FMAN45 Machine Learning

ML Assignment-4

Praveenkumar Hiremath

August 17, 2023

Exercise 1: Tabular methods

For a small snake game, the total size of the grid is 7 X 7. The snake has a constant length of 3 (cells). This means that the interior grid where the snake moves is of size 5 X 5. Now the state-action value function, $Q(s,a)$ where s and a stand for state and action, respectively, can be stored in a tabular form. The table will be of size K x 3 where K is the total number of states in the game and the 3 possible actions are moving $left$, $right$, and $forward$. This snake game's interior grid (size 5 X 5) is shown below.

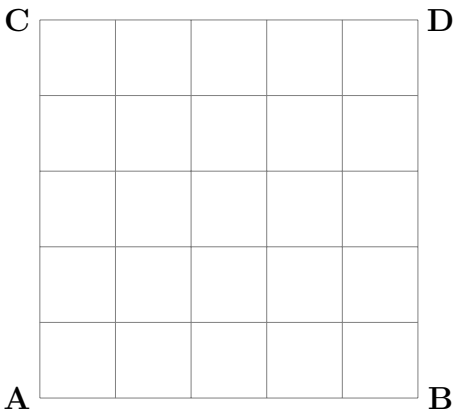

Figure 1: Interior grid where the snake moves and apple appears.

The snake can be straight-line or L-shaped. Considering the straight-line shape, placing the tail of the snake on one of the 4 sides (eg. side **AB**) gives 5 straight-line configurations for the snake. Moving these 5 configurations 1 cell away from the side **AB** and towards the side **CD**, gives 5 more configurations. If these 5 configurations are once again moved 1 more cell away from side **AB** and towards the side **CD**, the snake head will be on side **CD**. This will give another 5 configurations. Thus starting with the tail on one of the 4 sides, in this case, **AB**, and moving towards side **CD**, results in 5x3 = 15 straight-line configurations of the snake. Now repeating this process starting with the other 3 sides (individually) gives (5x3)x3 = 45 more straight-line configurations. Thus the total number of **straight-line configurations of the snake = (5x3)x4 = 60**.

Because the snake can move $left$ and $right$ as well, it can have L-shaped configurations. Let 2 units of the length of the snake (2 of a total of 3 units/cells) lie on the side **AB**. The other unit is on either of these 2 units thus forming L-shape. Then there will be 4 such L-shaped configurations along the side **AB**. Now moving these 4 configurations 1 cell away from side **AB** and towards **CD** will give 4 more configurations. These 4 configurations can be moved towards side **CD** until only 1 unit of the snake is on side **CD**. Thus starting with 2 units of snake length on side **AB**, this process gives a total of (4x4) = 16 L-shaped snake configurations. Similarly, starting with the other 3 sides (where 2 units of length of the snake are lying on a side and resulting configurations are moved 1 cell at a time towards the opposite side) gives (4x4)x3 = 48 more configurations. So far the total number of L-shaped configurations is (4x4)x4 = 64. However, the head position of the snake can be changed to the other end of the L-shape, giving 64 more configurations. As a result, **there are ((4x4)x4)x2 = 128 L-shaped snake configurations in total.** Therefore,

Total number of snake configurations = # Straight-line configurations + # L-shaped configurations = 60 + 128 = 188.

When the snake occupies 3 cells, an apple can be in any of the remaining 22 cells of the 5 X 5 interior grid. Therefore, the total number of non-terminal states (configurations), K is
K = (((5x3)x4) + (((4x4)x4)x2))x22 = 4136.

Exercise 2: Bellman optimality equation for the Q-function

Bellman equation for state-action value (Q-) function for any given policy π is as follows.

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

Here, γ is the discount factor, $T(s,a,s')$ is the state transition probabilities function and $R(s,a,s')$ is the reward function depending not just on current state,s, and action, a in this current state, but also on future state, s'. Bellman optimality equation for Q-function is

$$Q^{*}(s,a) = \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q^{*}(s',a') \right]. \tag{2}$$

Here, $Q^{*}(s,a)$ is the optimal Q-function. An optimal policy π^{*} can be obtained from $Q^{*}(s,a)$ as

$$\pi^{*}(s) = a^{*} = \arg\max_{a} Q^{*}(s,a) \tag{3}$$

(a) Rewriting equation (2) as an expectation using the notation $\mathbb{E}[\cdot]$.

If x is a discrete random variable and $f(x)$ is a function of x, under probability distribution $p(x)$, then the expectation of x and $f(x)$ are

$$\begin{aligned} \mathbb{E}_{p(x)}[x] &= \sum_{i=1}^{N} p(x_i) x_i \\ \mathbb{E}_{p(x)}[f(x)] &= \sum_{i=1}^{N} p(x_i) f(x_i) \end{aligned} \tag{4}$$

Using equation (4) and the fact that $T(s,a,s')$ is the state transition probabilities function in equation (2), equation (2) can be re-written as

$$Q^{*}(s,a) = \mathbb{E}_{T(s,a,s')} \left[R(s,a,s') + \gamma \max_{a'} Q^{*}(s',a') | s_t = s, a_t = a \right] \tag{5}$$

In the above equation, the recursion is maintained.

(b) Re-writing equation (2) as an infinite sum without recursion.

