Assignment 4 - Machine Learning

Andreas Timürtas

May 2021

Exercise 1

Our board with size 7x7 have an interior space with size 5x5 where the snake can be located. This gives us a total of 25 squares. Our snake has a fixed length of 3, meaning that the snake is always located on three squares, and we always have one apple on out board. This means that given a current location of the snake the apple can be in 22 places. Next we want to derive how many combinations that the snake can be located on. We begin by noticing that with the fixed length of 3 the middle square of the snake can be located in 25 different places. Depending on which square that is the front and back of the snake can be located in ether two, three or four squares. In figure 1 we see the board nad the squares colored. The black squares represent the wall. If the middle square of the snake is located on the green squares then the front and back squares of the snake have two possibilities to be located on. The possible combinations if the middle square is on a green square can then be calculated to be $4 \cdot 2 \cdot 1 = 8$ combinations. If the middle square is however located on a yellow square the head and tail can then be located on three squares. This gives us $12 \cdot 3 \cdot 2 = 72$ combinations. Lastly if the middle square is located on a red square then the head and tail can be located on four different squares. This gives us $9 \cdot 4 \cdot 3 = 108$ combinations. This gives us a total of $8 + 72 + 108 = 188$ combinations that the snake can be located on. With every combinations the apple can be located on 22 squares, meaning that the total number of states is equal to $188 * 22 = 4136$ combinations. The value of K is then equal to 4136.

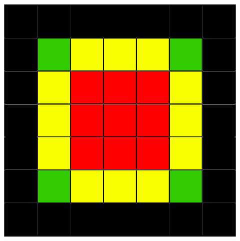

Figure 1: Our 7x7 board. The black squares are the walls, the green are the edges with only two adjacent playable squares, the yellow squares with three adjacent playable squares and red squares with four adjacent playable squares.

Exercise 2

The Bellman optimality equation is given by

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

This equation can be rewrite as an expectation with the form

$$Q^*(s,a) = \mathbb{E}\left[R(s,a,s') + \gamma \max_{a'} Q^*(s',a') \mid s,a \right] \quad (2)$$

The equation is calculating the expected utility if we are in state s taking action a and thereafter following the optimal policy. In more detailed explanation of the different components in 1 can be made.

- s is the current state the agent is in. In our snake game this is the possition of the apple and the location of the player on the board.
- a is the action the player is taking. This can either be *forward*, *left* and *right* for our snake game.
- s' is the next state the player ends up in.
- $Q^*(s,a)$ is the expected reward when if we are in state s and taking action a followed by using the optimal policy.
- $T(s,a,s')$ is the state-transition probability to start in state s, taking action a and ending on state s'.
- $R(s,a,s')$ is the reward for taking action a at state s and ending at state s'.
- γ is the discount rate and is a parameter with value between 0 and 1. This reduces the importance of future reward. Higher values of γ means more importance in future rewards and making the player more farsighted while lower values of γ makes the player focused on closer rewards.
- $\max_{a'}$ is an operation that returns the highest value after testing every action a'. In equation 1 the operation $\max_{a'}$ returns the highest expected reward in state s' after taking an action and following the optimal policy.
- The sum $\sum_{s'}$ sums over all the possible next states s'. This means that we calculation the possibility of ending on state s' ($T(s,a,s')$) and multiplying the immediate reward ($R(s,a,s')$) and the future reward ($\gamma \max_{a'} Q^*(s',a')$) and sums for all ending states s'. All this results in expected reward at state s taking action a ($Q^*(s,a)$).

In our small version of the snake game the movement of the player is fully deterministic. This means that the action is taking the player to the desired state, i.e. the player moves forward when the *forward* action is made. The state-transition probabilities $T(s,a,s')$ is then equal to 1 if the next state s' is reached if action a is taken from state s. $T(s,a,s')$ is then equal to 0 if the state s' is not reached with the action a at state s. One exception for this is when the snake is about to eat the apple. Assume that we are at a state s and taking the action a which will lead to that the snake eats the apple. The next state s' must have an apple on the board but the previous apple has been eaten so the game need to respawn a new apple. In our small snake game the apple can spawn in 22 different squares, each with the same probability. This means that $T(s,a,s') = \frac{1}{22}$ if action a on state s leads to the current apple being eaten and that the location of the snake in state s' is reached by taking action a on state s.

