FMAN-45 Machine Learning, Spring 2021
Assignment 4

January 27, 2021

This assignment consists of 7 questions with a maximum total score of 100. The first 4 questions constitute 50 points and are more theoretical. The last 3 questions constitute the remaining 50 points and are more programmatical/experimental. ***A score of*** 70% ***is needed to pass the assignment.*** *Write a detailed report. All solutions, plots and figures should be in* **one *pdf***. *It should be possible to understand all material presented in the report without running any code.* Submit your solutions and code using your individual Canvas account *as two files (a pdf and a single archive with all the code) by the deadline;* ***note that there will be no extensions***.

1 Introduction

Feel free to play the game Snake *at* `http://playsnake.org/` *to see how it works. The particular version of* Snake *used in this assignment is described below.*

In this assignment you will train reinforcement learning agents[1] to play the classic video game *Snake*. 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 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. The game ends when the snake hits itself or a wall; the final score is typically proportional to the number of apples "eaten" by the snake.

1.1 Details of the game

The following *Snake* variant is used in this assignment; see Figure 1 for some visualizations.

[1] Agents will sometimes be referred to as *players*.

- The game is played on a two-dimensional grid of size $M \times N$ pixels. In this assignment, $M = N = 30$.
- 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.
- The player will be controlling a connected[2] set of pixels known as the *snake*.
- 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.
- 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, and this is already implemented in the code you will use.*
- 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 (see above item). Again, also see Figure 1 for visualizations.
- 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.
- 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).
- At each time-step there is exactly one apple in the grid, occupying exactly one pixel.
- 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).
- The game ends as soon as the head of the snake moves into a wall or the body of the snake.

[2] 4-connectivity is considered here, i.e., two pixels are connected only if they are horizontal or vertical neighbours, *not* diagonal.

- 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.

2 Reinforcement learning for playing Snake

2.1 Tabular methods

Before considering the *Snake* game described so far, let's consider *small Snake* game: 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) and the snake has constant length 3. Everything else is as for the full game (see Section 1). In this small, special case, it is possible[3] to maintain a table which stores[4] 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 $\boldsymbol{Q}$, 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.**

> **Exercise 1 (10 points):**
>
> Derive the value of K above.
>
> *Hint: How many apple locations are possible for a given configuration of the snake? How many configurations of the snake are there, including its movement direction (or to phrase differently: how many configurations of the snake are there, if one also knows which pixels is the head)?*

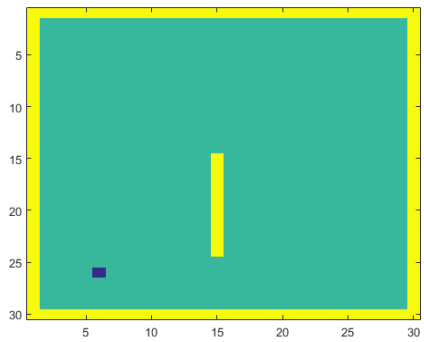
2.1.1 Bellman optimality equation for the Q-function

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

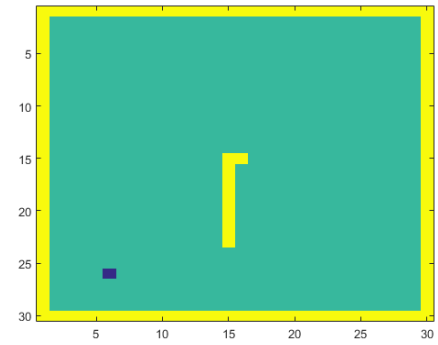
$$Q^*(s, a) = \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma \max_{a'} Q^*(s', a') \right]. \tag{1}$$

[3] In principle, this is possible even for much larger state-action spaces, but it quickly becomes practically difficult as the size of the state-action space increases.

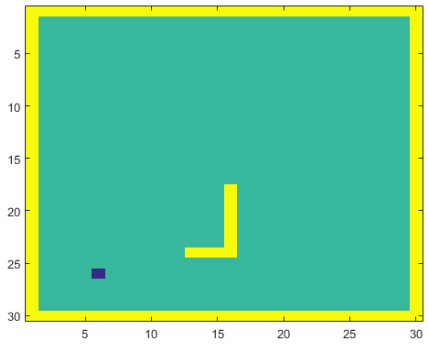
[4] Note that $Q(s, a)$ assumes an underlying policy $\pi(a|s)$ being followed; $Q(s, a)$ is then interpreted as the state-action value of taking action a in state s and thereafter following policy π.



