Assignment 4 FMAN45

Kalle Josefsson ka5532jo-s

May 20, 2024

1 Introduction

In the last lab of the course we are to use machine learning to train a model which plays the well known snake game. First we do it on a much smaller grid than the actual game and with the snake not growing when eat the apple. For the last part of the lab we will try to make a good agent which can play the original snake game and hopefully score some high points.

2 Task 1

For the first task we have the simplified snake game where the snake can only move within a 5x5 grid and is three pixels long and does not increase in length after scoring points by eating apples. Now the purpose was to find the size of the statespace. In order to do this we have to look at all the different combinations of snake and apple configurations in the grid. The snake is either straight or bent, when straight it can only be either vertical or horizontal, if it is vertical it can be in 3 different positions per column and if it is horizontal it can be in 3 different positions per row resulting in 3x5x2 = 30 combinations. When the snake is bent it has 4 different shapes which has 4 different combinations in each 2x2 section of the 5x5 grid. There are 16 distinct 2x2 sections in the 5x5 grid which results in 16x4 = 64 positions for the bent snake. Now that we have the amount of distinct snake positions we also need to take the apple position into the equation. The apple can be anywhere on the grid except for where the snake is giving it (5x5-3) = 22 positions for each snake position. We also have that the snake head can be at either side of the snake resulting in twice as many state. Hence the total amount of states would be 94x22x2 = 4136 different states, i.e K = 4136.

3 Task 2

3.1 2a

In this task we were to rewrite the expression for Q^*. using expected values. The original equation for Q^* is:

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

Here T is the transition probability for a state and action pair to end up in the new state s'. R is the reward you get from transitioning from state s to s', and γ is the decay constant, i.e how much value future rewards.

We have the following condition for T :

$$T(s,a,s') = \mathcal{P}(s'|a,s) \tag{2}$$

We can now rewrite Equation (1) as:

$$Q^*(s,a) = E_{T(s,a,s')}[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')] \tag{3}$$

3.2 2b

Now to rewrite Equation (1) as an infinite sum, i.e we do not want the recursion that is present in Equation (1). To first interpret this we can write it as:

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

Which we then can rewrite into the final form:

$$Q^*(s_0,a_0) = E_{T(s_0,a_0,s_1)}[R(s_0,a_0,s_1) + \sum_{t=1}^{\infty} \gamma^t \max_{a_t}(R(s_t,a_t,s_{t+1})] \tag{5}$$

3.3 2c

Equation (1) returns the expected sum of rewards when you take action a in state s and after that acting optimally throughout.

3.4 2d

The equation in question is the Bellman equation using Q^π:

$$Q^\pi(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma Q^\pi(s', \pi(a'|s')] \tag{6}$$

Here we use the optimal policy π^* which uses the max operator to optimize the reward over all the possible actions in a state. Here we want continue following that policy and if the policy π is indeed optimal $Q^* = Q^\pi$.

$$\pi^*(s) = a^* = \arg\max_a Q^*(s,a) \tag{7}$$

3.5 2e

γ is the decay value which is recursively multiplied with itself the deeper we go in the value function. Hence valuing scores later on less than current ones, based on what we choose gamma to be, $\gamma \in [0,1]$, we either value future rewards high or low which alters our decision making from the values we form Q^*

3.6 2f

We have two cases of how T(s,a,s') will look like based on what the action does. If we are in state s and take action a which does not result in the snake eating an apple, then the apple will not change its location in the grid and we will end up in state s' with 100 percent certainty, hence we will have T(s,a,s') = 1 for state s' and zero for the other possible states. If we however are in state s and take action a which results in the snake eating the apple we will get other results from the transition state matrix. The apple will now change location randomly and equally distributed to one of the positions in the grid which is not occupied by the snake, which occupies three spots in the grid, hence we have 22 spots left from the apple to appear. Hence we have 22 possible states to end up in, all equally likely, these states represent the snake in its position after the action and the apple in one of the 22 free spaces. Hence we have T(s,a,s') = $\frac{1}{22}$ for these states and zero for all other states.

4 Task 3

4.1 3a

The on-policy and off-policy are different ways to update the policy during training of the decision making agent. On-policy learning methods evaluate and improve the policy it is currently using to

make decisions. While off-policy agents learn the value of target policy while following a different behaviour policy.

4.2 3b

The difference between model-free and model based reinforcement learning is that in model based we learn or estimate the transition probability matrix and the rewards of the problem prior to using MDP algorithms. For model free reinforcement learning we start learning directly from interaction without trying to find an explicit model.

