Assignment 4 in Machine Learning FMAN45

Eliot Montesino Petrén, el6183mo-s, 990618-9130, LU Faculty of Engineering F19 , eliot.mp99@gmail.com

I. Exercises

Exercise 1

To derive the value of K for the small Snake game, we need to consider the number of possible states the game can be in. This involves determining the number of possible configurations of the snake and the number of possible locations for the apple.

First, we determine the number of possible apple locations. The game is played on a 7x7 grid, which means the interior grid where the snake can move and the apple can appear is 5x5. Thus, the total number of possible locations for the apple is

$$5 \times 5 = 25.$$

Next, we determine the number of possible snake configurations. The snake has a constant length of 3 pixels. There are 3 possible shapes for the snake: straight, bent to the left, and bent to the right (with the latter two being reflections of each other). For each of these configurations, the head can point in 4 different directions (N/E/S/W).

For the straight snake, the head can be placed in 3×5 different cells. For the bent snake, the head can be placed in 4×4 different cells. Thus, the total number of snake configurations is:

$$(3 \times 5 + 4 \times 4 + 4 \times 4) \times 4 = 188.$$

For each of these configurations, the apple can be placed in any of the remaining $5 \times 5 - 3 = 22$ cells. Therefore, the total number of possible states K is:

$$K = 188 \times 22 = 4136.$$

Thus, the value of K is 4136, calculating the number of possible states for the specific configurations of this Snake game, considering both the placement of the apple and the configurations and orientations of the snake.

Exercise 2

(a) Rewrite eq. 1 as an expectation using the notation $E[\cdot]$, where $E[\cdot]$ denotes expected value.:

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

The expectation of a discrete random variable X with possible values $x_1, x_2, \ldots, x_n$ and probabilities $P(X = x_i)$ is defined as:

$$E[X] = \sum_{i=1}^{n} P(X = x_i) \cdot x_i$$

In our context, we are dealing with the probabilities of transitioning to the next state s' given the current state s and action a. Here, $T(s,a,s')$ is the transition probability and represents the probability of transitioning to state s' from state s after performing the action a.

Using the concept of expectation for a discrete random variable, we can rewrite the summation as an expected value. Applying above to our equation, the answer to the question is obtained in eq 2:

$$Q^*(s,a) = E_{s' \sim T(s,a,\cdot)}[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')] \quad (2)$$

Where the notation $E_{s' \sim T(s,a,\cdot)}$ indicates that we are taking the expectation over the distribution of the next state s' given by the transition probabilities $T(s,a,\cdot)$. This means s' is drawn from the probability distribution defined by $T(s,a,\cdot)$. The optimal Q-value $Q^*(s,a)$ is the expected value of the immediate reward $R(s,a,s')$ plus the discounted optimal Q-value of the next state s', given that s' follows the distribution defined by $T(s,a,\cdot)$.

In simpler terms, this equation tells us that $Q^*(s,a)$ is the average of the future rewards, weighted by the probability of each possible next state s', and accounts for the immediate reward and the best possible future reward from state s'.

(b) Rewrite eq. 1 as an infinite sum without recursion: We can expand the Q-function for Bellman optimality recursively to obtain an infinite sum:

$$\begin{aligned} Q^*(s,a) &= \sum_{s'} T(s,a,s')R(s,a,s') + \\ &\gamma \sum_{s''} T(s',a',s'')[R(s',a',s'') + \gamma \max_{a''} Q^*(s'',a'')] \\ &= \sum_{t=0}^{\infty} \gamma^t \sum_{s_t} T(s_{t-1},a_{t-1},s_t)R(s_t,a_t) \end{aligned}$$

Here, $T(s_{t-1},a_{t-1},s_t)$ represents the probability of transitioning to state s_t from state s_{t-1} after taking action a_{t-1}. This infinite sum captures the expected discounted rewards over an infinite horizon.

(c) Explain what eq. 1 is saying in one sentence.: Equation 1 states that the optimal Q-value for taking action a in state s is the sum of the expected reward of that action and the discounted value of the best possible action in the subsequent state.

