FMAN45 Machine Learning, spring 2020
Assignment 4

Salma Kazemi Rashed

May 2020

1 Reinforcement learning for playing Snake

Exercise 1: Derive the value of K above.
For finding the maximum number of states (some of them could never happen), we can assume the different states of one cell of the game screen as:
1) contains an apple
2) contains the head of snake
3) contains the body of the snake (first part)
4) contains the body of the snake (second part)
Thus, each of the four elements could be in any of the 5×5 cells of the grid.
$K \ll 25 \times 24 \times 23 \times 22$.
The real number of states could be exactly calculated by considering the different cases that the head of snake could be according to Figs. (1–3).

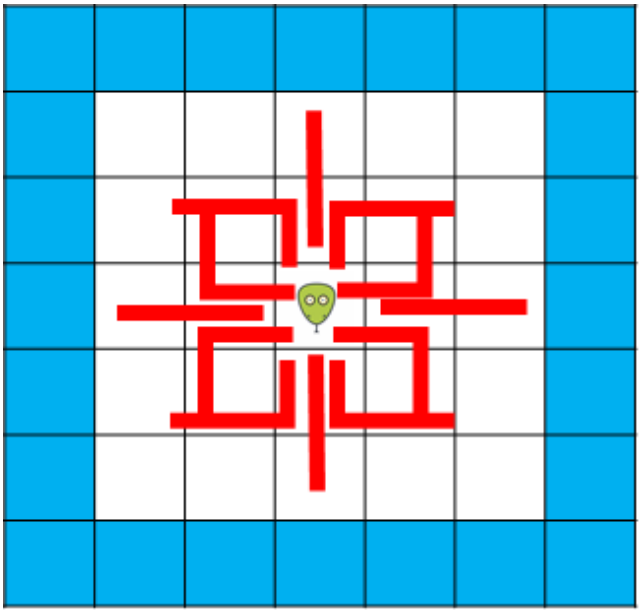

Figure 1: When the head is in the center of the grid. The possible shape of snake could be: $4 + 4 + 4 = 12$.

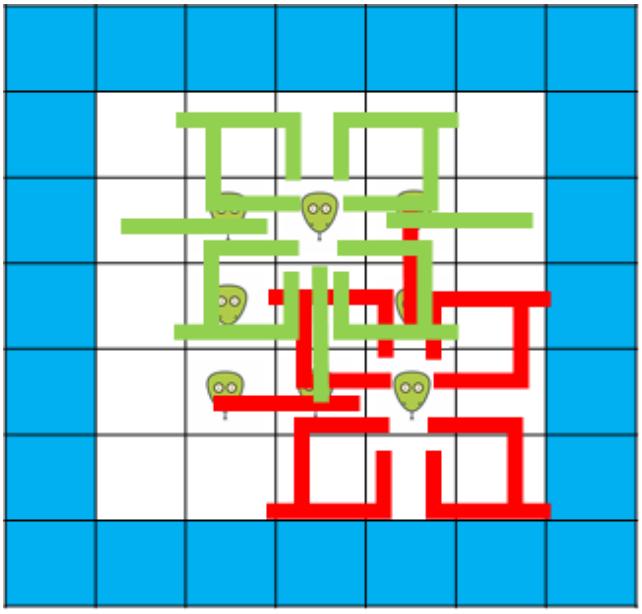

Figure 2: When the head is in the cells around the center cell of the grid. The possible shape of snake could be: $4 * (3 + 4 + 4) + 4 * (2 + 4 + 4) = 84$.

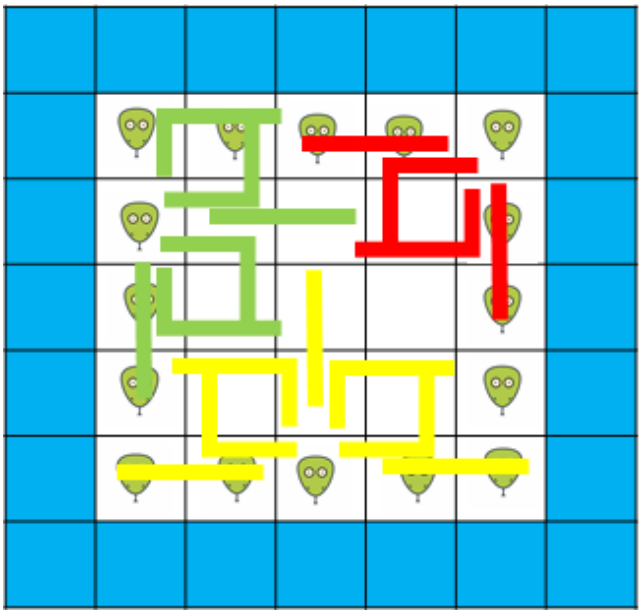

Figure 3: When the head is in the cells in the border of the grid. The possible shape of snake could be: $4 * (2 + 1 + 1) + 8 * (2 + 2 + 2) + 4 * (3 + 2 + 2) = 92$.