4.3 3c

The difference between passive and active reinforcement learning is that for passive the agent's goal is to evaluate an already existing policy rather than finding an optimal policy. The policy is predetermined and the agent follows it strictly during the process without changing it. For active the agent tries to learn the optimal policy which maximizes the sum of current and future rewards. The policy is not fixed and the agent updates and improves the it based on feedback from interactions.

4.4 3d

Reinforcement learning, unsupervised learning and supervised learning are three vital learning methods in the world of machine learning. They differ in the learning processes. In reinforcement learning we have an agent which takes an action in an environment which it gets a reward or a penalty from and the objective is that the agent learns from these interactions so that it in the future makes better decisions. The goal is to find an optimal policy which maximizes the reward over time. For supervised learning we train a model with a training dataset with a task which the model is supposed to solve. The goal is to during training find a way to map the input data to match the target set that comes with the training set. Based on how close our predictions of the output is we update our weights in the model. The goal is that after training we can solve problems outside of the training data but from the same distribution. Unsupervised learning is similar to supervised learning in the sense that the task is to map input to output, but here we do not have targets for the training data, i.e no labeled output. The learning process is of training is to learn the structure or distribution of the data and discover these hidden structures or patterns.

4.5 3e

For dynamic programming (policy iteration) we need a full and correct knowledge about the Markov Decision Process. That means we need to know both the transition probabilities T(s,a,s') and also the rewards from the reward function R(s,a,s'). This is not needed for Reinforcement learning, it learns the policy from interacting with the environment. For policy iteration we have two main steps being policy evaluation and policy improvement. First we evaluate the policy by computing value function V^{π} for policy π and then we update and improve the policy with respect to the current value function. They are guaranteed to converge to the optimal policy π^* and value function if we have an accurate model and perform calculations correctly. Policy iterations is also quite computationally demanding since we have to iterate over all actions and states which becomes troubling with large state action spaces. For reinforcement learning we need to explore in order to learn correctly which makes it more suitable for complex environments when the model is not known or it is to big so the computational load is too big for policy iteration. RL methods such as Q-learning converge to the optimal police if enough exploration is done and also specific conditions met such as decaying learning rate. However we need to find a good balance between exploration and exploitation, we need to take random actions sometimes in order to explore and try new actions but also follow and yield high rewards in order to learn effectively.

5 4

´

5.1 4a

We solve this task in the same manner as we did for task 2a. We rewrite the original expression:

$$V^*(s) = \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^*(s)] \tag{8}$$

Which we then rewrite into an expected value as:

$$V^*(s) = \max_a E[R(s, a, s') + \gamma V^*(s')|a, s] \tag{9}$$

5.2 4b

The $V^*(s)$ equation gives the expected value of a given state s given that your action a is optimally chosen in that state and all following moves.

5.3 4c

The purpose of the max operator is to choose the action a in state is that maximizes the value function as:

$$V^*(s) = \max_a Q^*(s, a) \tag{10}$$

5.4 4d

Again $\pi^*(s)$ is the optimal police and follows:

$$\pi^s(s) = \arg\max_a Q^*(s, a) \tag{11}$$

In other words it returns the action which maximizes Q^* and $V^*(s) = \max_a Q^*(s, a)$ so we get the following expression between the two.

$$\pi^*(s) = \arg\max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma v^*(s')] \tag{12}$$

5.5 4e

The relation between π^* and Q^* is not as simple as the relation between π^* and V^* is due to the information these two functions provide. Q^* provides information of ow valuable it is to take an action in a specific state and directly evaluates the expected return of each action in a state while V^* gives the information about the overall value of being in a state considering the best possible actions in that state. So V^* does not provide information about which actions to make instead it summarizes the values of being in state assuming that we are following the optimal policy π^*