The optimal state value function, $V^{*}(s)$ can be written as

$$V^{*}(s) = \max_{a} Q^{*}(s,a) \tag{6}$$

Substituting equation (6) for $Q^*(s',a')$ in equation (5) gives

$$Q^*(s,a) = \mathbb{E}_{T(s,a,s')}\left[R(s,a,s') + \gamma V^*(s') | s_t = s, a_t = a\right]$$

where $\mathbb{E}[V^*(s')]$ is the expected future return following an optimal policy and $R(s,a,s')$ can be simplified to the reward $R(s) = r_{t+1}$ depending only on the current state, s. (7)

Further for next state, s',

$$V^*(s') = \left(\sum_{k=0}^{\infty} \gamma^k r_{t+k+2}\right) \tag{8}$$

Substituting equation (8) in equation (7) leads to

$$Q^*(s,a) = \mathbb{E}_{T(s,a,s')}\left[r_{t+1} + \gamma\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k+2}\right) | s_t = s, a_t = a\right] \tag{9}$$

Here, the actions, a, follow an optimal policy.

(c) Equation (2) says:

$Q^*(s,a)$ is the expected return (or utility) that the agent gets for taking action, a, in the state (starting) s and then following the optimal policy $\pi^*(s)$ after taking action a. Therefore, $Q^*(s,a)$ is the expectation of the sum of the current state reward ($R(s,a,s') = R(s)$) and the γ times the maximum future return ($V^*(s') = \max_{a'} Q^*(s',a')$).

(d) What policy is assumed in equation (2) and where in the equations?

By comparing the Bellman optimality equation (equation (2)) with the Bellman equation (equation (1)), it can be seen that an optimal policy $\pi^*(s)$ is followed in equation (2). This assumption is made by replacing the term $\gamma Q^{\pi}(s',\pi(a',s'))$ in the Bellman equation (equation (1)) by $\gamma \max_{a'} Q^*(s',a')$ (to get the Bellman optimality equation (equation (2)).

(e) What is γ and what is its effect on the Q-values?

γ is called the discount factor. It controls how far into future the agent can look at rewards. In other words, after taking action, a, in state s, and then following an optimal policy π^* (in case of $Q^*(s,a)$), how many future rewards should be included to calculate the $Q^*(s,a)$. If $\gamma = 1$, $Q(s,a)$ becomes an infinite sum. $Q(s,a)$ depends on only immediate rewards, if $\gamma = 0$ thus making the agent myopic. For $0 < \gamma < 1$, $Q(s,a)$ is finite.

(f) Explain what is $T(s,a,s')$.

$T(s,a,s')$ in equation (2) is called state transition probability function. It provides the probability of agent ending state s' after taking an action, a, in current state s (stochastic environment). For the small version of the snake game (entire grid of 7 X 7 with an interior grid of 5X5 where the snake finds an apple and takes actions),
(i) if action a in state s (a state next to the wall) takes the snake to state s' in the wall (and apple

is quite far from the head), $T(s,a,s') = 0$. $T(s,a,s')$ should be zero because such an action will end the game (snake dies).
(ii) other possibility is that the next state s' could be the state where apple is located. So it is very much preferred that the action a in state s leads to snake eating the apple and hence $T(s,a,s')$ can be very high or even equal to 1 (if it is 1, then it is zero for other states at that point in time). The sooner the snake eats the apple, the better it is.
(iii) if action a in previous state resulted in the current state s where the snake has eaten an apple, then new apple can appear with equal probability of 1/22 (uniform distribution) in any of the remaining 22 locations (because snake length = 3 and the interior grid is 5 X 5). Therefore, $T(s,a,s')$ can be 1/22.
(iv) state s' could be a state where there is no apple and no wall. Apple is located somewhere on the grid more than one state away from head. Let's call this s' as '0' reward state. In such case, the current state, s, will be surrounded by 3 (up, left and right) '0' reward states. Then the $T(s,a,s')$ value can be decided based on which of the 3 '0' reward states is closer to apple location. If all the 3 '0' reward states are equidistant from apple, $T(s,a,s')$ can be 1/3. If any of these 3 '0' reward states is closer to apple, then for the closer '0' state s', $T(s,a,s')$ can be close to or equal to 1.

These few possible values of $T(s,a,s')$.

Exercise 3: Bellman optimality equation for the Q-function

Off-policy vs on-policy

An agent in an active RL algorithm can be thought to have two types of policies, namely,
(i) behavioural policy using which the agent takes actions in the environment and sample data is collected.
(ii) learning policy: This is the optimal (target) policy the agent is supposed to learn eventually by using the experience gained through interactions with environment.

In on-policy, the behavioural policy is the same as the learning policy. Because of this, behavioural and learning policy depend on each other. The policy used is non-deterministic so that the agent explores the environment enough. Also the convergence to optimal policy may be faster because behavioural and learning policies are the same. Eg. State Action Reward State Action algorithm (SARSA) [1].

In off-policy, the behavioural policy is different from the learning policy. Therefore, the convergence may be slower. The agent learns the optimal policy (target policy) independently of the actions taken by the agent during the environment exploration. Eg. Q-learning [1].