(d) Compare eq. 1 to the Bellman equations of Q^π in 3. What policy do we assume to follow in 1? Where in the equations do we make this assumption?:

$$Q^\pi(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma Q^\pi(s',\pi(s'))] \quad (3)$$

In Equation 1, the policy assumed is the optimal policy π^*. This assumption is made in the term $\max_{a'} Q^*(s',a')$,

where we select the action that maximizes the Q-value in the subsequent state, indicating the assumed optimal policy.

(e) What is γ? What is the effect of γ on the Q-values?: The term γ is the discount factor in reinforcement learning. It determines the value of future rewards compared to immediate rewards. A γ value close to 0 makes the agent prioritize immediate rewards heavily, while a γ value close to 1 makes the agent consider future rewards almost as important as immidiate rewards.

(Warning for blabber!) The discount rate might be a familiar term to economists where the discount rate, r, represents the loss/increase of nominal value in terms of equivalent real value through time...

(f) 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).: The term $T(s, a, s')$ represents the probability of transitioning from state s to state s' after taking action a. While transition probabilities in practical applications are often stochastic, in this game, the transition probabilities are (almost) completely deterministic because the next state s' is completely determined by the current state s and the action a.

If the action a does not make the snake eat an apple, there is only one possible subsequent state s', which is the state specified by the movement of the snake. In this case, the transition function $T(s, a, s')$ is:

$T(s, a, s') = 1$ if taking action a in state s leads directly to state $s', T(s, a, s') = 0$ otherwise.

There is a (single) stochastic element to the game. If the action a makes the snake eat an apple, a new apple must appear somewhere on the screen. There are 22 possible following states s' corresponding to the 22 possible placements of the apple within the 5x5 grid. Since the position of the apple is drawn from a uniform distribution, each of these 22 states has the transition function:

$T(s, a, s') = \frac{1}{22}$ if s' is one of the 22 states resulting from placing the apple in any unoccupied cell, $T(s, a, s') = 0$ otherwise.

For example: - $T(s, \text{move\_forward}, s') = 1$ if s' is the state resulting from moving forward from s. - $T(s, \text{turn\_left}, s') = 1$ if s' is the state resulting from turning left from s. - $T(s, \text{turn\_right}, s') = 1$ if s' is the state resulting from turning right from s.

In summary, the transition probabilities are deterministic when no apple is eaten, and uniformly distributed among 22 possible states when an apple is eaten.

Exercise 3

(a) Explain the difference between off-policy and on-policy.: Off-policy learning refers to learning the value of the optimal policy independently of the agent's actions. The agent learns from actions taken by a different policy, which is typically a random or exploratory policy. Q-learning is off-policy learning.

On-policy learning refers to learning the value of the policy being executed by the agent. The agent learns from actions taken by its own current policy, and updates its policy based on these actions. SARSA (State-Action-Reward-State-Action) is on-policy learning.

(b) Explain the difference between model-based and model-free reinforcement learning.: Model-based reinforcement learning involves learning a model of the environment's dynamics (the transition probabilities and the reward function) and then using this model to make decisions. The agent uses the model to simulate the environment and plan future actions.

Model-free reinforcement learning does not involve learning a model of the environment. Instead, the agent learns a policy or value function directly from its interactions with the environment. This approach typically requires more data and exploration but can be simpler to implement. Q-learning and SARSA are examples of model-free reinforcement learning.

(c) Explain the difference between active and passive reinforcement learning.: Active reinforcement learning involves the agent actively interacting with the environment to learn the best policy. The agent makes decisions about which actions to take to maximize its cumulative reward based on its current knowledge.

Passive reinforcement learning involves the agent learning from a fixed policy. The agent observes the outcomes of the policy but does not make decisions about actions to take. Instead, it evaluates the policy's performance and learns the value function associated with the policy.

(d) Explain the difference between reinforcement learning, supervised learning and unsupervised learning.: Reinforcement learning involves an agent learning to make decisions by interacting with an environment and receiving feedback in the form of rewards or punishments. The goal is to learn a policy that maximizes the cumulative reward over time.

Supervised learning involves learning a function from labeled training data. The algorithm is provided with input-output pairs, and the goal is to learn a mapping from inputs to outputs that generalizes well to new, unseen data.

Unsupervised learning involves learning patterns or structure from unlabeled data. The algorithm is not provided with explicit input-output pairs but must find hidden structures or relationships in the data. Common tasks include clustering and dimensionality reduction.