In all cases the apple could be in any cell other than the head and body of the snake (22 different cases.)
Thus, in total we will have $22 * (12 + 84 + 92) = 4136$.

Exercise 2: a) Rewrite eq. (1) as an expectation using the notation E[·], where E[·] denotes expected value. Use the same notation as in (1) as much as possible. The expression should still be a recursive relation – no infinite sums or similar.

$$Q^*(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')]. \tag{1}$$

We can rewrite eq. (1) as following: We know that the optimal Q-value (action-value) function is the maximum expected reward given an state-action (s,a) pair. Thus, we have:

$$\begin{aligned} Q^*(s,a) &= \sum_{s'} T(s,a,s')R(s,a,s') + \gamma[\sum_{s'} T(s,a,s') \max_{a'} Q^*(s',a')] \\ &= E(R(s,a,s')|s,a) + \gamma E(\max_{a'} Q^*(s',a')|s,a) \\ &= E(R + \gamma \max_{a'} Q^*(s',a')|s,a) \end{aligned} \tag{2}$$

The expectation comes from the randomness of the environment.

b) Explain what eq. (1) is saying in one sentence.
Eq. (1) is a recursive equation decompose the optimal action-value function into the expectation of immediate reward and expectation of discounted action-value of the successor state (next state).
Q^* is the total expected discounted reward of acting optimally after taking action a in state s.

c) Elaborate your answer in b) with a bullet list referencing all the various components and concepts of the equation $(Q^*, \sum, T, R, \gamma, \max, s, a, s')$.
Eq. (1) is a recursive equation decompose the optimal action-value function $(Q^*(s,a))$ of current state (s) into the expectation $(\sum_{s'} T(s,a,s'))$ of immediate reward (R) and expectation of discounted (γ) optimal action-value $(\max_{a'} Q^*))$ of the successor state (s'). The optimal Q-value (action-value) function,is the maximum expected reward given an state-action (s,a) pair. The maximum operator over a' comes from achieving the optimal policy by taking the action with the highest Q-value.

d) Explain what $T(s,a,s')$ in eq. (1) is for the small version of Snake. You should give values for $T(s,a,s')$, e.g. "$T(s,a,s') = 0.5$ if ... and $T(s,a,s') = 0.66$ if ..." (the correct values are likely different, though).
According the way I calculated the total states in previous question, the transition matrix would be a $3K \times K$ matrix. The elements of such matrix could be 1, 0, or $\frac{1}{22}$. If the position of the apple would be in one the adjacent cells of head of the snake in state s that by taking one of the actions (right, left, forward) the snake could eat the apple, the transition probability would be $(T(s',a,s) = \frac{1}{22})$ to any of the new positions of the apple (25-3=22). In this case 22 of the columns would be $\frac{1}{22}$ and the rest of the columns would be zero. If The apple would be far from the head of the snake that by taking any of the actions, apple could not be eaten by snake, in that case $T(s',a,s) = 1$ for only one column and $T(s',a,s) = 0$ for rest of the columns in state s. The two different cases for state s is shown in Fig. (4)