Model Based vs Model Free Reinforcement Learning

In reinforcement learning (RL), the model of the environment is not available. In model based RL, the model of the environment, that is, state transition probabilities ($P(s'|s,a)$ or $T(s,a,s')$) and rewards ($R(s,a,s')$) are estimated first. Then with the help of a simulator, interaction between agent and environment can be studied.

Contrary to this, in model free RL, $T(s,a,s')$ and $R(s,a,s')$ are not estimated. Instead, optimal policy $\pi^*(s)$, value functions $V^*(s)$ and $Q^*(s,a)$ are directly obtained.

Active vs passive Reinforcement Learning

In passive RL, the agent does not actively pick a policy. Instead, the agent follows a fixed policy and evaluates how good it is. In active RL, the agent tries different actions and picks a suitable policy i.e. optimal policy. In an active RL, either on-policy or off-policy algorithms are adopted. The agent also needs to balance between exploration and exploitation.

Difference between Reinforcement Learning, supervised learning and unsupervised learning

(i) In supervised learning, labelled data is used whereas in the unsupervised learning, the data is unlabelled. In RL, the data is not available ($T(s,a,s')$ and $R(s,a,s')$ are not available) but an agent interacts with the environment and samples of the outcomes are used as data (feedback).
(ii) Supervised learning algorithms learn to predict an output based on the training data that contained inputs and corresponding output labels (or values). In unsupervised learning, an algorithm is trained on data with only inputs and no corresponding outputs, and learns to find clusters in data or predict patterns in data. In RL, an agent learns to perform actions by interacting with the environment.
(iii) In supervised and unsupervised learning, a model learns by minimizing a loss function. In RL, the agent tries to maximize the return (or value functions) for any sequence of actions it undertakes, thereby learning to make decisions.
(iv) Supervised learning is used for tasks like regression and classification. Unsupervised learning is used for segmentation and clustering. RL is used for decision making like agent learning to take actions. Eg. Self-navigating robot, mars-rover, snake game in this assignment, pacman game using RL.

Dynamic programming approach vs Reinforcement Learning to solve MDP problem

(i) In dynamic programming (DP), the state transition probabilities ($T(s,a,s')$) and rewards ($R(s,a,s')$) are predefined or available. These are not known beforehand in reinforcement learning (RL).
(ii) In DP, using the $T(s,a,s')$ and $R(s,a,s')$, value functions ($V(s)$ and $Q(s)$), and an optimal policy can be evaluated. Because $T(s,a,s')$ and $R(s,a,s')$ are not available in RL, they are either estimated first to subsequently evaluate optimal policy, or are completely ignored and optimal policy and value functions are directly computed without $T(s,a,s')$ and $R(s,a,s')$.
(iii) Unlike in DP, environment-agent interaction sampling plays a big role in RL. In RL, the agent first interacts with the environment by taking actions and gets feedback, if the action was good or bad. Such repeated interactions of the agent with the environment give samples of outcomes that are used to estimate $\pi^*(s)$, $V^*(s)$ and $Q^*(s,a)$.
(iv) In RL, the agent has to learn to make decisions based on the sampling outcomes of its interaction with the environment. Based on how the DP approach policy iteration works, there is no decision making in DP. The policy iteration simply finds the best possible policy based on predefined $T(s,a,s')$ and $R(s,a,s')$.
(v) If either the environment or the agent is evolving (in the current assignment, agent "snake" elongates by eating apples and changes the environment i.e. the number of pixels corresponding to '1' keeps increasing.), it is almost impossible to obtain an optimal policy using DP. But it is

still possible in RL because the agent learns from the sampled outcomes of its interaction with the environment.
(vi) Because of the iterative method to obtain optimal policy, the DP method can require a lot of memory (to save intermediate results of policy evaluation and improvement) when the state space is very large. However, RL method Q-learning should work fine despite large state space (though not very large in this assignment as the maximum size of the grid is 30X30).

Exercise 4: Policy iteration

The Bellman optimality equation for the state value (V-) function is

$$V^*(s) = \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s')\right]. \tag{10}$$

Here, $V^*(s)$ is the optimal V-function, γ is the discount factor, $T(s,a,s')$ is the state transition probabilities function and $R(s,a,s')$ is the reward function depending not just current state (s) and from this state, action (a), but also on future state (s').

(a) Rewriting equation (10) as an expectation using the notation $\mathbb{E}[\cdot]$.

Similar to in Exercise 1, equation (10) can be re-written in terms of $\mathbb{E}[\cdot]$ using equation (4). Again in equation (10), $T(s,a,s')$ is the state transition probabilities function. Thus, the equation (2) can be re-written as

$$V^*(s) = \max_a \left(\mathbb{E}_{T(s,a,s')} \left[R(s,a,s') + \gamma V^*(s') | s_t = s, a_t = a\right]\right) \tag{11}$$

In the above equation, the recursion is maintained.

(b) Equation (10) says:

Equation (10) says that for stochastic environment, the optimal value of a state s i.e. $V^*(s)$ function can be obtained as the maximum expected value (under probability distribution $T(s,a,s')$) of the sum of reward for action a taking agent from state, s, to state, s' ($R(s,a,s')$) and future state value functions ($V^*(s')$) that are dependent on optimal policy ($\pi^*(s)$) and discounted by γ.