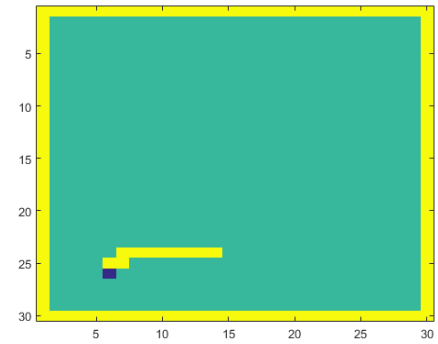
(a) Initial state. The length of the snake is 10 pixels. It forms a straight line of pixels, with its head located at the center of the grid, facing a random direction (N/E/S/W); in this case the direction is north (N). An apple is located at a random location, in this case in $(m, n) = (26, 6)$.



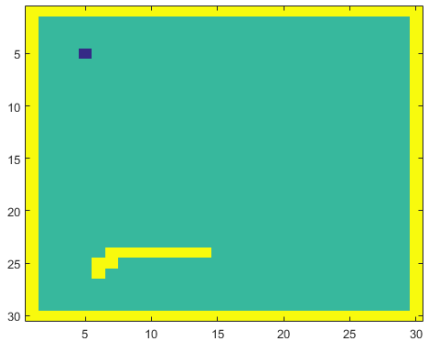
(b) At each time-step, the player must perform precisely one of three possible actions (movements): *left*, *forward*, or *right* (relative to the direction of the head). In this example, the player chooses to go right as their first action. The body of the snake moves along, keeping the snake intact.



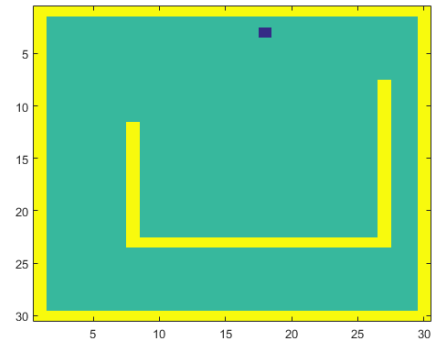
(c) After a few more time-steps, this is where the snake is located. Its direction is west. The initial apple has not yet been eaten.



(d) 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.



(e) Given the state shown in Figure 1d, 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.



(f) After several more time-steps, the snake has gotten significantly longer by eating apples. Each apple eaten gives one point. The game ends when the player chooses an action such that the head ends up in an occupied space (meaning the body of the snake or a border). The final score is the total number of apples eaten during the game.

Figure 1: A few situations, in chronological order, in a game of Snake on a 30×30-grid. The snake can only occupy any space in the interior 28×28 grid.

Once we have $Q^*(s,a)$, an optimal policy π^* is given by[5]

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

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

Exercise 2 (15 points):

a) (2 points) **Rewrite** eq. (1) as an expectation using the notation $\mathbb{E}[\cdot]$, where $\mathbb{E}[\cdot]$ denotes expected value. Use the same notation as in (1) as much as possible. The expression should still be a recursive relation – no infinite sums or similar.

b) (2 points) Explain what eq. (1) is saying in **one sentence**.

c) (3 points) Elaborate your answer in b) with a **bullet list** referencing *all* the various components and concepts of the equation $(Q^*, \Sigma, T, R, \gamma, \max, s, a, s')$.

d) (8 points) Explain what $T(s,a,s')$ in eq. (1) is for the small version of *Snake*. You should give values for $T(s,a,s')$, e.g. "$T(s,a,s') = 0.5$ if ... and $T(s,a,s') = 0.66$ if ..." (the correct values are likely different, though).