(e) What's the difference between using a dynamic programming approach (such as policy iteration) vs using Reinforcement Learning (such as Q-learning) when solving an MDP problem?: Dynamic programming approaches, such as policy iteration, require a complete and accurate model of the environment's dynamics, including the transition probabilities and reward function. These methods use this model to compute the optimal policy by iterating over all possible states and actions.

Reinforcement learning methods, such as Q-learning, do not require a model of the environment. Instead, they learn the optimal policy through trial and error by interacting with the environment. This makes reinforcement learning more flexible

and applicable to problems where the environment's dynamics are unknown or difficult to model.

Dynamic programming methods are generally more computationally intensive and require more memory, as they need to store and update values for all states and actions. Reinforcement learning methods can be more efficient in terms of memory and computation, as they focus on learning the optimal policy through sampled experiences rather than exhaustively computing values for all states and actions.

Exercise 4

(a) Rewrite eq. (4) using the notation $E[\cdot]$, where $E[\cdot]$ denotes expected value. Use the same notation as in (4) as much as possible. The expression should still be a recursive relation – no infinite sums or similar.: Equation (4) is:

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

Using the expectation notation $E[\cdot]$, we can rewrite this as:

$$V^*(s) = \max_a E_{s' \sim T(s,a,\cdot)}[R(s,a,s') + \gamma V^*(s')]$$

(b) Explain what eq. (4) is saying in one sentence.: Equation (4) states that the optimal value of being in state s is the maximum expected return achievable by taking any action a and then following the optimal policy thereafter.

(c) In eq. (4), what is the purpose of the max-operator?: The max-operator in Equation (4) is used to select the action a that maximizes the expected return from state s. It ensures that the value function $V^*(s)$ represents the highest possible value obtainable by choosing the best action in state s and following the optimal policy subsequently.

(d) 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 $\pi^*(s) = ...$? What is the equation called?: The relation between $\pi^*(s)$ and $V^*(s)$ is given by:

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

This equation is called the Bellman optimality equation for the value function.

(e) 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.: The relation between π^* and V^* is not as straightforward as that between π^* and Q^* because $V^*(s)$ represents the maximum expected return from state s under the optimal policy, without explicitly considering the actions. In contrast, $Q^*(s,a)$ represents the expected return from taking a specific action a in state s and then following the optimal policy. Therefore, Q^* directly incorporates the action choices, making it easier to derive the optimal policy $\pi^*(s)$ by simply selecting the action with the highest Q^* value.

Furthermore, V^* is the expected utility of a state, while Q^* is the expected utility of a state-action pair. The optimal policy π^* provides a "recipe" of what action to choose given that the environment is in a particular state. In a deterministic environment, the optimal action is the one leading to the state with the highest utility ($\max_{s'} V^*(s')$). However, in a Markov Decision Process (MDP), we are not certain which state we will end up in after taking an action. Therefore, the optimal policy is to choose an action a that maximizes Q^*, and V^* is then the expected value of the state, given that the optimal policy is followed.

Exercise 5

(a)

From what I can tell, I find that the function `policy_iteration` is implemented correctly and it yields similar results to what is desired, see table I and see the Python code attached in the appendix. There is only a difference in the number of policy evaluations (11 evaluations are desired), and I will proceed, I have spent enough time on this task anyways...

TABLE I
SUMMARY OF POLICY ITERATION

Description	Value
Number of policy iterations	6
Number of policy evaluations	10
Elapsed time	0.049 seconds

(b)

The learning rate hyperparameter γ seems to increase the number of policy iterations and policy evaluations as future actions are discounted at a lower rate. When $\gamma = 1$, the Delta converges $> \epsilon = 1$ and policy evaluation is never terminated, resulting in an infinite loop; i.e., the policy evaluation never converges.

