FMAN45 - Machine Learning Assignment 4

Erik Waldemarson

1 Tabular methods

1.1 Exercise 1

For every given snake configuration the apple has: 5 x 5 - 3 = 22 states.

The snake can either be vertical, horizontal or L-shaped. If it's vertical then there are 3 x 5 possible states for the snake. The problem is symmetric for horizontal so for vertical & horizontal there are 2 x 3 x 5 states. For an L-shaped snake there are 4 x 4 possible states. There are 4 unique rotations for an L-shaped snake so for L-shape there are 4 x 4 x 4 states. So in total there are 2 x 3 x 5 + 4 x 4 x 4 = 94 possible states for the snake.

Accounting for the position of the head doubles the states of the snake: 2 x 94 = 188 states.

Total states = 22 x 188 = 4136.

2 Bellman optimality equation for the Q-function

2.1 Exercise 2

a) $T(s, a, s')$ is the transition probability of going to state s' given that we have taken action a in state s:

$$T(s, a, s') = \mathbb{P}(s'|s, a) \tag{1}$$

Then we use the fact that $\mathbb{E}[X|Y] = \sum_i \mathbb{P}(X = x_i|Y)x_i$ and simplify:

$$\begin{aligned} Q^*(s, a) &= \sum_{s'} [\mathbb{P}(s'|s, a)R(s, a, s') + \gamma \mathbb{P}(s'|s, a) \max_{a'} Q^*(s', a')] \\ &= \sum_{s'} \mathbb{P}(s'|s, a)R(s, a, s') + \gamma \max_{a'} \sum_{s'} \mathbb{P}(s'|s, a)Q^*(s', a') \\ &= \mathbb{E}[R(s, a, s')] + \gamma \max_{a'} \mathbb{E}[Q^*(s', a')], \end{aligned} \tag{2}$$

where I've made the simplification $E[\cdot|s,a] = E[\cdot]$ since it can be inferred from $Q^*(s,a)$.

b) The immediate reward at time-step t is given by $R(s_t, a_t, s') = r_{t+1}$ and $\max_{a'} Q(s', a') = r_{t+2} + \gamma r_{t+3} + \gamma^2 r_{t+4} + ...$ so the expression becomes:

$$Q^*(s,a) = \sum_{k=0}^{\infty} \sum_{s'} \mathbb{P}(s'|s_t = s, a_t = a)\gamma^k r_{t+k+1} = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s, a_t = a]. \tag{3}$$

c) The optimal value of taking an action in a certain state is the expected immediate reward of taking that action plus the discounted expected reward of all future actions assuming we act optimally.

d) We assume that we are following the optimal policy, since we are calculating the value of the future as $\max_{a'} Q^*(s', a')$, i.e. the maximum of all possible future actions.

e) γ is a discount factor and tells how much we value the rewards of future actions. $\gamma = 1$ we care about the long-term a lot while $\gamma = 0$ means we only care about the immediate future.

f) Snake is mostly a completely deterministic game, the only random element is the placement which can be placed in 22 possible places for a given snake configuration. There are three additional possible states with probability 0, i.e. the apple respawning on a square already occupied by the snake. Thus the transition probability becomes

$$T(s,a,s') = \mathbb{P}(s'|s,a) = \begin{cases} \frac{1}{22}, & \text{if the snake eats the apple in state } s, \\ 0, & \text{the apple being on a square occupied by snake,} \\ 1, & \text{otherwise.} \end{cases} \tag{4}$$

2.2 Exercise 3

a) Off vs on-policy is essentially exploitation vs exploration.

On-policy means that an active agent is using the same policy to explore the world and find to learn an optimum policy, and is thus more exploitative.

Off-policy means that an active agent is using different policies to explore the world and to learn an optimum policy, and is thus more exploratory.

b) Model-based (MB) reinforcement learning tries to estimate a model for $T(s,a,s')$ and $R(s,a,s')$ and then create an optimum policy using that model.

Model-free (MF) reinforcement learning tries to directly learn an optimal policy $\pi^*(s)$ by trial-and error (direct mapping).

c) Passive reinforcement learning means that the agents acts according to a fixed policy $\pi(s)$ and we try to either estimate the model (MB) or the Q-value

function (MF). I.e. the agent is told what to do from an outsider and is punished/rewarded based on that action.

