Lecture 8 Reinforcement Learning Advanced: Q-Learning

8.1 Review of Reinforcement Learning

A key differentiator of reinforcement learning from supervised or unsupervised learning is the presence of two things:

- An environment: This could be something like a video game or the stock market, etc.
- An agent: This is the AI that learns how to operate and succeed in a given environment.

The way the agent learns how to operate in the environment is through an iterative feedback loop. The agent first takes an action, and as a result, the state will change based on the rewards either won or lost.

Mathematically, we use the Bellman equation to describe the fundamental problem in reinforcement learning. In particular, the Bellman Equation include:

- S: State
- a: Action
- R: Reward
- γ: Discount factor

Here is the Bellman equation for **deterministic** environments:

$$V(s_t) = \max_{a^*} R(s_t, a_t) + \gamma V(s_{t+1})$$

A Q-table is a lookup table that calculates the expected future rewards for each action in each state. This lets the agent choose the best action in each state. We learn the value of the Q-table through an iterative process using the Q-learning algorithm, which uses the Bellman Equation.

Here's a brief explanation of the Bellman Equation:

- The value of a given state s_t is equal to the reward of reward-max action a_t, which means of all the available actions in the state we're in, we pick the one that maximizes the reward value of a given state s_t.
- We take the reward of the optimal action $a *_t$ in state s_t and take into account of a multiplier of γ, which is the discount factor that diminishes our reward over time.
- Each time we take an action, we switch to the next state: s_{t+1}. This is where dynamic programming comes in, since it is recursive we take s_{t+1} and put it back into the value function $V(s_{t+1})$ to seek reward-max action $a *_{t+1}$.

However, the transition of states is stochastic over time. We actually do not know what s_{t+1} will be. In. Instead, we have to use the expected value of the rewards from the next state.

$$V(s_t) = \max_{a^*} R(s_t, a_t) + \gamma \sum_{n=1}^{N} P(s_t, a_t, s_{t+1} | \pi^n) V(s_{t+1})$$

The value function $V(s_t)$ is essentially a critic. It does not determine the agent's actions directly. It evaluates how good or how bad is the agent's action. For any agent π, the critic $V^{\pi}(s_t)$ evaluates how much is the maximum reward made by the agent's action. $V^{\pi}(s_t)$ takes a state s_t as the input and outputs a scalar. It represents when agent π sees the state s_t, the agent takes the action that leads to the optimal expected reward.

$$s_t \rightarrow V^{\pi}(s_t) \rightarrow scalar = \begin{cases} 1, if\ V^{\pi}(s_t)\ is\ large \\ 0, if\ V^{\pi}(s_t)\ is\ small \end{cases}$$

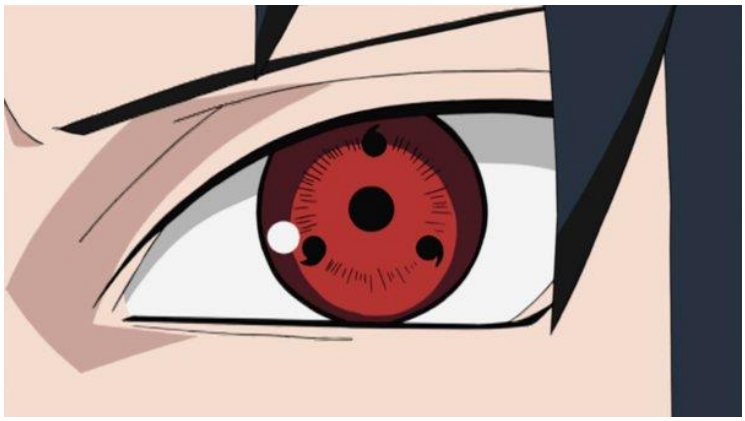
It is noteworthy that even when the state s_t is the same, as long as the agent is different, the rewards can be different. This is due to two reasons:

- First, different agents may have different time discount factor γ.
- Second, the actions taken by different agents may influence the state transitions in completely different ways. For the same state, one agent π^i may take an optimal action that receives receive high score, while the other agent π^j may take an optimal action that leads to game over. The output of the value function depends on the state s_t, the action a_t and the agent π.