Exercise 3

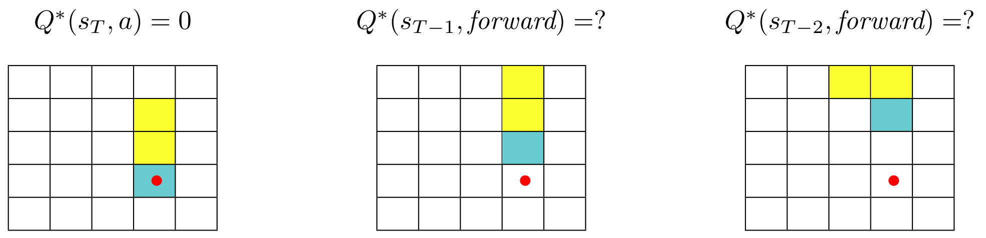
$Q^*(s_T, a) = 0$
$Q^*(s_{T-1}, \textit{forward}) = ?$
$Q^*(s_{T-2}, \textit{forward}) = ?$

Figure 2: The figure illustrate a trajectory at timesteps T, $T-1$ and $T-2$. The red dot is the apple, the blue square the head of the snake and the two yellow squares the tail of the snake.

In the figure above we have a scenario where the trajectory for three consecutive timesteps is illustrated. In the exercise we are given a suggested algorithm to calculate the Q^*-values and return the corresponding $\mathbf{Q}^*$ table. The algorithm starts by setting the Q^*-values equal to 0 for each terminal state s_T. In the algorithm we have a *while* loop that runs while $n \geq 0$, where n starts at the terminal timestep T and decreases by 1 for every turn in the loop. In the loop we are calculating $\mathbf{Q}^*(s_n, a_n)$ using the current $\mathbf{Q}^*$ table and equation 1.

We will show that this algorithm has a fatal problem by considering the trajectory in figure 2. We remember that this game is deterministic and can therefore ignore $T(s, a, s')$ and the sum. The modified equation for calculating the Q^*-values is therefore

$$Q^*(s, a) = R(s, a, s') + \gamma \max_{a'} Q^*(s', a') \tag{3}$$

We start by calculating equation 3 on timestep T. The rewards for eating an apple is +1 while dying has a reward of -1, default is set to 0. The game ends when the snake eats the apple and therefore the Q^*-value is set to 0 for the next (non-existing) state.

$$Q^*(s_T, a) = R(s_T, a, s') + \gamma \max_{a'} Q^*(s', a') = R(s_T, a, s') + 0 = 1 \tag{4}$$

For the previous timestep in our trajectory we get the Q^*-value

$$Q^*(s_{T-1}, \textit{forward}) = R(s_{T-1}, \textit{forward}, s_T) + \gamma \max_{a'} Q^*(s_T, a') = 0 + \gamma * 1 = \gamma \tag{5}$$

The algorithm has not yet had any problems but for the calculation we will see now the problem will show itself.

$$Q^*(s_{T-2}, \textit{forward}) = R(s_{T-2}, \textit{forward}, s_{T-1}) + \gamma \max_{a'} Q^*(s_{T-1}, a') = 0 + \max \left\{ \begin{array}{l} Q^*(s_{T-1}, \textit{left}) = ?, \\ Q^*(s_{T-1}, \textit{forward}) = \gamma, \\ Q^*(s_{T-1}, \textit{right}) = ? \end{array} \right\} \tag{6}$$

Here we see the problem the algorithm has, the Q^*-values for the actions *left* and *right* at state s_{T-1} has yet not been calculated and stored in the $\mathbf{Q}^*$ table. Therefore the algorithm will raise an error.

