Rasmus Helander Machine Learning, FMAN45

Assignment 3

May 26, 2020

Exercise 1

There are 3 possible configurations of the snake (not including the position or orientation, but only the "shape" pf the snake); straight, bent to the left and bent to the right - the latter two are reflections of each other. For each of these configurations, the head can point in 4 different directions (N/E/S/W). Consider, for simplicity, only the case where the head points north. Depending on the configuration of the snake, the head can be placed in a number of different cells in the 5×5 matrix:

- In the case of a straight snake, the head can be placed in 3×5 different cells.
- In the case of a bent snake, the head can be placed in 4×4 different cells.

This means that if we disregard the apple, there are a total of $(3\times5+4\times4+4\times4)\times4 = 188$ different states, where the factor 4 derives from the 4 possible directions of the head. Now, for each of these states, the apple can be placed in any of the remaining $5 \times 5 - 3$ cells, so the total number of possible states is

$$K = 188 \times 22 = 4136 \tag{1}$$

Exercise 2

a)

Since $T(s, a, s') = P(s'|s, a)$, we have

$$Q^*(s,a) = \mathbb{E}\left[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')\right] \tag{2}$$

b)

The equation says that the Q-value for action a in state s is given by the expectation of the utility (reward plus the discounted utility of subsequent states) given that the agent acts optimally after a.

c)

I do not understand this question.

d)

If a does not make the snake eat an apple, then there is only one possible subsequent state s', namely the state specified by the movement of the snake. The transition function $T(s, a, s')$ is then 1 for this state and 0 for all other states.

On the other hand, if a makes the snake eat an apple, there must appear a new apple somewhere on the screen. Thus, there are 22 possible following states s'. In all states the position of the snake is given by the movement in a, and the 22 possibilities are givin by the 22 possible placements of the apple. Since the position of the apple is drawn from a uniform distribution all of these states have the transition function $T(s, a, s') = \frac{1}{22}$. For all other states (i.e. all states where the position of the snake is different) the transition function is 0.

Exercise 3

Starting from the terminal state s_t where the snake dies, each Q-value is 0 as prescribed in the assignment. For the state-action pairs leading up to this terminal state (s_{t-1}, a_{t-1}), the Q-value is updated to -1, as the reward of the action is -1. Going one step further back, we can not calculate the Q-values $Q^*(s_{t-2}, a_{t-2})$ any longer. The reason for this is that we would need to know $\max_a Q^*(s_{t-1}, a_{t-1})$ in order to calculate the Q-value. That is, we must know the utility of the optimal action taken in state s_{t-1}, which is not possible as we only know the value of a sub-optimal action (namely the action that leads to the terminal state, rewarded with -1).

However, the optimal policy can be found by instead starting at the terminal state s_t where the snake eats an apple. Once again, the Q-value of all such state-action pairs is. This time however, the Q-value for all state-action pairs leading up to s_t is 1, as $R(s_{t-1}, a_{t-1}, s_t) = 1$. Note that it is not possible for any state-value pairs to get $Q^*(s, a) > 1$. This is because $R = 1$ only when the terminal state is reached, and $0 < \gamma < 1$ which means that the Q-value will be discounted for all actions that do not reach the terminal state directly. Thus, the action leading to s_t is the optimal action. Taking one more step back - to s_{t-2} - the Q-value can now be calculated for the action for the action leading up to s_{t-1}, as we know the subsequent optimal policy. If it is not possible to reach the terminal state from s_{t-2} then the action leading to s_{t-1} is optimal (if s_t can be reached than that will be the optimal action). Continuing to go back in this way, we will always know the optimal policy of subsequent steps, so the Q-value can always be calculated.

The above means that if we start from the terminal states where an apple is eaten, it is possible to calculate $\max_a Q^*(s, a)$ for all states that can be reached by going back one step at a time. For this game, going back in this direction will reach all possible states and the optimal policy for the game can thus be calculated.

Exercise 4

a)

Similarly to exercise 2, we have

$$V^*(s) = \max_a \mathbb{E}\left[R(s, a, s') + \gamma V^*(s') | a, s\right] \tag{3}$$

b)

The value of state s is given by the expected value of the utility of the next state, given that the action a is chosen in an optimal fashion.

c)

I am not sure that I understand the question but I guess that the purpose is to state state the value of a state is the utility of the optimal action, i.e. the action a that maximizes $Q^*(s, a)$.

d)

The fundamental difference between Q^* and V^* is that V^* is the expected utility of a *state*, while Q^* is the expected utility of a *state-action pair*. The optimal policy π^* prescribes a "recipe" of what action to choose given that the environment is in a given state. If the environment is entirely deterministic, then the optimal action is the one leading to the state with the highest utility (i.e $\max_{s'} V^*(s')$), but with an MDP we are not sure what state we will end up in. The optimal policy is to choose a in order to maximize Q^* and V^* is then the expected value of the state given that the optimal policy is followed.