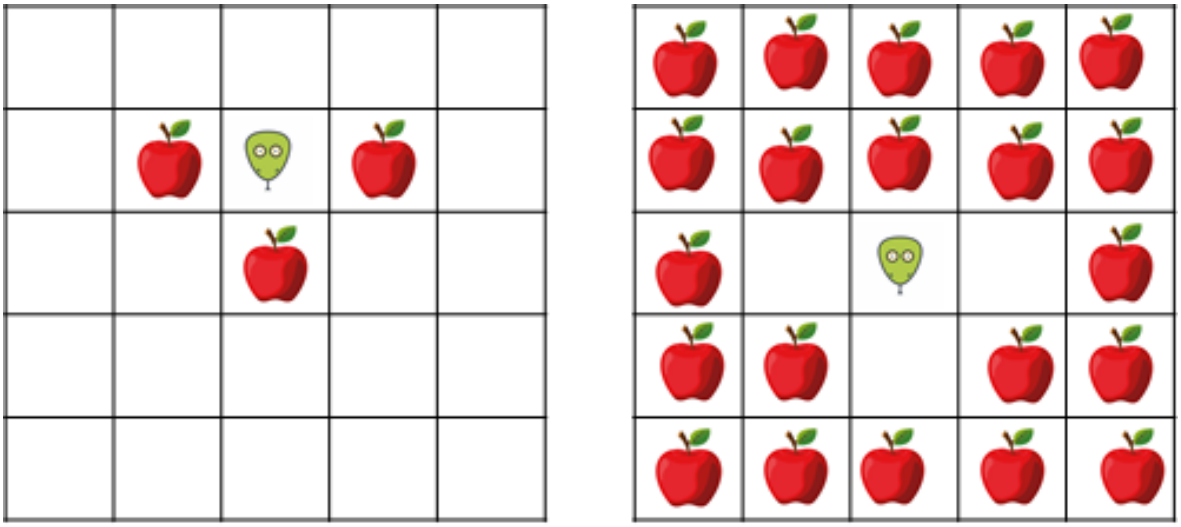

Figure 4: The left figure shows the case that the apple is adjacent to the head and after the apple is eaten by the snake, the successor state could be any of the 22 states depends on the new position of the apple $T(s', a, s) = \frac{1}{22}$. The right figure shows the case that in the current state s, taking any of the actions will change the state of the game screen to only one other state where $T(s', a, s) = 1$.

Exercise 3: a) Beginning from terminal state(s), perform the first two steps of the dynamic programming procedure, by using eq. (1). Explain what is the problem. Draw a figure (you may draw by hand, take a picture, and attach that picture of your drawing in the report) of the game with the snake one and two steps from the terminal state(s) to motivate your answer.

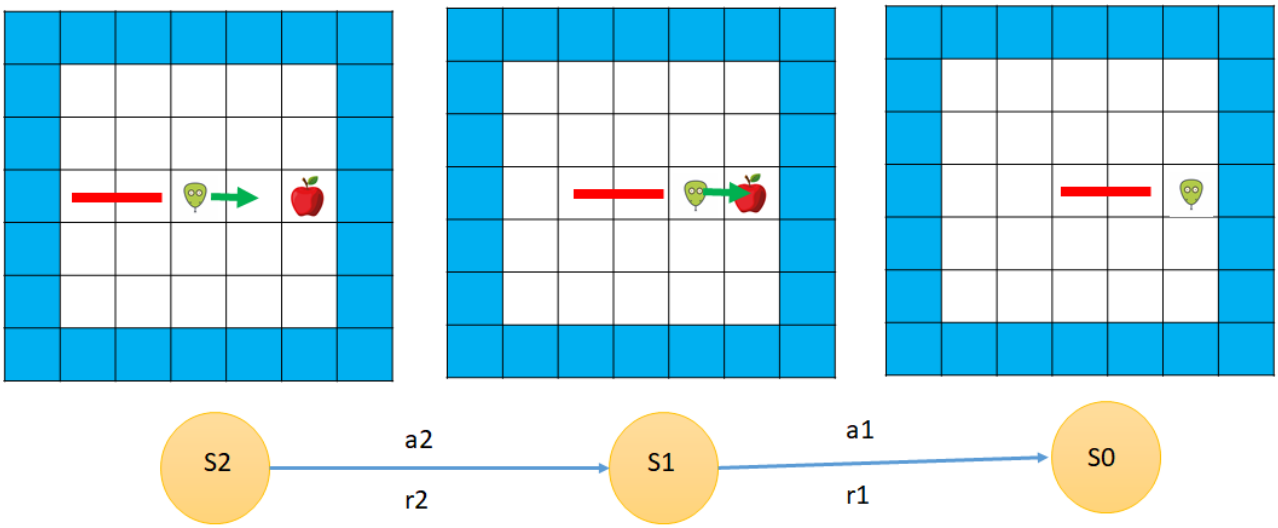

Figure 5: Two step backward from terminal state.

In Fig. (5), the most right screen shows the terminal state where the snake ate the apple by taking forward action. We assume all the Q-values of terminal states are zero. We need

to find Q-values only for the actions that takes us to the terminal state. And for s_2 we only need to calculate Q for the action that takes us to s_1. Thus, we only need to calculate $Q(s_1, a_1 = \text{forward})$ and $Q(s_2, a_2 = \text{forward})$.

$$Q(s_1, a_1 = \text{forward}) = T(s_1, a_1 = \text{forward}, s_{terminal})(R(s_1, a_1 = \text{forward}, s_{terminal}) + \gamma * 0) = +1$$

Now, for calculating the Q-values of forward action in s_2, which takes us to the state s_1, we will have the following equation. If we assume that the s_2 is as the example state (the left screen of Fig. (5), we will have:

$$\begin{aligned} Q(s_2, a_2 = \text{forward}) = T(s_2, \text{forward}, s_1)(R(s_2, \text{forward}, s_1) + \\ \gamma * max[Q^*(s_1, \text{forward}), Q^*(s_1, \text{left}), Q^*(s_1, \text{right})]) = \\ 1(0 + \gamma * max(1, \text{unknown}, \text{unknown})) = ???\gamma \end{aligned}$$

