Assignment 4
FMAN45
Machine Learning

Reinforcement learning for playing Snake

Author

Kajsa Hansson Willis

Lund University

Spring 2025

1 Exercise 1

In the first exercise, the aim is to, for the small Snake game, derive the value of K, which is the number of non-terminal states. Thus, the amount of possible states must be considered which involves counting the amounts of configurations the snake can have, as well as the amount of positions for the apple.

In the 7x7 grid, there are only 5x5 non-edge pieces, which corresponds to the valid places for the apples. However, it cannot be placed in any of the pixels where the snake is. Thus, the number of possible positions for the apples given a snake configuration is described by $5 * 5 - 3 = 22$.

To determine the amount of configurations for the snake one must consider all the shapes and the positions the snake can have. To simplify the calculations, it will be assumed that the head will be the pixel closest to the top right corner. If the snake forms a straight horizontal line, there are three positions in each row it can be on and five rows. This gives fifteen possible configurations in this scenario. The same reasoning holds for a straight vertical line, thus contributing fifteen additional configurations.

Further, the shapes where the snake is bent must be considered. There are four possible scenarios for this – assuming a straight vertical snake, it can bend to the upper right, upper left, lower right or lower left. For each of these options, there are $4*4 = 16$ possible pixels on the grid to place out the head of the snake.

As it was assumed that the head was in one position, the head being in the opposite end of the snake must also be considered. This is done by multiplying the amount of configurations found by two. This gives the following expression for the amount of non-terminal states:

$$(15 + 15 + 16 + 16 + 16 + 16 + 16) * 2 * 22 = 4136$$

Thus, the value of K has been determined as 4 136.

2 Exercise 2

2.1 Bellman optimality equation as expectation

The Bellman optimality equation can be found in Equation 1, where $R(s, a, s')$ corresponds to the reward for the state, $T(s, a, s')$ to the transition probability and $\gamma \max_{a'} Q^*(s', a')$ is the time-discounted Q-value for the optimal next state.

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

The expectation of a discrete variable can be described by Equation 2 for the i possible states of X. The expectation of a function of a stochastic variable is Equation 3.

$$\mathbb{E}(X) = \sum_i x_i * P(X = x_i) \tag{2}$$

$$\mathbb{E}(f(X)) = \sum_i P(X = x_i) \cdot f(x_i) \tag{3}$$

As $T(s, a, s')$ is the associated probability, ie $P(X = x_i)$ and the sum ensures that all possible states are summed over, the Bellman optimality equation can be rewritten as Equation 4. It is then formulated as an expectation.

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

2.2 Bellman optimality equation as an infinite sum without recursion

To rewrite Equation 1 as an infinite sum without recursion, it must be noted that the equation is recursive through its $\max_{a'} Q^*(s', a')$ element. This can be rewritten as:

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

Utilizing this, the Bellman optimality equation can be rewritten as Equation 5, where $i = 0$ is the current state and the values are the optimal ones. γ is the discount factor that typically attributes less importance to states far into the future.

$$\begin{aligned} Q^*(s, a) &= \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma \max_{a'} \sum_{s''} T(s', a', s'')[R(s', a', s'') + \gamma \max_{a''} Q^*(s'', a'')]] \\ &= \sum_{i=0}^{\infty} \gamma^i * T(s_t, a_t, s_{t+1}) * R(s_t, a_t, s_{t+1}) \end{aligned} \tag{5}$$

2.3 Interpretation of the Bellman optimality equation

The meaning of the Bellman optimality equation in Equation 1 is that the optimal Q-value for a given state-action pair (s, a) is equal to the expected reward if the next state as well as the discounted expected reward for all the future states for the best possible future actions.

2.4 Comparison of the Bellman equations of Q^π and the Bellman optimality equation

The Q-value for a given policy π is given by Equation 6. A comparison to Equation 1 shows that they are very similar in their structure and the first terms, but the latter term in the parenthesis vary. In the optimality equation it is $\gamma \max_{a'} Q^*(s', a')$, whereas it is $\gamma Q^\pi(s', \pi(a'|s'))$ in the Q^π-Bellman. The γ-term is the same, so the difference boils down to the Q-value equation. It is known that $\pi^*(s) = a^* = \arg\max_a Q^*(s, a)$, which can be observed as the policy that inserted into Equation 6 gives Equation 1. Thus, the optimal policy π^* is the policy assumed in the Bellman optimality equation.

$$Q^\pi(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma Q^\pi(s', \pi(a'|s'))] \tag{6}$$

2.5 Interpretation of γ

The value of γ represents a discount factor that determines the weight of future Q-values. If γ is high, distant Q-values will have a larger impact which represents high utility for long-term rewards. If γ is small, then short-term utility is premiered. A high value will typically lead to a higher Q-value if the rewards are non-negative though.