TABLE II
EFFECT OF DIFFERENT γ VALUES ON POLICY ITERATION

γ	Policy Iterations	Policy Evaluations	Elapsed Time (s)
0	2	4	0.0264
0.95	5	28	0.096
1	Infinite loop (no convergence) Delta ≈ 3.8		

The results indicate that as the discount factor γ increases, the algorithm requires more policy iterations and evaluations to converge. For $\gamma = 0$, the policy iteration converges quickly with only 2 policy iterations and 4 policy evaluations. This is because the agent only considers immediate rewards, ignoring future rewards entirely. When $\gamma = 0.95$, the agent takes future rewards into account to a significant extent, leading to a more complex optimization problem that requires several policy iterations and policy evaluations. However, when $\gamma = 1$, the Delta remains above the tolerance $\epsilon = 1$, causing the policy evaluation to never terminate and resulting in an infinite loop. This happens because with $\gamma = 1$, future rewards are not discounted at all, leading to potentially unbounded value functions and excessively large updates. Consequently, the policy evaluation step does not converge, demonstrating how things can go wrong with a bad choice of γ.

For $\gamma = 0.95$, the snake playing agent considers both immediate and future rewards. After training, the agent can play the game successfully. This indicates that the learned policy is "on the right track" and it achieves the main objectives of eating apples and staying alive. The behavior is perhaps still suboptimal, as γ = 0.95 is not necessarily optimal for balancing the complexity of a strategy that avoids risks (such as hitting walls or its own body) and captures opportunities (such as eating apples).

(c)

TABLE III
EFFECT OF STOPPING TOLERANCE ON POLICY ITERATION

Stopping Tolerance (ϵ)	Policy Iterations	Policy Evaluations
1×10^{-4}	5	195
1×10^{-3}	5	147
1×10^{-2}	5	107
1×10^{-1}	5	56
1×10^{0}	5	28
1×10^{1}	18	18
1×10^{2}	18	18
1×10^{3}	18	18
1×10^{4}	18	18

The effect of stopping tolerance ϵ in policy evaluation was investigated for values ranging from 1×10^{-4} to 1×10^{4}. As observed from Table III, the number of policy iterations and policy evaluations varied with different values of ϵ.

For lower values of ϵ (from 1×10^{-4} to 1×10^{0}), the number of policy iterations remained constant at 5, while the number of policy evaluations decreased as ϵ increased. This indicates that with stricter convergence criteria (lower ϵ), more evaluations were needed to meet the stopping condition within each policy iteration.

Interestingly, for ϵ values from 1×10^{1} to 1×10^{4}, both the number of policy iterations and evaluations stabilized at 18. This suggests that beyond a certain threshold of ϵ, the algorithm's behavior becomes consistent, possibly due to reaching a balance between exploration and exploitation in the policy space.

The consistent number of policy iterations (5) for lower ϵ values indicates that the algorithm was able to find a stable policy relatively quickly. However, the higher number of evaluations for lower ϵ values suggests that more precise convergence was achieved in these cases.

For the higher ϵ values ($\geq$ 10), the increase in policy iterations to 18 might indicate that the algorithm was exploring more of the policy space before converging. The fact that the number of evaluations matches the number of iterations in these cases suggests that each iteration required only one evaluation to meet the stopping criteria.

The consistency in the number of policy iterations for lower ϵ values suggests that the algorithm was finding similar policies across these runs. The change in behavior for higher values ϵ indicates a change in how the algorithm explores and converges on policies, which could potentially lead to different final strategies for the snake game. Observing just the agent as snake with the naked eye does not reveal any significant insights I find. The agent plays a snake rather successfully for at least the first few apples observed for any of these runs.

To fully understand the impact on the snake's behavior, it would be necessary to observe and analyze the actual gameplay resulting from these different policies more in depth rather that what I have done. One could investigate how many apples the agent is able to eat over time and compare for different runs. This analysis could provide insights into whether the policies resulting from different ϵ values lead to optimal or suboptimal strategies in the game context.

Exercise 6

(a): The Q-update code implementation for non-terminal states and terminal states was written as follows:

```
# Q-value update for terminal state
sample = reward
pred = Q_vals[state_idx, action]
td_err = sample - pred # don't change this.
Q_vals[state_idx, action] += alph * td_err

# Q-value update for non-terminal state
sample = reward + gamm * np.max(Q_vals[
    next_state_idx, :])
pred = Q_vals[state_idx, action]
td_err = sample - pred # don't change this!
Q_vals[state_idx, action] += alph * td_err
```

Listing 1. Relevant Python Code for Q-Updates

b) and c): I will summarize attempts that showcase the thought process and the progress, even though I made more attempts that are not shown. These are summarized in table IV.