(c) Purpose of the max-operator

In equation (10), the purpose of the max-operator is that it provides the value of the state s when an optimal policy is followed by the agent. This is because the goal is always to get maximum value for the state s (in other words possibility to get maximum return by being in state s). During policy extraction, the max-operator helps to extract an optimal policy that gives maximum value for being in state s. Such a policy $\pi^*(s)$ means that the agent has learned to take appropriate actions.

(d) Relation between $\pi^*(s)$ and $V^*(s)$

While

$$\pi^*(s) = \arg\max_a Q^*(s,a) \tag{12}$$

$$\pi^*(s) = \arg\max_a \left(\mathbb{E}_{T(s,a,s')}\left[R(s,a,s') + \gamma V^*(s') | s_t = s, a_t = a\right]\right)$$

$$\pi^*(s) = \arg\max_a \left(\sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma V^*(s')\right]\right) \tag{13}$$

The relation between $\pi^*(s)$ and $V^*(s)$ is called optimal policy extraction from $V^*(s)$.

(e) Why is the relation between $\pi^*(s)$ and $V^*(s)$ not simple?

The optimal state-action value function $Q^*(s,a)$ is a function of both state, s and action, a. But, optimal state value function $V^*(s)$ is just a function of state, s. $Q^*(s,a)$ is a matrix of size (N_s X N_a) where N_s is the number of states and N_a is the number of possible actions. Thus $Q^*(s,a)$ already has the information about the optimal actions. Therefore, it is easy to get $\pi^*(s)$ from $Q^*(s,a)$ suggesting the simplicity of the relation between $\pi^*(s)$ and $Q^*(s,a)$.

But, $V^*(s,a)$ is a column vector i.e. it has a size (N_s X 1) with its entries being the maximum value of state s for an optimal action and does not carry information on the optimal action itself. Therefore, the relation between $\pi^*(s)$ and $V^*(s)$ is complicated as shown in equation (13).

Exercise 5: Policy iteration

Policy iteration to find optimal policy

The policy iteration algorithm as described in [1] is implemented in the file "policy_iteration.m". The implementation is attached below.

```
% This while-loop runs the policy iteration.
while 1
    % Policy evaluation.
    while 1
        Delta = 0;
        for state_idx = 1 : nbr_states
            % FILL IN POLICY EVALUATION WITHIN THIS LOOP.
            val = values(state_idx);
            act = policy(state_idx);
            next_s = next_state_idxs(state_idx, act);

            if and((next_s ~= -1), (next_s ~= 0))  % Snake moves
               without getting apple or dying
                values(state_idx) = rewards.default + values(next_s
                   )*gamm;
            elseif next_s == 0  % Snake dies
                values(state_idx) = rewards.death;
            else   % Snake eates an apple
```

```
                values(state_idx) = rewards.apple;
            end
            Delta = max(abs(val-values(state_idx)), Delta);
        end

        % Increase nbr_pol_eval counter.
        nbr_pol_eval = nbr_pol_eval + 1;

        % Check for policy evaluation termination.
        if Delta < pol_eval_tol
            break;
        else
            disp(['Delta: ', num2str(Delta)])
        end
    end

    % Policy improvement.
    policy_stable = true;
    for state_idx = 1 : nbr_states
        % FILL IN POLICY IMPROVEMENT WITHIN THIS LOOP.
        %next_s_id = next_state_idxs(state_idx);
        prev_act = policy(state_idx);
        vals = zeros(length(next_state_idxs(state_idx,:)),1);
        for i = 1:length(vals)
            if and((next_state_idxs(state_idx,i) ~= -1),(
               next_state_idxs(state_idx,i) ~= 0))
                vals(i) = rewards.default + values(next_state_idxs(
                   state_idx,i))*gamm;
            elseif next_state_idxs(state_idx,i) == 0
                vals(i) = rewards.death;
            else
                vals(i) = rewards.apple;
            end
        end
        [num, ind] = max(vals);
        policy(state_idx) = ind;
        if (prev_act ~= policy(state_idx))
            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).
  if policy_stable
      break;
  end
```

To check the correctness of the implementation, $\gamma = 0.5$ and $\epsilon = 1$ are set in "snake.m". Running "snake.m" with these values gave 6 policy iterations and 11 policy evaluations. Therefore, the implementation seems to be correct.

(b) Investigating the effect of γ while $\epsilon = 1$.

Table 1: Details of policy iterations and evaluations for $\gamma = 0$,1,0.95.

γ-value	# Policy iterations	# Policy evaluations
0	2	4
1	-	infinite
0.95	6	38

For $\gamma = 0$, the snake (agent) is myopic and tries to maximize only the immediate reward. As a result fewer policy evaluations and policy iterations are needed to obtain a policy. However, this policy is not optimal to make the snake game optimal. For $\gamma = 0$, the snake keeps going in circle without scoring any apple. This is attributed to the snake being myopic and all the 3 possible actions (left, right, up) have same V-value. If to begin with, the snake is next to wall, it avoids running into the wall. If the snake is right next to an apple, it eats the apple and then starts going in circle.

