Machine Learning, LTH 2021

Assignment 4: Reinforcement Learning for Playing Snake

Name: Isabelle Frodé
Date: May 20 2021

Assignment Overview

The assignment was given in the course Machine Learning FMAN45 at LTH spring 2021. The goal with the assignment was to train a reinforcement learning agent to play the game *Snake*. To begin with a simplified *Snake* game was studied. All possible states were derived and calculated. The Bellman optimality equation for Q and V was studied and explained. A dynamic programming procedure was performed for a truncated version of the simplified *Snake* game. A policy iteration was implemented to obtain the optimal policy for the reinforcement agent. A *Snake* playing agent was then trained for a *small Snake* game using tabular Q-learning and then for a full *Snake* game using Q-learning with linear function approximation. Different parameters, rewards and feature configurations were tested to find the final settings.

Introduction

The aim of this assignment was to train a reinforcement learning agent to play the game *Snake*. The goal of the game is to steer the snake to the apples while avoiding colliding with the walls and the body of the snake. Details of the game used in this assignment was given in the assignment description [1].

To start with, a simplified version of the game was studied. The snake could move inside a 5×5 pixel grid and the snake had a constant length of 3 pixels. Additional definitions are presented below in Table 1:

$Q(s, a)$	state-action value for each state-action pair (s, a)
K	number of states in the game
s	states, K possible
a	action, 3 possible: *left, forward, right*
$\mathbf{Q}$	$K \times 3$ matrix which stores all $Q(s, a)$

Table 1: Variable description for the simplified Snake game

The simplified version is referred to as the *small Snake* game.

Tabular Methods

1. Derivation of the value of K

A state describes the current situation and it is the base for the agent to make choices. In *Snake*, the environment could be described by various of configurations of the snake and the apple. To derive the number of possible states in the *small Snake* game, i.e. K in Table 1, the possible configurations of the snake was first derived and calculated followed by the possible configurations of the apple.

Configurations of the snake

Four parameters that affects the configuration of the snake were identified and shown in Table 2 below.

position	x- and y-position of the snake's "true" head	25
heading	direction the snake's head is pointing to (N/E/S/W)	4
shape	snake can be straight, bend to left, bend to right	3

Table 2: Parameter description for snake's configuration with corresponding number of possibilities for each parameter

For all of the 25 *positions*, there are not 4 *headings* and 3 *shapes* possible. The possible configurations for all *positions* are presented in Table 3 below.

position	Count	Number of configurations/ snake position					
		heading	*shape*				**Total**
			h_1	h_2	h_3	h_4	
corner_1	4	2	2	2			2+2=4
$\text{along side}_1^{(1)}$	8	3	3	2	1		3+2+1=6
$\text{along side}_1^{(2)}$	4	3	3	2	2		3+2+2=7
corner_2	4	4	3	3	2	2	3+3+2+2=10
along side_2	4	4	3	3	3	2	3+3+3+2=11
middle	1	4	3	3	3	3	3+3+3+3=12

Table 3: Configurations of the snake given the "true" head position. Position-index 1 indicates a pixel in connection to the grid and index 2 indicates one pixel away from the grid

The total number of snake configurations p_{snake} could be calculated by summing over the position frequency multiplied with the number of configurations/ snake position from Table 3 for each position i:

$$\begin{aligned} p_{snake} &= \sum_{i=1}^{5} \text{count}_i \times \text{configuration}_i \\ &= 4 \cdot 4 + 8 \cdot 6 + 4 \cdot 7 + 4 \cdot 10 + 4 \cdot 11 + 1 \cdot 12 = 188 \end{aligned} \tag{1}$$