From a practical perspective, calculating the right hand side of the Bellman Equation (1) is challenging because the maximization is over all actions and all non-terminal states are initially unknown and we can not calculate the maximum of three different values while $Q^*(s_1, \text{left})$ and $Q^*(s_1, \text{right})$ are unknown. And their values also depend on the other Q-values and a unique value should be stored for each Q-value.
b) What would be a solution to the issue found in a)? We can not find the optimal (maximum) Q-value (policy) of successor states for the unknown unvisited states optimal values. We can solve this the way value-based solvers tackle the problem. By 1) policy evaluation and 2) policy improvement. In the policy evaluation phase, the solver calculates the value function for some or all states given the fixed policy. In the policy improvement step, the algorithm improves the previous policy based on values obtained in the policy evaluation step. The process of iteratively evaluating and improving the policy continues until either the policy remains unchanged, a time limit has been reached, or the change to the value function is below a certain threshold [1].

[1] -A. Geramifard, T. J. Walsh, S. Tellex, G. Chowdhary, N. Roy, and J. P. How,"A Tutorial on Linear Function Approximators for Dynamic Programming and Reinforcement Learning", Foundations and Trends in Machine Learning ,Vol. 6, No. 4 (2013), pp. 375–454.

Exercise 4: a) Rewrite eq. (3) using the notation E[·], where E[·] denotes expected value. Use the same notation as in (3) as much as possible. The expression should still be a recursive relation – no infinite sums or similar.

$$V^*(s) = \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^*(s')]. \tag{3}$$

$$\begin{aligned} V^*(s) &= \max_a \sum_{s'} T(s, a, s') R(s, a, s') + \gamma [T(s, a, s') V^*(s')] \\ &= \max_a E(R(s, a, s') | s, a) + \gamma E(V^*(s') | s, a) \\ &= \max_a E(R + \gamma V^*(s') | s, a) \end{aligned} \tag{4}$$

b) Explain what eq. (3) is saying in one sentence.

Eq. (3) 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 (Sutton & Barto 2016).*

In one sentence, eq. (3) is a recursive equation decompose the optimal state-value function into immediate reward and the optimal discounted value of the successor state.

c) In eq. (3), what is the purpose of the max-operator?
Since the state value function defined over each state, value of a state under an optimal policy must equal the expected return for the best action from that state. Thus, the maximum over the choices of agent's actions is taken rather than the expected value given some policy.

d) For $Q^*(s,a)$, we have the relation $\pi^*(s) = argmax_a Q^*(s,a)$. What is the relation between π^* and V^* (the answer should be something like $\pi^*(s) = ...$?

We can write Q^* in terms of V as follows:

$$Q^*(s,a) = E(R + \gamma V^*(s')|s,a) \tag{5}$$

Thus, we have:

$$\pi^*(s) = argmax_a E(R + \gamma V^*(s')|s,a) \tag{6}$$

e) Why is the relation between π^* and V^* not as simple as that between π^* and Q^*? You don't need to explain with formulas here; instead, the answer should explain the qualitative difference between V^* and Q^* in your own words.
The reason is that the optimal action-value function $Q^*(s,a)$ indicates how good it is for an agent to pick action a being in state s. Thus, if we know the optimal $Q^*(s,a)$, the optimal policy can be easily extracted by choosing the action a that gives maximum $Q^*(s,a)$ for state s. However, the value function represents how good is state s for an agent to be in. The value function depends on the policy by which the agent picks actions to perform. Among all possible value-functions, there exist an optimal value function that has higher value than other functions for all states. The optimal policy $\pi^*$is the policy that corresponds to optimal value function.

$$\pi^* = argmax_\pi V^\pi(s) \tag{7}$$

Thus, $V^*(s)$ does not indicate the effect of picking an action in each state, separately.

Exercise 5: a) Attach your policy iteration code. To check that your code seems to be correct, try running it with $\gamma = 0.5$ and $\epsilon = 1$, and check that you get 6 policy iterations and 11 policy evaluations (if you do not get this, make sure you use Matlab's built-in max-function if doing any max-operation). Do not proceed until you get this result. The codes of policy evaluation and policy improvement are shown in Figs. (9 and 13) and the result of running the code is shown in Fig. 8. The temp_values are reintialized to zero in Policy improvement code.
b) Investigate the effect of γ (in this part, set $\epsilon = 1$). Try ($\gamma = 0$, $\gamma = 1$ and $\gamma = 0.95$).
Provide a table showing the number of policy iterations and evaluations for each γ.
How does the final snake playing agent act for the various γ? Is the final agent playing optimally? Why do you get these results?