For $\gamma = 1$, the snake keeps looking into future. $\gamma = 1$ mathematically leads to V-function being an infinite sum of future rewards. Thus the V-function never converges and number of policy evaluations and iterations cannot be obtained. The current snake game has 4136. During policy evaluation, all possible actions in state, s, would lead to same infinite V-function.

For $\gamma = 0.95$ (< 1), the V-function is finite as the future rewards are effectively discounted. Therefore, snake looks at enough future rewards and at a given state gets different V-value for different policies. Therefore, the required number of policy evaluations and iterations is significantly higher (than for $\gamma = 0$). Consequently, the snake tries different policies and follows an optimal policy enabling the snake game to continue forever (snake has constant length = 3. It scored more than 10000 apples. So assuming that the game continues forever).

(c) Investigating the effect of stopping criterion ϵ while $\gamma = 0.95$.

It is observed that for $\epsilon \leq 1$ only 6 policy iterations are required while the number of policy evaluations increases with decreasing ϵ. As the ϵ gets smaller, the algorithm needs to reduce the difference between two successive policy evaluations (Δ in code) to very small value. This makes the convergence require more number of evaluations. The rewards in this game are 'apple' = 1, 'death' = -1, 'default' = 0. This suggests that $-1 \leq V^{\pi}(s) \leq 1$. The maximum possible Δ is 2. So setting $\epsilon > 1$ results in always policy evaluation terminating with $\Delta = 2$. Therefore, 19 policy evaluations are observed for $\epsilon > 1$. For these same reasons, for $\epsilon \leq 1$, only 6 policy iterations are needed while for $\epsilon > 1$, 19 iterations are needed. The snake plays optimally in all these cases of ϵ.

Table 2: Details of policy iterations and evaluations for $\gamma = 0,1,0.95$.

γ-value	# Policy iterations	# Policy evaluations
10^{-4}	6	204
10^{-3}	6	158
10^{-2}	6	115
10^{-1}	6	64
0	6	38
10^{1}	19	19
10^{2}	19	19
10^{3}	19	19
10^{4}	19	19

All the ϵ values considered in this task eventually give an optimal policy for the snake. This is because (i) the only dying option for the snake is "running into wall" because of its constant length = 3. With this length, the snake can never bite itself. (ii) Because running into wall gives -1 reward, the algorithm never finds it as an optimal policy and always just moves to 'default' locations leading to eating apple.

Exercise 6: Tabular Q-learning

(a) Q-update code implemented in the file "rl_project_project_to_students/small_snake_tabular_q/snake.m"

For non-terminal states, the Q-update is implemented as follows:

```
sample = reward + (max(Q_vals(next_state_idx,:)))*gamm;
pred = Q_vals(state_idx, action);
td_err = sample - pred;
Q_vals(state_idx, action) = Q_vals(state_idx, action) + (alph*
   td_err);
```

For terminal states, the Q-update is implemented as follows:

```
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) Three different attempts

The default settings (as received) in "snake.m" are

- Learning rate $\alpha = 0.01$
- Greediness (for random actions) $\epsilon = 0.01$

- $\gamma = 0.9$
- α_update_iters = 0
- α_update_factor = 0.5
- ϵ_update_iters = 0
- ϵ_update_factor = 0.5
- Rewards: 'apple' = +1, 'death' = -1, 'default' = 0

With the above default settings, the snake died before even gathering a score = 1. So a number attempts were made to achieve atleast a score = 250. A few attempts are presented below.

Table 3: Parameters tuning to get better score in snake game using tabular Q-learning.

Parameters and test score	Attempt-1	Attempt-2	Attempt-3	Attempt-4	Attempt-5	Attempt-6
α	0.5	0.5	0.5	0.5	0.5	0.5
ϵ	0.1	0.1	0.1	0.1	0.1	0.1
γ	0.9	0.7	0.9	0.9	0.9	0.99
reward 'default'	0	0	0	0	0	0
reward 'apple'	1	100	200	50	100	100
reward 'death'	-1	-200	-20	-200	-200	-200
test score	122	281	546	3657	7849	32573

Initially, only α was changed while keeping other parameters equal to default setting. As expected, increasing α to 0.99 did not improve the score. By trial and error, making $\alpha = 0.5$ (small enough) and $\epsilon = 0.1$ (for exploration i.e. the snake can explore new actions so that it does not follow the same path always. Initial $\epsilon = 0.01$ was too low) gave a score = 122. This is the **score of Attempt-1** in Table 3.

Reducing α any further below 0.5 did not result in higher score. This is because Q-values are updated very slowly and don't learn the optimal value. Additionally, because $\epsilon = 0.1$ and $\alpha = 0.5$ are not too high, they are not gradually reduced further using α_update_iters, α_update_factor, ϵ_update_iters and ϵ_update_factor. Instead, increasing the reward for eating apple and punishing for running into wall by awarding more negative rewards, seemed logical to increase Q(s,a) such that the snake takes actions towards eating apple. This indeed increased the game score. Different reward values are tried for 'apple' and 'death' and better scores are obtained in **Attempt 3 - 5**, see Table 3. Very good score was obtained for 'apple' = 100, 'death' = -200 and 'default' = 0.

Using the settings same as in Attempt-5 but only reducing γ to 0.7 (**Attempt-2**), resulted in much lower score as the snake could now take action based on fewer future rewards (or Q(s',a')γ). Therefore, $\gamma = 0.99$ was chosen along with other settings in **Attempt-6** and maximum score of 32573 was obtained. This is the maximum score that was possible.

(c) Final settings:

The final settings shown below lead to a **score = 32573**, see Table 3.

- $\alpha = 0.5$
- $\epsilon = 0.1$
- $\gamma = 0.99$
- reward 'default' = 0
- reward 'apple' = 100
- reward 'death' = -200

With these settings, it was possible to train a very good policy for the small snake game using tabular Q-learning.

(d) Difficult to achieve optimal behavior within 5000 episodes.

The downsides of tabular Q-learning are (i) need to explore enough while training (exploration and exploitation concept) (ii) need to slowly lower the learning rate, α, to make it small enough.

In the 7 X 7 grid of snake game, there are already 4136 states and three possible actions. The (s,a) (i.e.(states,actions)) space is quite large to explore. The smaller α (and slower lowering of α) means that the Q-value is updated slowly as well. Further, making ϵ too high and slowly reducing it also doesn't help. This is because the snake always explores and doesn't learn the optimal policy. Exploring the (s,a) space well enough and updating Q-value at good rate prove to be challenging with less than 5000 episodes. Therefore, achieving optimal behavior within 5000 episodes can be difficult.

Exercise 7: Q-learning with linear function approximation

(a) Q weight update code implemented in the file "rl_project_project_to_students/linear_q/snake.m"

For non-terminal states, the Q weight update is implemented as follows:

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

For terminal states, the Q weight update is implemented as follows:

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

(b) Three different attempts:

List of engineered features (in the file "extract_state_action_features.m").
Feature (a): Distance between the snake head and the location of apple.
The distance between the snake head and apple's location is measured by taking both L_1-norm and L_2-norm. However, L_1-norm has given better score, see Table 4. This feature is included because snake eating the apple sooner by traversing a shorter distance is preferred. The initial weight used for this feature is +1 because reducing the distance to apple is the goal (good weight is negative so the initial weight is chosen as +1). This feature is implemented as follows.