Active reinforcement learning means that the agent chooses actions to find an optimal policy (and perhaps corresponding value functions).

d) Supervised learning: Learn a mapping from input data to output data based on labeled examples.

Unsupervised learning: Find patterns or structure in data that is not labeled.

Reinforcement learning: Learn from the environment through trial and error, given a reward function.

e) The main difference is that dynamic programming uses a model-based learning approach and iteratively improves the optimal policy, while reinforcement learning does not use a model and learns the optimal policy from trial and error.

3 Policy iteration

3.1 Exercise 4

a) Again from (1) we have $T(s, a, s') = \mathbb{P}(s'|s, a) =>$

$$\begin{aligned} V^*(s) &= \max_a \sum_{s'} \mathbb{P}(s'|s, a)[R(s, a, s') + \gamma V^*(s')] \\ &= \max_a \mathbb{E}[R(s, a, s') + \gamma V^*(s')]. \end{aligned} \tag{5}$$

b) The best value of the current state is the expected value of the immediate reward plus the discounted expected value of all next states, with respect to the best possible action that can be taken in the current state.

c) We want to find the best value of the current state. An underlying assumption is that the best value corresponds to the best action otherwise it doesn't really make sense.

d) Bellman optimality equation.

$$\pi^*(s) = \operatorname*{argmax}_a \mathbb{E}[R(s, a, s') + \gamma V^*(s')] \tag{6}$$

e) The Q-value is a function of the action-state pair, while the V-value is only a function of the state. So for Q one can simply take the argmax, while for V one has to consider the immediate reward and V-value for future states.

4 Policy evaluation and improvement

We solve this problem using policy iteration and evaluation. We set the snake eating the apple and dying as terminal states and thus the transition probability $T(s, a, s') = 1$ for all cases. We also use a deterministic policy $a = \pi(s)$. The update rule becomes

$$V_{k+1}(s) = \begin{cases} R(s, \pi(s), s'), & s' = \text{'death' or 'apple'}, \\ R(s, \pi(s), s') + \gamma V_k(s'), & \text{otherwise.} \end{cases} \tag{7}$$

The policy evaluation similarly simplifies from (6) to

$$\pi_{k+1}(s) = \underset{a}{\operatorname{argmax}} \begin{cases} R(s, a, s'), & s' = \text{'death' or 'apple'}, \\ R(s, a, s') + \gamma V_k(s'), & \text{otherwise.} \end{cases} \tag{8}$$

And then repeat until convergence.

4.1 Exercise 5

a) I implement the policy iteration and evaluation:

```
% FILL IN POLICY EVALUATION WITHIN THIS LOOP.
        V_state_old = values(state_idx); %old V-value state

        action = policy(state_idx);
        next_state = next_state_idxs(state_idx,action); %s'

        if next_state == -1 %if snake eats an apple then terminal
            values(state_idx) = rewards.apple;
        elseif next_state == 0 %if snake hits wall then terminal
            values(state_idx) = rewards.death;
        else %snake moves normally
            values(state_idx) = rewards.default + gamma.*values(next_state);
        end

        if abs(values(state_idx) - V_state_old) > Delta
            Delta = abs(values(state_idx) - V_state_old);
        end
```

```
% FILL IN POLICY IMPROVEMENT WITHIN THIS LOOP.
    V_actions = zeros(1, nbr_actions);

    for action = 1:nbr_actions

        next_state = next_state_idxs(state_idx,action);
        if next_state == -1 %if snake eats an apple then terminal
            V_actions(action) = rewards.apple;
        elseif next_state == 0 %if snake hits wall then terminal
            V_actions(action) = rewards.death;
        else %snake moves normally
            V_actions(action) = rewards.default + gamma.*values(next_state);
        end
    end
    %pi(s) = argmax_a ...
    [~,best_action] = max(V_actions);

    if best_action ~= policy(state_idx)
        policy_stable = false;
    end

    policy(state_idx) = best_action;
```

With $\gamma = 0.5$ and $\epsilon = 1$, I get 6 policy iterations and 11 policy evaluations.

b) I set $\epsilon = 1$ and try three different γ. The results have been gathered in Table 1.

