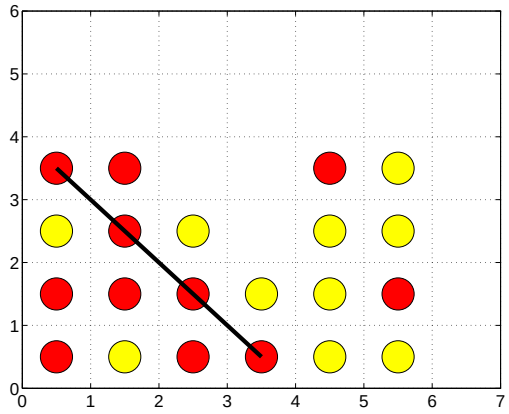
Computational Assignment 2 (Reikniverkefni 2) version 13/2.

Given January Wednesday 29th, due Monday February 17.

Topics: Monte Carlo methods, reinforcement learning and value approximation using n-tuples.

In this computational assignment your goal is to build an intelligent connect four player using reinforcement learning. Connect Four is a two-player game where the players take turn dropping coloured discs from the top into a seven-column by six-row suspended grid. The pieces fall straight down, occupying the next available space within the column. The object of the game is to connect four of one's own discs of the same color next to each other vertically, horizontally, or diagonally before the opponent does so. The MATLAB figure above illustrates an end game state where the "red" player has won.



You have been given the MATLAB function, `connect4simulate.m` which can be used to simulate the resulting states for the different actions in connect four. The function also returns a reward of $+1$ if the player who makes the first move wins, -1 if the other player wins, 0 for a draw, and empty ([]) if the game is still in progress. Using this function it is possible to simulate an entire game of connect four. For example, if purely random moves are selected the game may be simulated as follows:

```
state = zeros(6,7); % the initial grid is empty with all zeros 6x7
winner = [];        % the winner variable is empty since the game is not over
colour = 1;         % player with the first move has colour 1, other colour is -1
while isempty(winner)         % while empty then game not over
  A = find(0 == state(1,:));  % when zero there is no disk in the top row
  action = A(ceil(rand(1)*length(A)));  % random move selected
% now simulate the move and observe resulting state and reward
  [state, winner] = connect4simulate(state, action, colour);
% now its the other player's turn
  colour = -colour;
end
```

Now perform and answer the following (all parts are of equal weight). Make sure to hand in all programs (preferably MATLAB scripts) you write. Summarize your results in a 4-5+ page report with figures/tables, include the relevant code in an appendix. Give also a brief account of what you believe you have learned from this exercise.

1. Monte Carlo player:

   (a) Implement a Monte Carlo (MC) rollout where the rollout policy is purely random play for both players. The pseudocode for a rollout is given in the slides. To perform a rollout for move $a =$`A(i)` one may do the following:

```
[newstate, winner] = connect4simulate(state,A(i),colour);
na(i) = na(i) + 1; % record number of rollouts for move A(i)
if isempty(winner)
  winner = rollout(newstate, -colour); % note its the other player's turn
end
% note that player "colour" was performing the rollout, let the reward be in [0 1]:
R(i) = (winner == colour);        % True (1) if colour wins else false (0)
if (0 == winner), R(i) = 0.5; end % in case of a draw!
```

   Now let a Monte Carlo player, performing only 5 rollouts per feasible move, play against a purely random player. Out of 30 games in each colour, i.e. a total of 60 games, how many games does the MC player win?

   (b) Now modify the rollouts policy such that obvious moves are made, rather than purely random ones. That is, if a player can force a win (or block a win for the opponent) that move should be chosen. This is known as selective sampling for a MC rollout. Now let the Monte Carlo player using selective sampling compete with the Monte Carlo player above. Use 30 games for each colour, i.e. a total of 60 games to determine which player is better. Is there a statistically significant difference?

2. Heuristic value function:

   The heuristic value function can be seen as a function trying to estimate the utility of some state s. Let us denote this function by $h(s)$. Then the heuristic function can be used to determine which move is best at any given state, this is done as follows:

```
A = find(0 == state(1,:));  % when zero there is no disk in the top row (legal moves)
V = zeros(1,length(A));     % initialize memory for values V
for i = 1:length(A)         % examine one move at a time and estimate its value
% now simulate the move and observe the resulting afterstate
  afterstate = connect4simulate (state, A(i), colour);
  V(i) = h(afterstate);     % evaluate the value of the resulting afterstate
end
[dummy,index] = max(V);     % find the move with the highest utility
move = A(index);            % the best move
```

   The code above finds the utility of an afterstate s_a (sometimes called a half-state or post-decision state), which is a combination of the current state and move a taken. Now one is faced with two immediate problems: what should $h(s_a)$ look like and how to estimate $h(s_a)$?

   In this assignment we limit our discussion to linear functions, that is

$$h(s_a) = \sum_{i=1}^{n} w_i \phi_i(s_a)$$

where $\phi(s_a)$ are features of the after-state s_a (these features may be highly non-linear). Perhaps the simplest set of features for connect-4 is just the 6×7 game grid itself, where the value of a feature is $+1$ for a grid location occupied by a red disc, -1 for a yellow disc, and 0 for empty. An example illustrating these features is presented in figure 1 below. Rather than writing the feature as a matrix (variable `state`), one must form a vector of length 42, as follows:

```
>> phi=state(:); % or for the example in figure 1
phi=[0 0 0 -1 1 1 0 0 0 0 0 0 0 0 1 1 -1 -1 0 0 0 0 ...
                                        0 0 0 0 1 -1 -1 1 0 0 0 0 0 0 0 0 0 0 1 -1]';
```

