Machine Learning - FMAN45
Assignment 4 - spring 2019

Reinforcement Learning for Playing Snake
Policy Ieration and Q-learning
Q-learning with Linear Function Approximation

Hicham Mohamad - hi8826mo-s
hsmo@kth.se

July 17, 2019

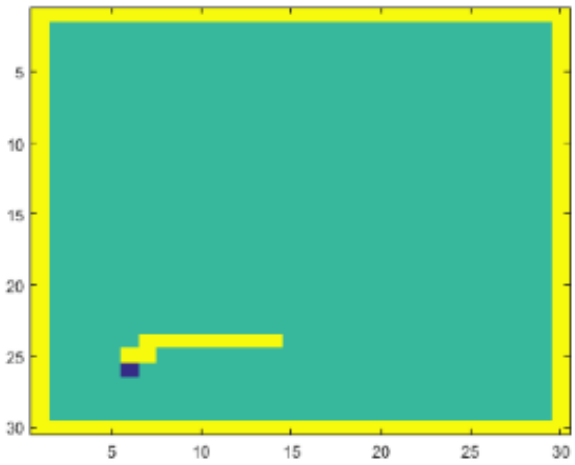
1 Introduction

In this assignment, we are going to train reinforcement learning agents to play the classic video game **Snake** on a two-dimensional grid. In the first part, we study five theoretical tasks regarding state spaces and Bellman equations. In the second part, we deal with three experimental tasks, that involves Policy Iteration, Q-learning and Approximate Q-learning. For solving these tasks, we consider two different sections regarding the Snake game. In the first section, we assume *tabular methods* where a smaller version of the Snake game is considered. Whereas in the second section a full version of the game is used.

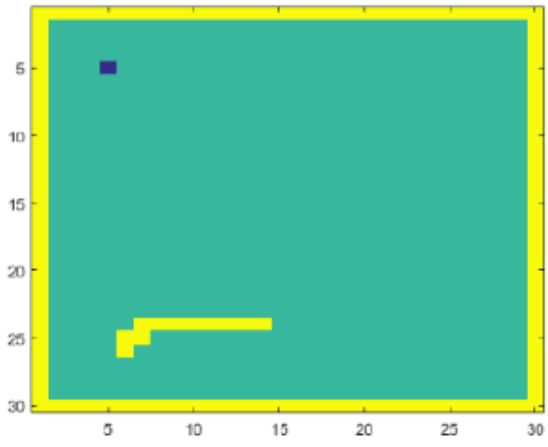
2 Objectives

The objectives of this assignment are to:

1. Derive the value of K, the number ot states in a smaller example of the Snake game.
2. Analyse the Bellman optimality equation for the Q-function.
3. Explain what a transition function T(s, a, s') is for the small version of Snake, in Bellman optimality equation.
4. investigate the issues of following the Dynamic Programming procefure for finding an optimal policy in the truncated game.
5. Analyse the Bellman optimality equation for the state value function.
6. implement Policy Improvement algorithm (Policy Iteration).
7. implement Q-learning algorithm.
8. implement Q-learning with linear function approximation.



(a)



(b)

Figure 1: a)The length of the snake is 10 pixels. An apple is located at a random location, in this case in (m; n) = (26; 6). After a few more time-steps, this is where the snake is located. Its direction is west. The initial apple has not yet been eaten. b) Given the state shown in Figure 1a, the player chooses to go left; the new snake direction is south. The head of the snake eats the apple, causing the length of the snake to increase by 1; the body remains in the same place during the next movement, growing one pixel in the movement direction. The apple disappears and a new one is randomly placed on an empty pixel.

3 Snake game

There exist several variants of this game, but in most variants the player is controlling a connected set of pixels known as the snake on a two-dimensional grid. Some pixels are apples. The goal of the game is to steer the snake to the apples - each apple gives a number of points - while at the same time avoiding colliding with the body of the snake and/or the walls of the grid. The following Snake variant is used in this assignment:

1. The game is played on a two-dimensional grid of size $M \times N$ pixels. In this assignment, $M = N = 30$.
2. The border pixels constitute walls, through which no movement is possible. In this assignment, it means that the player can only move around on the **interior** 28×28 grid, and apples can only exist in the interior of the grid.
3. The player will be controlling a **connected set of pixels** known as the snake.
4. The snake has a head (one pixel) and a body (the remaining pixels of the connected set of pixels). The head is at one end of the snake, but looking solely on one fixed state ("screen") of the game, it is not possible to determine at which end of the snake the head is located.
5. However, **the state of the game is automatically updated at some fixed frequency**, i.e., the duration for which the game is "paused" is fixed. What happens from a state to the next state is that the snake moves one pixel in the direction (N/E/S/W) of the head of the snake. Thus, the complete state of the game can be inferred by considering two **consecutive frames**.
6. At each time-step, the player must perform precisely one of **three possible actions** (movements): left, forward, or right (relative to the direction of the head). To be more precise, the player enforces a direction for the head of the snake, so that at the next time-step the snake moves in the chosen direction.
7. As mentioned, the state of the game is automatically updated at some fixed frequency. The higher the frequency, the harder the game becomes, as the player needs to choose one of the three actions above in a shorter amount of time.
8. Initially, **the length of the snake is 10 pixels**. It forms a straight line of pixels, with its head located at the center of the grid, facing a random direction (N/E/S/W).

9. Each time-step **there is exactly one apple in the grid**, occupying exactly one pixel.
10. When the snake eats an apple, **a new apple is simultaneously placed at uniform random** on an unoccupied pixel in the grid (meaning a pixel in the interior of the grid, which is not occupied by any pixel of the snake).
11. The game ends as soon as the head of the snake moves into a wall or the body of the snake.
12. **Eating an apple yields 1 point**; no other points (positive nor negative) are awarded throughout the game. The final score is thus the total number of apples eaten by the snake.

The crux is that each time the snake eats an apple, the snake becomes one pixel longer, making it increasingly difficult to control the snake so as to not hit itself or the walls, especially since the snake automatically moves with a certain frequency, even if the player is not making any active decisions.