Configurations of the apple

The apple could be located in any of the pixels in the grid except the ones that the already occupied. p_{apple} number of possible configurations of the apple could be calculated:

$$p_{apple} = 5 \cdot 5 - 3 = 22 \tag{2}$$

Configurations in the game

For each configuration of the snake, the apple can be in 22 different configurations, hence the total number of configurations in the game, K, could be calculated. K is found by multiplying the number of all possible configurations of the snake p_{snake} with the number of all possible configurations of the apple p_{apple}:

$$K = p_{snake} \cdot p_{apple} = 188 \cdot 22 = 4136 \tag{3}$$

Bellman optimality equation for the Q-function

The optimal Q-value $Q^*(s, a)$ is the expected quality of action a when in state s and from that point acting optimal. $Q^*(s, a)$ could be calculated using the state-action value Bellman optimality equation presented below:

$$Q^*(s, a) = \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma \max_{a'} Q^*(s', a') \right] \tag{4}$$

where $T(s, a, s')$ is the transition from state state s using action a to state s'. $R(s, a, s')$ is the reward and γ the discount.

A policy π is a stochastic rule that select actions based on states. The optimal policy π^* could be formulated in terms of optimal Q-value $Q^*(s, a)$ as shown in equation 5 below:

$$\pi^*(s) = a^* = \operatorname*{argmax}_a Q^*(s, a) \tag{5}$$

2a. Rewrite the Bellman optimality equation for Q as an expectation

The transition function could also be written as a probability $T(s, a, s') = P(s, a, s')$. Inserting in equation 4 gives:

$$Q^*(s, a) = \sum_{s'} P(s, a, s') \left[R(s, a, s') + \gamma \max_{a'} Q^*(s', a') \right] \tag{6}$$

The Bellman optimality equation for Q holds one equation for each state. Instead of summing over all possible next states, the expected value $\mathbb{E}$ could be inserted for a neater

notation. The definition of expected value follows:

$$\mathbb{E}[X] = \sum_{i=1}^{k} x_i p_i \tag{7}$$

By substituting using equation 7 in 6, the optimal Q-value could then be formulated

$$Q^*(s,a) = \mathbb{E}\left[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')\right] \tag{8}$$

or using timestep t notation:

$$Q^*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a} Q^*(S_{t+1},a) \;\middle|\; S_t = s,\ A_t = a\right] \tag{9}$$

where t is a discrete time step, S_t and A_t the state and action at t. R_{t+1} and S_{t+1} is the reward and state at $t+1$, dependent on S_t and A_t.

2b. Explanation of the Bellman optimality equation for Q

Equation 4 tell us for any state s, the expected long-term quality of the next state action pair (s,a) given the action is chosen in an optimal way.

2c. Component explanation of the Bellman optimality equation for Q

All components and concepts of the Bellman optimality equation for Q are explained below:

- Q^* is the estimate of the optimal Q-value, which is the expected "quality" or "utility" of taking action a when in state s
- The summation $\sum$ is there to calculate the cumulative rewards (immediate and future) so that the optimal Q-value is maximized w.r.t. rewards received over time and not only immediate rewards
- $T(s,a,s')$ is the transition function, i.e. the probability of transitioning to state s' from s using action a
- $R(s,a,s')$ is the reward received when going from state state s using action a to state s'. The reward is a simple number $R \in \mathbb{R}$
- γ is the discount rate parameter $0 \leq \gamma \leq 1$. It determines the present value of future rewards, adjusting so that a reward in the future is worth less than an immediate reward
- The goal of the agent is to maximize the total amount of reward it receives over time, hence the max component
- s is the current state
- a is the action from the current state
- s' is the state which the agent will end up in using action a when in state s

2d. Explain $T(s, a, s')$ for the small version of *Snake*

To explain the transition function for the small version of snake, we could look into two cases

1. Snake does not eat the apple
2. Snake eat the apple

If the snake does not eat the apple, there is only one possible new state the game can be in given a current state s and a specified action a. In terms of transition function, it means that the transition function will be $T(s, a, s') = 1$ for that only possible state s' and $T(s, a, s') = 0$ for all other values of s'.

