FMAN45 Machine Learning
-Assignment 4.

Nils Romanus

May 2022

1 Reinforcement learning for playing Snake

1.1 Tabular methods

Exercise 1

The K value can be derived by pondering how many configurations of apple- and snake there are. For each orientation of the grid the snake can have multiple configurations. If the snake is straight it can lie in three places for each row/column and have its head in two places. On the outer rows, if the snake is bent, it has eight possible orientations. In the middle rows it has sixteen possible orientations. There are two possible orientations of the grid. Since the snake takes up three spots in the 5×5 grid there are 22 possible locations for the apple. 2 grid orientations, 6 straight options on 5 rows, 8 bent options on the 2 outer rows, 16 bent options on the 3 inner rows and 22 apple locations yields $K = 4136$.

$$2 \times \big(2 \times 3 \times 5 + 2 \times 8 + 3 \times 16\big) \times 22 = 4136 \tag{1}$$

1.2 Bellman optimality equation for the Q-function

Exercise 2

a)
The optimal Q-value Q(s,a) for a given state-action pair (s,a) is given by the state- action value Bellman optimally equation 2, where T denotes the transition function or rather the probability of s leading to s' haven taken action a, and R the reward function.

$$Q^*(s,a) = \sum_{s'} T\left(s,a,s'\right)\Big[R\left(s,a,s'\right) + \gamma \max_{a'} Q^*\left(s',a'\right)\Big] \tag{2}$$

The definitions of the expectation of a discrete s.v X, with outcomes x_i with probabilities p_i, is displayed in expression 3.

$$E[X] = \sum_{i=1}^{n} x_i p_i \tag{3}$$

One may interpret the transition function from 2 as the probability of the reward R plus the future reward $\max_{a'} Q^*\left(s',a'\right)$ discounted by a factor γ when arriving at state s'. Where s' is an outcome from a possible set of states from the discrete s.v S' having taken the action a' which is an outcome from the possible set of future optimal actions from the discrete s.v A'. Noting the similarity between T from 2 and p_i, and s', a' and x_i we can rewrite 2 as an expectation using 3 as shown in 4.

$$Q^*(s,a) = E[R\left(s,a,S'\right) + \gamma \max_{A'} Q^*\left(S',A'\right)] \tag{4}$$

b)
...
c)
Equation 2 expresses that the optimal value of a Q state s, when taking action a, is given by the expected future utility, which is in turn given by the reward R and the future discounted rewards, given that the agent subsequently takes optimal the action a'.
d)
For a given policy the corresponding Q-value is given by the Bellman equation 5:

$$Q^{\pi}(s,a) = \sum T\left(s,a,s'\right)\left[R\left(s,a,s'\right) + \gamma Q^{\pi}\left(s', \pi\left(a' \mid s'\right)\right)\right] \tag{5}$$

The difference between equations 2 and 5 is that we in 2 assume that the optimal policy π^* is followed. This is expressed by the max operator on the discounted future Q value.
e)
The γ is the discounting factor used to represent how future rewards are less valuable than current rewards since we, in general, want to achieve the maximum amount of reward in as short amount of time as possible in practice. The γ makes future Q values $Q(s', a')$ less valuable than the current.
f)
The $T(s, a, s')$ is the transition function which can be interpreted as the probability that the state s leads to s' haven taken action a, i.e $P(s'|s, a)$. For the small version of the snake defined in exercise 1 we have the three different types of scenarios. $T(s, a, s') = 1$ if the snake starts in a state s that is not next the apple landing in an allowed position s' that is possible by taking the action a. $T(s, a, s') = 0$ under similar conditions but the action a makes the snake land in a position s' that is not possible. For example the head landing on a tile that is not next to the head. $T(s, a, s') = 1/22$ if the snake starts in a position next to the apple and the action a eats the apple and puts the snake in an allowed position. Since there are $25 - 3$ new positions for a new apple to appear with equal probabilities we arrive at the new state s' with a probability of $1/22$. $T(s, a, s)$ will of course also be zero for the states where the snake eats and apple and the apple re-appears in the snake or where the snake eats an apple and lands in an impossible position similar to the one that one previously described.

