game playing that stores position as a key and picked (randomly) legal move as a value.

Matchbox

plays n games with randplayer player. It updates the hashAll1 hashtable each game, when a player wins then the winning moves are duplicated 3 times but when it’s a loss then the move is deleted from legal moves. The algorithm loops through each move picked and updates the hashtable accordingly.



valueplayer1

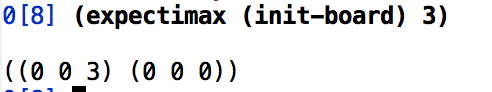
This is made for matchbox, It uses the hash table \*hashAll1\* in order to choose the best move to play. The best move would be the one repeated most in the values list.

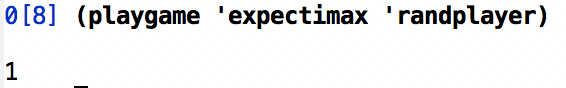




EXPECTIMAX

This player chooses the best move according to two levels in advance. It looks at each legal move and chooses the one that would give the greatest value. The value is calculated by: (sum of my positions)-(sum of other’s positions)





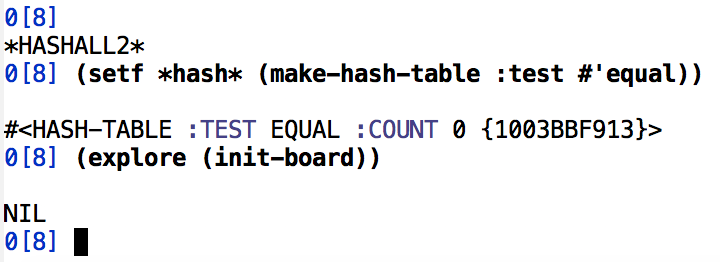
**E)  Bellman's Algorithm**.

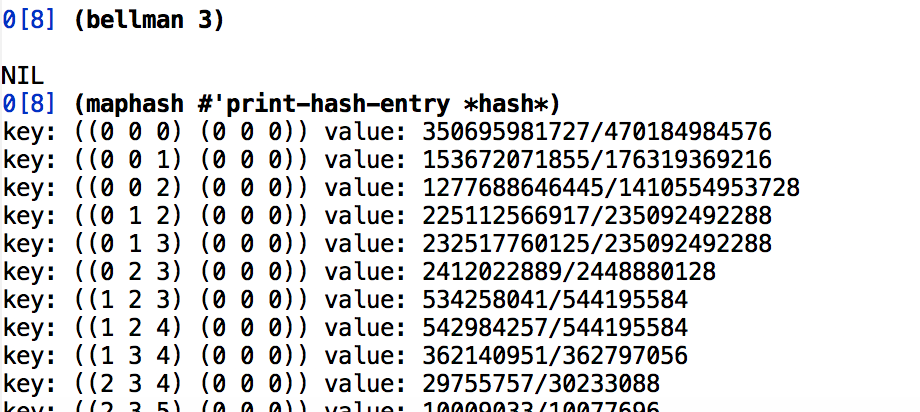
Bellman

Parameters:

\*hash\* - a hash table of each pos and its expected value (a number).

Bellman’s starts by calling explore; starting value of each position is a 0. Then it plays and updates each value to the value of the max next value taking into consideration the probability of each roll of die. At the end \*hash\* is updated.





Valueplayer2

This is made for bellman. It chooses from \*hash\* the legal move with the highest value.



**A)  Pollack & Blair Hillclimbing**.

Hillclimbing

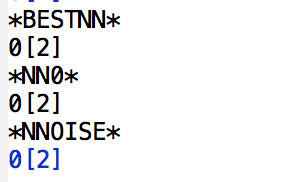
Parameters:

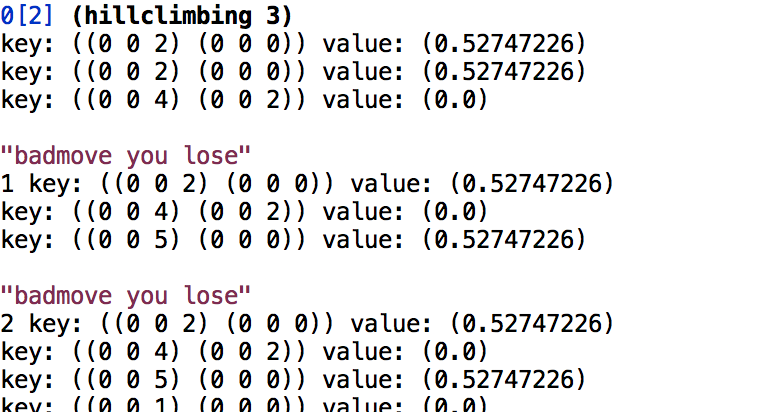
\*bestNN\*- a neural network created at the end with the best values