First I ran the baseline model and it performs not terrible. First I wanted to try to increase penalty, since my first thought was that optimizing around not dying would be priority number 1. The model is somewhat rediculous that I included, but it shows that just increasing penalty does not fix it and that it can in fact make the model worse. My reasoning for this is that when optimizing around just staying alive, then the agent does not try to get higher scores either and eventually the agent makes the snake die anyways. Risk averseness lowers risk of dying, but staying alive for too long also (ironically) highers the risk of dying per apple eaten.

Next I wanted to try to turn up the "heat" and I increased the exploration rate and the learning rate to see how that affected the model. It yielded some interesting results, so I continued to play around with it and also changed the alpha and epsilon update rate (introducing dynamics), since before it was set arbitrarily high so these hyperparameters were never updated. The exploration and learning rate starts high, then cools of further into the iterations once it has (hopefully) found a better policy.

The fourth model in the table is the result when I managed to balance the parameters somewhat and I took some conscious choices. I removed learning rate dynamics and made sure epsilon decreased even further. I re-balanced the scoring and made it more logical in my opinion. I re-introduced score for just staying alive which I think has synergy in the later iterations where the agent has longer games on average. The

TABLE IV
SUMMARY OF Q-LEARNING MODEL HYPERPARAMETERS AND PERFORMANCE (5000 ITERATIONS)

Parameter	Baseline	"Staying Alive"	Dynamic Model	Balanced Dyn. Expl.	Final model
Default Reward	0.1	1	0	0.5	-0.1
Apple Reward	10	10	10	20	20
Death Penalty	-100	-100000000000	-10	-100	-100
Gamma (γ)	0.9	0.9	0.9	0.9	0.9
Alpha (α)	0.2	0.2	0.5	0.5	0.5
Epsilon (ϵ)	0.1	0.1	0.2	0.3	0.3
Alpha Update Iterations	1,000,000	1,000,000	1,000	1,000,000	1,000,000
Alpha Update Factor	0.5	0.5	0.8	0.95	1
Epsilon Update Iterations	1,000,000	1,000,000	1,000	750	750
Epsilon Update Factor	0.5	0.5	0.8	0.7	0.5
Highest Score	**59**	**31**	**71**	**211**	**1209**

goal now really is to make sure the later iterations have longer games.

The fifth and final model in the table is the first model that reached the target (well and above). The differences are the decaying exploration rate decayed even faster and a negative score was added to just staying alive (following the argumentation of the staying alive model).

(d): It is difficult to achieve optimal behavior because the number of iterations is too low; the number of iterations is about the same as the number of states. Since states can be repeated, it means most likely that many states has not even been explored. Training requires many many iterations.

Exercise 7

(a): Code is written in Python.

```
# Non-terminal update
target = reward + gamm * np.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 - 1].reshape(-1, 1)

# 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 - 1].reshape(-1, 1)
```

(b): First I will attach the state-action feature engineering that I have implemented. I have implemented three state-action features for the Snake game:

1) Distance to apple: This feature measures the normalized L1 distance (L2 seem too much) between the snake's head and the apple. The normalization ensures the feature is always in the range [-1, 1], with 1 being closest to the apple and -1 being furthest. Initialize as -1.
2) The next cell in the snake's path. This feature evaluates what's in the next cell the snake would move to. Value 1 is for an apple (desirable), -1 for a wall or snake body (avoid), and 0 for an empty cell. Initialize as -1.
3) Size of open spaces on the board. This feature finds size of connected empty regions on the board. The snake wants to move into larger open areas, which helps prevent the snake from trapping itself as a preventive measure. The feature is the ratio of the size of the region the snake would move into compared to the largest empty region on the board (range [0,1]). Initialize as -1.

These features carry key information about the game state so the agent can guide the snake's decisions and is sufficient.