Exercise 5

a)

Policy evaluation:

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

Policy iteration:

```
possible_actions = next_state_idxs(state_idx , : );
i  = 1;
for a = possible_actions
    if a == -1
        v(i) = rewards.apple;
    elseif a == 0
        v(i) = rewards.death;
    else
        v(i) =  rewards.default + gamm*values(a);
    end
    i = i + 1;
end

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

b)

Table 1 shows the number of iterations and evaluations needed for the different discount parameters.

Table 1: Number of iterations until policy converges, using a tolerance $\epsilon = 1$

γ	**# Policy evaluations**	**# Policy iterations**
0	4	2
0.95	38	6
1	Does not converge	Does not converge

For $\gamma = 0$, we quickly get an optimal policy. However the snake does not play optimally, but instead gets stuck in an infinite loop and never eats the apple. This is not surprising, as it is easy to understand the logic of the agents decision with this γ:

- If connected to an apple, choose action to eat the apple.
- If connected to a wall, choose action that does not go in to wall
- otherwise, choose any action

If 2 actions have the same utility, the max operator in MATLAB will choose the action with the lowest index - corresponding to going left. Thus, the snake will get stuck in a loop where it always turns left. This problem could be fixed by forcing the agent to choose action randomly between all actions that have the same utility. This way, the agent will stay clear of the walls, and eventually reach an apple as a result of a random walk

For $\gamma = 0.95$, the convergence is rather slow, but the agent plays optimally after finding the optimal policy - i.e. the one that reaches the apple in the least amount of time.

For $\gamma = 1$ the policy iteration does not converge and the game is never played. If the iteration did converge, all actions that do not lead in to a wall would have utility 1, and the agent would get stuck in the same way as for $\gamma = 0$ (actually even worse, as the utility of eating an apple would be the same as when not eating an apple).

c)

See table 2 for the results.

Table 2: Number of iterations until policy converges, using a discount factor $\gamma = 0.95$

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

For $\epsilon \leq 1$, the only difference is that the policy needs to be evaluated more times in to converge for smaller ϵ. This is because smaller ϵ means that the requirements for convergence is harder. Note that if we disregard the first evaluation (directly after random initialization of the values) the biggest change in $V^\pi(s)$ is 2, since $-1 \leq V^\pi(s) \leq 1$ for all states s (worst value is if the policy forces the snake in to a wall, best value is if policy makes the snake eat an apple). This means that Δ will never be 10 or larger, so each policy will only be evaluated once.

For all values of ϵ tested, the snake eventually acts optimally. As I understand it, this has the following reasons.

- After first policy evaluation, all state-action pairs (s_{t-1}, a_{t-1}) leading to a positive terminal state will have the correct Q-value (1).
- Either the Q-value other state action pairs $(s_{t-1}, \hat{a}_{t-1})$ will be larger than 1 or smaller than one.
  - If smaller than 1, the policy can be changed so that the agent acts optimally if it is connected to an apple.
  - If larger than 1, the policy will be updated subtoptimally. However, the estimated value of $Q^\pi(s_{t-1}, \hat{a}_{t-1})$ will decrease in the next policy evaluation (since $\gamma < 1$)

and eventually be less than 1 - after which policy will be optimal for s_{t-1}

- The next policy evaluation, the next "layer" of state action pairs (s_{t-2}, a_{t-2}), i.e. the ones leading to s_{t-1}, will have correct Q-values.
- The same way as above, the policy is changed so that the agent acts optimally in s_{t-2}
- This goes on until the snake acts optimally in all states

Exercise 6

Terminal:

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

Non-terminal:

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

Table 3 below shows the test score for 4 different runs. I quickly realized that the default values of ϵ and α were too low to actually learn anything. Low ϵ means that the agent almost never tries new actions, and is thus stuck. Too low learning rate α means that the updates of the Q-values are too slow to make any real progress. After some testing I found $\epsilon = 0.2$ and $\alpha = 0.5$ to be quite good values, and had a hard time getting a better score than 158 by only tweaking these parameters. I then realised that making the rewards larger significantly improves the performance. This makes sence, because higher rewards give "higher incentives", meaning optimal actions will be rewarded greater.

Table 3: Score for a set of different hyperparameters

ϵ	γ	α	**Reward: apple**	**Reward: death**	**Score**
0.2	0.9	0.1	1	-1	1
0.2	0.9	0.5	1	-1	158
0.2	0.7	0.7	1	-1	42
0.2	0.9	0.5	10	-10	6368

The problem with only training the agent for 5000 iterations is that the number of states is almost 5000, and the number of state-action pairs significantly larger. This means that if we only train for 5000 iterations, it is quite possible that all states are not reached. A solution to this might be successively lowering the greediness of the agent. Then, after reaching a quite good policy, the agent will have a higher probability of following this policy and thus seeing more states before termination. This is just a theory, however, and I have not been able to make it work.

Exercise 7

Q-updates

Terminal:

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

Non-terminal:

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

State-action features

Feature 1: Normalized L1-distance (number of steps) to apple from head

