Reinforcement Learning

Assignment 01

Wei Chen

1 Dynamic programming

1.1 Methods

Dynamic programming is a model-based methods for reinforcement learning, which means the transition probabilities $p(s'|s,a)$ are accessible. To solve the game *Stochastic Windy Gridworld* using Dynamic programming, value iteration (not policy iteration) consisting of policy evaluation and policy improvement is needed.

Firstly, we need to infinitely execute full sweep until the absolute error ($|Q(s,a) - Q\hat{}(s,a)|$) given is less than the threshold, which means the optimal value is obtained. Basically, the process of the full sweep represents iterating over the action space for each state. Then, due to the fact that the transition probabilities $p(s'|s,a)$ is known, we can update the Q value functions (i.e., state-action value functions) based on the previous ones, where the estimate of a state-action value is shown as the equation below:

$$Q'(s,a) := \sum_{s'}(p(s'|s,a) \cdot (r(s'|s,a) + \gamma \cdot \max_{a'} Q(s',a'))) \quad (1)$$

where $p(s'|s,a)$ and $(r(s'|s,a)$ denote the probabilities transition and reward from state s taking action a, to state s', respectively. (s' and a' are the next possible state and its corresponding action for current state s). Finally, for a new full sweep, the agent likewise needs to select actions based on the Q-value table to guide itself on the game and correspondingly update the Q-value. Once it converges, the optimal policy is obtained after the optimal value is computed.

1.2 Progression of Q-value iteration

It can be seen from figure 1 that at the very beginning, the change of most of the state-action value is small except for those states close to the goal state. At the midway iteration, for many states close to the goal state, their state-action values are already almost fixed, while for a few states close to the start state, their state-action values still change obviously over time. Finally, at the iteration of convergence, for each state, its largest Q-value is obviously larger than the second-largest Q-value, which means there is no need to get into the next iteration for updating.

1.3 The converged optimal value at the start

From figure 1 (bottom left sub-figure), we can see that there are four state-action values corresponding to four actions. They are: 17.3, 17.3, 17.4 and 18.3. According to one of the optimal Bellman equations (below):

$$v_*(s) = \max_a q_*(s,a) \quad (2)$$

we have:

$$V_*(s=3) = 18.3$$

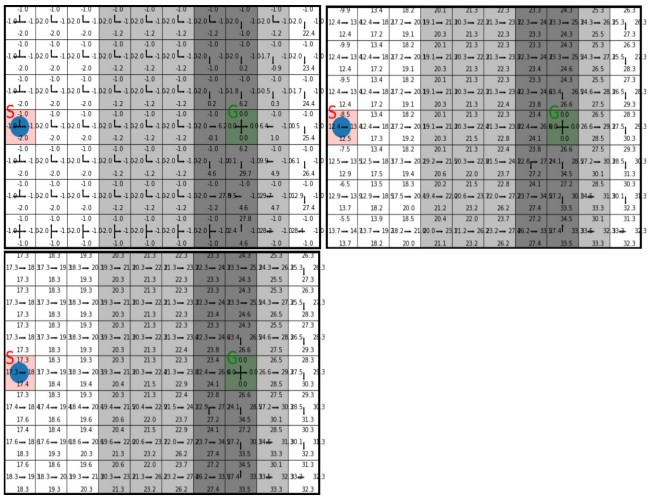

Figure 1: Progression of Q-value iteration, where the figures at top left, top right and bottom left represent the estimates of Q-value at the beginning, midway and convergence respectively. The choice of (hyper)parameters is as Table 1 shown

Selection of (hyper)parameters for Dynamic programming			
threshold	gamma	policy	lr
0.001	1.0	greedy	0.25

Table 1: Selection of (hyper)parameters for Dynamic programming, where lr represents the learning rate

1.4 Average reward per time step

Approximately equal to 1.1176.