(b):

```
# HERE BEGINS YOUR STATE-ACTION FEATURE ENGINEERING
# 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.
apple_loc = np.argwhere(grid == -1)[0]  # Added this
     line of code to find apple location
snake_len_init = 10
for action in range(3):  # Evaluate all the
    different actions (left, forward, right).
    next_head_loc, next_move_dir = get_next_info(
        action + 1, movement_dir, head_loc)

    # Feature 1: Normalized distance to apple
    dist_to_apple = np.sum(np.abs(next_head_loc -
        apple_loc))
    max_dist = 2 * (N - 1)  # Maximum possible L1
        distance on the grid
    state_action_feats[0, action] = 1 - (
        dist_to_apple / max_dist)*2  # Now closer is
         better (range [-1,1])

    # Feature 2: Looking ahead to the next cell
    next_cell_value = grid[next_head_loc[0],
        next_head_loc[1]]
    if next_cell_value == -1:  # Apple
        state_action_feats[1, action] = 1
    elif next_cell_value > 0:  # Snake body or wall
        state_action_feats[1, action] = -1
    else:  # Empty cell
        state_action_feats[1, action] = 0

    # Feature 3: Find largest region of empty cells
    empty_cells = (grid == 0).astype(int)
    labeled_array, num_features = label(empty_cells)
    if num_features <= 1:
        state_action_feats[2, action] = 1  # Only
            one region, full exploration possible
    else:
        next_region_size = np.sum(labeled_array ==
            labeled_array[next_head_loc[0],
            next_head_loc[1]])
        largest_region_size = np.max(np.bincount(
            labeled_array.flatten())[1:])
        state_action_feats[2, action] =
            next_region_size / largest_region_size
            # Ratio of sizes

return state_action_feats, prev_grid, prev_head_loc
```

TABLE V
SUMMARY OF LINEAR Q-LEARNING MODEL HYPERPARAMETERS AND PERFORMANCE

Parameter	Baseline	Model from E6	Balanced for weights 1	Balanced for weights 2
Default Reward	-0.1	0	-0.1	-0.05
Apple Reward	20	1	10	8
Death Penalty	-100	-1	-20	-10
Gamma (γ)	0.9	0.99	0.95	0.95
Alpha (α)	0.5	0.5	0.2	0.2
Epsilon (ϵ)	0.3	0.5	0.5	0.5
Alpha Update Iterations	1,000,000	1,000,000	1,000,000	1,000
Alpha Update Factor	1	1	1	0.85
Epsilon Update Iterations	750	1,000,000	500	500
Epsilon Update Factor	0.5	0.75	0.5	0.5
Highest Score	**58**	**4**	**86**	**101**
Average Score Last 100 Episodes	**4.76**	**0.0**	**36.76**	**40.65**

After the code implementation was deemed to be in a working state, fine-tuning of hyperparameters was performed. The baseline model's performance and hyperparameters are summarized in Table IV. As we can see, the initial performance of the baseline model is not very impressive, with a highest score of only 4 and an average score of a whopping 0. This provides a starting point for further improvements and adjustments to the model's parameters.

To proceed, we will apply the lessons learned from the previous exercise and implement a series of modifications to enhance the model's performance. The subsequent models and their respective performances will be documented in the same table, allowing for a comprehensive comparison of the different approaches. The next model that if interesting to test is simply taking the values from the previous exercise onto this one directly. The set of parameters evidently does not reach the same heights with this model as before, but I still believe that many principles from earlier (parameter dynamics) are still reasonable, I have to adapt it to the new model that are based on weights and feature extraction.

Moving on I will adjust the reward and penalties as I suspect their numerical value affect weight updates in a non-optimal way this time around. I have included two attempts in the order of when I did them. The general goal for these models is to reduce the weight updates for enhanced weight convergence. See the table Vl, first and foremost I have significantly reduced learning rate and introduced new lowered rewards/penalties. I also made sure to reduce the exploration (epsilon) more significantly through the iterations. This soon yielded a model that happened to above reach target (above 35 average score on last 100 training runs). The last model I am showing yielded ever so slightly better results after I introduced learning rate dynamics again and experimented some with rewards and penalties. It most likely possible to get more impressive results but this is the extent of this exercise I think.

(c): Now remains some analysis of the final models (but I have only a three features model to present). See table VII and weights in table VI. The final version of the 3 features model is very similar to the best one from subsection b) but apple reward was increased ever so slightly because when I tested the model I ran into infinite loops, so I adjusted apple reward so the snake would go for the apples instead of just staying alive and it worked.