If the snake eats the apple, the apple will appear in a new pixel in the grid. That yields for 22 possible new states s' according to equation 2. It further means that the transition function will be $T(s, a, s') = \frac{1}{22}$ for all of those possible new states s' and $T(s, a, s') = 0$ for all other (impossible) new states s'.

The *truncated* game

Then a *truncated* game version was studied. Proceeding from the *small Snake* game and assuming the following reward scores:

	Reward
snake dies	-1
snake eats apple	$+1$
otherwise	0

and $\gamma \in (0, 1)$. In this *truncated* game version, the game ends when reaching score $+1$ (snake eats an apple).

The goal for an optimal agent is to maximize the reward. In the *truncated* game that means for the snake to eat an apple. Since the length of the snake does not increase in the *small Snake* game, an optimal agent in the *truncated* game would act as an optimal agent in the normal *small Snake* game as well.

A terminal state is the final state before the game ends. In the *truncated* game the terminal states are all the possible states where the snake eats the apple or die by hitting the wall.

3a. Problem arising

A trajectory was presented in the assignment description [1]. At timestep T in the trajectory the snake eats the apple and $Q^*(s_T, a) = 0$. In the timesteps before that, the snake was one step away from the apple $(T - 1)$ and two step away from the apple $(T - 2)$.

An algorithm for dynamic programming applied to Q-value table learning was given in the assignment description [1]. $\mathbf{Q}^*(s_T, a_T) = 0$ was initialized for each terminal state S_T in the $\mathbf{Q}^*$-table. All other Q-value were unknown.

Using equation 8, the value for $Q^*(s_{T-1}, \text{forward})$ could be calculated:

$$Q^*(s_{T-1}, \text{forward}) = \mathbb{E}\left[R_T + \gamma \max_a Q^*(s_T, a)\right] = \mathbb{E}\left[1 + 0\right] = 1 \tag{10}$$

Equation 10 show that the algorithm would find the optimal path for the states leading up to the terminal state, i.e. for $T - 1$. If we then try to calculate two timesteps back $Q^*(s_{T-2}, \text{forward})$ we get

$$Q^*(s_{T-2}, \text{forward}) = \mathbb{E}\left[R_{T-1} + \gamma \max_a Q^*(s_{T-1}, a)\right] = \mathbb{E}\left[0 + ?\right] = ? \tag{11}$$

since $\max_a Q^*(s_{T-1}, a)$ was unknown in the $\mathbf{Q}^*$-table. The problem that arised using the dynamic programming algorithm was that the optimal Q-value was unknown when we went two timesteps back, as the $\mathbf{Q}^*$-table was not updated. The problem implicates that the agent would not have enough information to make the optimal decision in the state s_{T-2}.

3b. Solution to the problem in 3a

In order to find the optimal solution, the $\mathbf{Q}^*$-table needs to be updated for each iteration. If the $\mathbf{Q}^*$-table was updated with the Q^*-value obtained in equation 10, equation 11 could be calculated:

$$Q^*(s_{T-2}, \text{forward}) = \mathbb{E}\left[R_{T-1} + \gamma \max_a Q^*(s_{T-1}, a)\right] = \mathbb{E}\left[0 + 1 \cdot \gamma\right] = \gamma \tag{12}$$

Policy iteration

The Bellman optimality equation for the state value function could be described by

$$V^*(s) = \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma\, V^*(s')] \tag{13}$$

Once there is a policy π, it could be evaluated by computing $V_\pi(s)$. The policy could then be improved to π' and then once again evaluated $V_{\pi'}(s)$. This is called policy iteration.

