FMAN-45 Machine Learning, Fall 2016
Assignment 4: Reinforcement Learning for Playing *Snake*

November 15, 2016

Solve the problems and write down the solutions. If the assignment involves programming, download the code we provide and do the additional, required programming. 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 Moodle account *as two files (a pdf and a single archive with all the code) at* `http://moodle.maths.lth.se/course/` *by the deadline (note that there will be no extensions).*

1 Introduction

If you feel like there's too much text in this introduction, feel free to try playing the game Snake *yourself to see how it works, e.g. at* `http://playsnake.org/`*. Of course, you'll probably be interested in some specifics of the particular version of* Snake *game I have in mind 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]Agents will sometimes be referred to as *players*.

1.1 Details of the game

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

- 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.
- 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; we denote this a *time-step*). 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.
- 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.
- In reality, the state of the game is automatically updated at some fixed frequency (i.e., the duration for which the game is "paused" is fixed). 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 will be precisely one apple in the grid, occupying precisely 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).

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

- The game ends as soon as the head of the snake moves into a wall or the body of the snake.
- 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 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 (50 points): Derive a value for K above (30 points). Using appropriate symmetries, is it possible to reduce K, and if so, by how much (20 points)? *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?*

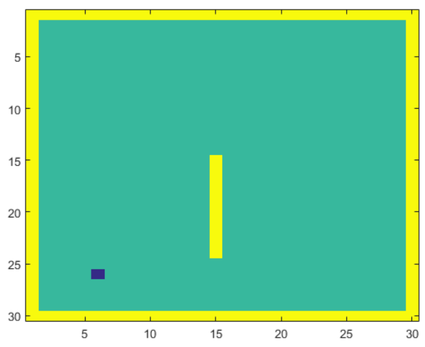
2.1.1 Bellman optimality equation for the Q-function

Assuming we have access to a table containing $Q(s, a)$ for each possible state-action pair (s, a), 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

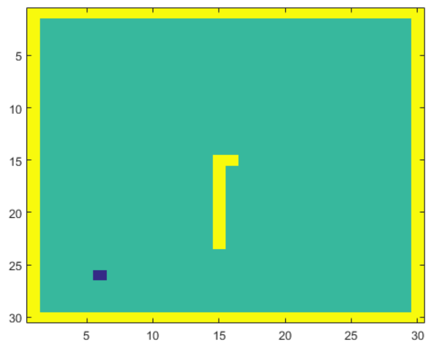
$$Q^*(s, a) = \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma \max_{a'} Q^*(s', a') \right] \tag{1}$$

[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.

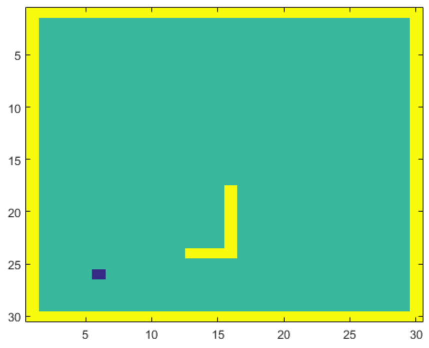
[4]Note that $Q(s, a)$ assumes an underlying policy $\pi(a|s)$ being followed; $Q(s, a)$ is then interpreted as the state-action value of taking action a in state s and thereafter following policy π.



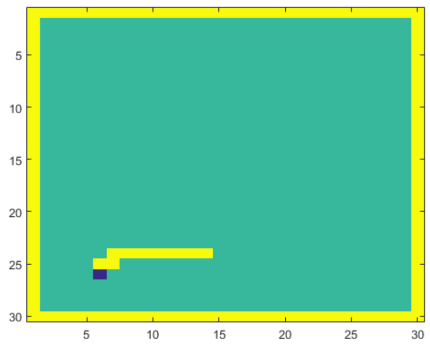
(a) Initial state. The length of the snake is 10 pixels. It forms a straight line of pixels, with its head located at the center of the grid, facing a random direction (N/E/S/W); in this case the direction is north (N). An apple is located at a random location, in this case in $(m, n) = (26, 6)$.