What could one learn using this feature set? What might the corresponding weights w_i be? The weights should tell us how important a particular grid location is for a player. Consider the bottom left grid location, this location is part of a set of 3 potential winning lines. However, the center bottom grid location is part of 7 potential winning lines, obviously more important in this respect. Indeed the number of potential winning lines, each grid location can partake in, is as follows:

```
     3     4     5     7     5     4     3
     4     6     8    10     8     6     4
     5     8    11    13    11     8     5
     5     8    11    13    11     8     5
     4     6     8    10     8     6     4
     3     4     5     7     5     4     3

% That is:
W=[3 4 5 5 4 3 4 6 8 8 6 4 5 8 11 11 8 5 7 10 13 13 10 7 5 8 11 11 8 ...
                                                    5 4 6 8 8 6 4 3 4 5 5 4 3];
% and the value of an afterstate is quickly computed as follows:
V = W*afterstate(:);
```

Could these be the weight we are looking for or is there an alternative?

```
     0     0     0     0     0     0     0
     0     0     0     0     0     0     0
     0     0     1     0     1     0     0
    -1     0     1     0    -1     0     0
     1     0    -1     0    -1     0     1
     1     0    -1     0     1     0    -1
```

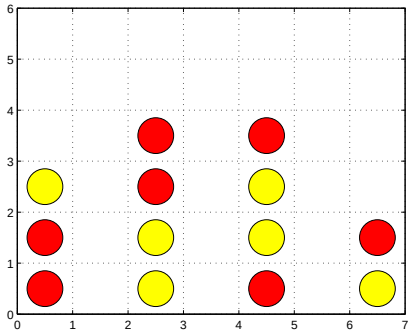


Figure 1: The example shows a simple set of features extracted from the game on the right.

As an example, an alternative features set may be to look the 69 different ways to connect four in a row. Lets say we had a matrix C of size 69×42 with ones at the four locations on the grid in each row. A function has been supplied for this purpose `C=winningpositions;`. Then this function can be used to generate features as follows:

```
function phi = getfeatures(state, C)
% GETFEATURES, usage phi = getfeatures(state, C)

phipos = C*(1 == state(:));
phineg = -C*(-1 == state(:));
phi = phipos + phineg;
phi((phipos ~= 0) & (phineg ~= 0)) = 0;
```

Now experiment (don't spend much time doing so) with a hand-crafted utility function of your own design, and play it against your MC player using selective sampling from part 1. Is it a good player[1]?

3. Reinforcement Learning

Now you are given the following code `connect4td0.m` which performs TD(0) learning using the after-state value function

$$V\big(\phi(s)\big) = \frac{1}{1 + \exp\big(-\sum_{i=1}^{n} w_i \phi_i(s)\big)} \tag{1}$$

for the game. The log-sigmoid squashing function[2] makes it easier to tune α for the learning task. Your task is to:

- extend this code and implement TD(λ) by including eligibility traces as done in chapter 8 in Sutton and Barto,
- and use n-tuple features like the ones in the paper titled: it Reinforcement Learning with N-tuples on the Game Connect-4 by Markus Thill, Patrick Koch and Wolfgang Konen.

Compare the performance of this player with the players in the previous parts, i.e. MC with selective sampling and your own heuristic player.

[1] Note that the performance of heuristic players can be improved by doing game tree search where you allow your opponent to use your heuristics function. When the opponent used your heuristic we typically do a "board inversion", i.e. `colour*state` for our task.

[2] See also the function `logsig` in MATLAB.

```
n = 42; % the number of features, depends on you "getfeatures" function
w = zeros(1,n); % initialize weights for linear value function
epsilon = 0.1; % epsilon-greedy policy, 0.1 is a good value to use here
alpha = 0.01;  % step size alpha, will need to tune this or decrease with episodes
for episodes = 1:10000 % the more the better ...
% initialize the game connect-four
  state = zeros(6,7); % the grid is empty with all zeros 6x7
  winner = []; % the reward is empty, i.e. game is not over
  colour = 1; % player with the first move has colour 1, other colour is -1
% player take turns dropping a disc in their respective colour
  phi = zeros(n,2); % previous feature vectors for both players
  counter = 0; % once both players have made a move we can start updating
  while isempty(winner) % while empty then game not over
    counter = counter + 1; % increment move counter
  % get all feasible actions, A(s)
    A = find(0 == state(1,:)); % when zero there is no disk in the top row
  % use a policy to select a move or action from A, for players 1 and -1
    if (rand(1) < epsilon) % random move
      move = A(ceil(length(A)*rand(1))); % random move
    else % greedy move, uses board inversion
      move = connect4heuristic(state, colour, w);
    end
  % now simulate the move and observe resulting state and reward
    [state, winner] = connect4simulate(state, move, colour);
  % TD update
    i = (colour+1)/2+1; % get index to player colour
    phinew = getfeatures(colour*state); % use board inversion
    if (counter > 2) % update using TD(0) and logsig squashing function
      w = w + alpha*(logsig(w*phinew)-logsig(w*phi(:,i)))*dlogsig(w*phi(:,i))*phi(:
    end
    phi(:,i) = phinew; % store feature to be used later
  % now its the other player's turn
    colour = -colour;
  end
  % let the reward reflect the probability of winning
  if (1 == winner)
    reward = 1;
  elseif (-1 == winner)
    reward = 0;
  else
    reward = 0.5;
  end
  % now the first player is +1 in colour and actually owns the "w" heuristic
  % so we update the first player with:
  w = w + alpha*(reward - logsig(w*phi(:,1)))*dlogsig(w*phi(:,1))*phi(:,1)';
  % however, since we are using board inversion the second player will be
  % updated with the reverse reward which will be (1-reward):
  w = w + alpha*((1-reward) - logsig(w*phi(:,2)))*dlogsig(w*phi(:,2))*phi(:,2)';
end % episode
```