FMAN45 Machine Learning

Assignment 4

(Van Duy Dang - va7200da-s)

May 22, 2023

1 Reinforcement learning for playing Snake.

1.1 Tabular methods.

Exercise 1:

In this task, we find the number of states K in the small Snake game. With a 7x7 grid and the constant length of 3 of the snake, the snake can move within the 5x5 grid. Therefore, we have (5 x 5 - 3) = 22 apple locations. Below are 6 sample configurations of the snake.

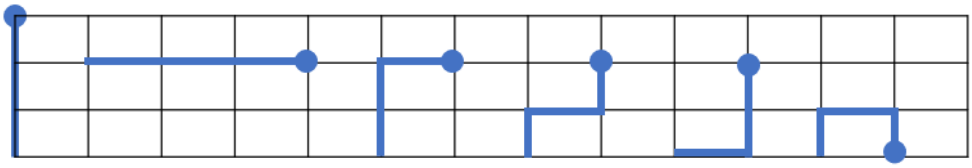

Figure 1: Configurations of the snake

The first 2 cases have 2 x 15 = 30 configurations (15 configurations each). The remaining 4 cases have 4 x 16 = 64 configurations (16 configurations each). In addition, we need to consider cases where the head is on the other side of the snake. Considering all above cases, we compute K = (30 + 64) * 2 * 22 = 4136.

1.1.1 Bellman optimality equation for the Q-function.

Exercise 2:

The optimal Q-value $Q^*(s,a)$ for a given state-action pair (s,a) is given by the state action value Bellman optimality equation

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

Once we have $Q^*(s,a)$, an optimal policy π^* is given by

$$\pi^*(s) = a^* = \underset{a}{\text{argmax}} Q^*(s,a) \tag{2}$$

For a given policy π the corresponding Q-value is given by the Bellman equation

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

a) We want to rewrite equation (1) as an expectation using the notation $\mathbb{E}[.]$, where $\mathbb{E}[.]$ denotes expected value. The expected value of a discrete random variable is calculated by summing the products of each possible value of X and its corresponding probability.

$$\mathbb{E}[X] = \sum_{i=1}^{n} x_i p(x_i) \tag{4}$$

Apply equation (4) to (1), we can rewrite equation (1) as

$$Q^*(s,a) = \mathbb{E}_{T(s,a,S')}\left[R(s,a,S') + \gamma \max_{a'} Q^*(S',a')\right] \tag{5}$$

b) Refer to lecture 8, we can rewrite equation (1) as an infinite sum without recursion

$$Q^*(s,a) = \mathbb{E}_{\pi^*}[r_{t+1} + (\gamma)r_{t+2} + (\gamma)^2 r_{t+3} + ...|s_t = s, a_t = a] \tag{6}$$

c) The state action value Bellman optimality equation defines the optimal Q-value for a given state-action pair as the expected value of the sum of the immediate reward $R(s,a,s')$ and the discounted maximum Q-value achievable from the next state $\gamma \max_{a'} Q^*(s',a')$.

d) In equation (1), we assume that we are following the optimal policy π^* and maximize over all actions to find the optimal action. In equation (3), we consider a specific policy π, which could be any policy, and calculate the Q-values based on the actions selected according to that policy.

In equation (1), we make an assumption in the maximization step $\max_{a'} Q^*(s',a')$. In equation (3), the assumption is made in the expression $\pi(a'|s')$ which represents the probability of selecting action a' in state s' according to the policy π.

e) $\gamma \in [0,1]$, is the discount factor that determines the importance of future rewards compared to immediate rewards.

A higher gamma value (close to 1) increases the weight given to future rewards in the Q-values. The agent considers long-term consequences and is more patient in its decision-making. The evaluation is based on the total sum of its future rewards. In contrast, a lower gamma value (close to 0) assigns more importance to short-term rewards, making the agent more myopic and concerned only actions with immediate rewards.

f) The term $T(s,a,s')$ represents the transition probability from state s to state s' when action a is taken.

$T(s,a,s') = 1$ if there is a 100% chance of reaching state s' when action a is taken.

$T(s,a,s') = 0$ if it is impossible to reach state s' when action a is taken.

$T(s,a,s') = 0.66$ if there is a 66% chance of successfully transitioning to state s' when action a is taken.

Exercise 3:

a)

- On-policy methods use the same policy for both behaviour and learning. The behavior policy, which determines the agent's actions during exploration, is also the target policy that is being learned. This means that the agent learns the effectiveness of the policy it is currently using, including the exploration actions. The behavior policy is often designed to be "soft" and non-deterministic, ensuring continuous exploration.