Exercise 3

a)
On-policy is fixing a set policy and writing the the sum of discounted rewards every time a state is visited in order to update the Q function. Off-policy is trying to converge to an optimal policy whilst (maybe) acting sub optimally. In off-policy the agent uses the experiance collected using different policies in order to converge to an optimal one.
b)

Model-based RL is learning an approximate model T through experiance and then solving the problem under the assumption that the learnt model is correct. Model-free RL, on the other hand, does not use T, rather it is using trial and error.

c)

In active RL we are trying to learn both the policy and the the values, in passive RL we just try to learn the values using a fixed policy. In active RL our learner *makes* choices as opposed to in passive where it only *evaluates* a fixed set of choices.

d)

The difference between supervised- and unsupervised learning an RL is that our machine acts in the RL case. In supervised- and unsuppervised our machine learns from a set of labeled our unlabeled data respectively, i.e what the machine can see. In LR the machine learns from its actions and the outcomes of those actions. The difference between unsupervised- and supervised learning is that the model in the supervised case learns from a set of labeled data whilst the unsupervised model does not use labels.

e)

When using a dynamic programming algorithm, such as policy iteration, a perfect model of the environment is assumed to be known as a Markov decision process (MDP). An RL algorithm, such as Q-learning, uses similar update rules that are based on the Bellman optimality equations but does not use a fixed dynamics that are assumed to be known. Rather the RL model tries to learn the dynamics through sampling the experience. In conclusion, when solving an MDP we have a set of states, a set of actions, a model and a reward function. Dynamic programming assumes the model and reward function to be known, whilst RL does not necessarily.

1.3 Policy Iteration

Exercise 4

a)

The Bellman optimality for the state value function is given by equation 6.

$$V^*(s) = \max_a \sum_{s'} T\left(s, a, s'\right)\left[R\left(s, a, s'\right) + \gamma V^*\left(s'\right)\right] \tag{6}$$

In a similar way as in exercise 1, one may note the similarity between equation 6 and the definition of the exceptation of a discrete s.v in expression 3. Noting this one may reformulate equation 6 as expression 7.

$$V^*(s) = \max_a E[R\left(s, a, S'\right) + \gamma V^*\left(S'\right)] \tag{7}$$

b)

Equation 6 expresses the fact that the optimal expected utility, i.e the value of state s, is given by by the expected reward of the next state plus the expected

future discounted rewards, given that a is the optimal action.

c)

The max operator fetches the largest possible expected utility w.r.t a, the actions the agent can take. This represents how we want the optimal expected utility, similarly to how the max in expression 2 fetched the maximum possible future utility based on future actions.

d)

Similary to expression $\pi^*(s) = \arg\max_a Q^*(s,a)$ the relation between the optimal policy π^* and the optimal value V^* is displayed in equation 8. This expression is used in value iteration for estimating the optimal policy is therefore called 'The policy improvement equation'.

$$\pi^*(s) = \arg\max_a V^*(s) = \arg\max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s')\right] \quad (8)$$

e)

The relationship between the π^* and Q^* is simpler since Q^* describes the optimal expected utility given a state-action pair s, a and V^* the optimal expected utility of a state. Since π^* gives the optimal policy, i.e next action, of a given state one would think that this relationship would be simple. Just pick the action that takes the agent to the state s' with the highest expected utility, i.e the subsequent state $s'^* = \arg\max_{s'} V^*(s')$. However in an MDP we have a stochastic mapping between the current state, picked action and subsequent state. Hence the optimal action a, given the otpimal policy, becomes the one that maximises the expectation given that the optimal policy is followed. Therefore we need the $\arg max_a$ even though V^* depends on s and hence the expression becomes more complicated.

