Lund University
Faculty of Engineering (LTH)

FMAN 45

Machine Learning

Reinforcement learning for playing Snake, Hand-In 4

Author:
Marcel Attar, 941127-2173

The report was handed in on: May 31, 2019

Lunds universitet
Lunds Tekniska Högskola

1 Tabular methods

Since we start with such a small game of snake, one where the snake doens't change in size, we can store all the different states in a big table or matrix. The table can be represented by a K x 3 matrix $\mathbf{Q}(\mathbf{S},\mathbf{a})$, where K is the number of states and the 3 columns the three different possible actions.

Exercise 1 (10 points):

Derive the value of K above.

There are 6 different configurations of the snake, two straight lines and 4 bent ones, and the head can be at two different parts of the snake. The two straight lines can be in 15 different positions each in the game and the bent snakes can be in 16 different positions each. The apple can be in $(5 \cdot 5 - 3 =)$ 22 positions (the minus three is because it cannot be inside the snake since that is not a non-terminal state). Therefore, the total number of states are

$$K = (2 \cdot 15 + 4 \cdot 16) \cdot 2 \cdot 22 = 4136$$

Bellman optimality equation for the Q-function

The optimal Q-value $Q^*(s,a)$ for a given state-action pair (s,a) is given by the state-action value Bellman optimality equation

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

From $Q^*(s,a)$ we get the optimal policy π^* from

$$\pi^*(s) = a^* = \arg\max_a Q^*(s,a) \tag{2}$$

Exercise 2 (10 points):

a) (1 point) Rewrite equation (1) as an expectation using the notation $\mathbb{E}[\cdot]$, where $\mathbb{E}[\cdot]$ denotes expected value.

b) (4 points) Explain what equation (1) is saying. Your answer must reference all the various components and concepts of the equation $(Q^*, \sum, T, R, \gamma, \max, s, a, s)$.

c) (5 points) Discuss the effect γ has on the optimal policy $\pi(s)$ in equation (2) by considering the following cases:
i) $\gamma = 0$, ii) $\gamma = 1$ and iii) $\gamma \in (0, 1)$.

a)

The definition of an exptected value X is

$$\mathbb{E}_{p(x)}[X] = \sum_{i=1}^{k} x_i p_i \tag{3}$$

where X is a random variable with a finite number of finite outcomes x_1,x_2,...,x_k with probabilities p_1,p_2,...,p_k. If we compare this with equation (1) we see some similarities. The transition function, T, is the probability and the reward, R, plus the future reward, $\max_{a'} Q^*(s',a')$, are the outcome X. Thus, equation (1) can be written as

$$Q^*(s,a) = \mathbb{E}_{T(s,a,S')}[R(s,a,S') + \gamma \max_{A'} Q^*(S',A)] \tag{4}$$

b)

Equation (4) can be interpreted as the expected value of the reward $R(s,a,S') + \gamma \max_{A'} Q^*(S',A')$ with the probability $T(s,a,S')$. T is the transition function, i.e. the probability that action a from state s will lead to state s'. R is the reward function, the reward assuming that we start in state s, do action a and end up in state s'. The expression $\max_{a'} Q^*(s',a')$ is the maximum reward we will get if we start in state s'. The parameter γ is a tuning parameter which work as a discount factor for future rewards. Since $Q^*(s,a)$ is an expectation we have to sum over all different possible state outcomes in one step.

c)

- i) $\gamma = 0$

Then the future rewards will be neglected, the system is only looking at what's best by looking at one step ahead.

- ii) $\gamma = 1$

Now the system will take into account 100% of the future rewards but it doesn't care if we eat the apple in 2 steps from now or 100 steps from now.

- iii) $\gamma \in (0,1)$

This is a tuning parameter of the future reward, how high or low do we want to value it and work as a discount factor. A smaller γ will lead to the snake being more short sighted and vice versa for a bigger γ.

Exercise 3 (10 points):

Explain what $T(s,a,s)$ in equation 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).

$T(s,a,s') = 1$ if $s = \{$the snakes head is more than one pixel away from the apple$\}$, $a = \{$any action$\}$ and $s' = \{$the snake is in the updated position depending on the action a and the apple is in the same place as before$\}$.