\*nn0\*- an initial neural network that is played against

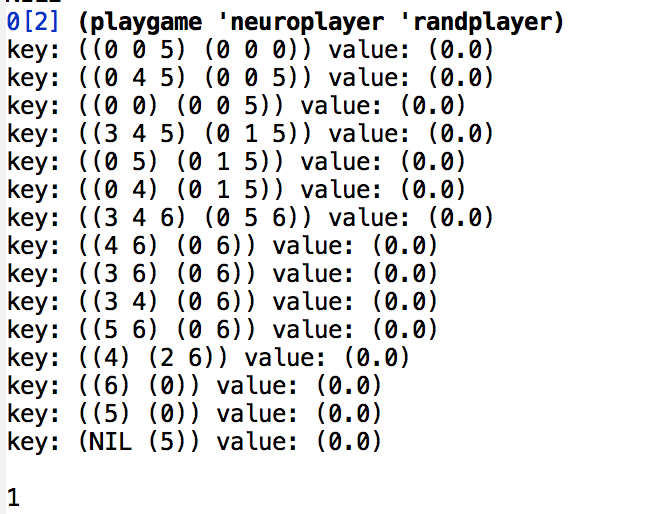
\*nnoise\* - a second neural network that is updated by adding noise to the initial one.

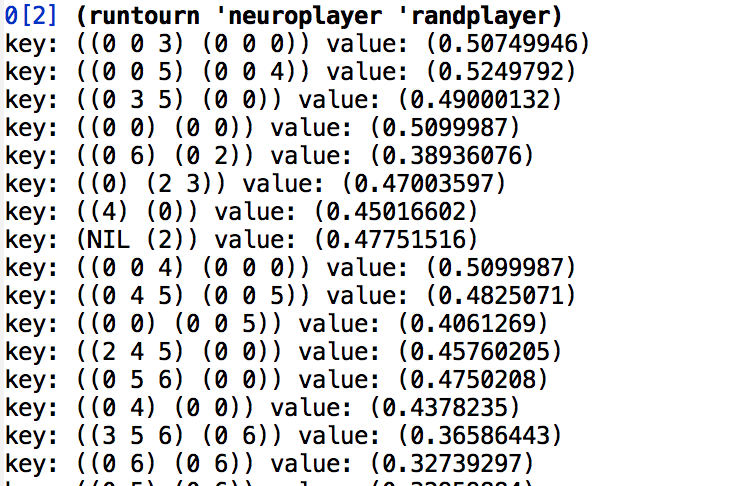
N games are played between \*nn0\* and \*nnoise\* each time updating them according to whoever wins and changing the loser to a better version. At the end, the best neural network is saved and used in the game.





NEUROPLAYER uses a simple 3-layer network \*bestNN\* to estimate values of positions by calling forward-pass and picks the position with the best value to be played.

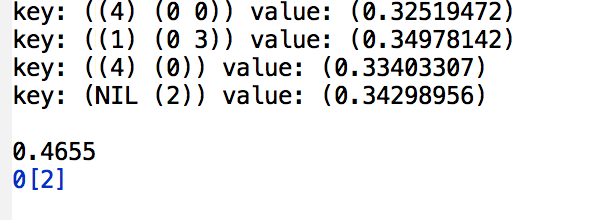




.

.

.



**Compare:**

When comparing the 3 players that I used to play the tournament, the highest number is using matchbox (0.5), while bellman is (0.49) and hillclimbing is (0.46). However, hillclimbing learned the fastest.

**Round-robin:**

Takes very long time.

**Code:**

(defvar \*boardsize\* 6)

(defvar \*pieces\* 3)

(defvar \*die\* 6)

;;this was easy

(defun roll nil (+ 1 (random \*die\*)))

;;instead of letting dice determine first player, we let the PLAY function do it

(defun firstroll ()

(let ((x (random \*die\*))

(y (random \*die\*)))

(if (= x y) (firstroll)(abs(- x y))))) ;;;this could be infinite recursion!