2.6 The transition probability for the small snake game

As $T(s, a, s')$ represents the probability of transitioning between states, it will depend heavily on whether an apple is consumed or not for the small snake game.

If an apple is not consumed, the next state will depend entirely on which direction the player decides to move the snake. Thus, the transition matrix will be $T(s, a, s') = 1$ if the action a leads directly to s' and $T(s, a, s') = 0$ if it does not.

If the apple is consumed, however, the scenario is a bit more complicated. Then, a new apple must be placed on the grid in any of the equally probable 22 pixels (the pixels where the snake and the apple were are ineligible). The movement of the snake is still deterministic so the transition matrix will be $T(s, a, s') = \frac{1}{22}$ if the movement of the snake leads directly to the new position of the snake, and $T(s, a, s') = 0$ if it does not, assuming that the snakes position overlaps the apples position.

3 Exercise 3

3.1 Off-policy and on-policy

An active reinforcement learning agent has two different policies. These are a behavior policy and a learning policy. The behavior policy is used to generate actions and interacts with the environment to gather sample data. The learning policy is the target policy to learn, ie the optimal policy π^*. The agent aims to eventually discover this policy through interactions with its environment.

If the learning policy, has been learned, which means the behavior policy and the learning policy are equal, then the reinforcement agent is on-policy. If the behavior policy and the learning policy are not equal, then the reinforcement learning agent is off-policy.

If the agent is on-policy it means that the agent learns the efficiency of the policy being used, including exploration actions. In this scenario, the policy is typically soft and non-deterministic in order to ensure possibilities of exploration.

If the agent is off-policy, the target policy is learned independently of the actions taken to explore the environment. This means that the agent follows one policy, but learns the efficiency of a different one.

3.2 Model-based and model-free reinforcement learning

In short, model-based reinforcement learning uses a model of the environment to select actions, whereas a model-free reinforcement learning selects its actions directly from experience without creating a model of the environment.

This means in the model-based, the aim is to first learn or estimate the probabilities and rewards, $P(s'|s,a)$ and $R(s,a,s')$ and to then use Markov Decision Process algorithms. In the model-free, the agent may try to directly learn the optimal policy, $\pi^*(s)$, or values functions, $V^*(s)$ or $Q^*(s,a)$, for instance.

3.3 Active and passive reinforcement learning

Active reinforcement learning involves learning the optimal policy while interacting with the environment. Passive reinforcement learning involves assuming a fixed policy with the aim of evaluating that policy.

In the active RL the agent chooses its actions and policy with the goal of learning the optimal policy. This involves exploration versus exploitation and on-policy versus off-policy.

In the passive RL, the agent has a given fixed policy and the goal of estimating state transition probabilities and reward function if it is model-based, and to learn value functions if it is model-free.

3.4 Reinforcement learning, supervised learning, and unsupervised learning

Reinforcement learning, supervised learning, and unsupervised learning are all machine learning methods. However, in reinforcement learning, the agent learns by interacting with an environment where it receives rewards or penalties based on actions with the aim of maximizing the rewards. In reinforcement learning, there are no clearcut answers/labels or errors. Instead, the agent must discover the optimal policy by testing different ones and evaluating them.

In supervised learning, the data is accompanied with labels. There is a correct answer to each classification, and it is learned through training data and then tested using test data to determine how well it works. This is mainly done for classification and regression.

Unsupervised learning does not have labels; instead the model tries to identify clusters or patterns without predetermined categories from the data.

To highlight the differences, one can say that reinforcement learning is used to optimize actions in order to maximize rewards, supervised learning is used to predict outcomes, and unsupervised learning is used to discover patterns or clusters hidden in data.

3.5 Dynamic programming approach versus using Reinforcement Learning for MDP problems

A Markov Decision Process (MDP) is a setup for sequential decision-making where subsequent rewards must be considered when determining the optimal action. Thus, the MDP involves a trade-off between immediate and future rewards.