	$\gamma = 0$	$\gamma = 1$	$\gamma = 0.95$
Number of policy iterations	2	∞	7
Number of policy evaluations	4	∞	37

```
% Policy evaluation.
while 1
    Delta = 0;
    for state_idx = 1 : nbr_states
        % FILL IN POLICY EVALUATION WITHIN THIS LOOP.
        temp_v = values(state_idx);

        next_state = next_state_idxs(state_idx,policy(state_idx));
        inst_r = 0;
        temp_value = 0;
        if next_state == 0
            inst_r = rewards.death;
        elseif next_state == -1
            inst_r = rewards.apple;
        else
            inst_r = rewards.default;
            temp_value = values(next_state);
        end

        values(state_idx) =  inst_r + gamm * temp_value;
        Delta =max(Delta, abs(temp_v-values(state_idx)));
    end
    % Increase nbr_pol_eval counter.
    nbr_pol_eval = nbr_pol_eval + 1;

    % Check for policy evaluation termination.
    if Delta < pol_eval_tol
        break
```

Figure 6: Policy evaluation.

```
% Policy improvement.
policy_stable = true;
for state_idx = 1 : nbr_states
    % FILL IN POLICY IMPROVEMENT WITHIN THIS LOOP.
    temp_policy = policy(state_idx);
    next_states = next_state_idxs(state_idx,:);
    inst_r = zeros(1,3);
    temp_values = zeros(1,3);
    for i =1:3
        if next_states(i)==0
            inst_r(i) = rewards.death;
        elseif next_states(i)==-1
            inst_r(i) = rewards.apple;
        else
            inst_r(i) = rewards.default;
            temp_values(i) = values(next_states(i));
        end
    end
    [val,policy(state_idx)] = max(inst_r+gamm*temp_values);
    if policy(state_idx)~=temp_policy
        policy_stable = false;
    end
end
% Increase the number of policy iterations .
nbr_pol_iter = nbr_pol_iter + 1;
% Check for policy iteration termination (terminate if and only if the
% policy is no longer changing, i.e. if and only if the policy is
% stable).
```

Figure 7: Policy improvement.

```
Successfully loaded states, next_state_idxs!
Running policy iteration!
Delta: 4.2773
Delta: 2.1387
Delta: 1.0693
Delta: 2
Delta: 1
Policy iteration done! Number of policy iterations: 6
Number of policy evaluations: 11, elapsed time: 0.25071 seconds
```

Figure 8: The result of policy iteration code.

Only the snake plays optimally with the $\gamma = 0.95$. In the case of $\gamma = 0$, if the apple would not be in the adjacent cells of the snake, it never reaches it since it has no sense of the further states (only based on instant reward). If $\gamma = 1$, the algorithm does not converge. Because all the states are identically good for the agent (does not matter how far the agent is far from apple). There is no optimal policy to converge to.

c) Investigate the effect of the stopping tolerance ϵ in the policy evaluation (in this part, set $\gamma = 0.95$). Try ϵ ranging from 10^4 to 10^4 (with exponent step size 1, i.e. 10^4 , 10^3 ,..., 10^4). Provide a table showing the number of policy iterations and evaluations for each.

How does the final snake playing agent act for the various ϵ? Is the final agent playing optimally? Why do you get these results?

ϵ	10^{-4}	10^{-3}	10^{-2}	10^{-1}	10^{0}	10^{1}	10^{2}	10^{3}	10^{4}
Number of policy iterations	7	7	7	7	7	19	19	19	19
Number of policy evaluations	214	168	125	65	37	19	19	19	19

For all values of ϵ, the agent plays optimal. However, for small values of ϵ the number of evaluations increases since it needs more steps to converge. However, for large ϵ the number of evaluations is the same as iterations since the termination condition is satisfied in first iteration. $(\Delta < \epsilon)$

Exercise 6: a) Attach your Q-update code (both for terminal and non-terminal updates). This should be 8 lines of Matlab code in total.
For terminal states the future $Q(s', a) = 0$.

```
if terminate

  % ------------------------------------------------------------
  %
  % FILL IN THE BLANKS TO IMPLEMENT THE Q-UPDATE BELOW (SEE SLIDES)
  %
  % Maybe useful: alph, reward, Q_vals(state_idx, action) [recall that
  % we set future Q-values at terminal states equal to zero].
  % Hint: Q(s,a) <-- (1 - alpha) * Q(s,a) + sample
  % can be rewritten as Q(s,a) <-- Q(s,a) + alpha * (sample - Q(s,a))
  sample              = reward;%+gamm*max(Q_vals(state_idx, :));% replace nan with something appropriat
  pred                =  Q_vals(state_idx, action);% replace nan with something appropriate.
  td_err              = sample - pred; % don't change this.
  Q_vals(state_idx, action) = Q_vals(state_idx, action) + alph * (td_err);% + ... (fill in blanks)
```