```
% Feature 1: Distance to apple (pixel '-1'). Weight is '+1'
max_pos_l1_dis = norm(size(grid),1);
[next_s_loc, next_s_dir] = get_next_info(action, movement_dir,
   head_loc);  % Getting head position and direction in next state
[minusones_rows, minusones_cols] = find(grid == -1); %Apple
   location
minus_ones_pix = [minusones_rows, minusones_cols];
dist_minusones = norm(next_s_loc - minus_ones_pix,1);
state_action_feats(1, action) = (dist_minusones/max_pos_l1_dis); %
   Normalizing
```

Feature (b): Where does the next move take the snake to?
This feature checks if a move taken by the snake lands it in no apple location or apple location or wall or snake's own body. It is preferred that the next state is an apple location. The feature is set to +1 if the next state is either apple or at least not wall or snake's body. Therefore, a good weight for this feature is positive meaning the initial weight is to be set to -1. This feature is implemented as shown below.

```
%   Feature 2: Next move landing in pixel of '0' or '-1'? Weight
   for this be '-1'
if (grid(next_s_loc(1),next_s_loc(2)) ~= 1)
    state_action_feats(2, action) = 1;
else
    state_action_feats(2, action) = -1;
end
```

Feature (c): Feature to capture maximum distance between snake's head and pixels of '1'.

This feature captures the importance of being away from states that will lead to snake's death. This feature is set equal to the maximum distance of the next state from wall and its body. Using minimum distance did not update the weight of this feature during training, hence the maximum distance was chosen. Because the need is to maintain a maximum possible distance, a good weight for this feature is positive making its initial weight to be set to -1. The feature is implemented as follows.

```
% Feature promoting maximum distance between head and pixels of
   '1'(snake/wall).  weight is '-1'.
```

```
[ones_rows, ones_cols] = find(grid == 1); %snake and wall
dist_ind_ones = zeros(length(ones_rows));
for i = 1:length(ones_rows)
    dist_ind_ones(i) = norm(next_s_loc-[ones_rows(i), ones_cols(i)
       ],1); % Tried L1 and L2 norms
end

[max_dist_ones,index] = max(dist_ind_ones,[],'all');
dist_ones = max_dist_ones;
state_action_feats(3,action) = (dist_ones/max_pos_l1_dis); %
   Normalizing
```

Feature (d): Feature to get information on more than one pixel ahead of snake head.

Just like feature (c), this feature tries to emphasize the need for next state to be either 'apple' or 'default'. While feature (b) only checks if only the next state is 'apple' or 'default' or 'death', this feature looks a bit ahead. If the states adjacent to next state lead to 'death'? If next action does not get closer to death, the feature is set to +1 and is set to -1 otherwise. Again, the initial weight is set to -1 because a good weight is positive. The implementation is as shown below.