Dynamic programming (DP) is a group of methods used to find optimal policies when the environment is fully known and can be described as a MDP. DP has some drawbacks stemming from their vast computational expense and assuming a perfect model. However, many reinforcement learning methods can be considered attempts of imitating dynamic programming with less computation and without the perfect environmental model assumption.

Reinforcement learning (RL) is, as previously explained, a learn-by-doing approach, where the agent interacts with the environment in order to discover the optimal policy.

Both dynamic programming and reinforcement learning share the idea of using value functions to organize and structure the pursuit of optimal policies. However, they vary vastly as DP requires a perfect model of the environment, whereas RL does not require a model of the environment at all. DP is planned initially, but RL is explored and solved through testing and learning. Dynamic programming is model-based, whereas reinforcement learning is model-free. RL can also be less computationally expensive.

4 Exercise 4

4.1 The Bellman optimality equation for the state value function as an expectation

The Bellman optimality equation for the state value function is given by Equation 7; again, where $T(s,a,s')$ corresponds to the transition probability, $R(s,a,s')$ to the reward for the state, and $\gamma V^*(s')$ to the discounted future state value. Utilizing Equations 2 and 3 in the same way as in exercise 2, this expression can be rewritten as the expectation in Equation 8.

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

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

4.2 The interpretation of the Bellman optimality equation for the state value function

Equation 7 states that the optimal value of a state is the maximum expected return from both the current and discounted future rewards from the optimal action.

4.3 The purpose of the max-operator

The purpose of the max-operator in Equation 7 is to ensure that the optimal policy is followed and the action, a, that maximizes the expected reward is selected. Thus, the state value function is the highest it can possibly be, which is the desired trait.

4.4 The relation between $\pi^*(s)$ and V^*

The relationship between π^* and V^*, knowing that $\pi^* = \arg\max_a Q^*(s,a)$, is given by Equation 9. This is because the optimal policy should be to do the action that maximizes the expected sum of the immediate reward and the discounted future rewards when continuing to choose the optimal action.

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

4.5 The difference between the relations between $\pi^*(s)$ and V^* and $\pi^*(s)$ and Q^*

The qualitative difference between the Equations 1 and 7 is that the state value function includes a maximization element and considers the entire set of possible actions. In turn, the state- action value function only considers the optimal Q-value for a given action. This adds complexity to the former.

Thus, it already considers the optimal policy, and the only thing needed is to take the argument of it to retrieve the optimal policy.

5 Exercise 5

5.1 Small snake game setup

The *policy_ iteration*-function in *Matlab* was updated with policy evaluation and policy improvement. The code for the policy evaluation can be observed here:

```
old = values(state_idx);
        a = policy(state_idx);
        next_state = next_state_idxs(state_idx, a);

        if next_state == 0
```

```
            reward = rewards.death;
            new = reward;
        elseif next_state == -1
            reward = rewards.apple;
            new = reward;
        else
            reward = rewards.default;
            new = reward + gamm * values(next_state);
        end

        values(state_idx) = new;
        Delta = max(Delta, abs(old - new));
```

The code for the policy improvement can be observed here:

```
actions = next_state_idxs(state_idx,:);
v = zeros(1,nbr_actions);
for aidx = 1:nbr_actions
    act = actions(aidx);
    if act == -1
        v(aidx) = rewards.apple;
    elseif act == 0
        v(aidx) = rewards.death;
    else
        v(aidx) =  rewards.default + gamm*values(act);
    end
end


[~,a] = max(v);
 if a ~= policy(state_idx)
      policy_stable = false;
      policy(state_idx) = a;
 end
```

It was ensured that with $\gamma = 0.5$ and $\epsilon = 1$, there were six iterations of the policy and eleven evaluations of the policy. Additionally, the elapsed time was 0.039609 seconds for the small snake game.

5.2 Varying γ

The amount of policy iterations and evaluations for the different values of γ can be found in Table 1.

In the case $\gamma = 0$, the optimal policy is reached very quickly which consists of the snake chasing its own tail counterclockwise in a tiny circle. This is because no reward past the first iteration holds any value for the agent. Therefore, it does not get any apples but still avoids the wall as it does not want to terminate the game. In the Matlab code, the maximum value of equal values will select the lowest index each time, which will result in the snake always turning in one direction, and thus going in a circle. If the code was changed so that an action was chosen randomly out of those with the same value, the snake would eventually collect apples, but not every efficiently. The snake is playing optimally, but the results are not very good which suggests that the value of γ is not the best.