Here is an example: $s_t = Sharingan, \pi^i = Itachi\ Uchiha, \pi^j = Sasuke\ Uchiha$

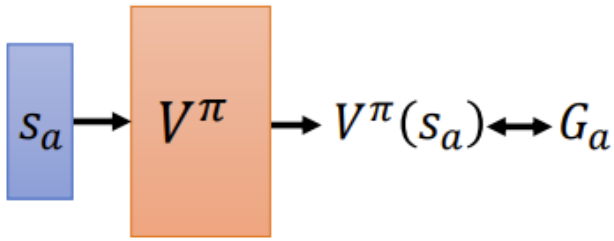
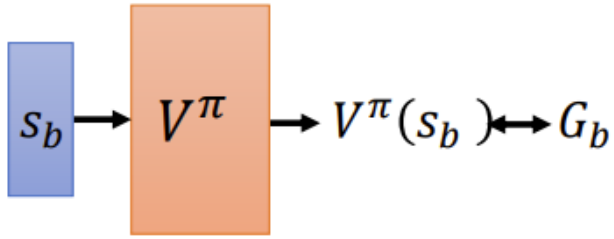
$$V^{Itachi\ Uchiha}(Sharingan) = strong\ (Mangekyo\ Sharingan)$$

$$V^{Sasuke\ Uchiha}(Sharingan) = weak\ (Three\ Tomoe)$$



Monte-Carlo Approach

The value function $V^{\pi}(s_t)$ watches agent π playing the game and then calculate the accumulated reward.

After seeing the state s_a Until the end of the episode, the accumulated reward is supposed to be G_a.	
After seeing the state s_b Until the end of the episode, the accumulated reward is supposed to be G_b.	

However, there is no way we could exhaust all possibilities. $V^{\pi}(s)$ is essentially a neural network, with the training being a regression problem.

8.2 Q Learning

Intuitively you can think of the Q-value as the quality (a critic) of each action. Instead of looking at the maximum reward value of each state $V(s_t)$. We're going to look at the reward values of each state-action pair, which is denoted by $Q(s_t, a_t)$.

Let's look at how we actually derive the value of $Q(s_t, a_t)$ by comparing it to $V(s_t)$. As we just saw, here is how we obtain $Q(s_t, a_t)$ from the Bellman equation for $V(s_t)$ in a stochastic environment:

By performing any action a_t given a state s_t, we can receive the reward $R(s_t, a_t)$.

Now the agent switches to the next state s_{t+1}. Because the next state can have N different possibilities, we add up the expected value of the next state, which is the probability weighted sum maximum reward of the next state:

$$Q(s_t, a_t) = R(s_t, a_t) + \gamma \sum_{n=1}^{N} P(s_t, a_t, s_{t+1} | \pi^n)\, V(s_{t+1})$$

What you will notice looking at this equation is that $Q(s_t, a_t)$ refers to all possible values of the first part in the Bellman equation. In this sense, the value of a state $V(s_t)$ is the maximum of all the possible Q-values $Q(s_t, a_t)$.

Let's take a relook at Bellman Equation. We now can get rid of $V(\cdot)$, since $V(\cdot)$ is a recursive function of $V(\cdot)$. We're going to replace the $V(s_{t+1})$ with $Q(s_{t+1}, a_{t+1})$.

$$Q(s_t, a_t) = R(s_t, a_t) + \gamma \sum_{n=1}^{N} P(s_t, a_t, s_{t+1} | \pi^n) \max_{a^*} Q(s_{t+1}, a_{t+1})$$

This is the recursive formula for the Q-value.

8.2 Deep Q Learning

Deep Q-learning is an extension of Q-learning, which is a reinforcement learning algorithm that learns to estimate the value of being in a particular state and taking a specific action.

For now, let's use the deterministic (I.e. ignore the probabilistic components) Bellman equation for simplicity, which to recap is:

$$Q(s_t, a_t) = R(s_t, a_t) + \gamma \max_{a^*} Q(s_{t+1}, a_{t+1})$$

The equation tells two facts:

- First, we know that before an agent π^n takes an action it has a Q-value map: $Q(s_t, a_t)$. After an action is taken, we know what reward the agent receives. The reward value at the timestamp t is: $R(s_t, a_t) + \gamma \max_{a^*} Q(s_{t+1}, a_{t+1})$. However, due to the stochastic market environment, the reward is uncertain, represented by stochastic process and should be estimated by deep neural networks.
- Second, the above equation represents the update rule of the Q-values. Q values are estimates of the expected future rewards for taking a particular action in a specific state. Q-value for a state-action pair should be updated towards the sum of the immediate reward $R(s_t, a_t)$ and the discounted estimate of the future rewards, based on the maximum Q-value of the next state.

We can add a learning rate η to control the trade-off between exploiting existing knowledge and incorporating new information.

$$Q(s_t, a_t) = (1 - \eta) \cdot Q(s_t, a_t) + \eta \cdot \left\{ R(s_t, a_t) + \gamma \max_{a^*} Q(s_{t+1}, a_{t+1}) \right\}$$

η is the learning rate, controlling the degree to which newly acquired information overrides old information. The Bellman Equation above sets $\eta = 1$.