Now, the question is how we find such an optimal policy π^*. That is, how should an algorithm look like for computing (1)? We could try to implement the Bellman equation (1) directly through dynamic programming.[6] The dynamic programming procedure would first fill in all entries (i,j) of $\boldsymbol{Q}^*$ that correspond to terminal state-action pairs with zeros (technically this means that $\boldsymbol{Q}^*$ has $K+K_0$ instead of K rows, where K_0 denotes the number of terminal states), and then go "backwards" – i.e. further and further away from terminal state-action pairs – and find $\boldsymbol{Q}^*$ for all action-state pairs. Then, at a given state s, we would simply look at the corresponding row in $\boldsymbol{Q}^*$ and pick the action a corresponding to the largest element of the row - this corresponds precisely to $\pi^*(s) = \arg\max_a Q^*(s,a)$.

[5] Sometimes it's interesting to consider stochastic policies π, in which case we use the notation $\pi(a|s)$ rather than just $\pi(s)$.

[6] For a good reference on dynamic programming have a look at Ch4 in Sutton's book or `http://www.cs.princeton.edu/~wayne/kleinberg-tardos/`

Algorithm 1: Dynamic programming applied to Q-value table learning

Result: Table $\boldsymbol{Q}^*$ of Q^*-values
Initialize $\boldsymbol{Q}^*(s_T, a_T) = 0$ for each terminal state s_T (for terminal states ***only***) ;
For each terminal state S_T, identify each trajectory $(s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_T)$ such that $s_i = s_j$ if and only if $i = j$ (this constraint avoids loops);
For each such trajectory $(s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_T)$ do the below;
Set $n = T$;
while $n \geq 0$ **do**
    Using the current table $\boldsymbol{Q}^*$ calculate $\boldsymbol{Q}^*(s_n, a_n)$ according to (1);
    $n = n - 1$
end

There are some issues of following the above dynamic programming procedure (see Algorithm 1). To see this, we shall now consider using dynamic programming to find $\boldsymbol{Q}^*$ for the *small Snake* game.

Exercise 3 (15 points): Consider the *small Snake* game. Assume we have a reward signal which gives -1 if the snake dies, $+1$ if the snake eats an apple, and 0 otherwise. Also assume that $\gamma \in (0, 1)$. 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*. Before proceeding with the exercises below, convince yourself that an optimal agent to the truncated game will be optimal also for the normal game. Also make sure you know which are the terminal states of the truncated game.

a) (10 points) Consider the trajectory in Figure 2. Beginning from the terminal state s_T, perform the first two steps of the dynamic programming procedure in Algorithm 1. Explain what problem arises.

b) (5 points) What would be a solution to the issue found in a)? *Hint: What values in the dynamic programming table are relevant for finding the optimal solution?*

2.1.2 Policy iteration

As is hopefully apparent from the previous problem, the straightforward dynamic programming approach based on (1) seems unnecessarily convoluted, and depends much on the particular structure of the reward signal. Fortunately the optimal policy π^* can be found 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

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

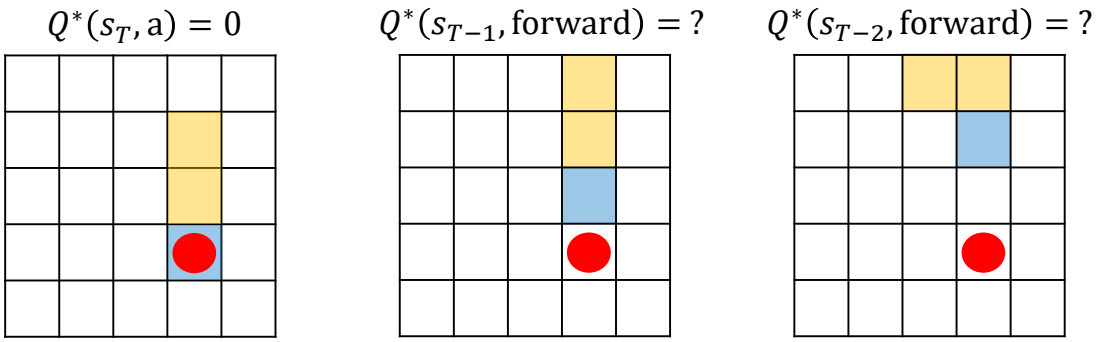


Figure 2: Figure associated to Exercise 3. The red circle is an apple, the blue square is the snake's head, and the two yellow squares are the snake's body. Figures from left to right show the snake at timestep T, $T-1$ and $T-2$.