If $\gamma = 0.95$ instead, the snake appears to behave much more optimally. The snake game continues for a long time while eating apples and moving according to the quickest route in order to reach the next apple. It does not appear to hit the wall easily either. The trial game was terminated at 1,000 apples as it showed no signs of dying anytime soon. Its amount of policy iterations and evaluations, found in Table 1 appear to be very reasonable, although convergence is slower than for $\gamma = 0$, and 0.95 seems to be an intuitively reasonable discount factor, as it premieres immediate rewards a little but still considers rewards far into the future.

For $\gamma = 1$, the policy evaluations are never ending as the value of delta converges at 5.5023 and never goes below ϵ, set to 1. This is because the future rewards are never discounted which leads to unbounded values and it has to continue to be calculated infinitely. As the policy evaluation loop is never left, there is only one policy iteration. If γ were less than one, it would decay over time and eventually reach the threshold ϵ, but now it never decays. There is thus no final snake-playing agent and no optimal policy.

γ	***Policy Iterations***	***Policy Evalutions***
0	2	4
0.95	6	38
1	1	∞

Table 1: The amount of policy iterations and policy evaluations for different values of γ with $\epsilon = 1$

When γ increases, the amount of policy evaluations increases as well, as can be understood from Table 1. This is because the discount factor increases the value of rewards down the line, so that a larger amount of time periods forward have to be run in order to shrink the delta value past the value of ϵ.

5.3 Varying ϵ

The amount of policy iterations and evaluations for the different values of ϵ can be found in Table 2.

ϵ	***Policy Iterations***	***Policy Evalutions***
1e-4	6	204
1e-3	6	158
1e-2	6	115
1e-1	6	64
1e0	6	38
1e1	19	19
1e2	19	19
1e3	19	19
1e4	19	19

Table 2: The number of policy iterations and evaluations for different values of ϵ and $\gamma = 0.95$

As shown in Table 2, increasing ϵ significantly reduces the number of policy evaluations needed per iteration, but beyond a certain point, it negatively impacts convergence, as can be seen by the number of policy iterations.

For small values of ϵ (up to one), the number of policy iterations stays at 6, but the number of evaluations per iteration decreases. A higher ϵ allows the policy evaluation step to converge more quickly, since the policy has larger allowances.

However, for very large values ($\epsilon \geq 10$), the number of policy iterations as well as number of policy evaluations skip to 19. This indicates that the algorithm converges immediately in each iteration, but requires more iterations overall in order to find the optimal policy. This happens because a very high ϵ makes the policy too random, preventing it from improving efficiently. It is likely impossible for the delta value to reach the largest values of ϵ which is why the loop is immediately exited.

For all the values of ϵ, the snake appeared to more or less follow an optimal policy. However, to truly investigate whether an optimal policy is reached or not, one would have to compare the apples eaten over time to a metric such as the largest amount possible. Without insight into these numbers, it is impossible to say whether the agent is acting optimally or not, but from observations it does appear to do so.

6 Exercise 6

6.1 Q-update code

The Q-update code for the terminal update can be seen below.

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

The code for the non-terminal update can be seen below.

```
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;
```

6.2 Training the agent

The initial test score of the agent without any parameter changes was zero, which is very far from the desired value of 250. The default parameter values can be seen in the first row of Table 3, where all the parameters and the corresponding test scores for the different variations are displayed.

The first modifications were to change the reward structure to [-0.01, 2, -1], and the learning rate, α to 0.1, see row two of Table 3. The reasoning for this was that a changed reward structure could incentivize getting to the apple quicker without unnecessary movements and perhaps quicker increase learning. This gave a test score of 2 before it gets stuck in an infinite loop, which is a small increase from the initial value but still very far from the desired one. It is likely that the agent found a way to survive for long without making any progress with collecting apples. Factors that might have impacted this poor performance is that the learning rate may be too low considering the amount of episodes and the reward may be too low considering it is rare to collect an apple in the beginning so the positive signals might need more strength.