The **temporal difference** is defined as follows:

$$TD(a_t, s_t) = R(s_t, a_t) + \gamma \max_{a^*} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$$

The TD equation essentially says that the Q-value for a state-action pair should be updated towards the sum of the immediate reward $R(s_t, a_t)$ and the discounted estimate of the future rewards, based on the maximum Q-value of the next state.

In deep Q-learning, neural networks are used to approximate the Q-values, and the TD equation is used to update the weights of the network during training. The network takes a state as input and outputs Q-values for all possible actions. The TD error, which is the difference between the current estimate and the updated estimate obtained from the TD equation, is used to adjust the network's weights through techniques like gradient descent.

8.3 Example: Monte Carlo vs Temporal Difference

A game has the following 8 trajectories. An agent π interacts with the environment 8 times and receives the following rewards. Let's use $V^{\pi}(s_t)$ to estimate the value of state.

For s_b: among the 8 trajectories it plays, there are 2 trajectories s_b receives reward 1 and 6 trajectories s_b receive 0. The accumulated rewards of s_b is thus ¾ when game ends. What is the expected rewards of s_a

$$V^{\pi}(s_b) = 1/4$$

$$V^{\pi}(s_a) = ?\, 0 \; or \frac{1}{4}?$$

Monte Carlo: $V^{\pi}(s_a) = 0$ because s_a only shows once and the reward is 0.

Temporal-difference: $V^{\pi}(s_a) = V^{\pi}(s_b) + r$, where $V^{\pi}(s_b) = 1/4$ and $r = 0$.

$\pi^1: s_a, r = 0, s_b, r = 0, END$

$\pi^2: s_b, r = 1, END$

$\pi^3: s_b, r = 1, END$

$\pi^4: s_b, r = 1, END$

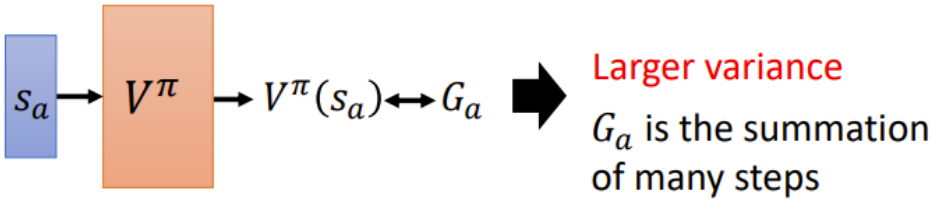
$\pi^5: s_b, r = 1, END$

$\pi^6: s_b, r = 1, END$

$\pi^7: s_b, r = 1, END$

$\pi^8: s_b, r = 0, END$

Monte-Carlo Approach has a big issue: the accumulated reward G_a is the summation of many steps (sum of many random variables, i.e. sum of rewards of all timestamps, e.g. G_a is a Brownian Motion). $Var[G_a]$, i.e. variance of reward, can grow very large as time goes (long episode). This leads to the Monte-Carlo approach to have very large variance.

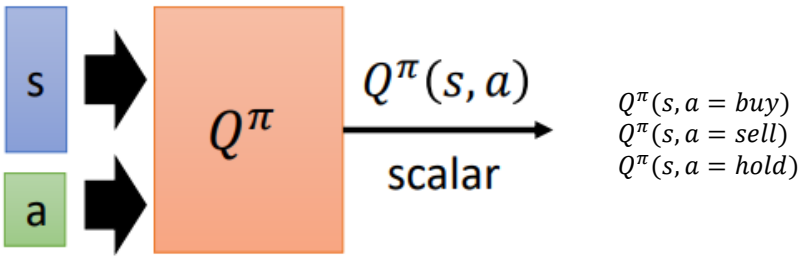


Temporal- difference (TD) Approach only needs to minimize the gap between the difference of $\max_{a^*} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$ and R_t. The key issue carried by Temporal- difference (TD) Approach is that the estimation of $\max_{a^*} Q(s_{t+1}, a_{t+1})$ may not be precise, which leads to the learning result of the critic function to be biased (inaccurate). Nowadays, Temporal- difference (TD) Approach is more common in training deep Q network.