Exercise 4 (10 points):

a) (2 points) **Rewrite** eq. (3) using the notation $\mathbb{E}[\cdot]$, where $\mathbb{E}[\cdot]$ denotes expected value. Use the same notation as in (3) as much as possible. The expression should still be a recursive relation – no infinite sums or similar.

b) (2 points) Explain what eq. (3) is saying in **one sentence**.

c) (2 points) In eq. (3), what is the purpose of the max-operator?

d) (2 points) 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^* (the answer should be something like $\pi^*(s) = \dots$)?

e) (2 points) Why is the relation between π^* and V^* not as simple as that between π^* and Q^*? You don't need to explain with formulas here; instead, the answer should explain the qualitative difference between V^* and Q^* in your own words.

Policy improvement consists 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

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

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. Note that

$\pi(a|s)$ may be stochastic above[7], which explains the outer summation over actions a and the conditional notation $a|s$. By turning the above Bellman equation into an update rule as

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

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 to V^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 (3), 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 $\tilde{\pi}(s) = \arg\max_a Q^\pi(s,a) = \arg\max_a T(s,a,s')[R(s,a,s') + \gamma V^\pi(s')]$ (convince yourself of the validity of this last equality - no motivation is however required in your report).

The two phases above are run back and forth to give the policy improvement algorithm; see Lecture 10 slides for details.

[7] We look at deterministic policies for the *Snake* game however.

Exercise 5 (15 points): Look at the code in `rl_project_to_students/small_snake_tabular_pol_iter/`. Begin by investigating `snake.m`. **Fill in the blanks to run policy iteration in order to find an optimal policy** $\pi^*(s)$ - look at `policy_iteration.m`. You don't need to think about the internal game state; nor will you need to think about how to implement the state representation, as this is already done for you. However, you of course need to understand how this state representation works in order to implement policy iteration. You are allowed to change settings **only where it states so** in `snake.m` and `policy_iteration.m`.

a) (3 points) Attach your policy iteration code. **To check that your code seems to be correct, try running it with** $\gamma = 0.5$ **and** $\epsilon = 1$**, and check that you get 6 policy iterations and 11 policy evaluations** (if you do not get this, make sure you use Matlab's built-in max-function if doing any max-operation). **Do not proceed until you get this result.**

b) (6 points) Investigate the effect of γ (in this part, set $\epsilon = 1$). Try $\gamma = 0$, $\gamma = 1$ and $\gamma = 0.95$.

- Provide a **table** showing the number of policy iterations and evaluations for each γ [3 points].
- How does the final snake playing agent act for the various γ? Is the final agent playing optimally? **Why do you get these results?** [3 points]

c) (6 points) Investigate the effect of the stopping tolerance ϵ in the policy evaluation (in this part, set $\gamma = 0.95$). Try ϵ ranging from 10^{-4} to 10^4 (with exponent step size 1, i.e. $10^{-4}, 10^{-3}, \ldots, 10^4$).

- Provide a **table** showing the number of policy iterations and evaluations for each ϵ [3 points].
- How does the final snake playing agent act for the various ϵ? Is the final agent playing optimally? **Why do you get these results?** [3 points]

2.1.3 Tabular Q-learning

One of the problems with the policy iteration approach described in Section 2.1.2 is that it requires full knowledge about the environment dynamics. That is, it assumes $T(s, a, s')$ and the corresponding rewards to be fully known. If the environment dynamics are unknown, trying things out and seeing what happens can be used to estimate the entries of $\boldsymbol{Q}^*$. There exist several approaches to such "trial-and-error" estimation; here we will be considering a particular algorithm known as *Q-learning*, which arguably is the most well-known algorithm in all of reinforcement learning.

Exercise 6 (15 points): Look at the code in `rl_project_to_students/small_snake_tabular_q/`. Begin by investigating `snake.m`. **Fill in the blanks to implement the tabular Q-updates (see lecture slides)**. You are allowed to change settings **only where it states so** in `snake.m`, so you can change the number of iterations, ε, α, γ, decrease learning rate, change the reward etc. **Write down and think about any observations you make as you try different settings** - you will be discussing this in b), c) below.