```
[apple_loc_r, apple_loc_c] = find(grid == -1);
apple_loc = [apple_loc_r, apple_loc_c];
[next_head_loc, next_move_dir] = get_next_info(action, movement_dir, head_loc);
dist = norm(next_head_loc-apple_loc,1);
max = norm(size(grid), 1);
state_action_feats(1, action) = dist/max; % normalized l1 distance to apple
```

This feature is calculated by finding the location of the apple and then calculating the L1-distance to it from the head of the snake after the action - corresponding to the number of steps needed to get to the apple. This distance was then normalized by dividing with the largest possible such distance. Since it should be positive to get closer to the apple, I anticipated this feature to have negative value and thus initiated it to 1.

Feature 2: What is at the location of the head after the action (apple, wall/self or nothing)

```
state_action_feats(2, action) = grid(next_head_loc(1), next_head_loc(2))
```

This feature looks at the next head location and sees what is there (-1 if an apple, 1 if wall/self, 0 otherwise). Since the agent should avoid walls and try to eat apples, this feature should be negative and I initiated it to 1.

Feature 3: Is the head in the largest possible field?

```
[f, n] = bwlabel(grid == 0, 8) ;
z = sum(grid == 0, 'all');
if n <=  1
    s = 0;
else
    c = f(next_head_loc(1), next_head_loc(2));
    if c == 0
        s = 0;
    else
        s = (f == c);
        s = sum(s, 'all');
        if s < z / n
            s = 1;
        else
            s = 0;
        end
    end
end
state_action_feats(3, action) = s;
```

This feature looks at weather the game board matrix is divided in multiple subsections (i.e. if there are some parts of the field that are currently impossible to reach for the agent). If only one subsection exists, the feature is set to 0. If the next head location given the action is in a wall, self or apple, the feature is set to 0. If the next location of the head is in a subsection that is smaller than the total number of 0's in the board divided by the number of subsections, the feature is set to 1. I.e. this feature is set to 1 if the next head location is in a smaller subsection of the matrix.

The point of this feature is to avoid the agent "catching itself". After experimenting for a while with the two first features I realized that the most common way for the snake to die was to get trapped by itself, and this feature made this behavior go away. As the snake should avoid the smaller subsections, the feature weight should be negative and was thus initiated to 1.

Tries

Table 4 below gives the results for the linear Q-learning.

Table 4: Results for the linear Q-learning

	2 features				**3 features**	
ϵ	0.5	0.5	0.5		0.5	0.5
ϵ_{upd_iter}	0	0	0		0	500
ϵ_{upd_factor}	N/A	N/A	N/A		N/A	0.3
γ	0.95	0.95	0.95		0.96	0.95
α	0.5	0.5	0.5		0.5	0.4
α_{upd_iter}	0	0	0		0	1800
α_{upd_factor}	N/A	N/A	N/A		N/A	0.5
Reward: apple	10	50	50		50	50
Reward: death	-1	-10	-10		-10	-10
Reward: default	0	0	-1		-1	-1
Average Score	Stuck in loop	Stuck in loop	42.52		71.71	91.58

Between the first and the second try, I only changed the rewards of apple and death. In both these, the agent ended up in an infinite loop. This indicates that the agent did not learn the importance of getting closer to the apple. To punish this behavior I changed the reward for default to -1 (because then, the infinite loop will receive large negative rewards). This improved the agent's performance drastically and it managed to get above the threshold of 35.

As stated above, an inspection of the behavior of the agent made me realize that it often trapped itself, and I created feature 3 (after quite a lot of experimentation). Using the same settings as previously this boosted the average score to over 70. In order to further improve the score, I started doing some experiments with ϵ and α decay. My reasoning was that after learning the basic features (i.e. avoid walls, try to get close to apple, avoid small subsections) it would be a good idea to raise the greediness of the agent. With $\epsilon = 0.5$ every other action is random, and the agent will die quickly during training because of this behavior. If the relative importance between the features is to be learned more effectively (e.g. what to choose if the apple is inside a smaller subsection) I figured that the agent needed to survive longer during training time. Introducing the decay, the agents performance once again improved, to 91.58. After this, I did not manage to get it any higher. I suspect that I would need to configure feature 3 a bit in order to get an average score above 100 (my feature 3 is quite primitive as it only tells whether the agent is in a smaller subsection or not).

Something interesting that I discovered is that the performance with the best settings improved when the weights were initialized randomly. I found this to be quite surprising, as the training with "bad" initialization still managed to find good values of the weights. To understand why this is the case, I trained the agent twice more; once using random initializations and once using the bad initialization. The results are in table 5. As the weights converge to the exact same values, I concluded that the problem is not in the training. Rather, I realized that the random number generator used to initialize the weights was called again when testing, meaning that the two agents were tested on slightly different games. After fixing this (by commenting out the random initialization during testing) I got the same average score no matter how the weights were initialized.

Table 5: Scores for different initializations

w_{1_i}	w_{2_i}	w_{3_i}	w_{1_f}	w_{2_f}	w_{3_f}	**Score**
1	1	1	-23.8106	-19.2161	-8.9589	91.58
$\in N(0,1)$	$\in N(0,1)$	$\in N(0,1)$	-23.8106	-19.2161	-8.9589	95.18