;;;a firstroll followed by n regular rolls

(defun dicestream (&optional (n 2000))

(cons (firstroll) (loop for i from 1 to n collect (roll))))

;;;

;;; My board represention is two lists in sorted order of the

;;; pieces on the board for player1 and player 2. winning means

;;; one of thet lists is empty because player is all off.

;;; it is easy to flip players

(defun swapplayer (board) (list (cadr board)(car board)))

(defun init-board nil (list (loop for i from 1 to \*pieces\* collect 0)

(loop for i from 1 to \*pieces\* collect 0)))

;;;subroutine: what spaces on the board are open for player 1 to move to?

(defun openp1 (board &aux p2)

(setf p2 (loop for x in (cadr board) collect (- (+ \*boardsize\* 1) x)))

(loop for i from 1 to \*boardsize\* unless

(or (member i (car board))

(and (member i p2)(member (+ 1 i) p2))

(and (member i p2)(member (- i 1) p2)))

collect i))

;;which of player 1 checkers can move (without duplicating 0)

(defun legalmoves (board roll)

(remove-duplicates

(loop for chex in (car board) with openp = (openp1 board)

when (or (> (+ chex roll) \*boardsize\*)

(member (+ chex roll) openp)) collect chex)))

;;; given a board, a checker to move, and the number,

;;; create a new board after the move, taking account of hitting

(defun sortup (l)(sort l #'<))

(defun makemove (board checker roll)

(let\* ((nb (copy-tree board));;make a new copy of board

(hit (member (- (+ \*boardsize\* 1) (+ checker roll)) (cadr nb))))

(if (> (+ checker roll) \*boardsize\*)

(setf (car nb) (delete checker (car nb) :count 1)) ;;bear off

(setf (car (member checker (car nb))) (+ checker roll))) ;;add diceroll

(if hit (setf (car hit) 0))

(list (sortup (car nb))(sortup (cadr nb)))));;there was a hit

;;; exploree the game using breadth first search and a hash table

;;; the table needs to compare using EQUAL

(defparameter \*hash\* (make-hash-table :test #'equal))

;;;initial value for the entries are win lose or unknown

(defun hashinit (pos)

(cond ((null (car pos)) 1)

((null (cadr pos)) -1)

(t 0)))

;;;explores the gamespace setting up a hashtable for values

(defun explore (pos)

(unless (gethash pos \*hash\*)

(setf (gethash pos \*hash\*) (hashinit pos))

(loop for position in

(loop for die from 1 to \*die\* append

(loop for move in (legalmoves pos die) append

(list (makemove pos move die))))

do (explore position)))

(unless (gethash (swapplayer pos) \*hash\*)(explore (swapplayer pos))))

(defun print-hash-entry (key value)

(format t "key: ~S value: ~S~%" key value))

;(maphash #'print-hash-entry \*hash\*)

(defun testsize (board piece dice)

(setf \*boardsize\* board)

(setf \*pieces\* piece)

(setf \*die\* dice)

(setf \*hash\* (make-hash-table :test #'equal))

(explore (init-board))

(print (list board piece dice (hash-table-count \*hash\*))))

;;(testsize 6 3 6)

;;(testsize 10 4 6)

;;(testsize 12 5 6)

(defun pick (l)

(and l (nth (random (length l)) l)))

;;make a random move

(defun randplayer (board roll)

(makemove board (pick (legalmoves board roll)) roll))

;;human player as player 1 checks for legal move

(defun human (pos roll &aux move)

(printboard pos)

(princ "you rolled: ")(princ roll)(princ " move? ")(terpri)

(setf move (read))

(if (member move (legalmoves pos roll))

(makemove pos move roll)

(progn (princ "try again")(terpri)(human pos roll))))

;;utility to do (boarddsize+1)-location

(defun bflip (halfpos)

(sortup (loop for i in halfpos collect (- (+ \*boardsize\* 1) i))))

;;;prints a board using 0 for player 1 and \* for player 2

(defun printboard (pos)

(loop for i from 1 to (- \*pieces\* (length (car pos))) do (princ " "))

(loop for x in (reverse (car pos)) do (princ (if (zerop x) "o" " ")))

(princ "|| ")

(loop for i from 1 to \*boardsize\* do

(princ (if (member i (car pos))

"o "

(if (member i (bflip (cadr pos))) "\* " " "))))

(princ "||")

(loop for x in (cadr pos) do (princ (if (zerop x) "\*" " ")))