4 Background

4.1 Markov Decision Processes (MDPs)

A Markov decision process is a tuple $(S, A, T(s, a, s'), \gamma, R)$, where:

- S is a set of **states**. (For example, in autonomous helicopter flight, S might be the set of all possible positions and orientations of the helicopter.)
- A is a set of **actions**. (For example, the set of all possible directions in which you can push the helicopter's control sticks.)
- $T(s, a, s')$ are the **state transition** probabilities. For each state $s \in S$ and action $a \in A$, $T(s, a, s')$ is a distribution over the state space. So briefly, $T(s, a, s')$ gives the distribution over what states we will transition to if we take action a in state s.
- $\gamma \in [0, 1)$ is called the **discount factor**.
- $R : S \times A \to \mathbb{R}$ is the **reward** function. (Rewards are sometimes also written as a function of a state S only.

The dynamics of an MDP proceeds as follows: We start in some state s_0 , and get to choose some action $a_0 \in A$ to take in the MDP. As a result of our choice, the state of the MDP randomly transitions to some successor state s_1 , drawn according to $s_1 \approx T(s_0, a_0)$. Then, we get to pick another action a_1 . As a result of this action, the state transitions again, now to some $s_2 \approx T(s_1, a_1)$. We then pick a_2 , and so on.

Actually MDPs are non-deterministic search problems. For MDPs, we want an optimal policy $\pi^* : S \to A$. A policy π gives an action for each state. An optimal policy is one that maximizes expected sum of rewards (utility) if followed.

4.2 Reinforcement Learning

Reinforcement has been carefully studied by animal psychologists for over 60 years. In Reinforcement Learning we examine how an agent can learn from success and failure, from reward and punishment.

In supervised learning, we saw algorithms that tried to make their outputs mimic the labels y given in the training set. In that setting, the labels gave an unambiguous "right answer" for each of the inputs x. In contrast, for many **sequential decision making** and **control** problems, it is very difficult to provide this type of explicit supervision to a learning algorithm. For example, if we have just built a four-legged robot and are trying to program it to walk, then initially we have no idea what the "correct" actions to take are to make it walk, and so do not know how to provide explicit supervision for a learning algorithm to try to mimic.

In the reinforcement learning framework, we will instead provide our algorithms only a reward function, which indicates to the learning agent when it is doing well, and when it is doing poorly. In the four-legged walking example, the reward function might give the robot positive rewards for moving forwards, and

negative rewards for either moving backwards or falling over. It will then be the learning algorithm's job to figure out how to choose actions over time so as to obtain large rewards.

Reinforcement learning has been successful in applications as diverse as autonomous helicopter flight, robot legged locomotion, cell-phone network routing, marketing strategy selection, factory control, and efficient web-page indexing. The study of reinforcement learning relies on the theory of the Markov decision processes (MDP), which provides the formalism in which RL problems are usually posed.

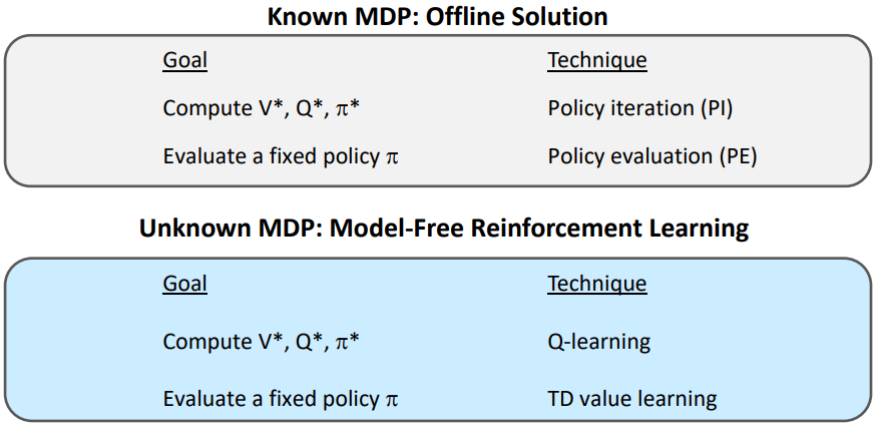


Figure 2: Known MDP: Offline Solution and Unknown MDP: Model-Free Reinforcement Learning.

5 Tabular Methods - Smaller Snake game

Before considering the Snake game, in this first section of the assignment we consider a much smaller example in which the Snake length does not increase by eating apples. Specifically:

- The grid is 7×7, so the **interior grid** in which the snake can move is 5×5
- The snake has constant length 3.
- Everything else is as for the full game.
- In this small, special case, it is possible to maintain a **table** which stores the state-action value $Q(s, a)$ for each state-action pair (s, a). This table can be represented by a $K \times 3$ matrix $\mathbf{Q}$, *where K is the number of states in the game, and the 3 columns correspond to the three possible actions left, forward and right.*
- Note that while it is possible to store state-action values in **terminal states**, in the small Snake game this corresponds to the snake having its head in a wall location, it is not necessary to do so, since no action can be taken in a terminal state. In practice, the state-action value of such terminal state-action pairs are set to zero, or they can simply be ignored. We will do the latter, so K refers to the number of non-terminal states.

5.1 Configuration of the smaller Snake game - Derivation of K

Exercise 1

Derive the value of K above.

For solving this task, we need to investigate the different possible configurations of the snake including its movement direction. As a possible strategy to do this, these snake configurations can be divided into two categories depending on whether the three connected pixels of the snake, head + body, form a straight line or no. Thus, we need to study two different cases as following:

- **Aligned head and body:** In this case we have 15 possible vertical lines and 15 possible horizontal line, of 3 pixels each, which can be occupied by the snake. Then we need to take into account the movement directions as the double possible locations of the head pixel for each straight line. In this way we count 60 possible linear configurations of the snake.

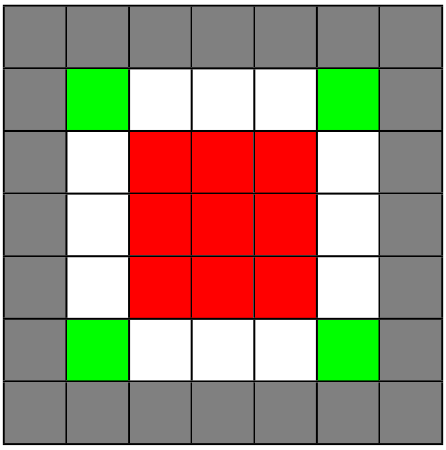
- **Non-aligned head and body:** For this we consider the 5×5 grid shown in Table 1. The gray pixels refer to the border pixels which constitute walls where no movement si possible. So the snake can move in the interior grid 5×5. The other colors represent the location of head when the body is not aligned with. So depending on whether the location is in a corner, along the wall or at least one field away from the wall we have different possible configurations for the snake. These can be 2, 4 or 8 configurations depending on whether the pixel color is green, white or red respectively.