- On the other hand, off-policy methods use different policies for behaviour and learning. The target policy is learned independently of the actions chosen during exploration. The agent follows a specific policy but it learns and evaluates the effectiveness of a different policy.

b)

- In the Model-Based approach, the primary objective is to learn/estimate the transition $P(s'|s,a)$ and the reward function $R(s,a,s')$. Model-Based methods can utilize Markov Decision Process (MDP) algorithms to compute optimal policies $\pi^*(s)$.

- In contrast, the Model-Free approach avoids explicitly learning/estimating the model of the environment. Instead, it focuses on directly learning the optimal policy $\pi^*(s)$ or value functions (such as $V^*(s)$ and/or $Q^*(s,a)$).

c)

- In active reinforcement learning, the agent explores the environment by taking actions, receives feedback in the form of rewards, and learns from these experiences to improve its policy or value functions. The agent actively explores and influences the environment by selecting actions based on its current policy or exploration strategy.

- Passive reinforcement learning, on the other hand, refers to scenarios where the agent does not have control over its actions during the learning process. The agent observes and learns from samples from the environment to evaluate the fixed policy.

d)

- Reinforcement learning: an agent learns to make sequential decisions in an environment to maximize cumulative rewards. It involves an agent interacting with an environment, receiving feedback in the form of rewards, and learning through trial and error to discover optimal actions or policies.

- Supervised learning: a model learns to map input data to corresponding output labels based on labeled training examples. In supervised learning, the model is trained using pairs of input-output examples, and its objective is to generalize and accurately predict the output labels for new, unseen input data.

- Unsupervised learning: a model learns patterns, structures, or representations from unlabeled data without explicit output labels. The goal of unsupervised learning is to discover hidden structures or relationships in the data, such as clustering similar data points or extracting meaningful features.

e)

The main difference between using a dynamic programming approach (such as policy iteration), and using reinforcement learning (such as Q-learning) when solving an MDP problem, is how they handle the learning and optimization process.

Dynamic programming approaches require knowledge of the complete MDP model and iteratively improve the policy and value functions based on the model's dynamics. Reinforcement learning methods do not require knowledge of the complete MDP model and learn directly from interactions with the environment, updating Q-values based on observed experiences.

1.1.2 Policy iteration.

The Bellman optimality equation for the state value function given by

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

Exercise 4:

a)

Similar to exercise 2, we rewrite (7) as

$$V^*(s) = \max_a \mathbb{E}_{T(s,a,s')} \left[R(s,a,s') + \gamma V^*(s')\right] \tag{8}$$

b)

The Bellman optimality equation states that the optimal value of a state is equal to the maximum expected sum of immediate rewards $R(s,a,s')$ and discounted future values $\gamma V^*(s')$ that can be achieved by taking the best action in that state.

c)

The max-operator is used to select the action that maximizes the expected sum of immediate rewards and discounted future values. In other words, it determines the optimal action to take in a given state.

d)

The optimal policy is given by

$$\pi^*(s) = \operatorname*{argmax}_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s')\right]$$

e)

V^* represents the maximum expected sum of immediate rewards and discounted future values, regardless of the specific actions taken. Therefore, π^* can be derived straightforwardly from V^* by selecting the actions that maximize the expected returns in each state.

On the other hand, Q^* represents the expected value of the sum of the immediate reward and the discounted maximum Q-value achievable from the next state. It requires an additional step of selecting the action with the highest expected return for each state, which introduces an extra layer of complexity.

Exercise 5:

Policy evaluation.

```
for state_idx = 1 : nbr_states
    value_old = values(state_idx);

    % Get the action chosen by the current policy.
    action = policy(state_idx);
    % Get the index of the next state.
    next_state_idx = next_state_idxs(state_idx,action);

```

```
    if next_state_idx == -1
        % Eat an apple.
        next_value = rewards.apple;
    elseif next_state_idx == 0
        % Death.
        next_value = rewards.death;
    else
        % Non-terminal state.
        next_value = rewards.default+gamm*values(next_state_idx);
    end

    % Update the value of the current state.
    values(state_idx) = next_value;

    % Update Delta
    Delta = max(Delta, abs(value_old-next_value));
end
```