(terpri)

(loop for i from 1 to \*pieces\* do (princ " "))

(princ "||")

(loop for i from 1 to \*boardsize\* do (if (< i 10) (princ " ")) (princ i))

(if (< \*boardsize\* 10) (princ " "))

(princ "||")

(loop for i from 1 to \*pieces\* do (princ " "))

(terpri))

(defun gameoverp (pos) (or (null (car pos))(null (cadr pos))))

;;similar to hashinit but that cant use nil

(defun whowon (pos)

(cond ((null (car pos)) 1)

((null (cadr pos)) 2)

(t 0)))

;;;plays two funarg as strategies

;;recursively called with dice-stream and position for player 1

(defun playgame (strat1 strat2 &optional dstream pos verbose &aux lm roll mv)

(if (null dstream)(setf dstream (dicestream)))

(if (null pos) (setf pos (copy-tree (init-board))))

(if (gameoverp pos)

(whowon pos)

(progn (setf roll (pop dstream))

(setf lm (legalmoves pos roll))

(if lm

(if (member (setf mv (funcall strat1 pos roll))

(loop for i in lm collect (makemove pos i roll)) :test #'equal)

(- 3 (playgame strat2 strat1 dstream (swapplayer mv)))

(progn (print "badmove you lose") 2))

(progn (if verbose (print "forfeit trn"))

(- 3 (playgame strat2 strat1 dstream (swapplayer pos))))))))

;;(playgame 'human 'randplay) works

(defun runtourn (strat1 strat2 &optional (pairs 1000) &aux dstream total)

(setf total 0)

(loop for i from 0 to (- pairs 1) do

(setf dstream (dicestream))

(if (= 1 (playgame strat1 strat2 dstream))

(setf total (+ 1 total)))

(if (= 2 (playgame strat2 strat1 dstream))

(setf total (+ 1 total))))

(/ total (\* 2.0 pairs)))

(defun roundrobin (list-of-strats &optional( pairs 5000))

(loop for x on list-of-strats do

(print (cons (runtourn (car x)(car x) pairs)(loop for y in (cdr x) collect (runtourn (car x) y pairs))))))

(defun lastplayer (pos roll)

(makemove pos (car (legalmoves pos roll)) roll))

(defun firstplayer (pos roll)

(makemove pos (car (reverse (legalmoves pos roll))) roll))

(defvar testpos '((3 4)(0 1 2)))

;;human player as player 1 checks for legal move

(defun human (pos roll &aux move)

(printboard pos)

(princ "you rolled: ")(princ roll)(princ " move? ")(terpri)

(setf move (read))

(if (member move (legalmoves pos roll))

(makemove pos move roll)

(progn (princ "try again")(terpri)(human pos roll))))

;;utility to do (boarddsize+1)-location

(defun bflip (halfpos)

(sortup (loop for i in halfpos collect (- (+ \*boardsize\* 1) i))))

;;;prints a board using 0 for player 1 and \* for player 2

(defun printboard (pos)

(loop for i from 1 to (- \*pieces\* (length (car pos))) do (princ " "))

(loop for x in (reverse (car pos)) do (princ (if (zerop x) "o" " ")))

(princ "|| ")

(loop for i from 1 to \*boardsize\* do

(princ (if (member i (car pos))

"o "

(if (member i (bflip (cadr pos))) "\* " " "))))

(princ "||")

(loop for x in (cadr pos) do (princ (if (zerop x) "\*" " ")))

(terpri)

(loop for i from 1 to \*pieces\* do (princ " "))

(princ "||")

(loop for i from 1 to \*boardsize\* do (if (< i 10) (princ " ")) (princ i))

(if (< \*boardsize\* 10) (princ " "))

(princ "||")

(loop for i from 1 to \*pieces\* do (princ " "))

(terpri))

(defun gameoverp (pos) (or (null (car pos))(null (cadr pos))))

;;similar to hashinit but that cant use nil

(defun whowon (pos)

(cond ((null (car pos)) 1)

((null (cadr pos)) 2)

(t nil)))

;;;plays two funarg as strategies

;;recursively called with dice-stream and position for player 1

(defun playgame (strat1 strat2 &optional dstream pos verbose &aux lm roll mv)

(if (null dstream)(setf dstream (dicestream)))

(if (null pos) (setf pos (copy-tree (init-board))))

(if (gameoverp pos)