$T(s,a,s') = 0$ if $s = \{$the snakes head is more than one pixel away from the apple$\}$, $a = \{$any action$\}$ and $s' = \{$the snake is in a state that was not possible given action a and the previous state of the snake and/or the apple is not in the same place as before$\}$.

$T(s,a,s') = \frac{1}{22}$ if $s = \{$the snakes head is one pixel away from the apple$\}$, $a = \{$the snake moves in the direction of the apple$\}$ and $s' = \{$the snakes position is updated according to its action and the apple is now in one of the 22 possible states$\}$.

Exercise 4 (10 points): Consider the small Snake game. Assume we have a reward signal which gives 1 it the snake dies, +1 if the snake eats an apple, and 0 otherwise. Also assume that $\gamma \in (0, 1)$.

a) (2 points) Consider a version of the game in which the game ends as soon as the snake eats the first apple (yielding a total score of +1 for the game). Let us call this game the *truncated game*. Explain the connection between the truncated game and the normal game (in which the snake can keep eating apples forever). In particular, explain in what sense an optimal agent to the truncated game will be optimal also for the normal game.

b) (1 point) What are the terminal states in the dynamic programming problem for the truncated game?

c) (5 points) Beginning from terminal state(s) in b), perform the first step of the dynamic programming procedure, by using equation (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 step from the terminal state(s) to motivate your answer. *Your answers to Exercise 2 and 3 may help you.*

d) (2 points) What would be a solution to the issue found in c)? *Hint: What values in the dynamic programming table are relevant for finding the optimal solution?*

a)

Since the snake has a constant size the game has the exact same state space, independent of how many apples that have been eaten. Therefore, if we optimize the snake for one truncated game we've "solved" the entire game.

b)

The terminal states are when the snake has its head in the wall or when it has eaten the apple. Since it is only 3 pixels long it cannot die by hitting its tail.

c)

Lets say that we start in the state below

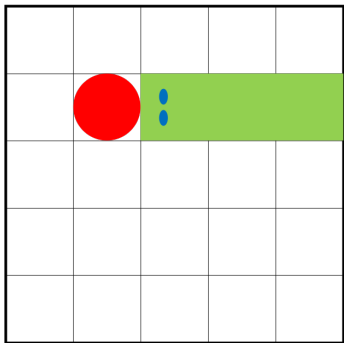

Figure 1: Here the snake is one pixel away from the apple and we define this as the state s. If the action is moving to the left the game will terminate and the agent will be rewarded with $+1$.

Using equation (1) we get the optimal Q value, for action $a = \leftarrow$,

$$Q^*(s, \leftarrow) = R(s, \leftarrow, s') + \gamma \max_{a'} Q^*(s', a') = 1 + 0 = 1 \tag{5}$$

The reason for the why the $\sum$ disappear is because $T(s, \leftarrow, s')$ will be zero for all states except for the state in figure 2 where it will be one. Below we see the updated terminated state.

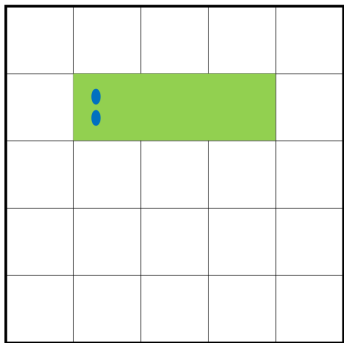

Figure 2: This is the updated state, s', of the action $a = \leftarrow$, assuming the state s is figure 1.

If we now start in the same state as figure 1 but instead of $a = \leftarrow$ we have the action $a = \uparrow$. Now equation (1) will be

$$Q^*(s, \uparrow) = R(s, \uparrow, s') + \gamma \max_{a'} Q^*(s', a') = 0 + ? = ? \tag{6}$$

since we do not know what $\gamma \max_{a'} Q^*(s', a')$ is. We will get the same for $a = \downarrow$. Therefore, in our $\mathbf{Q}^*$ matrix, on row s, we will have two unknowns and a one.

d)

The problem of having two unkowns is that it is hard to compare these with the one. However, we know that $\max_{a'} Q^*(s',a')$ is less than one and if we force γ to be $\in (0,1)$ we know that the unknowns will be less than one.