Exercise 5

a)

The implementation of the policy evaluation algorithm is displayed in code snippet 1. The implementation of the policy improvement algorithm is displayed in code snippet 2.

```
Delta = 0;
for state_idx = 1 : nbr_states
    % FILL IN POLICY EVALUATION WITHIN THIS LOOP.
    v = values(state_idx);
    action = policy(state_idx);
    next_state_idx = next_state_idxs(state_idx, action);
    switch next_state_idx
        case -1
            values(state_idx) = rewards.apple; %Found apple
        case 0
            values(state_idx) = rewards.death; %Died
        otherwise
            values(state_idx) = rewards.default + gamm * values(
    next_state_idx); %Nothing happened
```

```
    end
    Delta = max(Delta, abs(v - values(state_idx)));
end
```

Listing 1: Policy evaluation

```
policy_stable = true;
for state_idx = 1 : nbr_states
    % FILL IN POLICY IMPROVEMENT WITHIN THIS LOOP.
    old_action = policy(state_idx);
    possible_states = next_state_idxs(state_idx,:);
    temp = zeros(3,1);

    for i = 1:3
        next_idx = possible_states(i);
        switch next_idx
            case -1
            temp(i) = rewards.apple; %Found apple
            case 0
            temp(i) = rewards.death; %Died
            otherwise
            temp(i) = rewards.default + gamm * values(next_idx); %
    Nothing happened
        end
    end
    [~, best_action] = max(temp);
    policy(state_idx) = best_action;
    if policy_stable
        policy_stable = (old_action == policy(state_idx));
    end
end
```

Listing 2: Policy improvement

b)

Trying different values of γ yields the result in table 1. The reason for the infinite amount of evaluations using $\gamma = 1$ is because future and present rewards are regarded to be equally valuable. The software gets stuck in an infinite loop since, the snake has no incentive to eat the apple now rather than later making us unable to iterate to a policy. When using $\gamma = 0$ future rewards have no value. Hence the snake keeps going around in a circle which is a policy that can be generated in 2 iterations. Using $\gamma = 0.95$ works well.

γ	# Policy iterations	# Policy evaluations
1	NaN	inf
0.95	6	38
0	2	4

Table 1: Number of policy iterations and -evaluations for different values of γ

c)

Trying different values of ϵ yields the result in table 2

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 2: Number of policy iterations and -evaluations for different values of γ

A larger ϵ increases the difference tolerance causing the estimator to make more policy evaluations. The estimates will be worse since we are tolerating a larger difference and we hence need to make more policy iterations to make up for this. Vice versa having a low tolerance decreases the number of iterations and increases the number of evaluations.

1.4 Tabular Q-learning

Exercise 6

a)

The terminal update scheme for Q-learning is displayed in code snippet 3 and the non-terminal update scheme for Q-learning in code snippet 4.

```
sample = reward; % No future rewards.
pred = Q_vals(state_idx, action); % replace nan with something
    appropriate.
td_err = sample - pred; % don't change this.
Q_vals(state_idx, action) = Q_vals(state_idx, action) + alph *
    td_err; % + ... (fill in blanks)
```

Listing 3: Terminal state update scheme for Q-learning

```
sample = reward + gamm*max(Q_vals(next_state_idx,:)); % replace nan
     with something appropriate
pred = Q_vals(state_idx, action); % replace nan with something
    appropriate
td_err = sample - pred; % don't change this!
Q_vals(state_idx, action) = Q_vals(state_idx, action) + alph *
    td_err; % + ... (fill in blanks)
```

Listing 4: Non-terminal state update scheme for Q-learning

b)

The results from three different test attempts are displayed in table 3. The default configuration from the template did not work well at all. The snake learnt nothing and scored zero points during testing. This might because of the small α leading to only a small fraction of the sample beeing included in the new

estimate. This α fulfills a similar role to the learning rate α in a non-RL ML algorithm. A small α might have caused the model to not learn at all during the first attempt. After the initial failure the α was increased to 0.5. Looking at the problem at hand it also seemed intuitive to increase the punishment, i.e negative reward, of death. After all, a dead snake can't eat any apples and the punishment for dying should therefore be much higher compared to the reward of eating an apple. Increasing the punishment to -10 gave surprisingly good results with such a high score that the testing had to be manually terminated, hence the ≥ 176583. Increasing the punishment of death felt a bit like cheating however so the negative reward was therefore reverted back to -1. Instead the apple reward was increased to 5, the other parameters were also tweaked. This yielded an ok score of 54. In summary, the punishment for death seems the most important parameter to tweak to reach a high performance in the simplified snake game.