4a. Rewrite the Bellman optimality equation for V as an expectation

The Bellman optimality equation for V in equation 13 could be rewritten as an expectation in the same way as the Bellman optimality equation for Q described in section 2a. The transition function was written as a probability $T(s, a, s') = P(s, a, s')$ and then by using the definition of expected value presented in equation 7 we get

$$V^*(s) = \max_a \mathbb{E}\left[R(s, a, s') + \gamma\, V^*(s')\right] \tag{14}$$

or using timestep t notation:

$$V^*(s) = \max_a \mathbb{E}\left[R_{t+1} + \gamma\, V^*(S_{t+1}) \mid S_t = s,\ A_t = a\right] \tag{15}$$

where t is a discrete time step, S_t and A_t the state and action at t. R_{t+1} and S_{t+1} is the reward and state at $t+1$, dependent on S_t and A_t.

4b. Explanation of the Bellman optimality equation for V

Equation 14 tell us for any state s, the expected quality of the next state s', assuming that the action a was chosen in an optimal way.

4c. Explanation of the max-operator

The max-operator look at the expected values of the quality of the next state s' for *all* actions a. The purpose of the operator is then to return the maximum expected quality of a next state.

4d. The relation between $\pi^*(s)$ and V^*

The optimal policy $\pi^*(s)$ could be obtained by using the argmax-operatior in equation 14 instead of the max-operator. The argmax operator returns the *point* or *points* a instead of the *values* $f(a) = \mathbb{E}[R(s,a,s') + \gamma\, V^*(s')]$. The optimal policy $\pi^*(s)$ could therefor be formulated

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, \mathbb{E}\left[R(s,a,s') + \gamma\, V^*(s')\right] \tag{16}$$

4d. Difference between V^* and Q^*

$\mathbf{Q}^*$ is a $K \times 3$ matrix in which all $Q^*(s,a)$-values, one for each action, are saved. In other words $Q^*(s,a)$ keep track of which action a generated the expected value of the quality of a state-action pair. $\mathbf{V}^*$ on the other hand is a $K \times 1$ matrix, that keeps only the maximal expected value of the quality of a the next state. It means that $V^*(s)$ keep no information about what action a that actually generated the expected value.

π^* is the policy that will generate the optimal action a. Since V^* does not keep any information about a as opposed to Q^*, the relationship between π^* and V^* is less simple than for π^* and Q^*.

5a. Policy iteration code

A policy iteration code was run to find an optimal policy for the *small Snake* game. The policy iteration code that was filled in is presented below:

```
% POLICY EVALUATION

s_prime = next_state_idxs(state_idx, policy(state_idx));
if s_prime == -1
    v_new = rewards.apple;
elseif s_prime == 0
    v_new = rewards.death;
else
    v_new = rewards.default + gamm*values(s_prime);
end
Delta = max(Delta,abs(values(state_idx)-v_new));
values(state_idx) = v_new;



% POLICY IMPROVEMENT

actions = next_state_idxs(state_idx, :);
Q = zeros(1,nbr_actions);

for i = 1:nbr_actions
    if actions(i) == -1
        Q(i) = rewards.apple;
    elseif actions(i) == 0
        Q(i) = rewards.death;
    else
        Q(i) = rewards.default + gamm*values(actions(i));
        end
end

[~, action] = max(Q);
if policy_stable && policy(state_idx) ~= action
    policy_stable = false;
end
policy(state_idx) = action;
```

5b. Effect of the discount γ

The number of policy iterations and evaluations for different γ was then investigated using the policy iteration code above. The tolerance was set to $\epsilon = 1$. The results are presented in Table 4.

γ	Policy iterations	Policy evaluations
0	2	4
1		∞
0.95	6	38

Table 4: Number of policy iterations and evaluations using different values of the discount parameter γ and policy evaluation tolerance $\epsilon = 1$

The final snake playing agent act differently for the various values of γ.

Table 4 show that for $\gamma = 0$ an optimal policy was obtained quickly. The agent however did not play optimally. The agent got stuck in a loop, only going left. The agent did not manage to eat any apple using this policy. The obtained optimal policy corresponds to choosing any action if agent is not one step away from dying or one step away from eating the apple. When the max-operator in Matlab is suppose to choose any action it will always choose the action with the lowest index. That action is $a =$ going left, hence the agent get stuck in the going-left-loop.