Policy iteration

$$V^* = \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s') \right] \tag{7}$$

Exercise 5 (10 points):

a) (1 point) Rewrite equation (7) using the notation $\mathbb{E}[\cdot]$, where $\mathbb{E}[\cdot]$ denotes expected value.

b) (4 points) Explain what equation (7) is saying; your answer must reference all the various components and concepts of the equation $(V^*, \sum, T, R, \gamma, \max, s, a, s)$.

c) (1 point) For $Q(s,a)$, we have the relation $\pi^*(s) = \arg\max_a Q^*(s,a)$. What is the relation between $\pi^*(s)$ and $V^*(s)$?

d) (4 points) 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.

a)

Here we do similarly as we did for exercise 2.

$$V^*(s) = \mathbb{E}_{T(s,a,S')} \left[R(s,a,S') + \gamma V^*(S') \right] \tag{8}$$

b)

Equation (7) can be interpreted as the expected value of the reward $R(s,a,S') + \gamma V^*(S')$ with the probability $T(s,a,S')$. T is the transition function, i.e. the probability that action a from state s will lead to state s'. R is the reward function, the reward assuming that we start in state s, do action a and end up in state s'. $V^*(S')$ is the maximum expected total reward of the future. The parameter γ is a tuning parameter. Since $V^*(s)$ is an expectation we have to sum over all different possible state outcomes in one step.

c)

$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s') \right] \tag{9}$$

d)

In $Q^*(s,a)$ the action is saved for each Q^* but in V^* the action isn't saved. In other words, $\mathbf{Q}^*$ is a K x 3 matrix, with the K different states and for each state the 3 different actions. $\mathbf{V}^*$ is a K x 1 vector, only the highest value for each state is saved but not what action to take to get that value.

Exercise 6 (15 points): Look at the code in `rl_project_to_students/small_snake_tabular_pol_iter/`. Begin by investigating `snake.m`. Fill in the blanks to run policy iteration in order to find an optimal policy $\pi^*(s)$ - look at `policy_iteration.m`. You don't need to think about the internal game state; nor will you need to think about how to implement the state representation, as this is already done for you. However, you of course need to understand how this state representation works in order to implement policy iteration. You are allowed to change settings only where it states so in `snake.m` and `policy_iteration.m`.

a) (3 points) Attach a printout of your policy iteration code. *Hint: 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.*

b) (6 points) What is the effect of γ (in this part, set $\epsilon = 1$)? Try $\gamma = 0$, $\gamma = 1$ and $\gamma = 0.95$ and explain what happens, and **why you get those results**, by answering the following. What happens in the algorithm (i.e., how does γ affect the number of iterations of policy evaluation and policy iteration)? How does the final snake playing agent act for the various γ? Is the final agent playing optimally?

c) (6 points) What is the effect of 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) and explain what happens, and **why you get those results**, by answering the following. What happens in the algorithm (i.e., how does ϵ affect the number of iterations of policy evaluation and policy iteration)? How does the final snake playing agent act for the various ϵ? Is the final agent playing optimally?

a)

Here is the policy evaluation

```
        for state_idx = 1 : nbr_states
            % FILL IN POLICY EVALUATION WITHIN THIS LOOP.
            v = values(state_idx);

            % Checking the action for a given state
            action = policy(state_idx);

            % Checking what state that action will take us to
            next_index = next_state_idxs(state_idx,action);

            % Updating the values
            if next_index == -1 % This is when we eat an apple
                values(state_idx) = rewards.apple + 0;
```

```
            elseif next_index == 0 % We died
                values(state_idx) = rewards.death + 0;
            else % We lived
               values(state_idx) = rewards.default + gamm*values(
    next_index);
            end

            Delta = max(Delta, abs(v-values(state_idx)));
        end
```

Here is the policy improvment

```
    for state_idx = 1 : nbr_states
        % FILL IN POLICY IMPROVEMENT WITHIN THIS LOOP.
        old_action = policy(state_idx);
        next_idxs = next_state_idxs(state_idx,:);

        temp_values = zeros(3,1);

        for i=1:3
            next_index = next_idxs(i);

        % Updating the values
            if next_index == -1 % This is when we eat an apple
                temp_values(i) = rewards.apple + 0;
            elseif next_index == 0 % We died
                temp_values(i) = rewards.death + 0;
            else % We lived
               temp_values(i) = rewards.default + gamm*values(
    next_index);
            end
        end
        [~,idx] = max(temp_values);
        policy(state_idx) = idx;

        if policy_stable
            policy_stable = (old_action == policy(state_idx));
        end
    end
```

b)

	Number of policy iterations	**Number of policy evaluations**
$\boldsymbol{\gamma = 0}$:	2	4
$\boldsymbol{\gamma = 1}$:	N/A	∞
$\boldsymbol{\gamma = 0.95}$:	6	38

When $\gamma = 0$, the agent isn't looking at future rewards. The snake goes in a circle and never eats the apple.