Figure 9: Q-update for terminal updates.

```
% ------------------------------------------------------------
%
% FILL IN THE BLANKS TO IMPLEMENT THE Q-UPDATE BELOW (SEE SLIDES)
%
% Maybe useful: alph, max, reward, gamm, Q_vals(next_state_idx, :),
% Q_vals(state_idx, action)
% Hint: Q(s,a) <-- (1 - alpha) * Q(s,a) + sample
% can be rewritten as Q(s,a) <-- Q(s,a) + alpha * (sample - Q(s,a))
sample              = reward+gamm*max(Q_vals(next_state_idx, :));% replace nan with something appro
pred                =  Q_vals(state_idx, action);% replace nan with something appropriate.
td_err              = sample - pred; % don't change this.
Q_vals(state_idx, action) = Q_vals(state_idx, action) + alph * (td_err);
```

Figure 10: Q-update for nonterminal updates.

b) Explain 3 different attempts (parameter configurations and scores - note that the 3 attempts don't need to be completely different, but change some settings in between them)

you did in order to train a snake playing agent. Comment, for each three of the attempts, if things worked well and why/why not.

attempt 1	attempt 2	attempt 3
$R_{death} = -100$	$R_{death} = -100$	$R_{death} = -100$
$R_{apple} = 10$	$R_{apple} = 10$	$R_{apple} = 10$
$\gamma = 0.9$	$\gamma = 0.9$	$\gamma = 0.9$
$\alpha = 0.01$	$\alpha = 0.01$	$\alpha = 0.1$
$\epsilon = 0.01$	$\epsilon = 0.1$	$\epsilon = 0.01$
avg_train_Score =61.19	avg_train_Score = 2.85	avg_train_Score = 57.4
test_score=331	test_score=5	test_score=∞

I tried to change a bit rewards, learning rate(α) and the random action selection probability in epsilon-greedy policy. Increasing the reward of eating an apple and the penalty of death improves the score alot. Increasing the random exploration chance of actions did not work well and I got worse scores.
In last attempt, I increase the learning rate to $\alpha = 0.1$. "That results to obtain a snake keeps eating apples forever". I stopped the run in score 63729 which is shownn in Fig. (11). The reason is that larger learning rates result in rapid changes and require fewer training iterations to achieve optimal behaviour.

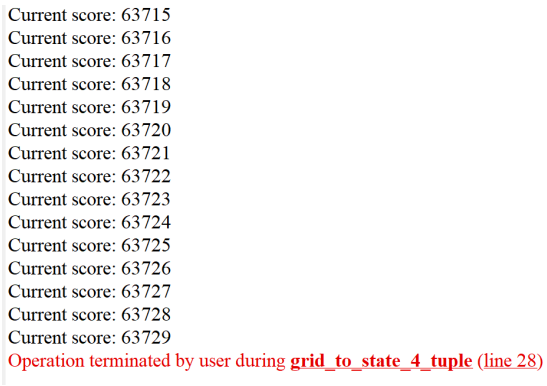

Figure 11: Eating apple forever.

c) Write down your final settings. Were you able to train a good, or even optimal, policy for the small Snake game via tabular Q-learning?
The final settings is the parameters of attemp3 where, $R_{death} = -100$, $R_{apple} = 10$, $\epsilon = 0.01$, $\gamma = 0.9$, and $\alpha = 0.1$. The Q-learning reaches the optimal policy since my snake is eating apple forever....

d) Independently on how it went, explain why it can be difficult to achieve optimal behavior in this task within 5000 episodes. It is because of the large number of states we have here and we need to go through all of them in the defined number of iterations. Generally learning rate totally affect the number of required steps for the convergence of the algorithm to optimal value..

Exercise 7: a) Attach your Q weight update code (both for terminal and non-terminal

updates). This should be 8 lines of Matlab code.

```
if terminate

    % ------------------------------------------------------------

    % FILL IN THE BLANKS TO IMPLEMENT THE Q WEIGHTS UPDATE BELOW (SEE SLIDES)
    % Maybe useful: alph, reward, Q_fun(weights, state_action_feats, action),
    % state_action_feats(:, action) [recall that
    % we set future Q-values at terminal states equal to zero]
    target  = reward;%+gamm*Q_fun(weights,state_action_feats_future,action) % replace nan with something
    pred    =  Q_fun(weights,state_action_feats,action);% replace nan with something appropriate
    td_err  = target - pred; % don't change this
    weights = weights + alph*(td_err)*state_action_feats(:,action); % + ... (fill in blanks)
```