(whowon pos)

(progn (setf roll (pop dstream))

(setf lm (legalmoves pos roll))

(if lm

(if (member (setf mv (funcall strat1 pos roll))

(loop for i in lm collect (makemove pos i roll)) :test #'equal)

(- 3 (playgame strat2 strat1 dstream (swapplayer mv)))

(progn (print "badmove you lose") 2))

(progn (if verbose (print "forfeit trn"))

(- 3 (playgame strat2 strat1 dstream (swapplayer pos))))))))

;;(playgame 'human 'randplay) works

(defun runtourn (strat1 strat2 &optional (pairs 1000) &aux dstream total)

(setf total 0)

(loop for i from 0 to (- pairs 1) do

(setf dstream (dicestream))

(if (= 1 (playgame strat1 strat2 dstream))

(setf total (+ 1 total)))

(if (= 2 (playgame strat2 strat1 dstream))

(setf total (+ 1 total))))

(/ total (\* 2.0 pairs)))

(defun roundrobin (list-of-strats &optional( pairs 5000))

(loop for x on list-of-strats do

(print (cons (runtourn (car x)(car x) pairs)(loop for y in (cdr x) collect (runtourn (car x) y pairs))))))

;;some simple strategies to test in tournaments

(defun lastplayer (pos roll)

(makemove pos roll (car (legalmoves pos roll))))

(defun firstplayer (pos roll)

(makemove pos roll (car (reverse (legalmoves pos roll)))))

(defun index (x l)

(cond ((null l) (error "member not in list"))

((equal x (car l)) 0)

(t (1+ (index x (cdr l))))))

;;look at myscore-yourscore after prospective moves, the best is the

;;minimum, not the maximum! this could be a model for a value-function

;;or neural net value player

(defun scoreplayer (pos roll &aux boards values)

(setf boards (loop for chex in (legalmoves pos roll)

collect (makemove pos chex roll)))

(setf values (loop for move in boards

collect (- (apply #'+ (car move))

(apply #'+ (bflip (cadr move))))))

(nth (index (apply #'min values) values) boards)))

;;returns upper triangular (including self-play diagonal) tournament values

;;could be improved with better display

(defun roundrobin (list-of-strats &optional( pairs 5000))

(loop for x on list-of-strats do

(print (cons (runtourn (car x)(car x) pairs)

(loop for y in (cdr x) collect

(runtourn (car x) y pairs))))))

(defparameter \*hashAll1\* (make-hash-table :test #'equal))

(defparameter \*hashgame\* (make-hash-table :test #'equal))

;-----------------------------------------------------------

(defun matchbox (n) ;-works

(if (= 0 n) \*hashAll1\* ;played n games

(progn

(setf \*hashgame\* (make-hash-table :test #'equal))

(playgamemb 'randplayer (init-board)) ;play game

(matchbox (- n 1)))))

(defparameter dstreamroll (dicestream));25 random rolls

(defun playgamemb (other pos)

(if (gameoverp pos);game over

;if won- add 3 to each box

;if lost-delete to each box

(let ((w (whowon pos)))

(print w)

(if (= w -1);I lost

(loop for key being the hash-key of \*hashgame\* do

(let ((picked (gethash key \*hashgame\*)))

(setf (gethash key \*hashAll1\*)

(delete picked (gethash key \*hashAll1\*)))))

(loop for key being the hash-key of \*hashgame\* do

(let ((picked (gethash key \*hashgame\*)))

(setf (gethash key \*hashAll1\*)

(append (gethash key \*hashAll1\*)

(loop for j from 0 to 3 collect picked)))))))

;game not over

(progn

;hashAll: save state and legalmoves if not already there

;hashgame: save state and picked move, chosen randomly

;play it

;let other play

; recursive call

(print pos)

(let ((lm (if (null (gethash pos \*hashgame\*))

(legalmoves pos (pop dstreamroll))

(gethash pos \*hashAll1\*))));if exit already dont redo legalmoves

(setf (gethash pos \*hashAll1\*) lm);save all legal moves

(print pos)

(let ((pickedmove (nth (random (length lm)) lm)))

(setf (gethash pos \*hashgame\*) pickedmove);save chosen move

(let ((swapmymove (swapplayer (makemove pos pickedmove (pop dstreamroll)))))

;(print swapmymove)

(let ((nextpos (swapplayer (funcall other swapmymove (car dstreamroll)))));after I play and my opponent plays