For $\gamma = 1$ the policy evaluation did not converge, hence the game could never be played. Equation 12 show that if the iterations however would converge, all state-action pairs that would not get the agent into the grid would have the same Q-value $Q^*(s, a) = 1$. The policy would get the agent stuck in the going-left-loop as it did for $\gamma = 0$, but additionally it would not even break the loop and go right if there was an apple next to it (since eating an apple would have the same utility as not eating an apple as long as the agent is not dead).

For $\gamma = 0.95$ the obtained policy got the agent to play optimally. The *Snake* playing agent went straight to the apples and managed not to die.

5c. Effect of the stopping tolerance ϵ

The number of policy iterations and evaluations for different values of the policy evaluation stopping tolerance ϵ was then investigated. The discount was set to $\gamma = 0.95$. The results are presented in Table 4.

ϵ	Policy iterations	Policy evaluations
10^{-4}	6	204
10^{-3}	6	158
10^{-2}	6	115
10^{-1}	6	64
10^{0}	6	38
10^{1}	19	19
10^{2}	19	19
10^{3}	19	19
10^{4}	19	19

Table 5: Number of policy iterations and evaluations using different values of the policy evaluation tolerance ϵ and discount parameter $\gamma = 0.95$

The agent snake seem to act optimally for all different values of ϵ presented in Table 5 even though the number of policy evaluations and policy iterations varied a bit.

Smaller ϵ generates a more accurate policy evaluation. A too small value will come with a higher cost, i.e. longer computation time. For $\epsilon \leq 1$, 6 policy iterations are enough to find the optimal policy π^*. Table 5 indicates that 10^0 was the largest tolerance that would still generate an optimal policy in only 6 iterations.

The computed Δ in the policy evaluation was always < 6. Therefor $\epsilon \geq 10$, will have a policy iteration for each policy evaluation, as the tolerance is larger than Δ.

Tabular Q-learning

6a. Q-update code

In this section tabular Q-updates was implemented. The code that was filled in with terminal and non-terminal updates is presented below:

```
# TERMINAL UPDATES

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);


# NON-TERMINAL UPDATES

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);
```

6b. Three attempts to train a *small Snake* playing agent

Then three attempts to train a *Snake* playing agent was performed by changing parameter configurations and rewards. The results are presented below in Table 6.

Attempt	Parameters			Rewards			**Score**
	γ	ϵ	α	default	apple	death	
1	0.9	0.1	0.5	0	1	-10	1775
2	0.9	0.1	0.5	0	10	-10	122
3	0.9	0.1	0.8	0	10	-10	1922

Table 6: Parameter and score configurations for each attempt to train a *small Snake* playing agent and the score for each attempt

For the first attempt the score was quite good. The reward for the death was set to -10 and the reward for the apple 1. It was the reason for the *Snake* playing agent to actually survive as long as it did. In attempt 2 the apple and death reward had the same absolute value. The reason was for the agent to be able to eat apples fast as well. It however made the agent worse in terms of scores. The *Snake* agent died much faster.

For attempt 3 the learning rate parameter α was increased. $\alpha \in [0, 1]$ is the learning rate and it determines how much each Q-value is updated with in each step. $\alpha = 0$ means that the Q-values will not get updated and nothing could be learned. A high value on α means it could learn quickly but it could also loose accuracy. The results show that increased α made the agent perform better.

The $\epsilon \in [0, 1]$-parameter is there to guarantee some randomness. It was not changed in these three tests but a pure greedy approach, $\epsilon = 0$ means that the highest Q-value would

always be chosen and there would be a risk of getting stuck in a local optima. By having the ϵ parameter, random actions could instead be chosen with a probability ϵ. I would be careful with having ϵ too large as some structure in the search is preferable. It should not be too small either because of the risk to get stuck in local optima.