Figure 12: Q-weight update code for terminal updates.

```
% FILL IN THE BLANKS TO IMPLEMENT THE Q WEIGHTS UPDATE BELOW (SEE SLIDES)
%
% Maybe useful: alph, max, reward, gamm, (Q_fun(weights, state_action_feats_future)),
% Q_fun(weights, state_action_feats, action), state_action_feats(:, action)
target  = reward+gamm*max(Q_fun(weights,state_action_feats_future,action)); % replace nan with something appropriate
pred    =  Q_fun(weights,state_action_feats,action);% replace nan with something appropriate
td_err  = target - pred; % don't change this
weights = weights + alph*(td_err)*state_action_feats(:,action); % + ... (fill in blanks)
```

Figure 13: Q-weight update code for non-terminal updates.

b) Write down at least 3 different attempts (show the scores, parameter configurations and state-action feature functions - note that the 3 attempts don't need to be completely different, but change some settings in between them) you did in order to train a good snake playing agent. Comment on if your attempts worked well and why/why not.
For this part, I tried multiple configurations (mostly on feature part) and I gained very good performance which is described in details in the following table.

	attempt 1	attempt 2	attempt 3	attempt 4
$\boldsymbol{\phi}(s,a)$	$\phi_1(s,a) = d_{head-apple}$ $\phi_2(s,a) = d_{head-tail}$	$\phi_1(s,a) = d_{head-apple}$ $\phi_2(s,a) = d_{head-tail}$ $\phi_3(s,a) = I_{hit-wall}$ $\phi_3(s,a) = I_{reward}$	$\phi_1(s,a) = x_{head-apple}$ $\phi_2(s,a) = y_{head-apple}$ $\phi_3(s,a) = I_{hit-wall}$	$\phi_1(s,a) = x_{head-apple}$ $\phi_2(s,a) = y_{head-apple}$ $\phi_3(s,a) = I_{hit-wall}$ $\phi_4(s,a) = I_{trapped}$
W_{0bad}	[1,-1]	[1,-1,1]	[1,1,1]	[1,1,1,1]
W	[-87.9067, -75.6457]	[-133.41 -131.8 -31.89] [-36.99 -29.74 -11.87]	[-21.7 -22.7 -24.8]	[-9.4 -15.0 -26.6 -18.3]
R_{death}	-35	-35	-35	-35
R_{apple}	1	1	1	1
γ	0.9	0.9	0.9	0.9
α	0.1	0.1	0.1	0.1
ϵ	0.1	0.1	0.1	0.1
S_{test}	0.02	got_stuck, 12.08	31.27	105.96

In feature part, I tried first the two features of relative distance of the head of snake from the apple ($d_{head-apple}$) and from the tail of the snake $d_{head-tail}$ (shown in Fig. (14)). We know that we want our snake to decrease the distance to apple all the time. Thus, the good weight for it is a negative weight and for a bad decision I initialized it with a positive value (1).
For the second feature, first I thought it would be good if the snake increase the distance with its own tail and the weight should be a positive value. Thus, I initialized it with a negative value. However, I realized (in attempt 2) it was not a good feature and really does not help to improve the performance. The snake actually follows its own tail and the weight of that converges to a big negative number almost equal to the first feature's weight and the snake got stuck in some case between following its own tail or the apple....

```
state_action_feats(1, action) = sum((next_head_loc-apple_loc).^2)/2/N^2;
state_action_feats(2, action) = sum((next_head_loc-tail_loc).^2)/2/N^2;
state_action_feats(3, action) = 0;
```

Figure 14: distance to the apple and to the tail of the snake.

For the second attempt, I also have added a new boolen feature called $I_{hit-wall}$ in which if the snakes head would hit the wall in next state it would be 1, otherwise 0. I initialized its weight with a positive number since the weight should be a negative number at the end. In attempt 2 the snake got stuck in a case following its own tail with out following the apple.
Then, I changed the third feature to I_{reward} in which it would be -1, 0, 1 if the snakes' head would eat apple, nothing, or hit the wall or itslef and try attempt 2 with that. It was much better and it did not get stuck in any case.

In attempt 3, I removed the $d_{head-tail}$ feature and changed the first feature to two features include the $x_{head-apple}$ and $y_{head-apple}$ which helps the snake goes more straight rather than diagonally and the third feature was again hit the wall feature. The results were not bad and I got 31.27 test score (Fig. 15).

```
state_action_feats(1, action) = abs(next_head_loc(1)-apple_loc(1)) / N;
state_action_feats(2, action) = abs(next_head_loc(2)-apple_loc(2)) / N;
state_action_feats(3, action) = hit_the_wall;
```

Figure 15: Third attempt features.