1.5 Why does it still converge?

As stated, a full sweep is to iterate over the state space and their corresponding actions. Once the goal (or terminal) state is visited, function *env.model()* in *Environment.py* is called to return $p(s'|s,a)$. However, the probabilities of the transition from the goal state to the other 69 states are all zero. Therefore, the current process of updating the Q-value table is meaningless. In addition, when we let the agent play the game following the optimal policy, in each step, *env.step()* in *Environment.py* is recalled and there is no further step that will be performed if the new state is the terminal state, which is determined by this code: *np.all(self.agent_location == (7,3))*

Another way to solve this issue could be to check the last reward in the list of rewards. Only if the value of the last reward is equal to +35, means the agent just stepped into the terminal state.

1.6 Reflection

Essentially, Dynamic reinforcement divides a complicated problem into several smaller sub-problems. In comparison to RL, an advantage is the transition probability is available. In the simulated environment, applying DP would be a better choice. Moreover, a potential weakness could be the curse of dimensionality when solving those more complex problems.

2 Exploration

2.1 Methods

The ϵ-**greedy policy** is as the equation (3) shown:

$$\pi(a|s) = \begin{cases} 1.0 - \epsilon, & \text{if } a = arg\max_{a \in A} Q(s,a) \\ \epsilon / length(A), & \text{otherwise} \end{cases} \tag{3}$$

The **Boltzmann policy** (or softmax) is as the equation (4) shown:

$$\pi(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_{a' \in A} e^{Q(s,a')/\tau}} \tag{4}$$

where the A represents the action space given a state s for both two policies, and ϵ and τ are an parameter to tune the policies, respectively.

For the updating process of the q-learning method, the total time step (or budget) is given beforehand, which means the length of all episodes should not add up to a value greater than this (i.e., n_timestep).

Within an episode, the agent starts from the start state and then repetitively selects the action based on the ϵ-greedy or softmax policy to reach a new state until the terminal state is visited. After each step (or action) a, the agent step into a new state s' with a reward and need to correspondingly update the Q-value (Q(s,a)) based on the greedy policy. The updating of the Q-value table is as the equation (5) shown below:

$$Q(s,a) = Q(s,a) + \alpha \cdot (r + \gamma \cdot \max_{a'} Q(s',a') - Q(s',a')) \tag{5}$$

On the whole, for an updating, the agent needs to select an action a based on the ϵ-greedy in current state s, and step into the next state s' with a reward r. Then in the state s', the agent selects an action a' based on greedy policy (but not perform this action a') to get the maximum Q(s',a').

2.2 Comparison of exploration methods

The figure 2 draws a comparison between the explorations with 2 repetitions and 50 repetition. As the number of repetitions increases, all the curves tend to converge (be stable). In addition, after 50 repetitions, most of the curves almost overlap with each other, which means they are similar in performance. However, it can be obviously seen that the ϵ-greedy policy with ϵ equal to 0.02 (the green curve) has the worst performance. Personally, it is because of the greater ϵ value it takes, leading to a higher probability that it does not select the action with the corresponding maximum Q-value.

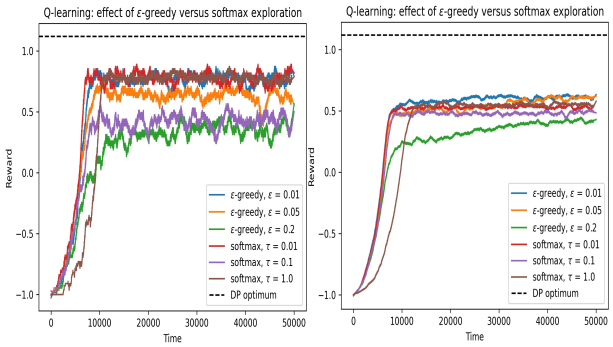

Figure 2: Comparison of exploration methods where the parameters *repetition* are 2 (left) and 50 (right), where the choices of the (hyper)parameters are both as Table 2 shown

2.3 Reflection

By and large, I prefer ϵ-greedy policy to sotfmax policy. Another better way than ϵ-greedy for exploration, would be Upper Confidence Bound.

Furthermore, it can be found that Q-learning (including other RL methods in the subsequent sections) does not achieve the optimal performance found from Dynamic programming. Personally, the main reason is that Q-learning is not a model-based method and then the transition probability is not available, that is, the MDP is unknown.

