Home Assignment 4

Arvid Gramer

FMAN45 Machine Learning
Lunds Tekniska Högskola, Lunds Universitet
May 2023

Reinforcement learning for playing Snake

Exercise 1: *Derive the number of possible non-terminal states, K, in the simplified game of Snake.*

The number of non-terminal states is an exercise in combinatorics. Each state consist of an apple position as well as a snake position. The snake position consist of the positions of the three pixels, as well as the direction it is heading.

I begin with calculating the number of possible snake positions. The following is a count of the possible positions as functions of where the head of the snake is positioned. A five by five matrix consists of three rims. The inner rim is just one pixel, the middle one. The first rim is the eight pixels one step from the middle. The outer rim is the 16 outermost pixels.

- Inner pixel: one possible position, four adjacent pixels each of which has three non-occupied adjacent pixels: $1 \cdot 4 \cdot 3 = 12$ combinations.
- First rim: four corners, each having two adjacent pixels that in turn has three possible positions for the last pixel, and two adjacent pixels that in turn has three possible positions for the last position. First rim also has four "inner" pixels. From these there is one way that has three choices for the last pixel, and three other ways that has three choices for the last pixel. $4(2 \cdot 3 + 2 \cdot 3) + 4(1 \cdot 3 + 3 \cdot 3) = 96$
- The outer rim has eight pixels. Four corners, four middle and four between corner and middle. The corners has two choices for the second pixels, each of them having two choices for the last pixel. The middle pixels have two choices that each render two possible combinations for the last pixel and one choice that renders three combinations for the last pixel. The four inbetweeners have one choice that renders one final choice, one that renders three final choices and one that renders two final choices. This yields $4(2 \cdot 2 + 1 \cdot 3) + 4(2 \cdot 2 + 1 \cdot 3) + 4(1 \cdot 1 + 1 \cdot 3 + 1 \cdot 2) = 80$

This yields a total of 188 ways to position the snake. For all of these the apple can be placed in the other 22 pixels, making the total number of states $K = 22 \cdot 188 = 4136$

Exercise 2:
a) *Rewrite equation 1 as an expectation.* The equation to rewrite is the following:

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

It describes the optimal Q-value as a function of the current state and action, and can be interpreted as a sum of all of the rewards that are receivable in the next state $s' \in S'$ when approached from state s by action a,

together with the discounted highest optimal Q-value from the new state s', weighted on the transition probability $T(s, a, s')$ of going to state s' from s by performing action a. Since a sum of different objective function values of a random variable, weighted on the probability of that random variable is the same as the expected value of that function and that variable, we can re-express $\sum_{s'} T(s, a, s')R(s, a, s') = \mathbb{E}[R(s, a, s')]$. The maximal $Q^*(s', a')$ is just a recursive way of expressing the future states optimal Q-value, as it is the value if we keep doing the optimal future actions $a' \in A'$. The probability weighted sum of the optimal Q^* from each possible next state s' is just the expected value of Q^* over all of the future state space S' and action space A'. We can move the γ out since it is not stochastic and rewrite equation 1 as

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

b) The recursive part of the equation comes when we include $Q^*(s', a')$. If we re-express this, using 1, we get $Q^*(s', a') = \sum_{s''} T(s', a', s'') \left[R(s', a', s'') + \gamma \max_{a''} Q^*(s', a')\right]$ which of course goes on and on and on. This implies that the value Q is just the sum of all future rewards, discounted to today. An analogy can be made to asset pricing, were one prices for example a stock or a bond as the discounted value of all future dividends. To simplify we re-express as a sum of expectations, where $S^{(k)}$ implies the state space after taking k actions.

$$Q^*(s, a) = \mathbb{E}\left[R(s, a, S')\right] + \sum_{k=1}^{\infty} \gamma^k \mathbb{E}\left[R(s^{(k)}, a^{(k)}, S^{(k+1)})\right] \tag{3}$$

c) Equation 1 states that the optimal Q-value is the expected reward from this step plus the discounted expected future Q-value.

d) In equation 1 we assume that we follow the optimal policy. We make this assumption when we set $\pi(a'|s')$ to be the a' that maximizes Q^*.

e) γ is the discounting factor. It makes the present value of future rewards less important when we are calculating Q-values. A smaller γ promotes rewards closer in time more over distant rewards, while a $\gamma = 1$ makes all rewards count as equal.