When $\gamma = 1$ the agent isn't discounting future reward, i.e. future and present

reward are valued equally. Now the code will get stuck in an infinite loop, the policy evaluation will never finish. This is because the snake cannot decide *when* to eat the apple, since it has no incentive to get to the apple quickly

When $\gamma = 0.95$ the snake eats all the apples successfully. Now the future reward is discounted by 5 % for each time step away it is.

c)

	Number of policy iterations	**Number of policy evaluations**
$\boldsymbol{\epsilon = 10^{-4}}$:	6	204
$\boldsymbol{\epsilon = 10^{-3}}$:	6	158
$\boldsymbol{\epsilon = 10^{-2}}$:	6	115
$\boldsymbol{\epsilon = 10^{-1}}$:	6	64
$\boldsymbol{\epsilon = 1}$:	6	38
$\boldsymbol{\epsilon = 10^{1}}$:	19	19
$\boldsymbol{\epsilon = 10^{2}}$:	19	19
$\boldsymbol{\epsilon = 10^{3}}$:	19	19
$\boldsymbol{\epsilon = 10^{4}}$:	19	19

A higher ϵ will lead to the value vector having a worse estimate of the true value since we will take fewer policy evaluations. Therefore the policy iterations will have to be higher to make up for the bad estimates to improve the policy. When we instead have a low tolerance, we will have to do a lot of policy evaluations and thus get a good value estimate which will lead to fewer policy iterations.

Tabular Q-learning

Exercise 7 (15 points): Look at the code in `rl_project_to_students/small_snake_tabular_q/`. Begin by investigating `snake.m`. Fill in the blanks to implement the tabular Q-updates (see lecture slides). You are allowed to change settings only where it states so in `snake.m`, so you can change the number of iterations, ε, α, γ, decrease learning rate, etc. Write down and think about any observations you make as you try different settings - you will be discussing this in b), c) below.

Once you have trained the agent for 5000 episodes, you should test its performance (the code automatically saves your Q-values). It is clear in the code of `snake.m` how to perform testing (set test_agent = true). This will set α and ε to 0 (meaning no Q-value updates will occur and the policy is completely greedy), load the automatically saved Q-values, and run the game.

a) (3 points) Attach a printout of your Q-update code (both for termi- nal and non-terminal updates). This should be 8 lines of Matlab code in total.

b) (3 points) Explain 3 different attempts (parameter configurations - 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.

c) (6 points) 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? When running this final test, you should be able to get a score of at least 250 to pass this task (write down the score you get prior to dying / getting stuck in a loop), but hopefully you can get something even better. For example, it should be possible to obtain a snake playing agent which keeps eating apples forever, although it doesn't necessarily reach them in the shortest amount of steps. An optimal snake player is one which keeps eating apples forever and reaching each apple as fast as possible. Note that, if your test run never terminates and the score just keeps increasing, you should write this in your report (this is a good thing).

d) (3 points) Independently on how it went, explain why it can be difficult to achieve optimal behavior in this task.

a)

Here's the Q-update code for the non-terminal update

```
1        sample                          = reward + ...
2                                              gamm*max(Q_vals(next_state_idx
    ,:));
3        pred                            = Q_vals(state_idx,action);
4        td_err                          = sample - pred;
5        Q_vals(state_idx, action)       = Q_vals(state_idx, action) ...
6                                                  + alph*td_err;
```

Here it is for the terminal update

```
sample                          = reward;
pred                            = Q_vals(state_idx,action);
td_err                          = sample - pred;
Q_vals(state_idx, action) = Q_vals(state_idx, action) ...
                                                + alph*td_err;
```

b)

- **Attempt 1:** $\varepsilon = 0.1$, $\alpha = 0.5$, $\gamma = 0.9$ and reward = $\{default : 0, apple : 1, death : -1\}$. **Score = 122**

This setup worked fairly well. An ε of 10% seem to be a good balance between exploration and exploitation. The discount factor γ also seemed to be decent.