My discussion in previous subsection and previous exercise is sufficient to explain the success of the final model and the general idea of choices of hyperparameter values. It is satisfying to see that in the end every hyperparameter fullfills a purpose and imporoves upon the model. There are surely ways to improve performance even further. Further tuning of hyperparameters can most surely yield even better results, but I feel that the initialization of the weight vector would be also an interesting place to start. Training has apparently found local minimum weight values and there is still things that could be explored here by starting at values close to the local minimum, or try training with randomly initialized weights in hope of finding a new optimal combination. I very quickly tried exactly this, but it found new weights slightly different from the one I consistently get with the normal initialization. Testing the new weights the snake unfortunately got stuck in infinite loop in the testing runs, thus highlighting the notorious nature of parameter tuning that there is no "fix-all", except for maybe increasing number of training iterations!

I find it slightly unclear from instructions but as I understand a 2 features model is also ideal to report here but I choose to opt out as it seems to not be required and I could use some extra hours right now! I started directly at 3 features and the code is designed that way...

TABLE VI
FINAL MODEL WEIGHTS (ALL INITIALIZED AT -1)

Weight Number	Category	Final 3 features model
1	Distance to Apple	9.37
2	Next Cell Content	4.31
3	Open Spaces	0.0302

II. REFERENCES

TABLE VII
SUMMARY OF FINAL LINEAR Q-LEARNING MODELS AND THEIR HYPERPARAMETERS AND PERFORMANCE

Parameter	Final model 3 features
Default Reward	-0.05
Apple Reward	10
Death Penalty	-10
Gamma (γ)	0.95
Alpha (α)	0.2
Epsilon (ϵ)	0.5
Alpha Update Iterations	1,000
Alpha Update Factor	0.85
Epsilon Update Iterations	500
Epsilon Update Factor	0.5
Highest Score	**111**
Average Score Last 100 Episodes	**46.8**

III. Appendix

```
def policy_iteration(pol_eval_tol, next_state_idxs, rewards, gamm):
    # Get number of non-terminal states and actions.
    nbr_states, nbr_actions = next_state_idxs.shape

    # Arbitrary initialization of values and policy.
    values = np.random.randn(nbr_states)
    policy = np.random.randint(0, nbr_actions, size=(nbr_states,))

    # Counters over the number of policy iterations and policy evaluations,
    # for possible diagnostic purposes.
    nbr_pol_iter = 0
    nbr_pol_eval = 0

    # This while-loop runs the policy iteration.
    while True:
        # Policy evaluation.
        while True:
            Delta = 0
            for state_idx in range(nbr_states):
                # Save the old value
                v = values[state_idx]

                # Determine the reward based on the state and action
                action = policy[state_idx]
                next_state_idx = next_state_idxs[state_idx, action]

                if next_state_idx == -1:
                    reward = rewards['death']
                elif action == 1:
                    reward = rewards['apple']
                else:
                    reward = rewards['default']

                # Update the value function based on the current policy
                next_state_idx = int(next_state_idx)
                values[state_idx] = reward + gamm * values[next_state_idx]

                # Calculate the change for convergence check
                Delta = max(Delta, abs(v - values[state_idx]))

            # Increase nbr_pol_eval counter.
            nbr_pol_eval += 1

            # Check for policy evaluation termination.
            if Delta < pol_eval_tol:
                break
            else:
                print('Delta:', Delta)

        # Policy improvement.
        policy_stable = True
        for state_idx in range(nbr_states):
            # Save the old action
            old_action = policy[state_idx]
```

```
            # One-step lookahead to find the best action
            action_values = np.zeros(nbr_actions)
            for action in range(nbr_actions):
                next_state_idx = next_state_idxs[state_idx, action]
                if next_state_idx == -1:
                    reward = rewards['death']
                elif action == 1:
                    reward = rewards['apple']
                else:
                    reward = rewards['default']

                next_state_idx = int(next_state_idx)
                action_values[action] = reward + gamm * values[next_state_idx]

            # Choose the best action
            best_action = np.argmax(action_values)
            policy[state_idx] = best_action

            # Check if the policy has changed
            if old_action != best_action:
                policy_stable = False

        # Increase the number of policy iterations.
        nbr_pol_iter += 1

        # Check for policy iteration termination
        if policy_stable:
            break

    return values, policy, nbr_pol_iter, nbr_pol_eval
```