Policy improvement.

```
for state_idx = 1 : nbr_states
    old_action = policy(state_idx);

    % Evaluate the expected return for each action.
    action_returns = zeros(1, nbr_actions);
    for action = 1 : nbr_actions
        next_state_idx = next_state_idxs(state_idx,action);

        if next_state_idx == -1
            % Eat an apple.
            action_returns(action) = rewards.apple;
        elseif next_state_idx == 0
            % Death.
            action_returns(action) = rewards.death;
        else
            % Non-terminal state.
            action_returns(action) = rewards.default+gamm*values(
    next_state_idx);
        end
    end

    % Find the optimal action.
    [~, optimal_action] = max(action_returns);
    policy(state_idx) = optimal_action;

    % Check if the policy has changed.
    if old_action ~= optimal_action
        policy_stable = false;
    end
end
```

γ	**Number of policy iterations**	**Number of policy evaluations**
0	2	4
1	NA	∞
0.95	6	38

Table 1: Number of policy iterations and evaluations for γ.

b)

With $\gamma = 0$, the agent focuses on immediate rewards and does not consider the long-term consequences of its actions. The snake goes in a circle without caring about the apple.

With $\gamma = 1$, the agent considers future rewards with equal importance as immediate rewards. This causes the code to get stuck in the policy evaluation loop. We never find the optimal action.

With $\gamma = 0.95$, the agent balances the desire for immediate gains with considerations for future consequences. It explores the map and the snake can find and eat all the apples. This agent plays optimally compared to the previous ones.

c)

ϵ	**Number of policy iterations**	**Number of policy evaluations**
10^{-4}	6	204
10^{-3}	6	158
10^{-2}	6	115
10^{-1}	6	64
1	6	38
10^{1}	19	19
10^{2}	19	19
10^{3}	19	19
10^{4}	19	19

Table 2: Number of policy iterations and evaluations for ϵ.

For all values of ϵ from the table above, the agent's performance is really good. It plays optimally and the snake eats all the apples. When we increase ϵ, the number of policy evaluations will reduce as the algorithm converges quicker and the check for policy evaluation terminates earlier. This looser convergence leads to an increased number of policy iterations to find the optimal action.

1.1.3 Tabular Q-learning.

Exercise 6:

a)

Q-update code for terminal update.

```
sample                    = reward;
pred                      = Q_vals(state_idx,action);
td_err                    = sample - pred; % don't change this.
Q_vals(state_idx, action) = Q_vals(state_idx, action) + alph*td_err;
```

Q-update code for non-terminal update.

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

b)

- First attempt: I used $\alpha = 0.05$, $\epsilon = 0.001$ and rewards = struct('default', 0, 'apple', 1, 'death', -1), because I wanted the algorithm to reach a more precise convergence. It turned out that I got 0 points. I think the reason was that the model studied too slow so it could not converge.
- Second attempt: We have only 5000 iterations so I changed $\alpha = 1$, $\epsilon = 0.1$ and kept the default rewards to learn faster. Then I still got 0 points. Maybe the learning rate was too fast or still not enough for the model to converge.
- Third attempt: In this attempt, I used the same setting as the second attempt and lowered $\alpha = 0.6$ to check if the model could converge or not. Eventually, I got 125 points. It meant that the learning rate in the second attempt was too fast. After this attempt, if I try more to find better values of α, ϵ and rewards, I may get better results.

c)

I used $\alpha = 0.6$, $\epsilon = 0.1$, rewards = struct('default', 0, 'apple', 1000, 'death', -1000). I changed the apple reward to 1000 and death to -1000 to encourage the agent to eat the apples and punish it if it hits the wall. It turned out that my test run never ended and the score kept increasing. In summary, I was able to train an optimal policy for a small Snake game via tabular Q-learning.

d)

Because we have many parameters in this model, it's difficult to find a good set to train the model with a good convergence. For instance, if α is too small then the model studies too slow and it will never converge within 5000 iterations. Need to try a lot to find a good set.