According to this strategy, we can calculate the different possible configurations of the snake as following

- Green pixels give $4 \times 2 = 8$.
- White pixels give $3 \times 4 \times 4 = 48$.
- Red pixels give $9 \times 8 = 72$.
- Straight lines give $30 \times 2 = 60$.

At the end we need to consider the possible apple locations for every given configuration of the snake. Thus, we get the total number of states in the game as: $K = (8 + 48 + 72 + 60) \times (25 - 3) = 4136$ states

Table 1: Small Snake grid where the colors refer to different possible locations of one end of the snake (head) when the body does not form a straight line. The gray color refers to the border pixels which constitute walls where no movement si possible. So the snake can move in the interior grid 5×5.



5.2 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

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

Once we have $Q^*(s, a)$, an optimal policy π^* is give by

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

Note that in the tabular case we could store all $Q^*(s, a)$ in a matrix Q^* , which would be $K \times 3$ in the small Snake game.

Exercise 2

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

By rewritting the Equation in 1 as an expectation, we get

$$Q^*(s, a) = \mathbb{E}_{s'}[f(x)] = \sum_{s'} p(x) f(x) = \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma \max_{a'} Q^*(s', a') \right] \tag{3}$$

From this we can conclude that

$$Q^*(s,a) = \mathbb{E}_{s'}\left[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')\right] \tag{4}$$

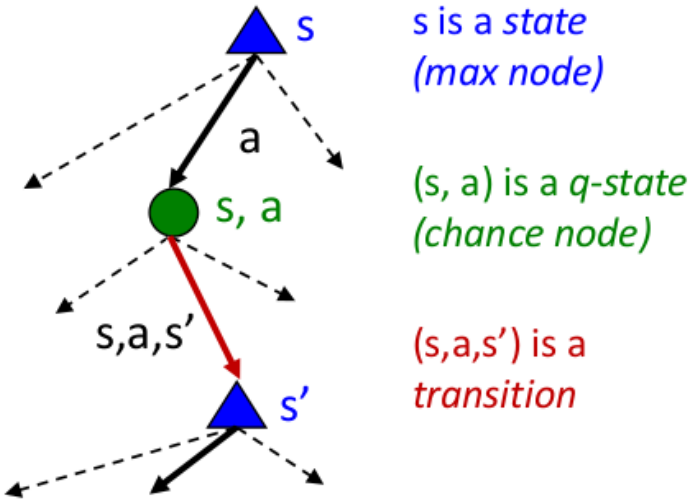


Figure 3: Markov Decision Processes (MDPs) as non-deterministic search problems.

b) Explain what Eq. 1 is saying. Your answer must reference all the various components and concepts of the equation $(Q^*, \Sigma, T, R, \gamma, \max, s, a, s')$**.**

First we should notice that $Q(s,a)$ and $R(s)$ are quite different quantities; $R(s)$ is the "short term" reward for being in s, whereas $Q(s,a)$ is the "long term" total reward from s onward.

For the state–action pair (s,a), the **optimal action-value** function $Q^*(s,a)$ is the expected sum of the immediate reward and the future discounted rewards we'll get upon after taking action a, assuming that the agent chooses the optimal policy and follows it thereafter. Actually we should note that $\pi^*(s)$ gives the action a that attains the maximum in the "max" in Equation . And the transition function $T(s,a,s')$ is the probability that an action a from s leads to s', i.e., $P(s'|s,a)$.

Examining the equation in more detail, we have seen before that the summation term above can be rewritten

$$Q^*(s,a) = \mathbb{E}_{s'}\left[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')\right] \tag{5}$$

This is the expected sum of discounted rewards for starting in state s' , where s' is distributed according $T(s,a,s')$, which is the distribution over where we will end up after taking the first action $\pi(s)$ in the MDP from state s.

All optimal policies achieve the optimal action-value function, i.e.

$$Q_{\pi^*}(s,a) = Q^*(s,a) \tag{6}$$

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

A remarkable consequence of using discounted utilities with **infinite horizons** is that the optimal policy π^* is independent of the starting state. Of course, the **action sequence** won't be independent; we remember that a policy is a function specifying an action for each state.

The discount $\gamma \in [0,1]$ weihgts the future rewards. The value of a reward R that is k+1 steps into the future is $\gamma^k R$. We can distinguish the effect that γ can have on the optimal policy $\pi^*(s)$ in Equation 2 by considering the following cases:

- when γ is close to 0 we get a **short-sighted** policy.

- when γ is close to 1 we get a **far-sighted** policy.
- when $\gamma = 0$ we get that only the next state is rewarded, the **immediate reward**.
- when $\gamma = 1$ we get equally weighted values, **undiscounted utilities**. Thus, in this case the optimal policy π^* depends of the starting state.

Note: All the answers to this exercise is with respect to general Markov decision processes, not with respect to the Snake game in particular.

5.3 The transition function T(s,a,s') for the smaller Snake game

A transition function $T(s, a, s')$ is the probability that an action a from s leads to s', i.e., $P(s'|s, a)$. It is also called the model or the dynamics and it should not be confused with the meaning of the policy π.

Exercise 3

In this exercise we need to explain what $T(s, a, s')$ in Bellman equation in 1 is for the small version of Snake. Here we can consider that there are two cases for a state-action pair (s, a):

- Deterministic case
- Uniform random case

How many possibilities are there for s' in each of these cases? Suppose that an agent is situated in the 5×5 Snake environment shown in Table 1 and start moving. It must choose an action at each time step. The actions available to the agent in each state are `left`, `forward` and `right`. All these actions are deterministic and result in a particular next state s' with probability 1. However, the new apple is simultaneously placed at uniform random on an unoccupied pixel, so the outcome occurs with probability 1/22, where we substract the 3 pixels in the grid the snake can occupy by its body.

5.4 Dynamic Programming procedure for finding an optimal policy

For finding an optimal policy π^*, one could try to implement the Bellman equation in 1 directly through Dynamic Programming. The dynamic programming procedure would first fill in all entries (i, j) of Q^* terminal state-action pairs with zeros, and then go **backwards** , i.e. further and further away from terminal state-action pairs. Once the dynamic programming procedure is done, we would have Q^*. Then, at a given state s, we would simply look at the corresponding row in Q^* and pick the action a corresponding to the largest element of the row, this corresponds precisely to

$$\pi^*(s) = \arg\max_a Q^*(s, a) \tag{7}$$

Exercise 4

In this task, we consider the small Snake game assuming $\gamma \in (0, 1)$ and the reward signal is

$$R(s, a, s') = \begin{cases} -1 & \text{if the snake dies} \\ +1 & \text{if the snake eats an apple} \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

a)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.