ϵ	γ	α	R: Apple	R: Death	**Score**
0.01	0.9	0.01	+1	-1	0
0.01	0.7	0.5	+1	-10	≥ 176583
10^{-3}	0.99	0.6	+5	-1	54

Table 3: Number of points for different configurations

c)

The configuration displayed in the second row of table 3 gave a snake that never (I think) dies and keeps on eating apples. Using this setup had the run never terminating and the score steadily ticking up. There is no way to be sure that this is a fully optimal policy since we can't know that the snake is taking the shortest amount of possible steps. However, snake is not a timed game, and since the score keeps on increasing indefinitely one could consider this agent optimal.

d)

A drawback of using only 5000 iterations is that the number of states is 4136 which is quite close to 5000. The number of state-action pairs is even greater. It is therefore quite unlikely that all states are reached during training, and thus our agent does not gain access to the experience of all states, making it difficult to learn the optimal policy.

2 Q-learning with linear function approximation

Exercise 7

a)

The terminal Q weight update is displayed in code snippet 5 and the non-terminal Q weight update in code snippet 6.

```
target  = reward; % replace nan with something appropriate
```

```
pred    = Q_fun(weights, state_action_feats, action); % replace nan
     with something appropriate
td_err  = target - pred; % don't change this
weights = weights + alph * td_err * state_action_feats(:,action);%
    + ... (fill in blanks)
```

Listing 5: Terminal Q weight update

```
target  = reward + gamm * max(Q_fun(weights,
    state_action_feats_future)); % replace nan with something
    appropriate
pred    = Q_fun(weights, state_action_feats, action); % replace nan
     with something appropriate
td_err  = target - pred; % don't change this
weights = weights + alph * td_err * state_action_feats(:,action);%
    + ... (fill in blanks)
```

Listing 6: Non-terminal Q weight update

b)

Initially the feature f_1, described in table 4 and implement using the code snippet 7, was developed. This feature describes the outcome of the tile assosiated with the next location of the snake head and therefore felt like a good starting point. Since the outcome of the game in every state is highly dependant on the tile type of the next tile it is quite intuitive to include a feature describing this when updating the weights. Unfortunately using the default configuration with only this feature produced unintended results. The snake got stuck in an infinite loop and zero points where scored. The parameter configuration and result of this test is displayed in the first row of table 5

```
%Feature 1 What type of tile is the next loc?
[next_head_loc, ~] = get_next_info(action, movement_dir, head_loc);
state_action_feats(1, action) = grid(next_head_loc(1),
    next_head_loc(2));
```

Listing 7: Feature 1. The type of tile in next head location.

In order to increase the performance the second feature f_2, described in table 4 and implemented using code snippet 8, was developed. In order for the snake to get a better view of the objective of the game the goal of feature f_2 was to describe how to snake should move closer to the apple. The selected feature therefore became the distance to the apple, normalised by the size of the grid to get the value in the range $[-1, 1]$. Since the game is played out in a grid world, where the snake is unable to move diagonally, the Manhattan norm was chosen as a suitable measure of distance. Taking inspiration from the previous exercise, the punishment for death was also increased to -10. This new configuration produced far better results, yielding an average score of 42.52 which exceeded the requirement of 30 in the assignement instructions. The full configuration, and result, of the second attempt is displayed in the second row of table 5.

```
%f2 Manhattan distance from new head loc to apple
[r,c] = find(grid == -1);
apple_loc = [r,c];
```

```
state_action_feats(2, action) = pdist([next_head_loc;apple_loc],'
    cityblock') / pdist([0 0 ; size(grid)],'cityblock');
```

Listing 8: Feature 2. Manhattan distance from next tile to apple normalised by grid size.