- **Attempt 1:** $\varepsilon = 0.1$, $\alpha = 0.8$, $\gamma = 0.9$ and reward = $\{default : 0, apple : 1, death : -1\}$. **Score = 54**

I now increased the learning rate α from the previous attempt which resulted in a worse performing model. This is probably because I stepped over a local maximum.

- **Attempt 1:** $\varepsilon = 0.1$, $\alpha = 0.5$, $\gamma = 0.9$ and reward = $\{default : 0, apple : 10, death : -1\}$. **Score = 43**

This time I changed the reward for eating an apple with everything else remaining constant from attempt 1. This yielded a worse result which led me to believe that the absolute value of the positive and negative reward need to be the same.

c)

- **Final Attempt:** $\varepsilon = 0.09$, $\alpha = 0.4$, $\gamma = 0.7$ and reward = $\{default : 0, apple : 100, death : -100\}$. **Score = ∞**
  For this setting I was able to get an infinite score, i.e. my code never terminates. I needed to lower my discount factor and the learning rate. The rewards were increased by a lot which I think led to the model converging quicker to the optimum weights.

d)

It was very hard to find the optimal parameters, it required a lot of tuning. Since we have 6 parameters who all seem to be dependent on each other there are a huge amount of possibilities. Also, the model was very sensitive, small changes could have a big effect on the outcome.

Q-learning with linear function approximation

Exercise 8 (20 points): Look at the code in `rl_project_to_students/linear_q/`. Begin by investigating `snake.m`. It contains most things you need to run the game and to train a reinforcement learning agent based on Q-learning using linear function approximation. You are allowed to change settings only where it states so in `snake.m` and `extract_state_action_features.m`. You need to do a few things in order to train the agent:

- Fill in the blanks in `snake.m` to implement the Q weight updates (see lecture slides).
- Engineer state-action features $f_i(s,a)$ and implement these in the given function `extract_state_action_features.m`.
- Tune learning parameters in the main file `snake.m`, including reward signal, discount γ, learning rate α, greediness parameter ε, and so on (similar to what you did in the previous exercise). Experimentation is necessary to train a successful agent.
- We have a weight vector weights, which needs to be initialized. We will initialize the weights in a bad way. Doing this "adversarial" initialization will force the agent to actually learn something, rather than just work perfectly right away. Hence do as follows:
  - You should set each entry w_i of w to 1 or -1, where the sign is set based on what you believe is bad. So if you believe that a good weight for $f_2(s,a)$ is positive, then in initialization set $w_2 = 1$. Clearly motivate for each w_i why you believe the sign is bad. Of course, during experimentation, you can do sanity checks by using good initial weights, but in the end you should make it work for bad initial weights.

Once you have trained the agent for 5000 episodes, you should test its performance (the code automatically saves your Q-weights). It is clear in the code of `snake.m` how to perform testing (set test_agent = true). This will set α and ε to 0 (meaning no weight updates will occur and the policy is completely greedy), load the automatically saved Q-weights, and run the game for 100 episodes.

a) (3 points) Attach a printout of your Q weight update code (both for terminal and non-terminal updates). This should be 8 lines of Matlab code.

b) (6 points) Write down at least 3 different attempts (parameter configurations and/or 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.

c) (11 points) 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 report your final weight vector w for Q_w and associated final state-action feature functions f_i. Comment on why it works well or why it doesn't work so well.

a)

Here's the Q weight update code for the non-terminal update

```
        target  = reward + gamm*max(Q_fun(weights,
    state_action_feats_future));
        pred    = Q_fun(weights, state_action_feats, action);
        td_err  = target - pred;
        weights = weights + alph*td_err*state_action_feats(:, action);
```

Here it is for the terminal state

```
            target  = reward;
            pred    = Q_fun(weights, state_action_feats, action);
            td_err  = target - pred;
            weights = weights + alph*td_err*state_action_feats(:,
    action);
```

b)