Here we can say that the normal game is a sequence of consecutive truncated games, where an optimal agent to the truncated game cannot be necessarily optimal to the normal game because he does not take into account the future states in his policy. But because the apple is always drawn randomly we can say that an optimal agent to the truncated game will be optimal to the normal game.

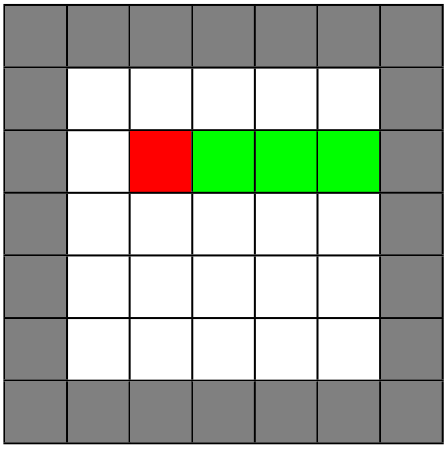
b) What are the terminal states in the dynamic programming problem for the truncated game?

For the truncated game, we can distinguish between two different terminal states in the dynamic programming problem:

- As the head of the snake moves into a wall.
- As the snake eats an apple.

c) Beginning from terminal state(s) in b), perform the first step of the dynamic programming procedure, by using eq. 1. Explain what is the problem. Draw a figure of the game with the snake one step from the terminal state(s) to motivate your answer.

Table 2: The snake in green at the terminal state eating an apple in red.



By begining from terminal state as shown in the game screen drawn in Table 2, we can apply the Bellman equation in 1 and get

$$\begin{aligned} Q^*(s, \leftarrow) &= \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma \max_{a'} Q^*(s', a') \right] \\ &= R(s, a, s') + \gamma \max_{a'} Q^*(s', a') = +1 + 0 = +1 \end{aligned} \tag{9}$$

Here we take into account the fact that $T(s, a, s') = 1$ because it is deterministic as mentioned in Exercise 3. Then we have $\max_{a'} Q^*(s', a') = 0$ because we are dealing with a terminal state, i.e. eating an apple. In this way, The dynamic programming procedure fill in the corresponding entry of the dynamic programming table Q^* with 1, but the other entries in the same row remain unknown and then go backwards. This operation can be shown in the drawn figure be in Table 3.

Table 3: The Dynamic Programming table Q^* .

left	forward	right
⋮	⋮	⋮
?	1	?
⋮	⋮	⋮

Now one step from the terminal state, as shown in Table 4, we need to apply Bellman equation again at this state and write

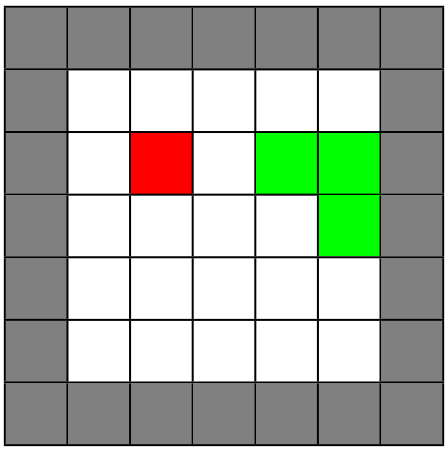
$$\begin{aligned} Q^*(s, \leftarrow) &= \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma \max_{a'} Q^*(s', a') \right] \\ &= R(s, a, s') + \gamma \max_{a'} Q^*(s', a') = 0 + \gamma \max_{a'} Q^*(s', a') = \gamma \max_{a'} [?, 1, ?] \end{aligned} \tag{10}$$

where $R(s, a, s') = 0$ in this case according to the given assumptions above regarding the reward signal. Then, we look at the corresponding row in Q^* table that we computed before, in this case values to the

next state, i.e. $Q^*(s', a') = [?, 1, ?]$ and we need to pick the action a corresponding to the largest element of the row. This corresponds precisely to $[?, 1, ?]$. Now it is impossible to get this largest element which correspond to π^* because the values of the other actions in the same row are unknown.

This issue highlight the fact that following the dynamic programming procedure we could not find an optimal policy to our task which thus motivates the use of more sophisticated methods such as policy iteration in Exercise 6.

Table 4: The snake in green one step from the terminal state.



d) What would be a solution to the issue found in c)?

For solving the problem, we can look at the relevant values in the dynamic programming table and notice that with the specific assumptions about R given in this task the largest value must be 1 because anyway the other values can be 0 or -1.

It turns out that, for the small version of Snake, this can be solved by using some **specific assumptions** about the reward signal. But we would like to avoid using such highly task specific ideas to solve general reinforcement learning problems if possible.

5.5 Policy Iteration

From the solution and discussion to the previous problem, we can conclude that the dynamic programming approach based on Equation1 seems unnecessarily convoluted, and depends much on the particular structure of the reward signal. Fortunately, it turns out *there exists a general approach which can be used to find the optimal policy π^* without taking into account any particular structures of the problem at hand: policy iteration.* Before discussing policy iteration, recall the Bellman optimality equation for the state value function given by

$$V^*(s) = \max_a \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V^*(s') \right] \tag{11}$$

Exercise 5 - Bellman optimality equation for the state-value function

a) Rewrite Equation 11 as an expectation using the notation $\mathbb{E}[\cdot]$, where $\mathbb{E}[\cdot]$ denotes expected value.

By rewritting the Equation in 11 as an expectation, we get

$$V^*(s) = \mathbb{E}_{s'}[f(x)] = \sum_{s'} p(x) f(x) = \max_a \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V^*(s') \right] \tag{12}$$

From this we can conclude that

$$V^*(s) = \max_a \mathbb{E}_{s'} \left[R(s, a, s') + \gamma V^*(s') \right] \tag{13}$$

b) Explain what Eq. 1 is saying. Your answer must reference all the various components and concepts of the equation $(Q^*, \Sigma, T, R, \gamma, \max, s, a, s')$.