Selection of (hyper)parameters for exploration						
repetition	timesteps	gamma	policy	lr	ϵ	τ
2&50	50000	1.0	ϵ-greedy	0.25	0.01	0.01
			softmax		0.05	0.1
					0.2	1.0

Table 2: Selection of (hyper)parameters for exploration based on Q-learning, where lr represents the learning rate, and ϵ and τ are the hyperparameters of the ϵ-greedy and softmax policy

3 On-policy versus off-policy target

3.1 Methods

SARSA is similar to Q-learning in implementation to great extent. The major distinction between them is updating (or back-up). For SARSA, the agent likewise needs to follow the ϵ-greedy policy to select an action a and step into the next state s' based on the current state s. Then, instead of recalling greedy policy as what Q-learning does, ϵ-greedy policy is recalled again to select a new action a' (but not perform right now), based on the new state s'. The equation of an single updating is:

$$Q(s,a) = Q(s,a) + \alpha \cdot (r + \gamma \cdot Q(s',a') - Q(s',a')) \tag{6}$$

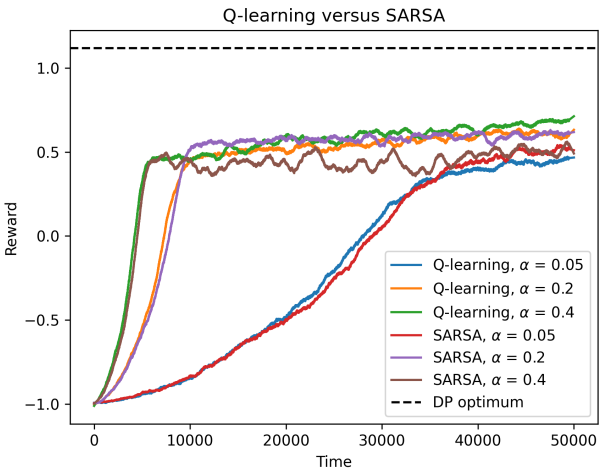

Figure 3: Comparison between On-policy (SARSA) and off-policy (Q-learning) with several different learning rates, where the choice of (hyper)parameters is as Table 3 shown

3.2 Comparison between the On-policy method and the off-policy method

As the figure 3 shown, Q-learning is almost the same as SARSA in terms of performance in the *Grid World* since the results for both two methods are all close to 0.5 finally. However, it can be seen that Q-learning is slightly better than SARSA in general. In addition, for both the two methods with 0.05 α (i.e., blue and green curves), they rise the slowest and finally reach values less than other curves, while the green and brown curves, which represent Q-learning and SARSA both with 0.4 α, go up quickly initially. The reason why the Q-learning method is at a slight advantage in such situation is that Q-learning directly learns and updates from the current optimal policy, while SARSA still needs to consider exploration, that is, update Q-value based on ϵ-greedy policy.

3.3 Reflection

For me Q-learning would be preferable, since it is better in this kind of simulated environment. However, SARSA is more conservative than Q-learning. Add to that Q-learning is prone to be influenced by data (e.g., the collected episodes), which means SARSA relatively ensures convergence to a certain extent. Therefore, if there are attempts to train robots in the real environment with costly negative rewards (e.g., real world), SARSA is better.

Finally, n-step Q-learning is similar to Q-learning to some extent (e.g., updating). But it is not strictly a full off-policy method, since the first n steps is based on the current behaviour policy (ϵ-greedy) and updating is mainly based on the behaviour policy.