A solution for the problem in the algorithm could be to first check if all the Q^*-values for the next timestep is not NaNs. If that is the case then the algorithm proceeds as usual but if not all relevant Q^*-values have been calculated we ignore them and take the maximum of the available values from the table. This mean that we would not get the Q^*-values directly but instead temporary Q-values. When the algorithm then runs for all trajectories the Q-values would converge towards the desiered Q^*-values.

Exercise 4

The Bellman optimality equation for the state value function is given by

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

This equation can be written as an expectation in the following way

$$V^*(s) = \mathbb{E}\Big[R(s, a, s') + \gamma V^*(s') | s, a \Big]. \tag{8}$$

This function is saying what utility we can expect if the player is on state s and following the optimal policy. The $\max_a$ operator is necessary because we want to obtain the optimal reward which is given by taking the action a that maximizes the sum.

The optimal policy $\pi^*(s)$ have a relation with $Q^*(s, a)$ through the formula $\pi^*(s) = \arg\max_a Q^*(s, a)$. A relation between $\pi^*(s)$ and $V^*(s)$ also exist with the formula

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

This relation is notably not as simple as the relation between $\pi^*(s)$ and $Q^*(s, a)$. This is because the actions are included in the Q-values and we can then simply choose the action that maximizes the utility for a given state. In the other hand actions are not included in the V-values and we need therefore to check the utility for the next states for each action.

Exercise 5

In this exercise we implemented a policy improvement routine. In the listings below the code used to implement the policy evaluation and policy improvement. When we run the routine on our small snake game with $\gamma = 0.5$ and $\epsilon = 1$ we got 6 policy iterations and 11 policy evaluations, which we expected.

In table 1 you can see the number of policy iterations and evaluations the routine performed for different values of γ. When γ is equal to 0 the routine converges quickly but the final player performed bad where the snake only went in a circle. This is because with $\gamma = 0$ the player only aims for the immediate reward and can not "see" the apples that are located two or more steps further from its location. The second playing agent with γ equal to 0.95 takes more iterations and evaluations then the previous agent but performs optimal. The agent keeps eating apples and is doing it as fast as possible and is therefore optimal. This agent performs good because the snake considers future rewards at the same time as immediate rewards is still more valuable, which makes the snake want to eat the apple as fast as possible. The final agent with γ equal to 1 does never end the policy evaluation step and keeps running until we ended it. This is because the agent considers all the future reward it can get and therefore every route to the apple maximizes the utility. The *Delta* error then only changes when we first find the apple but stays the same because we can not find any routes that increases the utility.

In table 2 the result after we performed the policy routine with different values of ϵ. With lower values of ϵ the policy evaluation needs to run more times to reach the threshold. This will make the state values V_k to converge to the state values if we follow the current policy V^π more precise. The result of this is that the policy improvement does not need to as many iterations to stabilise. On the other hand when ϵ is higher the threshold is meet immediately which makes the state values V_k less similar to V^π. The result of this is that the policy improvement needs more iterations to stabilize. Notice that the number of policy evaluations and iterations is the same, this is because for every policy improvement there is a policy evaluation before. Interesting is that all the different tests performed optimal, which may be because even though the threshold is high enough to always exit the loop the policy is not yet stable and therefore some evaluations are made, possibly enough to reach an optimal policy.

	$\gamma = 0$	$\gamma = 0.95$	$\gamma = 1$
Iterations	2	6	-
Evaluations	4	38	∞

Table 1: Number of policy iterations and evaluations for different values of γ, ϵ is set to be equal to 1.

	$\epsilon = 10^{-4}$	$\epsilon = 10^{-3}$	$\epsilon = 10^{-2}$	$\epsilon = 10^{-1}$	$\epsilon = 10^{0}$	$\epsilon = 10^{1}$	$\epsilon = 10^{2}$	$\epsilon = 10^{3}$	$\epsilon = 10^{4}$
Iterations	6	6	6	6	6	19	19	19	19
Evaluations	204	158	115	64	38	19	19	19	19
Optimal	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Table 2: Number of policy iterations and evaluations for different values of ϵ, γ is set to be equal to 0.95.