First it is important to notice that $V(s)$ and $R(s)$ are quite different quantities; $R(s)$ is the "short term" reward for being in s, whereas $V(s)$ is the "long term" total reward from s onward.

The **optimal state-value** function $V^*(s)$ is the maximum over all actions a of the expected sum of the immediate reward and the future discounted rewards we'll get upon after action a, assuming that the agent chooses the optimal policy and follows it thereafter. Actually we should note that $\pi^*(s)$ gives the action a that attains the maximum in the "max" in Equation 11. And the transition function $T(s, a, s')$ is the probability that an action a from s leads to s', i.e., $P(s'|s, a)$.

Examining the equation in more detail, we have seen before that the summation term above can be rewritten

$$V^*(s) = \max_a \mathbb{E}_{s'} \left[R(s, a, s') + \gamma V^*(s') \right] \tag{14}$$

This is the expected sum of discounted rewards for starting in state s' , where s' is distributed according $T(s, a, s')$, which is the distribution over where we will end up after taking the first action $\pi(s)$ in the MDP from state s.

All optimal policies achieve the optimal value function, i.e.

$$V_{\pi^*}(s) = V^*(s) \tag{15}$$

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

Using the Bellman optimality equation for the state-value function in 11, we can write

$$\begin{aligned} V^*(s) &= \max_a \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V^*(s') \right] \\ \implies \pi^*(s) &= \arg\max_a V^*(s) = \arg\max_a \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V^*(s') \right] \end{aligned} \tag{16}$$

This is called **policy extraction**, since it gets the policy implied by the values.

d) Why is the relation between π^* and V^* not as simple as that between π^* and Q^* ? The answer should explain the qualitative difference between V^* and Q^* in your own words.

First we should remember that π^* is a policy, so it is a function that recommends an action for every state; its connection with s in particular is that it's an optimal policy when s is the starting state.

Because the optimal value $Q^*(s, a)$ is the expected utility starting out having taken action a from state s and thereafter acting optimally, So the relation between π^* and Q^* must be simple comparing to the relation between π^* and V^* which does not have an action a as input. In other word, actions are easier to select from Q-values than from V-values.

5.6 Policy Iteration algorithm

Policy improvement consists of two phases (we will now explicitly write the policy dependencies such as V^π instead of just V).

1. **Policy evaluation:** Given an arbitrary policy π (thus not necessarily optimal nor deterministic), the Bellman equation for V^π says that

$$V^\pi(s) = \sum_a \pi(a|s) \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V^\pi(s') \right] \tag{17}$$

   where *existence and uniqueness of V^π are guaranteed as long as $\gamma \in [0, 1)$ or eventual termination is guaranteed from all states under the policy π.* By turning the above Bellman equation into an update rule as

$$V_{k+1}(s) = \sum_a \pi(a|s) \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V_k(s') \right] \tag{18}$$

   it follows that $V_k = V^\pi$ is a fixed-point, as guaranteed by the Bellman equality.

   The sequence $\{V_k\}$ can be shown to converge in general toV^π as $k \to \infty$ under the same conditions that guarantee existence of V^π . In practice, the iterative policy evaluation is stopped once

$$\max_s |V_{k+1}(s) - V_k(s)| < \epsilon$$

   for some threshold $\epsilon > 0$.

2. **Policy improvement:** Suppose we have determined the value function $V^{\pi}(s)$ for an arbitrary deterministic policy π (so instead of $\pi(a|s)$ we simply have $\pi(s)$). We now ask ourselves: should we deterministically take some action $a \neq \pi(s)$ in state s? The answer can be found by considering $Q^{\pi}(s,a)$; if it holds that

$$Q^{\pi}(s,a) > Q^{\pi}(s,\pi(s))$$

then we should take action a instead of action $\pi(s)$.

More generally, according to the **policy improvement theorem**, we should in any state s take action $\arg\max_a Q^{\pi}(s,a)$ instead of action $\pi(s)$. Doing this over all states s will gives us a new, improved, policy $\tilde{\pi}(s)$. Due to the state value Bellman optimality equation 11, we are done if we ever find a new policy $\tilde{\pi}(s)$ satisfying $V^{\tilde{\pi}(s)} = V^{\pi}(s)$. Instead of involving the Q-function, one instead uses

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

The two phases above are run back and forth to give the policy improvement algorithm shown in Figure 4.

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
   $\Delta \leftarrow 0$
   Loop for each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s))\left[r + \gamma V(s')\right]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   old-action $\leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma V(s')\right]$
   If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Figure 4: Policy Iteration - Full Algorithm.

Bandit Algorithms: epsilon-Greedy Algorithm

As in most cases, there is a **trade-off** between two extremes. If we **explore** all the time, we'll collect a lot of data and waste resources in testing inferior ideas. However, if we only **exploit** the available option and never try new ideas, we would be left behind.

By exploring solutions offered by **Multi-armed Bandit Algorithms** we have two main advantages over the traditional methods:

- *Smoothly decrease exploration over time* instead of sudden jumps.
- *Focus resources on better options* and not keep evaluating inferior options during the life of the experiment.

Bandit Algorithms are algorithms that try to learn a rule of selecting a sequence of options that balance exploring available options and getting enough knowledge about each option and maximize profits by selecting the best option. In this way, we need to have enough feedback about each option to learn the best option. As a result, the **best strategy** would be to explore more at the beginning of the experiment until we know the best option and then start exploiting that option.

epsilon-Greedy Algorithm

The $\epsilon-$greedy algorithm is very simple and occurs in several areas of machine learning. One common use of $\epsilon-$greedy is in the so-called multi-armed bandit problem. In short, epsilon-greedy means pick the current best option ("greedy") most of the time, but pick a random option with a small (epsilon) probability sometimes.

There are many other algorithms for the multi-armed bandit problem. But epsilon-greedy is incredibly simple, and often works as well as, or even better than, more sophisticated algorithms such as UCB ("upper confidence bound") variations.