;(print nextpos)

(playgamemb other nextpos))))))))

;(playgamemb 'randplayer (init-board))

(defun valueplayer1 (pos roll)

;(matchbox 5 'firstplay)

(let ((move (maxrepeated pos roll)))

(makemove pos move roll)))

(defun maxrepeated (pos roll)

(let ((lm (gethash pos \*hashAll1\*)));list of legal moves

(if (null lm) (pick (legalmoves pos roll))

(nth (+ 1 (random (- (length lm) 2))) lm))));did random but should choose max repeated\*\*\*\*\*

;-----------------------------------------------------------

;legal move with max value

(defun expectimax (pos roll);-works

(let ((boards (loop for chex in (legalmoves pos roll)

collect (makemove pos chex roll))))

(let ((values (loop for move in boards

collect (- (apply #'+ (car move))

(apply #'+ (bflip (cadr move)))))))

(nth (index (apply #'min values) values) boards)))) ;rerurn next pos that gives max value

; (expectimax (init-board) 2)

;-----------------------------------------------------------

(defparameter \*hashAll2\* (make-hash-table :test #'equal))

;(setf \*hash\* (make-hash-table :test #'equal))

;(explore (init-board))

(defun bellman (n) ;update 1 move ahead again and again and again -works

;(if (< n (hash-table-size \*hash\*))

(unless (= n 0)

(progn

(loop for pos being the hash-keys of \*hash\* do

(if (= 0 (hashinit pos))

(setf (gethash pos \*hash\*)

(loop for die from 1 to \*die\* sum ( \* (/ 1 6) (maxval pos die))))))

(bellman (- n 1)))))

; (maphash #'print-hash-entry \*hash\*)

(defun maxval (pos roll); get max value of keys

(defvar maxpos)

(defvar maxv 0)

(loop for p in (loop for move in (legalmoves pos roll) collect (makemove pos move roll))

maximizing (gethash p \*hash\*)));get values of all legal moves

(defun valueplayer2 (pos roll) ;legal move with max value

;(bellman)

(car (car (sort (loop for k in (loop for move in (legalmoves pos roll) collect (makemove pos move roll)) collect (list k (gethash k \*hash\*))) #'> :key 'cadr))))

;---------------------------------------------------------

;;(defvar net221 (setuplayers '(6 6 1)))

;;(noise net221 .1)

;;(learn net221 '((-5 -5)(-5 5)(5 -5)(5 5)) '((0)(1)(1)(0)))

(defparameter \*bestNN\* (setuplayers '(6 6 1)))

(defparameter \*nn0\* (setuplayers '(6 6 1)))

(defparameter \*nnoise\* (setuplayers '(6 6 1)))

(defun hillclimbing (n) ;HELP - nn are nil all the time

(progn

(noise \*nnoise\* .1)

(loop for i from 0 to n do

(let ((w (playgame 'neuroplayer1 'neuroplayer2)))

(progn ;(print w)

(cond ((= 1 w) (setq \*nnoise\* (addnoise \*nnoise\* 0.1)))

((= 2 w)

(progn (setq \*nn0\* \*nnoise\*)

(setq \*nnoise\* (addnoise \*nnoise\* 0.1))))))))

(setq \*bestNN\* \*nn0\*)))

(defun neuroplayer1 (pos roll); input pos and output best move from legal moves \*nn0\*

;(hillclimbing 10);train on 10 games

(nnplayer1 \*nn0\* pos roll))

(defun neuroplayer2 (pos roll); input pos and output best move from legal moves \*nn0\*

;(hillclimbing 10);train on 10 games

(nnplayer1 \*nnoise\* pos roll))

(defparameter \*lm-val\* (make-hash-table :test #'equal))

(Defun nnplayer1 (nn pos roll);\*\*\*closures

;legal move and val of nn in a table and take max

(progn

(loop for lm in (legalmoves pos roll) do

(let ((mm (makemove pos lm roll)))

(let ((fp (forward-pass nn (append (car mm) (cadr mm)) T)))

(setf (gethash mm \*lm-val\*) (if (null fp) '(0.0) fp)))));save legalmoves and their network value

;(maphash #'print-hash-entry \*lm-val\*);\*\*\*forward-pass???? sends nil

(car (car (sort (loop for k being each hash-key of \*lm-val\* using (hash-value v) collect (list k v)) #'> :key 'caadr)))))