The discount rate $\gamma \in [0, 1]$ was not changed in these three tests either, but it determines the present value of future rewards. I believe that it should be quite high so that the agent is not shortsighted.

6c. Final settings

A few more parameter and reward score configurations were tested. The final settings chosen for the parameter and rewards to train a *Snake* playing agent was:

Parameters	Rewards
$\gamma = 0.9$	*default*: 0
$\alpha = 0.7$	*apple*: 10
$\epsilon = 0.01$	*death*: -10

A good policy was able to be trained with tabular Q-learning using the parameters and rewards stated above. The test run with these settings did not terminate and the score just kept increasing. It means that the *Snake* playing agent learned most importantly how not to die. The small value on ϵ made a the agent take the greedy action selection in 99% of the cases and the algorithm (usually) selected the next action with the best reward. The choice of this parameter trained a "safe" agent, with a good enough policy even though it may have not been an optimal policy.

6d. Difficulties in training the agent within 5000 episodes

For the above tests, the agent was trained for 5000 episodes. As derived in section 1 of this report, the state space consists of 4136 different configurations which is almost as many states as training episodes. The difficulties with training the agent within 5000 episodes would be that if we let the agent try out different policies the trade-off would be the learning. It was hard to make the agent try more random actions and still get a good score in the test. By using a small ϵ and large α a local optimal policy could be found quickly thus the agent had time to learn the policy.

Q-learning with linear function approximations

In this section the full version of the *Snake* game was studied. Due to a much larger state space, the *Snake* playing agent was trained using Q-learning with linear function approximations.

7a. Q-weight update code

The Q-weight update was implemented. The code that was filled in with terminal and non-terminal updates is presented below:

```
# TERMINAL UPDATES

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


# NON-TERMINAL UPDATES

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);
```

7b. Three attempts to train a *Snake* playing agent

State-action features $f_i(s, a)$ was then implemented.

Feature function f_1

The first feature f_1 evaluates whether the action would lead the agent closer to the apples (+1), or not (-1 for the first version $f_1^{(1)}$ and 0 for the second version $f_1^{(2)}$). The goal of the agent is to reach the apples which means that a good weight should be positive and weights for this feature was therefor initialized to -1.

```
% FEATURE 1 version 1
%   - closer to apple?

apple_dist_2 = norm([apple_m-next_head_loc(1), ...
   apple_n-next_head_loc(2)],2);
apple_dist_1 = norm([apple_m-prev_head_loc(1), ...
   apple_n-prev_head_loc(2)],2);

if apple_dist_1 > apple_dist_2
    state_action_feats(1, action) = 1;
else
    state_action_feats(1, action) = -1;
```

```
end


% FEATURE 1 version 2
%   - closer to apple?

apple_dist_2 = norm([apple_m-next_head_loc(1), ...
   apple_n-next_head_loc(2)],2);
apple_dist_1 = norm([apple_m-prev_head_loc(1), ...
   apple_n-prev_head_loc(2)],2);

if apple_dist_1 > apple_dist_2
    state_action_feats(1, action) = 1;
else
    state_action_feats(1, action) = 0;
end
```

Feature function f_2

The second feature f_2 investigates if the action would get the snake into a wall or into itself and therefor die (-1) or not (0). The weight should be positive since the *Snake* agent are not supposed to die, hence the weight or this feature was initially set to -1.

```
% FEATURE 2
%   - die?

if grid(next_head_loc(1), next_head_loc(2)) > 0
    state_action_feats(2, action) = 1;
else
    state_action_feats(2, action) = 0;
end
```

Feature function f_3

The third feature f_3 checks whether an action could lead to the *Snake* agent to eat an apple in the following action ($+1$) or not (0). The weight should be positive as the possibility of eating an apple in the next step is something good, hence the weigh was initially set to -1.

```
% FEATURE 3
%    - eating an apple in next action?

if (grid(next_head_loc(1),next_head_loc(2)) == -1)
    state_action_feats(3,action) = 1;
else
    state_action_feats(3,action) = 0;
end
```

Attempts were then performed to train a *Snake* playing agent. Three different configurations with corresponding average test scores after 100 tests are presented in Table 7.