- **Attempt 1:** $\varepsilon = 0.5$, $\alpha = 0.7$, $\gamma = 0.99$ and reward = $\{default : 0, apple : 1, death : -1\}$. **Score = 28.72**

```
    % Feature 1. Does the action bring us closer to the apple?
    state_action_feats(1, action) = (norm(apple - prev_head_loc) - ...
                                     norm(apple - next_head_loc))/sqrt
    (2*(N-2)^2);
```

```
    % Feature 2. Will the next action kill us?
    if(grid(next_head_loc(1),next_head_loc(2)) == 1)
        state_action_feats(2, action) = -1;
    else
        state_action_feats(2,action) = 0;
    end
```

```
    % Feature 3. Will this action lead to us eating the apple?
    if (grid(next_head_loc(1),next_head_loc(2)) == -1)
        state_action_feats(3,action) = 1;
    else
        state_action_feats(3,action) = 0;
    end
```

weight 1	weight 2	weight
3.9759	1.0154	0.9809

The initial weights were $[-1, -1, -1]$.

- **Attempt 2:** $\varepsilon = 0.5$, $\alpha = 0.7$, $\gamma = 0.99$ and reward = $\{default : 0, apple : 1, death : -1\}$. **Score = infinity loop**

```
% Feature 1. Does the action bring us closer to the apple?
state_action_feats(1, action) = (norm(apple - prev_head_loc) - ...
                                 norm(apple - next_head_loc))/sqrt
(2*(N-2)^2);
```

```
% Feature 2. Will the next action kill us?
if(grid(next_head_loc(1),next_head_loc(2)) == 1)
    state_action_feats(2, action) = -1;
else
    state_action_feats(2,action) = 0;
end
```

```
% Feature 3. How "grouped up" is the snake?
cropped_grid = grid(2:N-1,2:N-1);
indices = find(cropped_grid == 1);
[I, J]  = ind2sub(size(cropped_grid),indices);
sn_pos = [I,J];
sn_pos = [sn_pos;next_head_loc];

state_action_feats(3, action) = max(pdist(sn_pos))/sqrt(2*(N-2)^2)
;
```

weight 1	weight 2	weight
-1	-1	-1

I now replaced the third feature from before, instead of looking if the action will lead us to eat the apple I replaced it with a feature that looks how grouped up the snake is. This will incentivize the snake to straighten out. However, since there's no feature to incentivize the snake to eat the apple it will never get to the apple and just loop around in a square.

- **Attempt 3:** $\varepsilon = 0.3$, $\alpha = 0.5$, $\gamma = 0.7$ and reward $= \{default : 0, apple : 1, death : -1\}$. **Score = 28.72**

```
% Feature 1. Does the action bring us closer to the apple?
state_action_feats(1, action) = (norm(apple - prev_head_loc) - ...
                                 norm(apple - next_head_loc))/sqrt
(2*(N-2)^2);
```

```
% Feature 2. Will the next action kill us?
if(grid(next_head_loc(1),next_head_loc(2)) == 1)
    state_action_feats(2, action) = -1;
else
    state_action_feats(2,action) = 0;
end
```

```
% Feature 3. Will this action lead to us eating the apple?
if (grid(next_head_loc(1),next_head_loc(2)) == -1)
    state_action_feats(3,action) = 1;
else
    state_action_feats(3,action) = 0;
end
```

weight 1	weight 2	weight
2.7633	0.9709	0.9712

This is the same features as in attempt 1 but with different parameters. However, it yielded the exact same score as in attempt 1 but with different weights, which doesn't make a lot of sense to me.

c)

- **Final attempt:** $\varepsilon = 0.5$, $\alpha = 0.7$, $\gamma = 0.99$ and reward $= \{default : 0, apple : 1, death : -1\}$. **Score = 42.52**

```
% Feature 1. Does the action bring us closer to the apple (Binary)
?
if (norm(apple - prev_head_loc) - norm(apple - next_head_loc)) > 0
    state_action_feats(1,action) = 0;
else
    state_action_feats(1,action) = -1;
end
```

```
% Feature 2. Will the next action kill us?
if(grid(next_head_loc(1),next_head_loc(2)) == 1)
    state_action_feats(2, action) = -1;
else
    state_action_feats(2,action) = 0;
end
```

```
% Feature 3. Will this action lead to us eating the apple?
if (grid(next_head_loc(1),next_head_loc(2)) == -1)
    state_action_feats(3,action) = 1;
else
    state_action_feats(3,action) = 0;
end
```

weight 1	weight 2	weight
0	1	1

I now have a binary function as feature 1 that tells if I moved closer to the apple or not. However, I got a weight of 0 for this. All three features are now binary.