1.2 Q-learning with linear function approximation.

Exercise 7:

In this exercise, I initialize init_weights to [-1;-1;-1]. I believe this is a bad way because it assigns a negative weight to Q value from linear predictor.

a)

Q weight update code for terminal update.

```
target  = reward;
pred    = Q_fun(weights, state_action_feats, action);
td_err  = target - pred; % don't change this
```

```
weights = weights + alph*td_err*state_action_feats(:, action);
```

Q weight code for non-terminal update.

```
target  = reward + gamm*max(Q_fun(weights, state_action_feats_future));
pred    = Q_fun(weights, state_action_feats, action);
td_err  = target - pred; % don't change this
weights = weights + alph*td_err*state_action_feats(:, action);
```

b)

Attempt 1:

```
% Feature 1: Calculate the distance from the apple to the next head location.
state_action_feats(1, action) = norm(apple_loc-next_head_loc);
state_action_feats(1, action) = distance_diff;
% Feature 2: Check if the snake can eat the apple in the next action
eat_apple = grid(next_head_loc(1), next_head_loc(2)) == -1;
state_action_feats(2, action) = eat_apple;
% Feature 3: Check if the snake hits the wall or eats itself
terminate = grid(next_head_loc(1),next_head_loc(2)) == 1;
state_action_feats(3, action) = -terminate;
```

I wanted the agent to study slowly with a good convergence so I used $\epsilon = 0.1$, $\alpha = 0.3$, $\gamma = 0.99$, rewards = struct('default', 0, 'apple', 1, 'death', -1).

The mean score after 100 episodes was 0. I observed the weights are NaN. It seemed that my feature 1 was too large.

Attempt 2:

I modified the feature 1 to calculate the difference in distance to the apple before and after moving.

```
% Feature 1: Calculate the difference in distance to the apple before and
    after moving.
distance_diff = norm(apple_loc-next_head_loc)-norm(apple_loc-prev_head_loc);
state_action_feats(1, action) = distance_diff;
% Feature 2: Check if the snake can eat the apple in the next action
eat_apple = grid(next_head_loc(1), next_head_loc(2)) == -1;
state_action_feats(2, action) = eat_apple;
% Feature 3: Check if the snake hits the wall or eats itself
terminate = grid(next_head_loc(1),next_head_loc(2)) == 1;
state_action_feats(3, action) = -terminate;
```

I wanted the agent to study slowly with a good convergence so I used $\epsilon = 0.1$, $\alpha = 0.3$, $\gamma = 0.99$, rewards = struct('default', 0, 'apple', 1, 'death', -1).

The mean score after 100 episodes was 28.72. The NaN error from state-action feature functions was fixed. It seemed that these ϵ and α were good. But I might need to change some parameters to improve the performance.

Attempt 3:

I updated the feature 1 to check if the snake moves toward the apple

```
% Feature 1: Check if the snake moves toward the apple.
distance_compare = norm(apple_loc-next_head_loc)<norm(apple_loc-prev_head_loc)
    ;
state_action_feats(1, action) = distance_compare;
% Feature 2: Check if the snake can eat the apple in the next action
eat_apple = grid(next_head_loc(1), next_head_loc(2)) == -1;
state_action_feats(2, action) = eat_apple;
% Feature 3: Check if the snake hits the wall or eats itself
terminate = grid(next_head_loc(1),next_head_loc(2)) == 1;
state_action_feats(3, action) = -terminate;
```

I used the same state-action feature functions as attempt 2. In addition, I use $\epsilon = 0.4$, $\alpha = 0.6$, $\gamma = 0.99$, rewards = struct('default', 0, 'apple', 1000, 'death', -1000).

This time I got only 42.52 points. It seemed that the model converged with these parameters.

c)

Final attempt:

I changed the way to check feature 1 and used $\epsilon = 0.01$, $\alpha = 0.7$, $\gamma = 0.95$, rewards = struct('default', 0, 'apple', 1, 'death', -1)

```
% Feature 1: Check if the snake moves toward the apple.
distance_compare = sum(abs(apple_loc - next_head_loc))>sum(abs(apple_loc -
    prev_head_loc));
state_action_feats(1, action) = -distance_compare;
% Feature 2: Check if the snake can eat the apple in the next action
eat_apple = grid(next_head_loc(1), next_head_loc(2)) == -1;
state_action_feats(2, action) = eat_apple;
% Feature 3: Check if the snake hits the wall or eats itself
terminate = grid(next_head_loc(1),next_head_loc(2)) == 1;
state_action_feats(3, action) = -terminate;
```

ω_1	ω_2	ω_3
-1	-1	-1

Table 3: Initial weights.

After training, I got $w_1 = 0$, $\omega_2 = 1$ and $\omega_3 = 1$.

ω_1	ω_2	ω_3
0	1	1

Table 4: Final weights.

Even though I used $\epsilon = 0.01$, I still got 42.52 points. In my opinion, that is the best point that my model can achieve. To get a higher score, I need better state-action features.