```
% Feature 4: Looking a bit more ahead to avoid pixels of '1' (snake
   /wall). weight is '-1'.
desired_pix = [0 -1];
%Testing head running into pixels of '1'.
if ((next_s_loc(2)+1<(N-1)) & (next_s_loc(1)+1<(N-1))) & ((
   next_s_loc(2)-1>0) & (next_s_loc(1)-1>0))
    if ((ismember(grid(next_s_loc(1), next_s_loc(2)+1),desired_pix)
       ) | (ismember(grid(next_s_loc(1)+1, next_s_loc(2)+1),
       desired_pix)) | (ismember(grid(next_s_loc(1)-1, next_s_loc(2)
       +1),desired_pix)) | (ismember(grid(next_s_loc(1)+1,
       next_s_loc(2)),desired_pix)))
         state_action_feats(4, action) = 1;
    elseif ((ismember(grid(next_s_loc(1), next_s_loc(2)-1),
       desired_pix)) | (ismember(grid(next_s_loc(1)+1, next_s_loc(2)
       -1),desired_pix)) | (ismember(grid(next_s_loc(1)-1,
       next_s_loc(2)-1),desired_pix)))
         state_action_feats(4, action) = 1;
    else
        state_action_feats(4, action) = -1;
    end
else
    state_action_feats(4, action) = -1;
end
```

Feature (e): Feature indicating if there are very narrow snake configurations.

This feature checks the presence of snake configurations that are too narrow (as the snake grows) that could lead to biting its body. This is done by checking if the snake is taking action resulting in its movement parallel to its own body. The initial weight of this feature is also -1. The feature implementation is below.