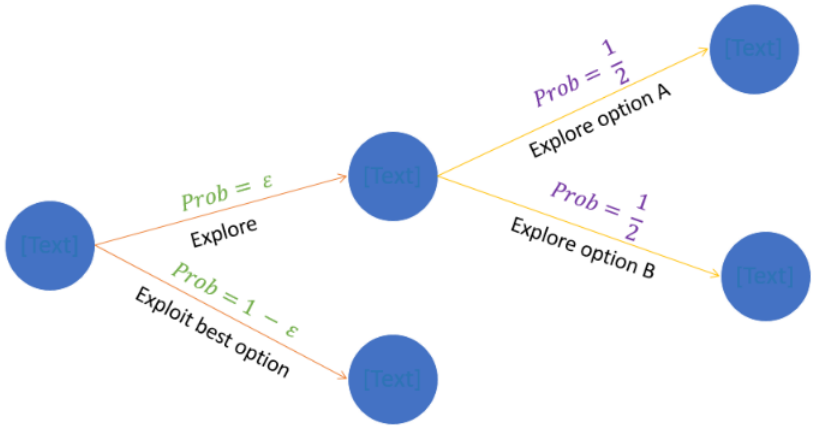


Figure 5: Epsilon-Greedy Algorithm.

- As ϵ increases $\rightarrow$ increase the exploration $\rightarrow$ increases the chance of picking options randomly instead of the best option.
- Algorithms with higher epsilon learn quicker but don't use that knowledge in exploiting the best option.

Exercise 6 - Implementation of the Policy Iteration code

Here in this task, in order to find an optimal policy $\pi^*(s)$, we need to fill in the blanks in the Maltlab code `policy_iteration` and run the policy iteration in `snake`.

a) Attach a printout of your policy iteration code.

In the following listing, we can see the relevant code to the algorithm Policy Iteration implemented in Matlab, as answer for the question (6a). To check that the code is correct, we tried running it with $\gamma = 0.5$ and $\epsilon = 1$, and verified that we got 6 policy iterations and 11 policy evaluations.

Listing 1: The implementation of the algorithm Policy Iteration in Question (6a).

```
1   % Policy evaluation.
2   while 1
3
4       Delta = 0;
5       for state_idx = 1 : nbr_states
6           % FILL IN POLICY EVALUATION WITHIN THIS LOOP.
7           v = values(state_idx);
8           nextState = next_state_idxs(state_idx, policy(state_idx));
9
10          if nextState == -1
11              values(state_idx) = (rewards.apple + 0);
12          elseif nextState == 0
13              values(state_idx) = (rewards.death + 0 );
14          else
15              values(state_idx) = (rewards.default + gamm * ...
16                                   values(nextState));
17          end
```

```
18              Delta = max(Delta, norm(v-values(state_idx)));
19          end
20
21          % Increase nbr_pol_eval counter.
22          nbr_pol_eval = nbr_pol_eval + 1;
23
24          % Check for policy evaluation termination.
25          if Delta < pol_eval_tol
26              break;
27          else
28              disp(['Delta: ', num2str(Delta)])
29          end
30      end
31
32      % Policy improvement.
33      policy_stable = true;
34      optimalV = zeros(1, 3);
35      for state_idx = 1 : nbr_states
36          % FILL IN POLICY IMPROVEMENT WITHIN THIS LOOP.
37          old_action = policy(state_idx);
38
39          for a = 1 : nbr_actions
40              nextState = next_state_idxs(state_idx,a);
41              if nextState == -1
42                  optimalV(a) = rewards.apple + 0;
43              elseif nextState == 0
44                  optimalV(a) = rewards.death + 0;
45              else
46                  optimalV(a) = rewards.default + gamm * values(nextState );
47              end
48          end
49
50          [~, idx] = max(optimalV);
51          policy(state_idx) = idx;
52
53          if old_action ~= policy(state_idx)
54              policy_stable = false;
55          end
56      end
```

b) What is the effect of γ. In this part, set $\epsilon = 1$?

By running the `snake` script and setting $\epsilon = 1$ we get the following results for the various values of γ:

- **When** $\gamma = 0$, we get the number of policy iterations: 2 and Number of policy evaluations: 4. The final snake playing agent act in cyclic movement and remains moving in the same 2×2 grid without growing or eating any apple. So the agent is playing bad, because in this case it ignores all the future rewards and get only the immediate one.
- **When** $\gamma = 1$, we don't get any results about the number of policy iterations and evaluations. The game didn't work and it get stuck in a loop without changing the value of Delta. In this case, we have undiscounted rewards or equally weighted values. So the algorithm doesn't converge and the optimal policy depends on the starting state. Actually, as mentioned above, existence and uniqueness of V^π are guaranteed as long as $\gamma \in [0, 1)$.
- **When** $\gamma = 0.95$, we get number of policy iterations: 6 and number of policy evaluations: 38. In this case the discount factor γ is close to 1 that makes the final snake agent acting best and keeping him eating apples forever. We are in the case of a far-sighted policy where the algorithm take into account all the future rewards.

c) What is the effect of ϵ in the policy evaluation, in this part, set $\gamma = 0.95$?

By running the `snake` script and setting $\gamma = 0.95$ we get the following results for the various values of ϵ:

- **When** $\epsilon = 10^{-1}$, we get number of policy iterations: 6 and Number of policy evaluations: 64.
- **When** $\epsilon = 10^{-2}$, we get number of policy iterations: 6 and number of policy evaluations: 115.
- **When** $\epsilon = 10^{-3}$, we get Number of policy iterations: 6 and Number of policy evaluations: 158.
- **When** $\epsilon = 10^{-4}$ we get number of policy iterations: 6 and number of policy evaluations: 204
- **When** $\epsilon = 10^{1}$, $\epsilon = 10^{2}$, $\epsilon = 10^{3}$, $\epsilon = 10^{4}$ we get the same number of policy iterations: 19 and same number of policy evaluations: 19.

In all these values for ϵ, the final snake agent is playing best and keeps eating apples forever. In practice, the iterative policy evaluation is stopped once

$$\max_s |V_{k+1}(s) - V_k(s)| < \epsilon$$

for some threshold $\epsilon > 0$.

What trends can be seen in relation between epsilon and the number of iterations?

Policy iteration and value iteration work well to find the optimal policy. They assume that the agent knows the transition function and the reward for all states in the environment, but this isn't always true. Q-learning goes about this through exploration and exploitation where exploration refers to taking an action that is not the best action and exploitation refers to using the knowledge that has already been built into the policy and picks the best action. This is similar to how simulated annealing makes bad decisions on purpose to avoid getting stuck in a local maximum. In the CMU Reinforcement Learning Simulator, this value is represented through "Epsilon" where epsilon refers to the probability that the best action is taken and (1-epsilon) is the probability where any of the other actions besides the best action is taken. Another important parameter is "Learning rate" which is a value between 0 and 1 and represents how much the new policy changes the existing policy.