Selection of (hyper)parameters for both Q-learning and SARSA					
repetition	timesteps	gamma	policy	lr	ϵ
50	50000	1.0	ϵ-greedy softmax	0.05 0.2 0.4	0.05

Table 3: Selection of (hyper)parameters for both Q-learning and SARSA, where lr represents the learning rate and ϵ is the hyperparameter for ϵ-greedy

4 Depth of target

4.1 Methods

Unlike Q-learning or SARSA, the implementation of N-step or MC here requires a complete episode to be collected beforehand - that is, this episode contains a set of continuous state-action-reward, e.g., $[S_1, A_1, R_2, S_2, A_2, R_3, ..., R_T, S_T]$, where S_T indicates the last state in this episode but not necessarily represents the terminal state (T is a hyperparameter we could tune). Additionally, Q-learning or SARSA generally counts only one reward before bootstrapping, n-step Temporal-Difference methods, including n-step Q-learning and Monte Carlo, need to bootstrap an estimate after n transitions (i.e., sum up the first n rewards).

4.1.1 N-step Q-learning

As stated above, given a parameter *n_timestep* that limits the sum of the length of all episodes, a single episode will be collected first. (Computing the returns G_t for n-step methods requires performing at least n steps) Next, the returns G_t for each time step t is computed:

$$G_t = \begin{cases} \sum_{i=0}^{m-1} (\gamma^i \cdot r_{t+i}) & \text{if } s_{t+m} = s_T \\ \sum_{i=0}^{m-1} (\gamma^i \cdot r_{t+i}) + \gamma^m \cdot \max_a Q(s_{t+m}, a) & \text{otherwise} \end{cases} \quad (7)$$

where $m = min(n, ep - t)$, i.e., take the smallest value of these two and s_T represents the last state for an episode.

Finally, update the Q-value function $Q(s_t, a_t)$ like what Q-learning or SARSA does but with different bootstraps depending on n:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \cdot (G_t - Q(s_t, a_t)) \quad (8)$$

4.1.2 Monte Carlo

As n-step Q-learning, given a parameter *n_timestep* (outer-loop) that decreases over time steps, for each iteration of the outer-loop, collect a complete episode. Then, bootstrap is omitted. Returns G_t for each time step t is:

$$G_t = \sum_{i=0}^{\infty} \gamma^i \cdot r_{t+i} = r_{t+1} + \gamma \cdot r_{t+2} + ...\gamma^{T-t} \cdot r_T \quad (9)$$

where r_T denotes the last resultant reward of an episode collected previously. Finally, the same updating equation (see equation 8) is introduced.

4.2 Comparison between back-up depths

The figure 4 shows the comparison of the n-step TD methods with different parameters *max_episode_length* (they are: 500, 1000, 2000, and 5000), where the selection of the (hyper)parameters (except for the parameter T) is as the table 4 shown. It can be seen that there is a proportional decrease in performance for all methods as the

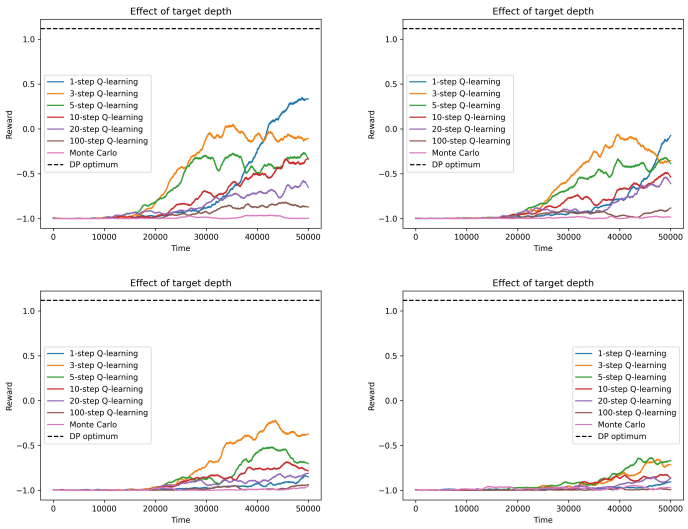

Figure 4: Comparison of the n-step TD methods with different parameters *max_episode_length*: 500(top left), 1000(top right), 2000(bottom left) and 5000(bottom right) over 50 repetitions, where the choice of (hyper)parameters is as Table 4 shown