(defun neuroplayer (pos roll); input pos and output best move from legal moves \*nn0\*

;(hillclimbing 10);train on 10 games

(setq \*lm-val\* (make-hash-table :test #'equal))

(nnplayer1 \*bestNN\* pos roll))

;---------------------------

;; -\*- Mode: LISP; Syntax: Common-lisp; Package: USER; Base: 10. -\*-

;; a network is a collection of nodes and links

(defstruct bp-node

in-list

(input 0.0 :type float)

(output 0.0 :type float)

(delta-in 0.0 :type float)

(delta-out 0.0 :type float))

(defstruct bp-link

from-node

(weight 0.0 :type float)

(delta 0.0 :type float)

(previous-delta 0.0 :type float))

;;;

;;;The main backprop function which uses two parameters mu and alpha

;;; the halting function is min-dif (i.e. the output is assumed boolean

;;; runs for 1000 cycles then stops but can be run again

(defun learn (network in-list out-list

&optional (mu 0.3) (alpha 0.9) (min-dif 0.2) (limit 1000)

&aux (error -1))

"Uses in-list and out-list to train the network."

;; initial value loop for momentum

(loop for link in (cdr network) do

(setf (bp-link-previous-delta link) 0.0))

;; now do the iterations

(format t "~%")

(loop for cycle from 1 to limit until (zerop error) do

(loop for link in (cdr network) do

(setf (bp-link-delta link) 0.0))

;;now do the forward and backward passes

(setq error

(loop for x in in-list as y in out-list ;cost function

sum (prog2

(forward-pass network x)

(backward-pass network y min-dif min-dif t))))

(format t "Cycle: ~d Error: ~d~%" cycle error)

;;now update weights

(loop for link in (cdr network) do

(incf (bp-link-weight link)

(- (setf (bp-link-previous-delta link)

(+ (\* mu (bp-link-delta link))

(\* alpha (bp-link-previous-delta link)))))))))

(defun forward-pass (network in-list &optional flag)

;; set inputs

(loop for node in (caar network) as i in in-list do

(setf (bp-node-output node) (float i)))

;; do loop forward pass

(loop for level in (cdar network) do

(loop for node in level do

(getinput node)

(setoutput node)))

;;flag to generate output values

(and flag (loop for node in (car (last (car network))) collect

(bp-node-output node))))

;;;subrouting for forward-pass

(defun getinput (node)

(setf (bp-node-input node) 0.0)

(loop for link in (bp-node-in-list node) do

(incf (bp-node-input node)

(\* (bp-link-weight link)

(bp-node-output (bp-link-from-node link))))))

;;subroutine for forward pass

(defun setoutput (node)

(setf (bp-node-output node) (sigmoid (bp-node-input node))))

;;these flags are for debugging

(defvar \*always-bp-flag\* t)

(defvar \*indiv-trace-flag\* t)

;;the backward pass code

(defun backward-pass (network desired tol edge-tol &optional addflag

&aux (error nil))

"Given a desired set of values and a cutoff, perloop forms a backward pass"

;; clear input values

(loop for level in (car network) do

(loop for node in level do

(setf (bp-node-delta-out node) 0.0)))

;; set inputs

(loop for node in (car (last (car network))) as d in desired do

(setf error (or (< (or (and (floatp d) tol) edge-tol)

(abs (setf (bp-node-delta-out node)

(- (bp-node-output node) (or d (bp-node-output node))))))

error))

(setdeltain node)

(errorprop node addflag))

;; unless the output is good enough, do the backward pass

(and \*indiv-trace-flag\* (write-char (or (and error #\\*) #\.)))

(cond

((or error \*always-bp-flag\*)

;; do backward pass

(bpfromtail (car network) addflag)))

(cond (error

;; compute error

(loop for node in (car (last (car network))) sum

(\* (bp-node-delta-out node) (bp-node-delta-out node))))

(t 0.0)))

;;subroutine for backward pass

(defun bpfromtail (levels addflag)

(and (cddr levels) (bpfromtail (cdr levels) addflag))

(loop for node in (car levels) do

(setdeltain node)

(errorprop node addflag)))

;;subrouting for bpfromtail

(defun setdeltain (node)

(setf (bp-node-delta-in node)

(\* (bp-node-delta-out node)

(bp-node-output node)

(- 1.0 (bp-node-output node)))))

;;subroutine from bp from tail

(defun errorprop (node &optional addflag)