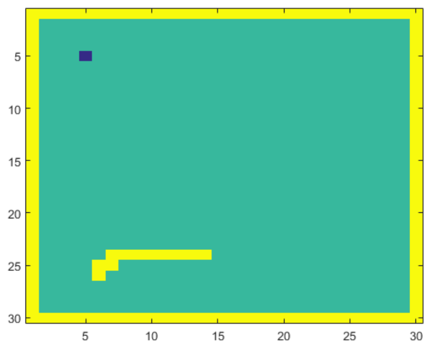
(b) At each time-step, the player must perform precisely one of three possible actions (movements): *left*, *forward*, or *right* (relative to the direction of the head). In this example, the player chooses to go right as their first action. The body of the snake moves along, keeping the snake intact.



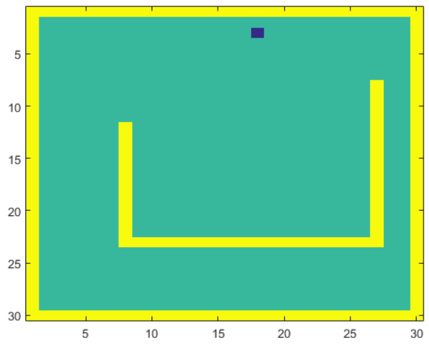
(c) After a few more time-steps, this is where the snake is located. Its direction is west. The initial apple has not yet been eaten.



(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.

where s' denotes state at the next time-step; a' denotes action at the next time-step; $T(s, a, s')$ is the *transition probability*[5] of landing in state s' by taking action a in state s; $R(s, a, s')$ is the reward obtained by the corresponding transition from s to s' by taking action a; and $\gamma \in [0, 1]$ is the *discount factor*. Once we have $Q^*(s, a)$, an optimal policy π^* is given by[6] $\pi^*(s) = a^* = \arg\max_a Q^*(s, a)$. 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 (50 points): Explain in your own words what (1) is saying and briefly comment (informally, no math needed!) on why it makes sense (25 points). Discuss the effect γ has on the policy $\pi^*(s)$, by considering the following cases (25 points): i) $\gamma = 0$, ii) $\gamma = 1$ and iii) $\gamma \in (0, 1)$.

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

Exercise 3 (25 points): Explain what $T(s, a, s')$ in (1) is for the small version of *Snake*. *Hint: It may help you to go back and think about what the K states are from Exercise 1. Also, there are two cases for a state-action pair (s, a) which must be considered.*

Exercise 4 (50 points): Assume we have a reward signal which gives -1 it the snake dies, $+1$ if the snake eats an apple, and 0 otherwise. Also assume that $\gamma = 1$. Then there are some major problems with straightforwardly following the above procedure for the small version of *Snake*. Explain these, and suggest some ways to make it work. *Hint: What would happen if the game instead was such that it ends as soon as the snake eats the first apple (yielding a total score of $+1$ for the game)?*

2.1.2 Policy iteration

As is hopefully apparent from your solution and discussion to the previous problem, the straightforward dynamic programming approach based on (1) seems unnecessarily convoluted, and depends much on the particular structure of the *Snake* game. 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

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

[5]Thus note that $\sum_{s'} T(s, a, s') = 1$.

[6]Sometimes it's interesting to consider stochastic policies π, in which case we use the notation $\pi(a|s)$ rather than just $\pi(s)$. You'll see this later in the assignment.

Exercise 5 (50 points): Explain in your own words what (2) is saying and briefly comment (informally, no math needed!) on why it makes sense (25 points). For $Q^*(s,a)$, we have $\pi^*(s) = \arg\max Q^*(s,a)$. Explain why such a direct connection between the optimal policy π^* and optimal value function $V^*(s)$ does not exist (25 points).