f) $T(s, a, s')$ is the transition probability, that is the probability going from state s to state s' with action a. The a state of the game consist of the positions of the two body pixels and the head pixel and the position of the apple. Below are a few transition probabilities depending on the current state, the action and the next state. Since the transition densities are calculated given the states and the action, they are all deterministic except for the spawn of a new apple, which happens randomly uniformly over all 22 empty pixels. The deterministic implies that the probability is = 1, which it then is if the action is legal. If the

action is illegal

$$T(s,a,s') = \begin{cases} & s' = \{\text{the snakes head is one step away from the apple}\} \\ \frac{1}{22} \quad \text{if} & a = \{\text{action that leads the snake to the apple }\} \\ & s' = \{\text{snake head is on old apple } \forall \text{ possible new apples}\} \\ \\ & s = \{\text{any legal state}\} \textbf{ and} \\ 1 \quad \text{if} & a = \{\text{any legal action}\} \textbf{ and} \\ & s' = \{\text{the next snake position and same apple position}\} \\ \\ & s = \{\text{any illegal state }\} \textbf{ or} \\ 0 \quad \text{if} & a = \{\text{any illegal action }\} \\ & s' = \{\text{any illegal state }\} \end{cases}$$

Exercise 3

a) On- and off-policy is determined by which policy our RL agent uses to explore (learn) about the environment it acts in. If the policy used to gather information about the world (the behavioural policy) is the same as the one that we eventually want our agent to follow, we say that the method is On-Policy. If they differ, we say that the method is Off-Policy.

b) In model based reinforcement learning we aim to model the environment the agents acts in, with transition probabilities and rewards, and form the optimal policy from it. When the approach is model free, the learning process is more trial and error. The agent then just updates the policy as the trials that was the most rewarded.

c) In passive reinforcement learning the agent does not update the policy nor vary the actions, it simply evaluates it and the environment accordingly. In active RL the agent chooses its actions in order to find the optimal policy and explores the environment it works in.

d) The difference between supervised learning, unsupervised learning and reinforcement learning is stated in the following bullets:

- Supervised learning: We provide a model with trainable parameters, together with labeled data and train the parameters by minimising some loss function. Mainly used for classification or prediction.
- Unsupervised learning: We provide a model structure and unlabeled data, and the model tries to label the data by finding patterns and hidden structures. Mainly used for clustering/anomaly detection or generative modeling.

- Reinforcement learning: learning an agent to take sequential decisions in an environment by using a reward function. It gathers data itself by acting in this environment and finding what states and actions that are rewarded. Mainly applied in autonomous systems such as robotics, games or recommendation systems.

e) In dynamic programming we assume that we know the underlying Markov Decision Process, meaning that we know the transition probabilities and reward function, and then we try to find the optimal policy w.r.t. it. We can do this by for example policy iteration where we in each iteration try to improve the policy. In Reinforcement Learning we instead send our agent out to explore the environment it acts in and learn from the outcomes of action. In the Q-learning case we try to understand the dynamics of the Bellman Optimality equations in our system by exploring it and generating samples.

Exercise 4:

a) This exercise has a direct analogy to exercise 2a). We have that the optimal state value function is the probability weighted sum of reward plus discounted future value functions, given that we take the action a that maximizes the value. Since a probability weighted sum is the discrete expected value we immediately see that

$$\begin{aligned} V^*(s) &= \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')] \\ &= \max_a \left(\mathbb{E}[R(s,a,s')] + \gamma \mathbb{E}[V^*(s')] \right) \end{aligned} \tag{4}$$

b) Equation 4 simply says that the optimal state value in the current state is equal to the expected reward of the next action plus the discounted expected value of future state values.

c) Since it is the *optimal* state value, we coount with taking the action that maximizes the state value. It is similar to the instrucitons "step 1: take correct first action, step 2: Keep being optimal"

d) The corresponding equation for $V^*(s)$ is called the Policy Extraction Theorem and it is very similar:

$$\pi^*(s) = \operatorname*{argmax}_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')] \tag{5}$$

e) The relation between π^* and V^* is not as simple as the one between π^* and Q^* mainly because of what they represent. $Q^*(s,a)$ is the expected future rewards given that we will take action a from state s. $V^*(s)$ is the expected future rewards given that we are in state s and will take an action that maximises the

rewards. Therefore we must, in each step, evaluate all possible actions and take the best one.