Finally, for attempt 4, I defined a new feature called $I_{trapped}$ in which in a recursive function called "AvailableSpace(grid,x, y, n)" (shown in Figs. (16, 17)) the white space trapped between the snake and wall is calculated and if it would be less than the length of the snake the boolean variable "trapped" would be set to 1. The forth feature is that variable.

```
% To get the trapped white spaces in front of the snake
function [newGrid, newN] = availableSpace(grid, x, y, n)
    grid(x, y) = 2;
    n = n+1;
    if (grid(x-1, y) == 0)
        [grid, n] = availableSpace(grid, x-1, y, n);
    end
    if (grid(x+1, y) == 0)
        [grid, n] = availableSpace(grid, x+1, y, n);
    end
    if (grid(x, y-1) == 0)
        [grid, n] = availableSpace(grid, x, y-1, n);
    end
    if (grid(x, y+1) == 0)
        [grid, n] = availableSpace(grid, x, y+1, n);
    end
    newGrid = grid;
    newN = n;
end
```

Figure 16: Calculate the white space between the snake and the wall, recursively.

```
trapped = 0;
if (hit_the_wall ~= 1)
    [tmpGrid, nSpace] = availableSpace(grid, next_head_loc(1), next_head_loc(2), 0);
    if (nSpace < snake_length)
        trapped = 1;
    end
end
```

Figure 17: Boolean "trappped" variable.

In this attempt, the performance improved alot and I could get the average score of 105.96 (shown in Fig. 18).

```
----------------
GAME OVER! SCORE:      129
AVERAGE SCORE SO FAR:  105.96
AVERAGE SCORE LAST 10:  104.3
AVERAGE SCORE LAST 100: 105.96
---------------------------------
Final weights:
  -9.4032
 -15.0465
 -26.6126
 -18.2655

Mean score after 100 episodes: 105.96
... SUCCESS! You got average score at least 35 (feel free to try increasing this further if you want)
Done testing agent!
```

Figure 18: Final score.

c) Report what average test score you get after 100 game episodes with your final settings; you must get at least 35 on average and not get stuck in an infinite loop (it is possible to get over 100 points with only 3 state-action features). Also attach a table with your initial weight vector w0, your final weight vector w and associated final state-action feature functions fi . Comment on why it works well or why it doesn't work so well. For final attept I used the features of attept 4 which were 1) $x_{head-apple}$ 2) $y_{head-apple}$ 3)$I_{hit-wall}$ and 4)$I_{trapped}$ shown in Fig. (19)

```
state_action_feats(1, action) = abs(next_head_loc(1)-apple_loc(1)) / N;
state_action_feats(2, action) = abs(next_head_loc(2)-apple_loc(2)) / N;
state_action_feats(3, action) = hit_the_wall;
state_action_feats(4, action) = trapped;
```

Figure 19: Final features.

In this case, a sample of game is shown in Fig. (20). By assuming "trapped" feature the snake does not go inside the trap areas where the white space is less than its own length and the scores goes up until even 124.. The death of the snake is in the case that e.g., after going inside the white space which was not trap at first the snake could eat an apple in that space and by increasing the length of it, it dies...All the weights converged to negative numbers since the snake tries to minimize all of them and for a bad initialization I assume a positive number..The final weights were $[-9.4\ -15.0\ -26.6\ -18.3]$ and the initial weights were $W_0 = [1\ 1\ 1\ 1]$.

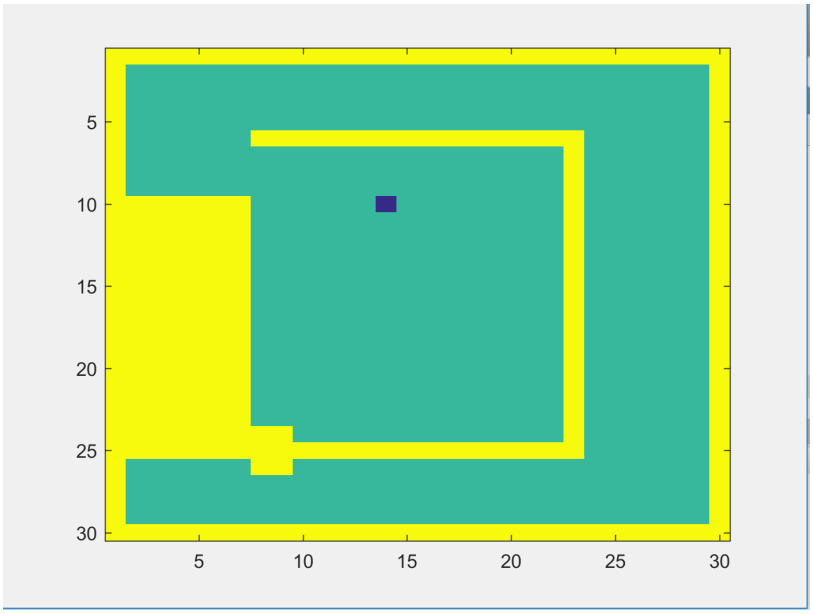

Figure 20: Good score.