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, 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 is clear 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 $\pi'(s)$. Due to the state value Bellman optimality equation (2), we are done if we ever find a new policy $\pi'(s)$ satisfying $V^{\pi'}(s) = V^\pi(s)$. Instead of involving the Q-function, one instead uses $\pi'(s) = \arg\max_a Q^\pi(s,a) = \arg\max_a T(s,a,s')[R(s,a,s') + \gamma V^\pi(s')]$ (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 slides for details.

Exercise 6 (50 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. Attach a printout of your policy iteration code. Also, in the code (`snake.m`) you will find two questions: i) What is the effect of γ? Try $\gamma = 0$, $\gamma = 1$ and $\gamma \in (0, 1)$ and explain what happens (in this part, set $\epsilon = 1$). ii) What is the effect of ϵ in the policy evaluation? Try ϵ ranging from 10^{-4} to 10^4 (with exponent step size 1, i.e. $10^{-4}, 10^{-3}, \ldots, 10^4$) and explain what happens (in this part, set $\gamma = 0.95$).

2.1.3 Tabular Q-learning

A problem with the policy iteration approach described in Section 2.1.2 is - other than that is impractical for any real problems with large/continuous state-action spaces - 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, *Monte-Carlo estimation* can be used to estimate the entries of $\boldsymbol{Q}^*$. There exist several approaches to such Monte-Carlo 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 7 (50 points): Look at the code in `rl_project_to_students/small_snake_tabular_q/`. Begin by investigating `snake.m`. You will not need to implement anything in this task (but you of course need to understand the code to understand what is going on); just play around with the various parameters in `snake.m`. Write down any observations you make as you try different settings; be extra clear about the final settings you used. Once you have trained the agent, save your Q-values (see `save` and `load` in Matlab), set α and ε to 0 (meaning no weight updates will occur and the policy is completely greedy) and run the game. Are you able to train a good, or even optimal, policy for the small *Snake* game via tabular Q-learning? By good is meant a snake which keeps eating apples although it doesn't necessarily reach them in the shortest amount of steps (so a good snake player doesn't end up in any infinite loops); by optimal is meant a snake which keeps eating apples and reaching each apple as fast as possible.

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

$$Q(s,a) \approx Q_{\boldsymbol{w}}(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \cdots + f_n(s,a), \tag{3}$$

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 8 (100 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 need to do two things in order to train the agent:

- Engineer state-action features $f_i(s, a)$ and implement these in the given function `extract_state_action_features.m`. Currently these features are random; each $f_i(s, a)$ is imply set to a value drawn from $\mathcal{N}(0, 1)$. Report what state-action features you end up using.
- Tune learning parameters in the main file `snake.m`, including reward signal, discount γ, learning rate α, greediness parameter ε (for ε-greedy Q-learning - see slides), and so on. I've set some values here to get you started, but don't take that to mean that they are particularly good - or bad - it depends a lot on what your feature functions $f_i(s, a)$ end up being. Experimentation (and of course reasoning to some extent) is necessary to train a successful agent. Report what settings you end up using.

Once you have trained the agent, save your weight parameters (see `save` and `load` in Matlab), set α and ε to 0 (meaning no weight updates will occur and the policy is completely greedy) and run the game for several episodes using your parameters. Report what average score you get after 100 game episodes. As a small guideline, it should be possible to get a score of ≈ 40 using only two very simple features, but higher scores are of course possible using additional/other features per state-action pair. Adding an additional particular feature, for a total of three features, makes it possible to get an average score of ≈ 100. This extra feature is not as simple as the two features which yields ≈ 40 in score, but it is also not overly complicated. You are not expected to find this feature (you may perhaps find more and/or different and/or better features), but this example is merely to emphasize that it is possible to train a successful *Snake* playing agent using very few, relevant features.