(loop for link in (bp-node-in-list node) do

(cond (addflag (incf (bp-link-delta link)

(\* (bp-node-delta-in node)

(bp-node-output (bp-link-from-node link)))))

(t (setf (bp-link-delta link)

(\* (bp-node-delta-in node)

(bp-node-output (bp-link-from-node link))))))

(incf (bp-node-delta-out (bp-link-from-node link))

(\* (bp-link-weight link)(bp-node-delta-in node)))))

;;this is a function to setup a multilayer network (see end of file)

(defun setuplayers (layers &optional otherconnections &aux \*levels\* \*links\* \*biasnode\*)

(setf \*biasnode\* (make-bp-node :output 1.0))

(setf \*levels\* (loop for i in layers collect

(loop for j from 1 to i collect (make-bp-node))))

(loop for level on \*levels\* do

(and (cdr level)

(loop for y in (cadr level) do

(push (make-bp-link :from-node \*biasnode\*) \*links\*)

(push (car \*links\*) (bp-node-in-list y))

(loop for x in (car level) do

(push (make-bp-link :from-node x) \*links\*)

(push (car \*links\*) (bp-node-in-list y))))))

;;i don't recall what otherconnections does, think it is for skipped links

(loop for pair in otherconnections do

(loop for x in (nth (car pair) \*levels\*) do

(loop for y in (nth (cdr pair) \*levels\*) do

(push (make-bp-link :from-node x) \*links\*)

(push (car \*links\*) (bp-node-in-list y)))))

(noise (cons \*levels\* \*links\*) .5)

(cons \*levels\* \*links\*))

;;reset the weights in a network to uniform noise, usually .1 or .5

(defun noise (network n)

"Small � weights."

(loop for link in (cdr network) do

(setf (bp-link-weight link)

(/ (- (random (\* 200 n)) (\* 100.0 n)) 100.0))))

;;this might be useful for hillclimbing

(defun addnoise (network n)

"add Small � weights."

(loop for link in (cdr network) do

(incf (bp-link-weight link)

(/ (- (random (\* 200 n)) (\* 100.0 n)) 100.0))))

;;;sigmoid used to be slow, so this uses linear interpolation to speed up sigmoid

(defun create-sigmoid-lookup-table (size max-value)

"Creates a sigmoid lookup table from -(max-value) to +(max-value) with size elements."

(let ((sigmoid (make-array (1+ size) :initial-element 0)))

(dotimes (i (1+ size))

(setf (aref sigmoid i)

(/ 1.0 (+ 1.0 (exp (- (/ (- i (/ size 2.0))

(/ (/ size 2.0) max-value))))))))

sigmoid))

(defvar \*coarse-sigmoid\* (create-sigmoid-lookup-table 400 20))

(defvar \*fine-sigmoid\* (create-sigmoid-lookup-table 500 5))

(defvar \*extra-fine-sigmoid\* (create-sigmoid-lookup-table 200 1))

(defun sigmoid (n)

"Triple resolution lookup table loop for sigmoid function."

(cond ((< n -20.0)

0.0)

((>= n 20.0)

1.0)

((> (abs n) 5.0)

(aref \*coarse-sigmoid\*

(+ 200 (truncate (+ (\* 10.0 n) (if (minusp n) -0.5 0.5))))))

((> (abs n) 1.0)

(aref \*fine-sigmoid\*

(+ 250 (truncate (+ (\* 50.0 n) (if (minusp n) -0.5 0.5))))))

(t

(aref \*extra-fine-sigmoid\*

(+ 100 (truncate (+ (\* 100.0 n) (if (minusp n) -0.5 0.5))))))))

;;these two functions are a cheap way to store and load weights into a network

(defun getweights (network)

(loop for link in (cdr network) collect (bp-link-weight link)))

(defun setweights (network weights)

(loop for link in (cdr network) as w in weights do (setf (bp-link-weight link) w))

network)

demonstrations

;;(defvar net221 (setuplayers '(2 2 1)))

;;(noise net221 .1)

;;(learn net221 '((-5 -5)(-5 5)(5 -5)(5 5)) '((0)(1)(1)(0)))

;(defvar net424 (setuplayers '(4 2 4)))

;(noise net424 .5)

;(learn net424 '((1 0 0 0)(0 1 0 0)(0 0 1 0)(0 0 0 1)) '((1 0 0 0)(0 1 0 0)(0 0 1 0)(0 0 0 1)))