```
% Feature 5: to avoid very narrow snake configuration. Weight = -1.
% Copying grid info
inner_grid = grid;
size(inner_grid);
inner_grid(:,1) = [];
inner_grid(:,end) = [];
inner_grid(1,:) = [];
inner_grid(end,:) = []; % This is 28X28
size(inner_grid);
[snake_i, snake_j] = find(inner_grid == 1);
row_dist = max(snake_i) - min(snake_i);
col_dist = max(snake_j) - min(snake_j);
if and((and(col_dist == 0, row_dist == 10)),(and(col_dist == 0,
   row_dist == 10)))
    state_action_feats(5, action) = 1; % straight config
elseif (or(row_dist == 1, col_dist == 1))
    state_action_feats(5, action) = 0; % very narrow config
else
    state_action_feats(5, action) = 1; % Not too narrow
end
```

At first, only the features (a) and (b) are used to train the model. Though α, ϵ and γ values were altered, the values $\alpha = 0.5$, $\epsilon = 0.1$ and $\gamma = 0.9$ gave better score. Changing the reward values did not improve the score (see Attempt-2 and Attempt-3 in Table 4). Additionally, slow reduction in α and ϵ was also in vain as the Q-updates for the used features do not change any more. This conclusion is drawn because, in Attempt-4 (Table 4), updating α for every 500 episodes using update factor 0.1, showed that the weights w_1 and w_2 converge to -246.48 and 96.70, respectively. This indicates that the $\alpha = 0.5$, $\epsilon = 0.1$ and $\gamma = 0.9$ are the best possible settings and now the most influencing parameter is the engineered features. The best average test score using just features (a) and (b) is 42.52.

In Table 5, the results using features (a), (b) and (c) are presented. Adding feature (c) did not improve the average test score. Only a score of 38.81 was achieved indicating that it is not such an influential feature.

In Table 6, features (a), (b), (c), (d) and (e) were used. Now the complexity of the model has increased quite a bit (i.e. because linear function to be approximated has 5 features $\implies Q(s,a) \approx Q_{\mathbf{w}}(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + w_3 f_3(s,a) + w_4 f_4(s,a) + w_5 f_5(s,a)$). With $\alpha = 0.5$, $\epsilon = 0.1$, and $\gamma = 0.9$, the snake got stuck in an infinite loop during testing. So the γ was reduced to 0.7 and α was slightly increased to 0.6. This allowed a bit better updating of weights for Q_w(s,a) and looking at fewer future Q(s',a'). This setting did not result in snake getting stuck in infinite loop during testing. Using 5 features yielded an average test score = 38. However, the features (c), (d) and (e) are not good because they have not allowed approximating Q(s,a) so as to mitigate game ending because of snake biting its body, see Figure (2).

Table 4: Parameters tuning to get better score in snake game using Q-learning with linear function approximation. Here, 2 features (a) and (b) are used. The final weights after training are labelled w_i for i = 1,2. Superscripts $L1$ and $L2$ on test score indicate that L_1- and L_2-norms, respectively, are used to measure distance in feature (a). The initial weights for feature (a) and feature (b) are +1 and -1, respectively.

Parameters and avg test score	using features (a) and (b)			
	Attempt-1	Attempt-2	Attempt-3	Attempt-4
α	0.5	0.5	0.5	0.5
ϵ	0.1	0.1	0.1	0.2
γ	0.9	0.9	0.9	0.9
α_update_factor	-	-	-	0.1
α_update_iters	-	-	-	500
ϵ_update_factor	-	-	-	-
ϵ_update_iters	-	-	-	-
reward 'default'	0	0	0	0
reward 'apple'	100	100	10	100
reward 'death'	-200	-200	-50	-200
w_1	-211.87	-193.99	-31.94	-246.48
w_2	122.38	109.45	26.52	96.70
avg test score (100 episodes)	28.72^{L2}	42.52^{L1}	42.52^{L1}	42.52^{L1}

Table 5: Parameters tuning to get better score in snake game using Q-learning with linear function approximation. Here, 3 features (a), (b) and (c) are used. The final weights after training are labelled w_i for i = 1,2,3. Superscripts $L1$ and $L2$ on test score indicate that L_1- and L_2-norms, respectively, are used to measure distance in feature (a). The initial weights for features (a), (b) and (c) are +1, -1 and -1, respectively.

Parameters and avg test score	using features (a), (b) and (c)	
	Attempt-1	Attempt-2
α	0.5	0.5
ϵ	0.1	0.1
γ	0.9	0.9
α_update_factor	-	-
α_update_iters	-	-
ϵ_update_factor	-	-
ϵ_update_iters	-	-
reward 'default'	0	0
reward 'apple'	100	100
reward 'death'	-200	-200
w_1	-193.55	-201.20
w_2	110.18	98.47
w_3	-35.07	-17.38
avg test score (100 episodes)	34.7^{L2}	38.81^{L1}

Table 6: Parameters tuning to get better score in snake game using Q-learning with linear function approximation. Here, 5 features (a), (b), (c), (d) and (e) are used. The final weights after training are labelled w_i for i = 1,2,3,4,5. The initial weights for features (a), (b), (c), (d) and (e) are +1, -1, -1, -1 and -1, respectively.

Parameters and avg test score	using features (a), (b), (c), (d) and (e) Attempt-1
α	0.6
ϵ	0.1
γ	0.7
α_update_factor	-
α_update_iters	-
ϵ_update_factor	-
ϵ_update_iters	-
reward 'default'	0
reward 'apple'	100
reward 'death'	-200
w_1	-274.79
w_2	101.15
w_3	-40.86
w_4	3.17
w_5	-23.26
avg test score (100 episodes)	38.0^{L1}

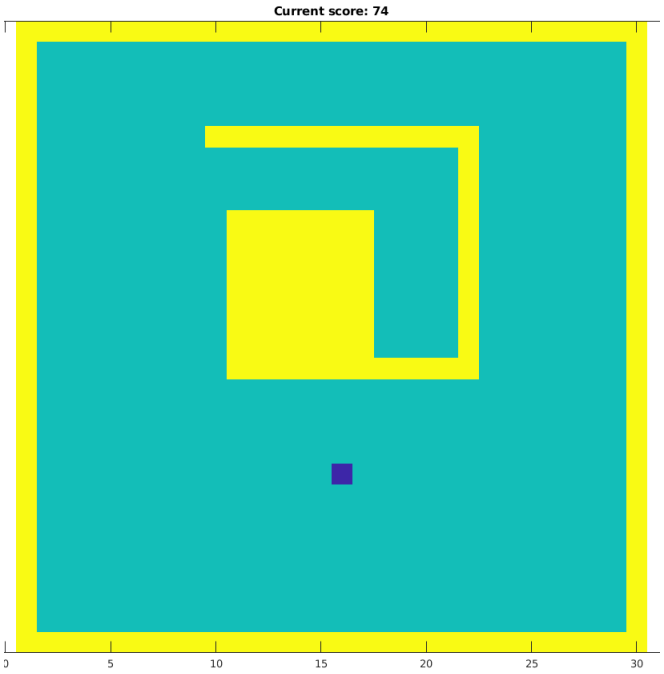

Figure 2: Exercise 7: Q-learning with linear approximation - Image of snake running into its tail and causing the game ending.

(c) Final settings

- Feature (a) Where does the next move take the snake to?
- Feature (b) Distance between the snake head and the location of apple.
- $\alpha = 0.5$, $\epsilon = 0.2$, $\gamma = 0.9$
- $\alpha\_update\_factor = 0.1$
- $\alpha\_update\_iters = 500$
- reward 'apple' = 100
- reward 'death' = -200
- reward 'default' = 0

Above final settings give an average score of 42.52 over 100 episodes.

Table 7: Initial and final weights for features (a) and (b) for above settings.

Initial weights, $\boldsymbol{w_0}$	Final weights, $\boldsymbol{w}$	$f(s,a_1)$	$f(s,a_2)$	$f(s,a_3)$
$w_1 = +1$	w_1 = -246.48	0.2	0.2	0.1667
w_2 = -1	$w_2 = 96.70$	-1	-1	-1

Then,

$$Q(s,a) \approx Q_{\mathbf{w}}(s,a) = -246.48 f_1(s,a) + 96.70 f_2(s,a)$$

The final weights (that seemed to have converged as they were not being updated after around ∼3000 episodes.) are $w_1 = -246.48$ and $w_2 = 96.70$. This indicates that the $\alpha = 0.5$ with reduction factor 0.1 every 500 episodes, $\epsilon = 0.2$ and $\gamma = 0.9$ along with rewards shown in Table 4 Attempt-4 are optimal settings possible. Choosing 2 features (a) and (b) works because only two Q-weights are to be optimized, thus the model is simpler. However, this is not the best result possible because the snake keeps biting its own body, see Figure 2.

It is important to engineer a feature that can capture the essence of snake biting its own body. Attempts made through the engineered features (c), (d) and (e) to achieve this are poor and did not really contribute to improving the score. All these (c), (d) and (e) features try to do more or less the same thing.

References

[1] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.