To further improve the performance the third feature f_3, described in table 4 and implemented using code snippet 9, was developed. The goal of this feature was to stop the snake from dying by giving it a sense of how close it was to the wall. Therefore the Manhattan distance to the closest wall was used. However, a longer snake might have to go closer to the wall since more of the grid is occupied by its tail. Therefore the length of the snake was added to the minimum distance to the wall. Hence a small distance to the wall would be offset by a large snake capturing the dynamic of some times having to go close to the wall if the tail of the snake is long. Everything was subsequently normalized by the size of the grid. Unfortunately this new feature did not enable the snake to reach a score higher than the previous one. In order to try and improve the performance some of the other parameters where tweaked, however this did not increase the performance. The full configuration, and result, of the third attempt is displayed in the third row of table 5.

```
%3 distance to wall
edges = [[ones(1,30);1:30]';[1:30;ones(1,30)]';[1:30;30 * ones
    (1,30)]' ; [30 * ones(1,30);1:30]'];
snake_size = length(find(grid == 1)) - 30 * 4 + 4;
temp = pdist([head_loc ; edges],'cityblock');
temp = min(temp(:,1));
state_action_feats(3, action) = (temp+snake_size)/man_grid_size;
```

Listing 9: Feature 3. Manhattan distance to closest edge + length of snake. Normalised by grid size .

Feature	**Description**
f_1	Tile type of next tile
f_2	Manhattan distance between next tile and apple, normalised by grid size
f_3	Manhattan distance between head and closest wall + the current size of snake, normalised by grid size

Table 4: Feature descriptions.

ϵ	γ	α	Apple	Death	Default	F1	F2	F3	**Score**
0.5	0.99	0.5	+1	-1	0	f_1	N/A	N/A	**0**
0.5	0.99	0.5	+1	-10	0	f_1	f_2	N/A	**42.52**
0.5	0.95	0.35	+10	-50	-1	f_1	f_2	f_3	**42.52**

Table 5: Configurations and scores for different attempts. F denotes feature and 'Apple' the reward for apple, etc.

c)

After the third failed attempt to increase the score a plethora of new configurations were experimented with. Multiple different combinations of ϵ, γ and α were tried. Different reward functions were also tried, for example letting the normal state having a small negative reward of -1 relative to a large punishment of -50 for death and +10 for apple. Also more complex features were tried, like adding non-linearities in terms of log, exp or tanh to the distance features. Unfortunately none of these experiments produced the desired outcome and the average score for 100 iterations remained at 42.52. A higher score could probably have been achieved by using a different type of feature that was not necessarily related to the distance between head and apple or head and wall. Some more exotic feature describing the emptiness of the grid in a certain direction is probably needed to increase the score above 42.52. The most likely way to loose a game of snake when one has achieved a sufficient amount of points is not to go in to the wall, rather for the snake to eat itself. Therefore a feature describing how curled up the snake is, or how empty each section of the grid is would probably be necessary. Since the goal of 30 points from the assignement instructions had been reached using only two features, this more advanced feature was never engineered. The concept of adversarial weight initialization was also experimented with, however having tried multiple initialization methods the score never went above 42.52.

In general it is better to use a simple model rather than a complex one if the performance of both models are equal. Therefore the less complicated model, using two features only, was chosen to be the final model. The final settings were therefore set to the ones displayed in table 6, with the inital and finalized weights displayed in table 7.

ϵ	γ	α	R: Apple	R: Death	R: Default	F1	F2	F3	**Score**
0.5	0.99	0.5	+1	-10	0	f_1	f_2	N/A	**42.52**

Table 6: Final settings

	w_1	w_2	w_3
Final Value	-8.8986	-8.3798	0
Initial Value	1	1	0

Table 7: Final weights