As one can see the behaviour is as expected. With $\gamma = 0$ the snake cares only for the short-term reward. For $\gamma = 1$ the infinite sum similar to (3) becomes divergent, basically the snake tries to predict an infinite number of moves in the future and so doesn't converge. For $\gamma = 0.95$ the snake considers both the immediate reward and the long-term reward.

Table 1: Number of policy iterations and evaluations for different γ with $\epsilon = 1$.

γ	Nr. Policy iter.	Nr. Policy eval.	Behaviour
0	2	4	Stuck in an infinite loop.
1	$+\infty$	$+\infty$	Doesn't converge.
0.95	6	38	Plays optimally.

c) I set $\gamma = 0.95$ and try three different ϵ. The results have been gathered in Table 2. As one can see when increasing the tolerance ϵ the number of policy

evaluations decrease as expected while the number of policy iterations increase to compensate.

Table 2: Number of policy iterations and evaluations for different ϵ with $\gamma = 0.95$.

ϵ	Nr. Policy iter.	Nr. Policy eval.	Behaviour
10^{-4}	6	204	Plays optimally.
10^{-3}	6	158	Plays optimally.
10^{-2}	6	115	Plays optimally.
10^{-1}	6	64	Plays optimally.
10^{0}	6	38	Plays optimally.
10^{1}	19	19	Plays optimally.
10^{2}	19	19	Plays optimally.
10^{3}	19	19	Plays optimally.
10^{4}	19	19	Plays optimally.

5 Tabular Q-learning

The previous methods become very difficult to use when the state space becomes increasingly big, as for the real game snake. In that case we will use Q-learning, which means that we update our Q-value with the following update rule:

$$Q^{\pi}(s,a) \leftarrow Q^{\pi}(s,a) + \alpha[\text{difference}], \tag{9}$$

where

$$\text{difference} = \begin{cases} R(s,a,s') - Q^{\pi}(s,a), & s' \text{ is terminal,} \\ R(s,a,s') + \gamma \max_{a'} Q^{\pi}(s',a') - Q^{\pi}(s,a), & \text{otherwise.} \end{cases} \tag{10}$$

The variable $\alpha \in [0,1]$ is the learning rate, a higher learning means that q^{π} adapts to new information more quickly and vice versa. There is also another variable $\epsilon \in [0,1]$ which states how random we are when exploring the action space. A higher ϵ means that we are more exploratory while a lower means that we are more exploitative.

5.1 Exercise 6

```
%Terminal state
    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
    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;
```

I will present the final model and then going over three changes I made along the way.

My final model keeps eating infinitely and does so optimally (in my opinion). The settings I used have been collected in Table 3.

One of the most important parameters were the rewards. When 'default' = 0 the snake would just go into an infinite loop so you have to punish it.

This problem is difficult to solve since there are up to 10 parameters that you can use to optimize and you basically have to experiment to reach a good value since (to my knowledge) there doesn't exist any better optimization methods than trial and error.

Table 3: Final settings

Hyperparameter	Value
`nbr_iter`	5000
`rewards`	{'default', -1, 'apple', 2, 'death', -100}
γ	0.9
α	0.5
ϵ	0.01
α-`update_iter`	0
α-`update_factor`	0
ϵ-`update_iter`	0
ϵ-`update_factor`	0

6 Q-learning with linear function approximation

We use the linear function approximation

$$Q(s,a) \approx Q_w(s,a) = \mathbf{w}^T \mathbf{f}(s,a), \tag{11}$$

where $\mathbf{w}$ is a vector weights w_i and $\mathbf{f}(s,a)$ is a vector of functions $f_i(s,a)$ known as the state-action feature functions.

Instead of updating Q directly we instead use the update rule

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[\text{difference}]\mathbf{f}(s,a), \tag{12}$$

where difference is the same as in (10).

6.1 Exercise 7

```
%Terminal state
    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 state
    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);
```

First I will show you my final solution, then I will some of the more interesting things I found while experimenting.

The model I used had the four state-action feature functions. Define the vector v_{apple} as the direction vector from the head of the next position to the apple. The first feature function is:

$$f_1(s,a) = \frac{||v_{\text{apple}}||_1}{2(N-2)}, \tag{13}$$

i.e. the Manhattan distance from next head position to the the apple.