Exercise 5:

a) The code for implementing policy evaluation is found below:

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

    action = policy(state_idx);
    value = values(state_idx);
    next_state = next_state_idxs(state_idx, action);
    if(next_state == 0) % death :(
        values(state_idx) = rewards.death;
    elseif(next_state == -1) % apple :)
        values(state_idx) = rewards.apple;
    else
        values(state_idx) = gamm*values(...
            next_state);
    end
    Delta = max(Delta, abs(value-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
```

The code for implementing policy improvement is found below:

```
policy_stable = true;
for state_idx = 1 : nbr_states
    % FILL IN POLICY IMPROVEMENT WITHIN THIS LOOP.

    old_action = policy(state_idx);
    action_rewards = zeros(nbr_actions,1);

    for action = 1:nbr_actions
        next_state = next_state_idxs(state_idx, action);
```

```
            if(next_state == 0)
                action_rewards(action) = rewards.death;
            elseif(next_state == -1)
                action_rewards(action) = rewards.apple;
            else
                action_rewards(action) = gamm*values(next_state);
            end
        end
        [~, policy(state_idx)] = max(action_rewards);
        if ~(old_action == policy(state_idx))
            policy_stable = false;
        end
    end
```

Both of the algorithms are implementations of the pseudo code found in the slides for lecture 10 (originally from Sutton and Barto).

b) In table 1 one finds the different number of policy iterations and evaluations for different values of γ. For $\gamma = 0$ we do not value future rewards at all, which does not make the snake favor going towards an apple. It therefore just goes around. When $\gamma = 1$ we consider all rewards equally important, no matter when in time we receive them. This makes algorithm iterate for ever, since there are always improvements to make. $\gamma = 0.95$ is a good value and makes the snake play optimally.

γ	Policy Iterations	Policy Evaluations
0	2	4
1	∞	∞
0.95	6	38

Table 1 Number of Policy Iterations and Evaluations for three values of discounting factor γ.

c) The number of policy iterations and evalutaions for different values of ϵ is found in table 2. The value is a threshold for when we consider value of each state and action to have converged. A low value requires more policy evaluations, but we also get a better resolution of how valuable the policy is in each state. This is why a high value evaluates the policy once in each iteration for $\epsilon > 1$, and considers the map to have converged. The poor mapping then results in the algorithm having to iterate more times to reach a stable policy. However, the snake is playing optimally in all cases.

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 Number of Policy Iterations and Evaluations for three values of discounting factor γ.

Exercise 6: a) The code for implementing an update of the Q-value is found below:

```
if terminate
    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
```

For the non terminal case we have:

```
sample                      = reward + gamm*...
                                 max(Q_vals(next_state_idx,:));
pred                        = Q_vals(next_state_idx, action);
td_err                      = sample - pred;
Q_vals(state_idx, action) = Q_vals(state_idx, action)...
                                 + alph*td_err;
```

b) The settings and score for the attempts of snake are found in the table 3. R(event) are the rewards for the different events. γ is the discount factor which tells us how much less we value rewards distant in the future. α is the learning rate which determines how much it should update its prediction of the value of a certain action at a certain state during training. ϵ is the rate at which the model does a random action during training. A higher value makes it explore more. The first attempt did not perform well. I hypothetizes that the very low α is what causes the snake not to learn very fast. Since it should be in the range (0,1) I begin with trying mid interval, $\alpha = 0.5$. Regarding the other parameters, I would like to experiment with the rewards and ϵ, but I begin with only α in order to easier attribute the effects to one parameter. This experiment also did

not work very well, with a score of zero as seen in the second row of table 3. Since my snake dies a lot, and fast, I adopt a new strategy. I want my snake to explore a lot in the beginning, to get a firm grasp of the reward structure. I also want it to be very afraid of dying. Furthermore I want it to explore more in the beginning. I therefore raise the penalty of death to -10, and initialise a parameter schema for α and ϵ where they are both starting high at 0.9 and then updated to 0.9 times the previous value every 100 iteration. This means that the snake does less and less exploring every 100th iteration and also learns less and less every 100th iteration. With these settings the snake becomes quite good at surviving, but it does not very often find the apple.

Attempt	R(apple)	R(death)	R(default)	γ	α	ϵ	Score
1	1	-1	0	0.9	0.01	0.01	0
2	1	-1	0	0.9	0.5	0.01	0
3	1	-5	0	0.9	0.9	0.9	0

Table 3 Tabular Q-learning Snake Game Attempt Results

c) I was not able to succeed.

d) The number of states is 4136 which makes it very hard to explore all possible states in only 5000 iterations. If one also considers the combinations of possible actions that the snake can take in each of the states one realises that it is tough to learn enough in only 5000 iterations.

Exercise 7:

a) The Q-update code for a terminated state is found below:

```
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);
```

The Q-update code for a non-terminated state is the following:

```
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) A summary of the three attempts are found in table 4. In the first attempt we carry on the "afraid to die"-approach, by penalising death quite high. We also use a feature, f_1 which is the value of the next pixel (for each of the three

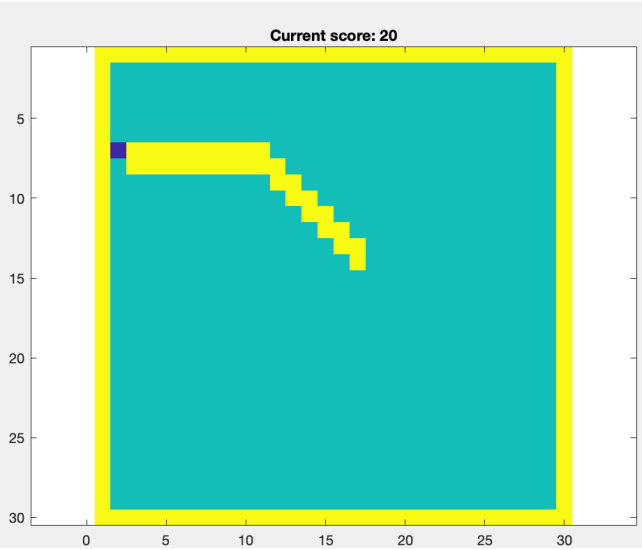

Figure 1 The zig-zag pattern that the snake creates when we have a feature that discriminates between going forward and turning. This results in a more compact placement of the snake, which proved to be slightly better.

possible actions). This makes sure that the snake is aware of the consequences of each action (except for crashing into itself. I tried to implement this but it was very difficult). The result was a snake that is very afraid of the border and almost only rattled around in the middle, eventually biting it's own tail. To give it a grasp of the grid, we initialise two new features, f_2, f_3 that are the horizontal and vertical distance to the apple. This results in a attempt 2 in the table, which is much better. Now the snake successfully targets the apple, and only has problems with it's own tail. A failed attempt, but with unexpected positive effects, to make it avoid the tail is found in attempt 3. Here I added a feature that is positive for going straight forward and negative for going to the sides. This idea came from an observation where the snake often took unnecessary turns which trapped itself. I thought that if I just nudge it slightly to preferring going straight instead of turning, it would get stuck less often. However, the weights for this was trained to be negative, which would imply that it eventually preferred turning. This made the snake zig-zag its way forward, as seen in figure 1. This was actually a quite efficient way to make the snake more compact and thus reducing the risk of hitting itself.

Table 4 Tabular Q-learning Snake Game Attempt Results

Attempt	R(apple)	R(death)	R(default)	γ	α	ϵ	Feats	Score
1	1	-5	0	0.9	0.5	0.1	f_1	0.02
2	1	-5	0	0.9	0.5	0.1	f_1, f_2, f_3	31.46
3	1	-5	0	0.9	0.9	0.9	f_1, f_2, f_3, f_4	32.01

$f_1 =$ content of next pixel in each direction
$f_2 =$ horizontal distance to apple
$f_3 =$ vertical distance to apple
$f_4 =$ difference between forward and sideways

The four weights for the four features were initialised as (where we want to initialise them with sign opposite to how they will end up):

- f_1: Positive. Since an apple has value -1 and the wall has value 1, we infer a positive value to be "bad". Thus we do not want to change sign by using our weight.
- f_2 and f_3: Positive, since we want to get closer to the apple, a large value of the distance to it is not good.
- f_4 Negative. I chose this feature to make the snake avoid unnecessary turns that would make it get stuck in itself. This was however a mistake from my side, since the snake actually preferred a negative value on this weight after training.

c) The final settings of the snake is found in table 5. The idea was to have high randomization and high learning in the beginning, and then to lower these values slowly in order to converge to a good strategy. This works quite well, the snake is erratic in the beginning but after most states are explored it slowly converges to a good strategy. What is not working very well is a lack of fear of its own tail. Crashing into itself is always the cause of death, and needs to be handled.

R(apple)	1
R(death)	-5
R(default)	0
γ	0.9
α	0.99
ϵ	0.99
α update interval	100
α update factor	0.9
γ update interval	100
γ update factor	0.8
$\mathbf{w}_0$	[1,1,1]
$\mathbf{w}$	[-1.1216, -0.0466, -0.0243]
$f_1 =$	content of next pixel in each direction
$f_2 =$	horizontal distance to apple
$f_3 =$	vertical distance to apple
Score	31.46

Table 5 The settings and initialisation for the final experiment of snake