The next modification was to set the random action selection probability, ϵ to 0.001, $\alpha = 0.2$, and the reward structure to [0, 1, -2] see row three of Table 3. The motivation for this was to favor more exploitative behavior to try to stimulate the apple collecting and increase the cost of death to survive longer. This gave a result of 34 apples before the snake died. This is a considerable increase from the previous test score and the initial test score, but still nowhere close to 250. However, it indicates that these changes were beneficial, and perhaps further adjustments in the same direction may increase the test score even more. It is possible the exploration rate is too low so no new information is learned and that the rewards should be larger to encourage apple-seeking further. The learning rate may still be too low, as well.

Building on the previous change, the next change was to set ϵ to 0.01 and $\alpha = 0.3$, the reward structure to [0, 5, -10] and γ to 0.9, which can be seen in the fourth row of Table 3. This gave a test result that kept growing and was finally terminated around 500,000. Presumably, the snake could have continued infinitely. This worked well because it balances a high reward for collecting apples and a large cost of death, as well as it has a beneficial ratio of exploration and exploitation and a learning rate that accelerates the convergence and a discount factor that balances immediate rewards with future situations.

ϵ	α	γ	***Rewards [default, apple, death]***	***Test score***
0.01	0.01	0.9	[0, 1, -1]	0
0.01	0.1	0.9	[-0.01, 2, -1]	2
0.005	0.2	0.9	[0, 1, -1.5]	34
0.01	0.3	0.9	[0, 5, -10]	Presumably ∞ ($>$ 500,000)

Table 3: Parameters and test scores for the Snake game with the initial settings on the first row

6.3 Changing the settings

The last modification worked extremely well for the snake game and are thus satisfactory final settings. They are displayed in Table 4.

ϵ	α	γ	***Rewards [default, apple, death]***	***Test score***
0.01	0.3	0.9	[0, 5, -10]	Presumably ∞

Table 4: Final setting parameters

In this configuration, the test run did not terminate and kept increasing in score, which means that the snake keeps eating apples without dying. However, it is uncertain if this is the optimal policy as it may be possible to reach the apples in fewer steps. The score did increase very quickly so if the policy is not optimal, it is at least reasonable to suspect it is very close. Thus, at least a good policy was trained for the small snake game via tabular Q-learning. Considering that the cost of death is not unreasonably high compared to the reward of eating an apple, the agent should prioritize getting to the apples quickly quite high, but still be avoidant of death which seems to have generated a successful and stable policy.

6.4 Difficulty of achieving optimal behavior within 5,000 episodes

There is some difficulty in achieving optimal behavior in the small snake game within 5,000 episodes, primarily because there are almost as many states. Within these 5,000 episodes, the 4,136 states must be explored and the optimal strategy learned. The rewards are also quite scarce and can be quite delayed which adds additional difficulty in the game and makes it harder to train the agent for all situations. Therefore, it would be considerably easier to train the agent with many more episodes.

7 Exercise 7

7.1 Q-weight update code

The Q-weight update code for the terminal update can be seen below.

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

The Q-weight update code for the non-terminal update can be seen below.

```
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);
```

7.2 Training adjustments

The first state-action feature is whether the head of the snake is next to a wall in the next move. If it is, the value is 1 and if it is not then the value is -1. As it is more desirable to be far from the wall in a safer space, this feature was initialized as -1.

The second state-action feature is the normalized Euclidean distance between the apple and the head of the snake. It is normalized to be between -1 and 1, where -1 means that their positions overlap and 1 is the furthest possible distance. As it is more desirable to be close to the apple, this feature was initialized at 1.

The third state-action feature was to check whether the next move is in the direction of the apple or not. If the next move is toward the apple, the value will be 1, whereas if it does not, the value will be -1. As it is desirable to move toward the apple, this feature was initialized at -1.

After the state-action features had been initialized, they were tested without a parameter change. This resulted in a mean score after 100 episodes of 0. This indicates that the parameters must be tuned or the features need to be adjusted as it is far from the goal of 35. This can be observed in Table 5, where the parameters, features and results can be viewed for all of the configurations.

The first parameter modification was to change the reward structure to [0, 100, -50] in order to encourage quicker learning when encountering apples, as there are very few episodes and it is unlikely to collect many apples randomly. This gave a mean score of 0.16 collected apples over 100 episodes, as can be seen in Table 5.