Attempt	Parameters			Rewards			Features				Weights	**Score**
	γ	ϵ	α	default	apple	death	$f_1^{(1)}$	$f_1^{(2)}$	f_2	f_3		**(average)**
1	0.9	0.4	0.01	0	1	-1	$\surd$		$\surd$		[0.47,0.89]	17.08
2	0.9	0.4	0.01	0	1	-1		$\surd$	$\surd$		[0.42,0.74]	42.52
3	0.95	0.1	0.5	0	10	-10		$\surd$	$\surd$	$\surd$	[7.1,17,0.56]	42.52

Table 7: Parameter, reward and feature configurations for each attempt to train a *Snake* playing agent and the average score for 100 tests for each attempt

Table 7 show that the first attempt using the first version of the feature f_1 did not get a very good average score. By changing so that the feature of not getting closer to an apple was instead set to 0, the average score got much better, see attempt 2. The though behind setting the feature of not getting closer to an apple to -1 was to make the *Snake* agent more optimal, always taking the shortest path. The results in Table 7 indicates that the focus should rather be on not dying and eating apples, than to eat apples fast.

For the last attempt, the best version of feature f_1, namely $f_1^{(2)}$ was used together with feature f_2 and f_3. The average score after 100 tests runs was however the same as for attempt 2 which indicates that the last feature was redundant.

Final settings

Some more configurations were tested. The average test score after 100 game episodes with the final settings was 42.52. The final settings to train a *Snake* playing agent was then chosen and presented below:

Parameters	Rewards
$\gamma = 0.9$	*default*: 0
$\alpha = 0.01$	*apple*: 1
$\epsilon = 0.4$	*death*: -1

The feature functions, initial weights and obtained final weights are presented below in Table 8.

Features	Initial weights	Final weights
$f_1^{(2)}$: closer to apple	-1	0.4172
f_2: die	-1	0.7431
	$\mathbf{w}_0 = [-1, -1]$	$\mathbf{w} = [0.4172, 0.7431]$

Table 8: Features, initial weights and obtained final weights for the final settings

The used features $f_1^{(2)}$ and f_2 are presented below:

```
% FEATURE 1 version 2
%   - closer to apple?

apple_dist_2 = norm([apple_m-next_head_loc(1), ...
   apple_n-next_head_loc(2)],2);
apple_dist_1 = norm([apple_m-prev_head_loc(1), ...
```

```
   apple_n-prev_head_loc(2)],2);

if apple_dist_1 > apple_dist_2
    state_action_feats(1, action) = 1;
else
    state_action_feats(1, action) = 0;
end


% FEATURE 2
%   - die?

if grid(next_head_loc(1), next_head_loc(2)) > 0
    state_action_feats(2, action) = 1;
else
    state_action_feats(2, action) = 0;
end
```

Different parameter configurations and reward scores were tested using these features and it seemed to always lead to the average score 42.52 during the 100 test game episodes. The reason behind it was that the *Snake* trapped itself as it got longer. The conclusion drawn from this was the the optimal weights could quite easily be learned with respect to the implemented features. A way to increase the average score would therefor be to create a feature that targets this issue. It could for example be implemented by counting the number of free pixels in connection to the head and punish when it is too few in relation to the rest of the free grid pixels.

References

[1] FMAN45 Machine Learning *Assignment Instructions: Assignment 4* Spring 2021. LTH. Downloaded: 2021-05-14. https://canvas.education.lu.se/courses/11000/files/1221084?wrap=1