Exercise 7 - Implementation of the tabular Q-updates

Here in this task, in order to find an optimal policy $\pi^*(s)$, we need to fill in the blanks in the Maltlab code `snake` to implement the **tabular Q-updates**.

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

In the code of `snake` a testing is performed by setting `test_agent = true`. This will set α and ϵ to 0, meaning no Q-value updates will occur and the policy is completely **greedy**. It will also load the automatically saved Q-values, and run the game.

In the following listing, we can see the 8 lines Q-update code implemented in Matlab, for answering the question (7a).

Listing 2: The implementation of the tabular Q-updates in Question (7a)).

```
1          % Check for termination.
2          if terminate
3
4              %
                 ------------------------------------------------------------------------
5              %
6              % FILL IN THE BLANKS TO IMPLEMENT THE Q-UPDATE BELOW (SEE
                   SLIDES)
7              %
8              % Maybe useful: alph, reward, Q_vals(state_idx, action) [recall
                   that
9              % we set future Q-values at terminal states equal to zero].
10             % Hint: Q(s,a) <-- (1 - alpha) * Q(s,a) + sample
11             % can be rewritten as Q(s,a) <-- Q(s,a) + alpha * (sample - Q(s
                   ,a))
12             sample = reward + 0;  % replace nan with something appropriate.
```

```
13              pred = Q_vals(state_idx, action);  % replace nan with something
                    appropriate.
14              td_err = sample - pred; % don't change this.
15              Q_vals(state_idx, action) = Q_vals(state_idx, action) + ...
16                                          alph * td_err; %(fill in blanks)
17
18          % ------------------------------------------------------------------
19          %
20          % FILL IN THE BLANKS TO IMPLEMENT THE Q-UPDATE BELOW (SEE SLIDES)
21          %
22          % Maybe useful: alph, max, reward, gamm, Q_vals(next_state_idx, :),
23          % Q_vals(state_idx, action)
24          % Hint: Q(s,a) <-- (1 - alpha) * Q(s,a) + sample
25          % can be rewritten as Q(s,a) <-- Q(s,a) + alpha * (sample - Q(s,a))
26
27          sample = reward + gamm .* max(Q_vals(next_state_idx, :)) % replace
                nan with something appropriate
28          pred = Q_vals(state_idx, action) % replace nan with something
                appropriate
29          td_err = sample - pred; % don't change this!
30          Q_vals(state_idx, action) = Q_vals(state_idx, action) + ...
31                                      alph * td_err; %(fill in blanks)
```

b) Explain 3 different attempts (parameter configurations) 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.

In order to train the snake playing agent, we consider the following 3 different attempts by using a positive scalar $m \neq 1$ and another scalar $n \neq 0$:

1. The rewards 'default','apple' and 'death' are multiplied with m. In other words, all the rewards are multiplied with the same positive scalar. The learning rate α and all other settings are kept unchanged.
2. The learning rate α is multiplied with m, yielding the learning rate $m\alpha$. The rewards 'default', 'apple' and 'death', and all other settings, are kept unchanged.
3. The scalar n is added to the rewards 'default', 'apple' and 'death'. In other words, all the rewards are shifted exactly the same amount. The learning rate α and all other settings are kept unchanged.

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?

After experimenting a lot with these parameters related to learning agent, we get the following final settings:

- Reward signal rewards = struct('default', -1, 'apple', 10, 'death', -1000);
- Discount factor in Q-learning $\gamma = 0.95$
- Learning rate in Q-learning (automatically set to zero during testing) $\alpha = 0.7$
- Random action selection probability in epsilon-greedy Q-learning (automatically set to zero during testing) $\epsilon = 0.01$

In addition, we play around also with these settings.

- Positive integer k: Update alpha every kth episode
  `alph_update_iter` = 100;
- At alpha update: new alpha = old alpha × `alph_update_factor`
  `alph_update_factor` = 0.95;
- , Positive integer k: Update eps every kth episode
  `eps_update_iter` = 100;

- At eps update: new eps = old eps × `eps_update_factor`
  `eps_update_factor` = 0.95;

Training results

After running the snake code during the training, the output result is

```
GAME OVER! SCORE:          1267
AVERAGE SCORE SO FAR:      482.9824
AVERAGE SCORE LAST 10:     1080.4
AVERAGE SCORE LAST 100:    1397.18
Successfully saved Q-values!
Done training agent!
You may try to set test_agent = true to test agent if you want
```

Test results - Score

An optimal snake player is one which keeps eating apples forever and reaching each apple as fast as possible. After running the final test, the test run never terminates and the score just keeps increasing.

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

The challenge is that the size of state space is extremely huge due to the fact that position of the snake affects the training results directly while its changing all the time. As we have seen above, basic Q-Learning keeps a table of all Q-values. In realistic situations, we cannot possibly learn about every single state. Thus, it is difficult to achieve optimal behavior because of the vast amount of states, i.e. because of the following issues:

- Too many states to visit them all in training.
- Too many states to hold the Q-tables in memory.

5.7 Q-learning with Linear Function Approximation - Full Snake Game

In this task, we look at the full version of the game. Tabular methods are more or less intractable for the full game. The bottleneck of course is the vast amount of states. As in most other areas of machine learning, this issue is tackled by introducing **features**, which summarize important concepts of the state.

This means that $Q(s, a)$ is approximated by some (possibly nonlinear) function $Q_w(s, a)$, parametrized by weights w. In this task we study linear function approximation. That is, $Q(s, a)$ is approximated as

$$Q(s, a) \approx Q_{\mathbf{w}}(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \cdots + w_n f_n(s, a) \tag{19}$$

where $\mathbf{w} = [w_1, w_2, \cdots, w_n]^T$ is the **weight vector** and ,$f_i(s, a)$ are state-action **feature functions**.

Valid assumptions

The rule of thumb is that the learning agent(s) are assumed to have the same kind of knowledge about the game state as would a human being with perfect vision. As humans, the input signal which we use to control the snake cleverly is only the **visual input from the game screens** that we observe.

In particular, the game screens are images with precisely three different colors; one background color (for unoccupied pixels), one snake/wall color and one color for the apple, as shown in Figure 1. In analogy, *the learning agent "sees" the game screen, which means that it gets as input the matrix representing the current game state.* In the code, **unoccupied pixels are represented by zeros, walls and the snake are represented by ones, and the apple pixel is represented by -1**.