The next parameter modification was to change the reward structure to [-5, 100, -100], ϵ to 0.8 and α to 0.6. The idea was to encourage quicker routes to the reward by adding a penalty for regular steps and increasing the exploration and learning rate to encourage quicker learning. This gave a result of 4.65 apples, which was a step in the right direction.

In the third parameter modification, γ was kept at 0.99, α changed to 0.4, and ϵ to 0.55. The reward structure was changed to [0, 3, -2]. The value of α was updated with factor 0.8 every 1000 iterations and the value of ϵ was also updated with factor 0.5 every 500 iterations. However, the largest change was that after seeing that the snake often got stuck in a loop, an additional feature was added to avoid being stuck in loops by avoiding tight spaces. This was done by counting the amount of unoccupied spaces the next placement has, excluding the apple, and normalizing it. This was initialized as -1, which is the equivalent of zero free spaces.

This led to a mean score after 100 episodes of 6.55, which was larger than the previously recorded values, but still far from the target. The increase can be explained by the added feature that helps the snake not get stuck in infinite loops which allows for better adjustments of the parameters. The significantly lowered reward structures also increases stability among the iterations, however, it may lead to slower learning. The decaying values of α and ϵ allow for quicker learning and more speedy adjustments and exploration in the beginning. Then, as more knowledge has been found and the need for adjustments decreases, there is more exploitation and a much lower learning rate.

As it felt like a more drastic change was needed, in the fourth and last parameter modification, the first and second feature were modified and the fourth feature was removed. The first feature was changed so that it checks if the snakes next step is in a wall or snake position, ie if the snake dies in the next move. It was still initialized at -1. The second feature was adjusted to be more impactful, by being discretely one if the snake gets closer to the apple, negative one if it gets further away and zero if the distance is the same, instead of the previous scale. The feature is still initialized as -1. The fourth feature was deemed to not have enough impact, so the next try would be without to see if it still could work. The value of ϵ and the rewards were drastically changed to encourage more apple-seeking behavior, which can be seen in Table 5.

This led to a mean score of 41.05 after 100 episodes, and was thus deemed successful. Most likely, the accuracy and impact of the changed features led to this improvement.

ϵ	α	γ	***Rewards [default, apple, death]***	***Features***	***Mean score after 100 episo***
0.5	0.5	0.99	[0, 1, -1]	Unchanged	0
0.5	0.5	0.99	[0, 100, -50]	Unchanged	0.16
0.8	0.6	0.99	[-5, 100, -100]	Unchanged	4.65
0.55	0.4	0.99	[0, 3, -2]	Added feature for finding empty spaces	6.55
0.06	0.445	0.985	[0, 30, -200]	Changed first, second and removed fourth feature	41.05

Table 5: The parameters and features and their respective mean scores after 100 episodes of different tests, with the initial value on the first row. Note that the update factors and iterations are excluded.

7.3 Final test scores

After testing countless feature variations and different parameter constellations, an average test score after 100 game episodes of 41.05 was achieved. This was done with the parameters found in Table 6.

ϵ	α	γ	**Rewards [default, apple, death]**	α **iteration update**	α **iteration factor**	ϵ **iteration update**	ϵ **iteration factor**	**Mean score after 100 episodes**
0.06	0.445	0.985	[0, 30, -200]	900	0.705	500	0.605	41.05

Table 6: The optimal parameters for the final Snake version

A table of the initial weights, the final weights and the associated final state-action feature functions can be found in Table 7.

	Initial	***Final***	f_i
w_1	-1	86.7114	If the next position is in the wall or the snake
w_2	1	-1.4729	If the snake gets closer to the apple
w_3	-1	3.1084	If the next move is in the direction of the apple

Table 7: The initial and final weights for the final three feature functions

Because the features had been suboptimal, the snake had in very many instances gotten stuck in an infinite loop. Additional features were added to counter this including the one presented in the previous task as well as a version of it looking further into the future to optimize the number of empty spaces in the future. However, it did not solve the original problem of the agent getting stuck in infinite loops. Therefore, the additional features proved excessive and it was enough to adjust the initial features and the parameters.

The parameters work well because they balance learning and finetuning, exploration and exploitation, and the cost of dying to the rewards of collecting apples. Additionally, the learning rate and exploration rate are updated as the process is learned in order to optimize results and learning.

Looking at the weights, it is interesting to see that the first feature has the most impact by far. This shows that is a very important feature, which makes intuitive sense, as it is very important for the snake not to die.