Listing 1: Policy evaluation

```
Delta = 0;
for state_idx = 1 : nbr_states
    temp = next_state_idxs(state_idx, policy(state_idx));

    if temp > 0
        new_value = rewards.default + gamm*values(temp);
    elseif temp == 0
        new_value = rewards.death;
    else
        new_value = rewards.apple;
    end

    Delta = max([abs(new_value-values(state_idx)) Delta]);
    values(state_idx) = new_value;
end
```

Listing 2: Policy improvement

```
policy_stable = true;
for state_idx = 1 : nbr_states
    temp = next_state_idxs(state_idx, :);

    q_temp = zeros(1,3);
    for i = 1:nbr_actions
        if temp(i) > 0
            q_temp(i) = rewards.default + gamm*values(temp(i));
            elseif temp(i) == 0
            q_temp(i) = rewards.death;
        else
            q_temp(i) = rewards.apple;
        end
    end
    [~, argmax] = max(q_temp);
    if argmax ~= policy(state_idx)
        policy_stable = false;
    end
    policy(state_idx) = argmax;
end
```

Exercise 6

In the listings below the code used to implement our Q-update can be seen, both for the non-terminal and terminal updates.

The original snake playing agent had a configuration with $\gamma = 0.9$, $\alpha = 0.001$, $\epsilon = 0.001$ and the rewards 0, 1 and -10 for default, apple respectively death. This parameter configuration obtained a test score of 6 before it got stuck in a loop. To improve this we first changed the default reward from 0 to -0.01. The idea here was to give the player agent an incentive to find the fastest route to the apple and penalize the agent for unnecessary actions. This attempt resulted in an improvement of the score and obtained 78 in the test until it got stuck in a loop. The second idea we had was to lower ϵ from 0.001 to 0.01 (the changed reward was kept). The motivation for this is that we thought that the snake may have a too small probability to take a random step. If this was true the agent would not try different routes and may miss better actions that would improve the performance. This however turned out to not be true and the score instead decreased to 9 (again stuck in a loop). We therefore ignored the change on ϵ and set it back to 0.001. For the third attempt we had a thought that it may not be the probability to try different actions but instead that new routes did not affect the policy enough. We therefore changed the learning rate α from 0.001 to 0.01. This would mean that the Q-values would change more rapidly. The change improved the performance significantly and the test run never terminated but instead the score kept increasing.

The final settings was therefore; discount factor $\gamma = 0.9$, learning rate $\alpha = 0.01$ and random action selection probability $\epsilon = 0.001$ with the rewards -0.01, 1 and -10 for default, apple respectively death cases. As mentioned above this configuration never terminated and instead kept increasing the score. Nevertheless the player agent is not optimal because the snake does not take the fasted route to the apple and instead sometimes takes some detours.

The difficulty with trying to achieve optimal behavior is this small snake game within 5000 episodes is because, as we derived in exercise 1, the number of non-terminal states is 4136. This means that within 5000 episodes we need to explore update almost as many states. The result is that the optimal behavior may not have been explored enough to be found. Another difficulty is that we have 6 different parameters that all change the outcome and influence each other. The possibilities are large and by trying to tune them with a trial-and-error method would mean a lot of attempts.

Listing 3: Q-update code when on non-terminal update

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

Listing 4: Q-update code when on terminal update

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

Exercise 7

The implemented Q weight update code for both the non-terminal and terminal updates can be seen in the listings below.

In our first attempt we implemented three features. The first state action feature we implemented was a value that described the normalized Manhattan distance if the action was performed (the next head location). The initial weight for this feature was set to be equal to 1 because a higher value on this feature is negative and therefore the weight will probably get updated to be negative. The second feature could have the values -1, 0 or 1 depending if the next head location would be on a apple, default respectively wall/body square. The idea here was that this feature would give the agent the behaviour to avoid obstacle and eat apples when possible. The initial weight here was set to be equal to 1 because a positive value on the feature is an bad indication of the state (a value of 1 means that we are hitting a wall). The third and final feature for this attempt was binary where 1 representing that the next head location was located closer to the apple. This was implemented in a way that if the next head location was closer either column wise or row wise the feature would be set to 1. Otherwise the feature was set to 0. The initial weight for the third feature was set to be equal to -1 because it is positive for the agent if the feature is 1 and would therefore be *bad* if the weight was set to -1. This attempt gave after 5000 episodes of updating an average test score of 30.85 on 100 episodes. This is a descent score for our first attempt and the player agent behaved reasonable, finding the fastest route on many occasions and avoiding obstacles on its path. A problem though was that the snake could make a poor decision and create an inescapable loop with the body. This was the main cause of death for our agents.