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

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

b) (3 points) Explain 3 different attempts (parameter configurations and scores - note that the 3 attempts don't need to be completely different, but change some settings in between them) you did in order to train a snake playing agent. **Comment, for each three of the attempts, if things worked well and why/why not.** [1 point per described attempt] *Hint: The most important things to experiment with are ε, α and the reward function.*

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

d) (3 points) Independently on how it went, explain why it can be difficult to achieve optimal behavior in this task within 5000 episodes.

2.2 Q-learning with linear function approximation

Now we are ready to have a look at the full version of the game as described in Section 1. If it is not immediately clear, you should convince yourself why 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 $Q_{\boldsymbol{w}}(s, a)$, parametrized by weights $\boldsymbol{w}$. In this task we study *linear* function approximation. That is, $Q(s, a)$ is approximated as

$$Q(s, a) \approx Q_{\boldsymbol{w}}(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \cdots + w_n f_n(s, a), \tag{4}$$

where $\boldsymbol{w} = [w_1, w_2, \ldots, w_n]^\top$ 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; see 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.**

In addition to the above rule of thumb, the following assumptions are valid (of which the two first are implemented already).

- The initial head location, i.e. the center of the screen, can be assumed to be *known*.
- The initial direction (N/E/S/W) can be assumed to be *known* (since the head location is known, and the snake is initially a straight line of pixels, the movement direction can be inferred).
- You don't need to think about the automatic game state updates, discussed in Section 1. The reason is that the computations that the agent perform when determining what action to take go very quickly. So even if there was a notion of automatic game state updates (in which the snake moves even without player actively telling it to do so), the agent would figure out its next action long before its "decision time" runs out. This is true only to some extent, of course, as you can imagine very complicated ways of figuring out what to do next. So as long as your agent is able to make a decision within, say, **1/20 second**, it's fine, although it likely will go much faster.

Exercise 7 (20 points): Look at the code in `rl_project_to_students/linear_q/`. Begin by investigating `snake.m`. It contains most things you need to run the game and to train a reinforcement learning agent based on Q-learning using linear function approximation. You are allowed to change settings **only where it states so** in `snake.m` and `extract_state_action_features.m`. You need to do a few things in order to train the agent:

- Fill in the blanks in `snake.m` to implement the Q weight updates (see lecture slides).
- Engineer state-action features $f_i(s, a)$ and implement these in the given function `extract_state_action_features.m`. **Note: Ensure all your features are within the [-1,1]-range** to ensure weights remain stable.
- Tune learning parameters in the main file `snake.m`, including reward signal, discount γ, learning rate α, greediness parameter ε, and so on (similar to what you did in the previous exercise). Experimentation is necessary to train a successful agent.
- We have a weight vector `weights`, which needs to be initialized. We will initialize the weights in a bad way. Doing this "adversarial" initialization will force the agent to actually learn something, rather than just work perfectly right away. Hence do as follows:
  - Set each entry w_i of $\boldsymbol{w}$ to 1 or -1, **where the sign is set based on what you believe is *bad*.** So if you believe that a good weight for $f_2(s, a)$ is positive, then in initialization set $w_2 = -1$. Clearly motivate for each w_i why you believe the sign is bad.

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

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

b) (6 points) Write down at least 3 different attempts (show the scores, parameter configurations and state-action feature functions - note that the 3 attempts don't need to be completely different, but change some settings in between them) you did in order to train a good snake playing agent. **Comment on if your attempts worked well and why/why not.** [2 points per described attempt] *Hint: The most important things to experiment with are state-action features, ε, α and the reward function.*

c) (11 points) Report average test score after 100 game episodes with your final settings; **you must get at least 30 on average and not get stuck in an infinite loop** (possible with only 2 state-action features, and it is possible to get over 100 points with only 3 state-action features). Also attach a **table** with your initial weight vector $\boldsymbol{w_0}$, your final weight vector $\boldsymbol{w}$ and associated final state-action feature functions f_i [8 points for this, if your snake agent got at least 30 points on average]. **Comment on why it works well or why it doesn't work so well**. [3 points for this commenting / discussion]