Exercise 8 - Q weight update code

In this task, we train a reinforcement learning agent based on Q-learning using linear function approximation. For this we need to Fill in the blanks in `snake` to implement the Q weight updates, engineer state-action features $f_i(s, a)$ and implement these in the given function `extract_state_action_features`.

Once the agent have been trained for 5000 episodes, we should test its performance, where the code automatically saves our Q-weights. In the code of `snake` a testing is performed by setting `test_agent = true`. This will set α and ϵ to 0, meaning no Q-value updates will occur and the policy is completely **greedy**. It will also load the automatically saved Q-values, and run the game.

Adversarial initialization of the weights

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, We should set each entry w_i of $\mathbf{w}$ to 1 or -1, where the sign is set based on what we 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.

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

In the following listing, we can see the 8 lines Q weight update code implemented in Matlab, for answering the question (8a).

Listing 3: The implementation of the Q weight updates in Question (8a).

```
% Check for termination
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 + 0; % 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)

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

% 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)));% 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)
```

b) Write down at least 3 different attempts (parameter configurations and/or state-action feature functions)

As we have done in Exercise 7b, we follow the same strategy and consider the following 3 different attempts by using a positive scalar $m \neq 1$ and another scalar $n \neq 0$:

1. The rewards 'default','apple' and 'death' are multiplied with m. In other words, all the rewards are multiplied with the same positive scalar. The learning rate α and all other settings are kept unchanged.
2. The learning rate α is multiplied with m, yielding the learning rate $m\alpha$. The rewards 'default', 'apple' and 'death', and all other settings, are kept unchanged.
3. The scalar n is added to the rewards 'default', 'apple' and 'death'. In other words, all the rewards are shifted exactly the same amount. The learning rate α and all other settings are kept unchanged.

Final settings

For training a successful agent for the fulll Snake game based on Q-learning using linear function approximation, we should experiment a lot with these parameters related to learning agent, we get the following final settings:

- Reward signal rewards = struct('default', -1, 'apple', 100, 'death', -10);
- Discount factor in Q-learning $\gamma = 0.99$
- Learning rate in Q-learning (automatically set to zero during testing) $\alpha = 0.05$
- Random action selection probability in epsilon-greedy Q-learning (automatically set to zero during testing) $\epsilon = 0.5$

In addition, we play around also with these settings.

- Positive integer k: Update alpha every kth episode
  `alph_update_iter` = 100;
- At alpha update: new alpha = old alpha × `alph_update_factor`
  `alph_update_factor` = 0.5;
- , Positive integer k: Update eps every kth episode
  `eps_update_iter` = 100;
- At eps update: new eps = old eps × `eps_update_factor`
  `eps_update_factor` = 0.5;

c) Report what average test score you get after 100 game episodes with your final settings.

Training results

After running the snake code during the training, the output result is

```
GAME OVER! SCORE:         57
AVERAGE SCORE SO FAR:     35.8498
AVERAGE SCORE LAST 10:    47.1
AVERAGE SCORE LAST 100:   42.95
Successfully saved Q-weights!
Done training agent!
You may try to set test_agent = true to test agent if you want
```

Test results - Score

An optimal snake player is one which keeps eating apples forever and reaching each apple as fast as possible. After running the final test, the test run never terminates and the score just keeps increasing. After running the snake code during the training, the output result is

```
GAME OVER! SCORE:         74
AVERAGE SCORE SO FAR:     42.52
AVERAGE SCORE LAST 10:    38.6
AVERAGE SCORE LAST 100:   42.52
```

```
Final weights:
    -0.8112
    -9.8867

Mean score after 100 episodes: 42.52
... SUCCESS! You got average score at least 35 (feel free to try increasing
     this further if you want)
Done testing agent!
```

Features engineering

```
% Evaluate all the different actions (left, forward, right).
for action = 1 : 3

    % Feel free to uncomment below line of code if you find it useful.
    [next_head_loc, next_move_dir] = get_next_info(action, ...
                                                                          m

                                                                          he

    % Replace this to fit the number of state-action features per features
    % you choose (3 are used below), and of course replace the randn()
    % by something more sensible.

    % XXX Feature 1: distance to an apple with normalizing
    [apple_x, apple_y] = find(grid == -1);
    apple_loc = [apple_x, apple_y];

    %state_action_feats(1, action) = randn();
    state_action_feats(1, action) = norm(apple_loc - next_head_loc, 1)/100;

    % XXX Feature 2: Hitting the wall
    %state_action_feats(2, action) = randn();
     state_action_feats(2, action) = ...
                          (grid(next_head_loc(1), next_head_loc(2)) == 1);

     % XXX Feature 3: Circular movement of the snake
    %state_action_feats(3, action) = randn();
    % ... and so on ...
%     curr_dist = norm(next_head_loc - round([N / 2, N / 2]), 1);
%     prev_dist = norm(prev_head_loc - round([N / 2, N / 2]), 1);
%     state_action_feats(3, action) = curr_dist == prev_dist;
end
```

References

[1] R. Sutton, A.G. Barto: Reinforcement Learning, An Introduction. MIT Press, 2017. Draft available at http://incompleteideas.net/sutton/book/the-book-2nd.html (March 2019).

[2] Bishop, C. M.: Pattern Recognition and Machine Learning. Springer, 2006. Online version available at https://www.microsoft.com/en-us/research/people/cmbishop/#!prml-book (March 2019).

[3] T. Hastie, R. Tibshirani, J. Friedman: The elements of statistical learning, data mining, inference and prediction, 2nd edition, Springer, 2009, online version available at https://web.stanford.edu/~hastie/Papers/ESLII.pdf (March 2019).

[4] Andrew Ng, Stanford course CS229: Machine Learning. Available online at http://cs229.stanford.edu/syllabus.html

[5] Roger Grosse, UToronto course CSC 411 Fall 2018: Machine Learning and Data Mining. Available online at http://www.cs.toronto.edu/~rgrosse/courses/csc411_f18/

[6] Ian Goodfellow, Yoshua Bengio, Aaron Courville. *Deep Learning.* MIT Press, 2016, ISBN: 9780262035613. HTLM version available at https://www.deeplearningbook.org/ (March 2019).

[7] Aurélien Géron: Hands-On Machine Learning with Scikit-Learn and TensorFlow, Concepts, Tools, and Techniques to Build Intelligent Systems. O'Reilly Media, 2017, ISBN: 9781491962299.