The problem with the inescapable loops the snake created was tried to be tackled in the second attempt. Here we implemented a fourth binary feature. The value of the feature would be set to 1 if the snake forms a loop if the action was chosen, otherwise 0. This was done by inverting the grid and using the MATLAB function *regionprops()* to get how any regions existed on the grid if the action was made. This in theory would give the agent insight if an action would form a loop which would be negative. As we saw when we trained the agent and tested it this was not the case. The average test score on 100 episodes increased to 33.61 but the player agents would still create inescapable loops. This was still the main cause of death. The problem here was that in many situations either of the three possible actions would result in creating a new region. Imagine a long snake that takes three consecutive *right* action (after some *forward* actions between). The snake would now be headed towards the tail and on the state right before contact the snake have three non-terminal actions, *left* or *right*. Either of the actions results in creating a new region, even though a left turn would probably be better. We tried to solve this problem by adding a new value to the feature, -1. When the snake was about to form a new region the code would check if the next head location would be further away from the center of the loop then the current head location. If this was true the feature would be set to 1, otherwise the feature would be set to -1. The initial weight was set to -1 because a feature value of 1 would mean that the snake avoided entering a loop, which would be positive for the agent. This new feature performed unfortunately even worse then the previous feature and resulted in an average test score of 21.68. This may be because the loop can remain on the grid for several steps and the feature may "confuse" the agent with the information to avoid the loop. If we could find a better way to implement this feature then the score may increase. The problem remains that the main cause of death is still inescapable loops. Another method that may work is to try to anticipate if the snake will hit the body if it continues on the same direction. This could be done by calculating the distance to the body and the remaining tail on that square. This will be a future attempt.

In the third attempt we returned to the first three features and realised that the first feature probably was redundant due to the second feature. We therefore removed the first and only used two features. The idea here was that the first feature did not give the snake any relevant information but instead could "confuse" the snake. We also tried different α, ϵ and reward values with no significant change in the score. With the two remaining features the average test score on 100 episodes was 35.08.

The third attempt was chosen for our the final configuration. The settings for this model was $\gamma = 0.95$, $\alpha = 0.03$ and $\epsilon = 0.15$ with the rewards 0, 1 and -1 for default, apple and death. With an average score of 35.08 after 100 episode the agent performs reasonable (even though the loop problem is something we want to solve). In table 3 the initial weights with the final weights for the two features are represented. As we can see the final weights has diverged from the initial weights which indicates that the agent have learned something. The reason why the agent works well is because the two features both makes the snake move towards the apple at the same time avoiding obstacles. This is enough for this simple game, especially when the snake have not grown so much in the early game. The problem with the loops though occurs when the snakes moves towards the apple, unaware that it is entering an inescapable loop. This is what we consider to be the main issue with our current agent and would need to be handled to reach an optimal player agent. In table 3 the initial weights with the final weights can be seen for both our features. The corresponding state-action feature functions is also described in words.

	Feature 1	Feature 2
$\mathbf{w}_0$	1	-1
$\mathbf{w}$	-0.9746	0.7004
f_i	Will *next head location* hit a wall/body (=1), an apple (=-1) or nothing (=0)?	Is *next head location* closer to the apple then current *head location*? (yes=1, no=0)

Table 3: Table with the initial weights and the corresponding final weights. The state-action feature functions is also described in words.

Listing 5: Q weight update code when on non-terminal update

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

Listing 6: Q weight update code when on terminal update

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