6 5

6.1 5a

Below my code for the policy iteration and evaluation is presented:

```
while 1

    % Policy evaluation.
    while 1

        Delta = 0;
        for state_idx = 1 : nbr_states
            % FILL IN POLICY EVALUATION WITHIN THIS LOOP.

            V = values(state_idx);
            a = policy(state_idx);
            next_idx = next_state_idxs(state_idx,a);

            if next_idx == -1
               values(state_idx) = rewards.apple;

            elseif next_idx == 0
                values(state_idx) = rewards.death;

            else
                values(state_idx) = rewards.default +gamm*values(next_idx);
            end

            Delta = max(Delta,abs(V-values(state_idx)));


        end


         % Increase nbr_pol_eval counter.
         nbr_pol_eval = nbr_pol_eval + 1;

         % Check for policy evaluation termination.
         if Delta < pol_eval_tol
             break;
         else
             disp(['Delta: ', num2str(Delta)])
         end
     end

     % Policy improvement.
     policy_stable = true;
     for state_idx = 1 : nbr_states
         % FILL IN POLICY IMPROVEMENT WITHIN THIS LOOP.
         a_old = policy(state_idx);
         next_idxs = next_state_idxs(state_idx,:);
         val_temp = zeros(3,1);

         for k=1:3
             next_idx = next_idxs(k);
             if next_idx == -1
                 val_temp(k) = rewards.apple;
             elseif next_idx == 0
                 val_temp(k) = rewards.death;
             else
                 val_temp(k) = rewards.default + gamm*values(next_idx);
             end
            end
         end
         [~,index] = max(val_temp);
         policy(state_idx) = index;

         if policy_stable
             policy_stable = (a_old == policy(state_idx));
         end

     end

     % Increase the number of policy iterations .
     nbr_pol_iter = nbr_pol_iter + 1;

     % Check for policy iteration termination (terminate if and only if the
     % policy is no longer changing, i.e. if and only if the policy is
     % stable).
     if policy_stable

         break;
```

Figure 1: Combined Code Snippets for Policy iteration and evaluation

Having γ set to 0.5 and ϵ to 1 I go the desired number of policy iterations and valuations, six and eleven respectively.

6.2 5b

The table for different values of γ and the following results are presented below.

	Policy iterations	**Policy evaluations**
$\gamma = 0$	2	4
$\gamma = 1$	N/A	∞
$\gamma = 0.95$	6	38

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

For γ being set to zero our agent totally neglects future rewards which makes it just going around in circles forever since if no current action leads to apple it will just continue turning left forever since it is first in the index all actions have same reward (0) resulting in a total score of zero.

For γ being set to 1 our agent values future rewards equally to present ones even. The code runs forever since the snake is cant decide if going for the apple now or later is better so it is stuck in an infinite loop which leads to the game not even being played.

Lastly for γ being 0.95 The snake plays optimally finally and it values rewards one timestep into to the future 5 percent less so it goes for the apple when it sees it. However as we can see the convergence is rather slow but it does actually find $\pi*$ and takes the shortest path possible towards the apple.

6.3 5c

Table 2 shoes the results from task 5c.

ϵ	**Policy iterations**	**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: Comparison of policy iterations and evaluations for different ϵ values increasing by 10x for every measurement and γ being constant at 0.95

The large ϵ means a loose convergence criterion, which is why the number of policy evaluations decreases. However, to make up for this, the number of policy iterations has to increase, as can be seen clearly in the table. The reason why each policy iteration only has one policy evaluation is that the maximum difference between two states is 2. The best score is 1 for an apple and the worst score is -1 for hitting a wall, hence having ϵ be more than 2 will let any policy evaluation through.

As we can see, the number of policy evaluations needed to find the optimal policy is much larger for stricter convergence criteria, which is not surprising. Since we are doing more policy evaluations, the number of policy iterations decreases. Therefore, we must find a trade-off for what should be the optimal way to train the agent.

The snake actually acted optimally for all different ϵ values. I think this is because, for policy iterations, as mentioned above, the policy will eventually converge towards π^* given that the state-action model is correct and our ϵ is not excessively large. Also, for the large convergence criteria, the policy evaluation step might be more approximate, but the policy evaluation still guides the snake to high-reward actions.

7 6

7.1 6a

```
sample                    = reward % 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)
```

Figure 2: Code for terminal

```
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)
```

Figure 3: Code for non terminal

7.2 6b and c

ϵ	γ	α	**Reward: apple**	**Reward: death**	ϵ **Factor/Freq**	α **Factor/Freq**	**Score**
0.2	0.9	0.5	1	-1	0/0	0/0	158
0.5	0.9	0.5	1	-1	0/0	0/0	1
0.3	0.99	0.7	100	-100	0.5/1000	0.5/1000	17169
0.1	0.99	0.5	100	-100	0.5/1000	0 / 0	∞