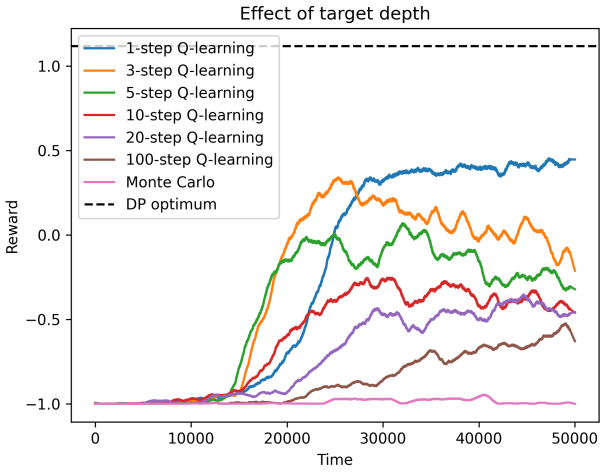

Figure 5: Comparison between n-step TD methods including n-step Q-learning and Monte Carlo method over 50 repetitions, where the choice of (hyper)parameters is as Table 4 shown

parameter *max_episode_length* increases. The reason might be that the parameter *n_timesteps* is not long enough, that is, collecting the first few episodes is similar to the random walk in which the agent is likely to take many more steps to reach the terminal state, leading to a small number of episodes for updating (or training). However, an intelligent agent generally needs a huge range of episodes (or samples).

Selection of (hyper)parameters for n-step Q-learning						
repetition	timesteps	gamma	policy	lr	T	Depth
50	50000	1.0	ϵ-greedy	0.25	100 500 1000 2000 5000	1 3 5 10 20 100
Selection of (hyper)parameters for Monte Carlo method						
repetition	timesteps	gamma	policy	lr	T	Depth
50	50000	1.0	ϵ-greedy	0.25	100 500 1000 2000 5000	None

Table 4: Selection of (hyper)parameters for n-step Q-learning and Monte Carlo method, where lr and T represent learning rate and the maximum time step of an episode, respectively

In addition, it can be concluded from figure 5 that the 1-step Q-learning has the best final performance and its average reward per time step is plateaued (close to 0.5) after 50 repetitions. The reason is that 1-step Q-learning is similar to Q-learning which the average rewards curves also reach around 0.5. Additionally, 3-step Q-learning (orange curve) and 5-step Q-learning (green curve) both learn faster initially.

4.3 Reflection

The benefits of the n-step TD methods could be that i) No need to collect an episode from the start state to the terminal state (but requires continuous episode) compared to MC. ii) Does not need transition probability compared to DP, while a potential problem is that the target (or returns) will have high variance as the n increases, which finally leads to the Monte Carlo method. For MC, it also does not require the transition probability compared to DP but needs the fully complete episode from the start state to the terminal state. Apropos of the bias-variance trade-off, 1-step TD methods and Monte Carlo are as if two extremes. The former one has low variance but high bias, while the latter one tends to have high variance but low bias in terms of the target (or returns) G_t.

Personally, according to the experiments and the results above, I prefer 1-step Q-learning. It can be found from the figure 5 that, however, 3-step Q-learning and 5-step Q-learning learn faster than other n-step methods initially and they gradually drop below 0. Hence, for this task, I believe 1-step Q-learning has the highest chance of convergence.

4.3.1 Curse of dimensionality

The tabular RL methods tend to quickly find the optimal solutions for those easy and small tasks. However, tabular RL methods prone to the "curse of dimensionality" when trying to sovle comlex problems that have exceptional number of dimensions resulting in huge data set, e.g., those games with a huge mass of features. For deep

learning, over-fitting is a direct result of the "curse of dimensionality". We may therefore introduce cross-validation or dimensional reduction methods (e.g., PCA) to overcome this "curse".

5 Conclusion

For a simulated environment, dynamic programming should be considered first. In contrast, it would be better to introduce model-free methods in the real world. To explore, ϵ-greedy policy and softmax policy will serve the purpose. Additionally, TD methods Q-learning and SARSA are similar in implementation. The main difference, however, is that the off-policy methods take two different policies for performing action and updating, while the on-policy methods take only one. Finally, 1-step TD methods and Monte Carlo can be seen as the two opposing extremes for n-step methods.

References