The second state action-feature function was defined as

$$f_2(s,a) = \begin{cases} -1, & \text{next move leads to terminal state death,} \\ 0, & \text{otherwise.} \end{cases} \tag{14}$$

This one was implemented to prevent the snake from dying all the time.

The third and fourth state-action feature functions were defined as

$$f_3(s,a) = \frac{\text{distance to closest wall in next direction}}{(N-2)}, \tag{15}$$

and

$$f_4(s,a) = \frac{\text{distance to snake segment in next direction}}{(N-2)}, \tag{16}$$

where $f_4(s,a) = 0$ if there is no segment in the next direction.

The code for the $\mathbf{f}$ has been collected below:

```
[next_head_loc, next_move_dir] = get_next_info(action, movement_dir, head_loc);

[apple_loc_m, apple_loc_n] = find(grid == -1);
%direction vector from head to apple
head_to_apple = [apple_loc_m, apple_loc_n] - next_head_loc;

state_action_feats(1, action) = norm(head_to_apple, 1) / (2*(N-2));

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

if next_move_dir == 1 % NORTH
    %looking up
    path = grid([1, next_head_loc(1)], next_head_loc(2));

elseif next_move_dir == 2 % EAST
    %looking right
    path = grid(next_head_loc(1), [next_head_loc(2), N]);

elseif next_move_dir == 3 % SOUTH
    %looking down
    path = grid([next_head_loc(1), N], next_head_loc(2));

else % WEST
    %looking left
    path = grid(next_head_loc(1), [1, next_head_loc(2)]);
end
```

```
%average distance to wall
state_action_feats(3, action) = length(path)/ (N-2);

path = path(1:length(path)-1);
dist_to_body = find(path == 1, 1);

if dist_to_body
    state_action_feats(4, action) = dist_to_body / (N-2);
else
    state_action_feats(4, action) = 0;
end
```

The settings for the training have been collected in Table 4. The initial weights $\mathbf{w_0} = [1, -1, 1, 1]^T$.

The test average was 36.23 and final weights $\mathbf{w} = [-9.2273, 90.2055, -101.8975, -0.0012]^T$.

I believe it works well because it always has an awareness of the distance to the apple by f_1, it's prevented from killing itself by f_2, and it has an idea of the distance to wall and itself by f_3. For the learning parameters I think having a dynamic learning rate ϵ is good because it's allowed to explore more in its infancy and and then exploit good strategies it has already explored (like a child). I set a very high 'death' punishment to really incentivise it to survive as long as possible.

Table 4: The settings used to train the final snake model.

Hyperparameter	Value
`nbr_iter`	5000
`nbr_feats`	4
`rewards`	{'default', -1, 'apple', 1, 'death', -100}
γ	0.9
α	0.1
ϵ	0.01
α-`update_iter`	500
α-`update_factor`	0.5
ϵ-`update_iter`	500
ϵ-`update_factor`	0.5

Here are some interesting changes I made along the way. For the results all other settings remained the same:

6.1.1 Changing normalization with tanh to just a constant.

In some of my early attempts I set $f_1(s, a) = \tanh(||v_{\text{apple}}||_1)$. This resulted in the snake kept running into an infinite loop if the apple was too far away from it and I couldn't even finish training because it took too long.

I believe the reason is due to the non-linearity of tanh. When the apple is far-away $\tanh(\cdot) \approx 1$ constantly, and thus the snake barely makes a benefit when making a move closer to the apple.

6.1.2 Changing distance to apple from L2 to L1.

For $f_1(s, a)$ I used to previously use the L2-norm instead of L1. I found that the snake moved a lot along the diagonal in a zig-zag pattern which made it make risky mistakes for the apple. So I decided to change to the Manhattan distance to encourage it to take more straight paths around instead.

The average test result before changing was 30.24.

6.1.3 Changing reward for 'apple' from 3 to 1.

I found that the snake was too greedy for the apple, which made it make stupid mistakes because it was taking the shortest path to the apple. I decided to make the reward of the apple lower to compensate.

The average test result before changing was 35.66.