Table 3: Total score achieved by agent for different sets of hyper-parameters

Table three shows how well our agent scored based on what hyperparameters we used during training. As you can tell from the table I was able to make the agent act optimally and never dying after finetuning the parameters, at least I tihnk so it did not die after achieving a score of 200 000 and I'm guessing that if it did not die after scoring that high it would never fail. During training I saw that the initial value of α was way to low and I quickly increased to be at least 0.5 for all other training sets this is probably due to the fact the model actually was not able to learn if it was set to low. Another factor that also had a big impact was that I increased the magnitude of the reward and the punishment of loosing by 100. What surprised was keeping the random action selection low at beginning actually worked really well, decreasing the randomness over the iterations was a good idea theoretically and worked really well but as stated having the initial value of ϵ as 0.1 worked really well. Not decaying the learning parameter also worked well which was kind of surprising since I thought that letting it decay would finetune better but it seemed like it worked optimally. The reason the high rewards and penalization's worked better is probably since it forces the snake to go for the apples at the same time as it avoids loosing at all cost which is a good combination for a game.

7.3 6d

The reason why it can be hard to find optimal behaviour in just 5000 iterations of training is that in our specific system we have 4136 states without actions taking into consideration, taking the actions into consideration we get a state action space which is larger than the amount of iterations which would possibly lead to not all state action pairs being taken which would like to avoid. Another issue might be that we have a lot hyperparameters to tune, 10 in total, and most if not all of them seem to depend on the others which makes it harder to finetune.

8 7

8.1 7a

```
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)
```

Figure 4: Code for terminal

```
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)
```

Figure 5: Code for non terminal

8.2 7b

The table below shows the performance achieved by using different sets of hyperparameters.

ϵ	γ	α	**Reward: default**	**apple**	**death**	ϵ **Factor/Freq**	α **Factor/Freq**	**Score**
0.1	0.9	0.5	0	1	-1	0/0	0/0	2.12
0.4	0.99	0.5	0	-10	10	1000/0.5	1000/0.5	41.52
0.5	0.9	0.4	-1	30	-100	500/0.3	1800/0.5	41.52

Table 4: Total score achieved by agent for different sets of hyper-parameters

Except for the features in table three presented above I also had 3 different state action features to evaluate the states and action and hopefully improve how well my agent played.

```
if(grid(next_head_loc(1),next_head_loc(2)) == 1)
    state_action_feats(1, action) = -0.05;
else
    state_action_feats(1,action) = 0;
end
```

Figure 6: State action feature 1

```
[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(2,action) = dist/max;
```

Figure 7: State action feature 2

```
if(grid(next_head_loc(1),next_head_loc(2)) == -1)
    state_action_feats(3, action) = 0.5;
else
    state_action_feats(3,action) = 0;
end
```

Figure 8: State action feature 3

The first feature checked if the snake would run into a wall or itself and if it did it got a score of minus one and if the action did not lead such a state it would get a reward of zero, the code for it is presented in figure 6. The second feature was to check how close the snake was to the apple after an action. Here I just gave a reward based on how close the snake was to the apple after an action and keeping the reward between zero and one note however that the reward grows with distance which is wrong but the weight becomes negative during training so the Q-learning algorithm penalizes longer distances. The last feature just gives an incentive to go for the apple if a direct action leads to eating an apple then we give that action a reward.

As we can see in table four from the results our model did not work well when ϵ as to low, this is probably due to our model not working well when do not, at least initially, have a lot of randomness to evaluate different actions. Hence I increased the value for both ϵ but also the rewards to really punish loosing but at the same time give incentive to go for the apples as we did in task 6. This as you can see yielded better results but still not great. For the last try I increased the randomness even more while actually lowering the learning rate just a little to have less variance during learning. I also introduced a reward of minus one for actions which did not lead to any outcome, death or apple, and also decreased the randomness as well as the learning rate over time as can be seen in table four. Now I got a an average score of 41.52 again during the test run, which I was not entirely satisfied with but it was good enough.

8.3 7c

The weights from my last try, best performing model, are presented below in table 5. I tried optimizing it even further but only got the same results at best after a lot of different variations of hyperparameters.

weight 1	**weight 2**	**weight**
1768	-282	453

Table 5: Weights for the three different state action features

The hyperparameter selection after finding state action features which worked okay, the ones presented above, did not really make that effect on the outcome except change the magnitude of the weights more than the proportions between them which I thought was really odd. Hence the state action features had way more impact than the choice of hyperparameters. As we can see the weight for not dying or running into itself is higher than the others since loosing the game is what we want to avoid so this is not weird at all. I think adding a feature which is able to keep track of the whole body of the snake, i.e look around the head to avoid having its body slithering around its head would be a good feature and would probably increase the average score by a lot compared to mine at around 42. However it did still perform better than 35 which I am grateful for but a little bit higher would have been satisfying.