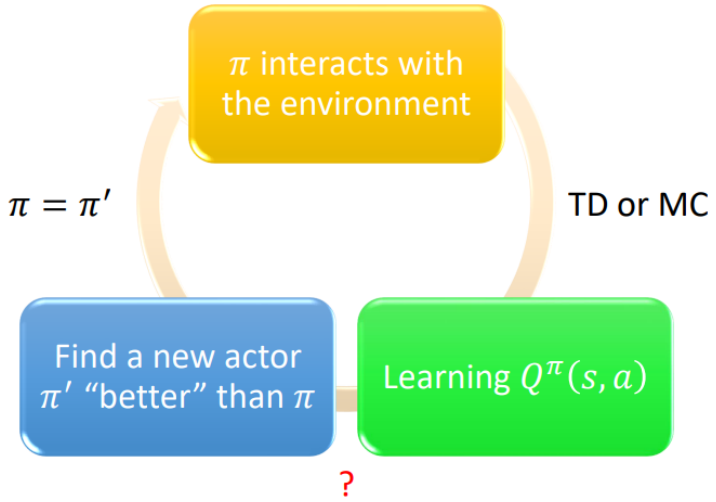
8.4 Algorithm Anatomy

8.4.1 Q-Value Function

The state-action value function $Q^{\pi}(s, a)$ means when using the agent π, the cumulated reward expects to be obtained after taking a at state s. The input of state-action value function is no longer the state s alone, but a state-action pair (s, a). The underlying fact is the agent may not take action a when observing the state s, but the state-action value function requires the agent π to take the action a.



Every time the state-action value function $Q^\pi(\cdot)$ receives a state s, the function returns several reward values, each corresponds to a particular action (This holds true for Q-learning with discrete actions, but not for continuous actions.) In this sense, the state-action value function $Q^\pi(s,a)$ can evaluate the performance of any one of the three possible actions taken by a particular agent π.



Now we arbitrarily select an agent π, who doesn't have to be the optimum one. The agent will collect data via its interaction with the environment. We then learn the Q function value (i.e. the expected reward of agent π by taking action a given the state s) using either Monte-Carlo (MC) approach or the Temporal- difference (TD) approach. Once you've learned the Q function value, you could find a new policy π'. The new policy is "better than" the original policy π. Now, you replace the policy π with the new policy π' to obtain the new Q-function $Q^{\pi'}(s,a)$ (i.e. the expected rewards of the new policy π'). You then look for a new agent π" who is better than π'.

Definition: Given $Q^{\pi'}(s,a)$, find a new agent π' that is "better" than π.

"Better" means $V^{\pi'}(s) \geq V^{\pi}(s)$ for all state s and

$$\pi'^{(s)} = argmax_a\, Q^\pi(s,a)$$

$Q^\pi(s,a)$ means given a state s, we calculate the Q function values for all possible actions a. We select action a that leads to the maximum of Q function for the new agent π' (given the state s).

- It is noteworthy that we are not building a new network to learn π', but π' is learned from Q value function. As long as we know $Q^\pi(s,a)$, we can solve for π'.
- We need to solve an $\arg max$ problem, if action a is discrete, we can calculate the Q value function iteratively. If the action is continuous, the Q value function is of the challenge.

Proof: Existence of $\pi'^{(s)} = argmax_a\ Q^\pi(s,a)$, which satisfies means $V^{\pi'}(s) \geq V^\pi(s)$ for all state s.

$$V^\pi(s) = Q^\pi(s, \pi(s))$$

$\pi(s)$ represent the action you are about to take when given the state s. $\pi(s)$ may not be optimum. $Q^\pi(s,a)$ represents the optimum reward value when taking the optimum action a, which is not necessarily equal to $V^\pi(s)$.

$$V^\pi(s) = Q^\pi\big(s, \pi(s)\big) \leq \max_a Q^\pi(s,a)$$

Therefore,

$$V^\pi(s) \leq Q^\pi\big(s, \pi'(s)\big)$$

It means if I take the action $\pi'(s)$, the expected reward I receive is higher than when I take the action $\pi(s)$. The intuition behind is that if we take action $\pi'(s)$ instead of $\pi(s)$ in every step, we will be able to receive higher total reward in the end. In particular,

$$Q^\pi\big(s, \pi'(s)\big) = E[r_t + V^\pi(s_{t+1}) | s_t = s, a_t = \pi'(s_t)]$$

Given the state s at t, we will take the action $a_t = \pi'(s_t)$, we receive the reward r_t then the state evolves to s_{t+1}. $V^\pi(s_{t+1})$ is the reward value for taking action $\pi(s_{t+1})$ in state s_{t+1}. The above equation satisfies the following condition.

$$Q^\pi\big(s, \pi'(s)\big) \leq E[r_{t+1} + Q^\pi(s_{t+1}, \pi'(s_{t+1})) | s_t = s, a_t = \pi'(s_t)]$$

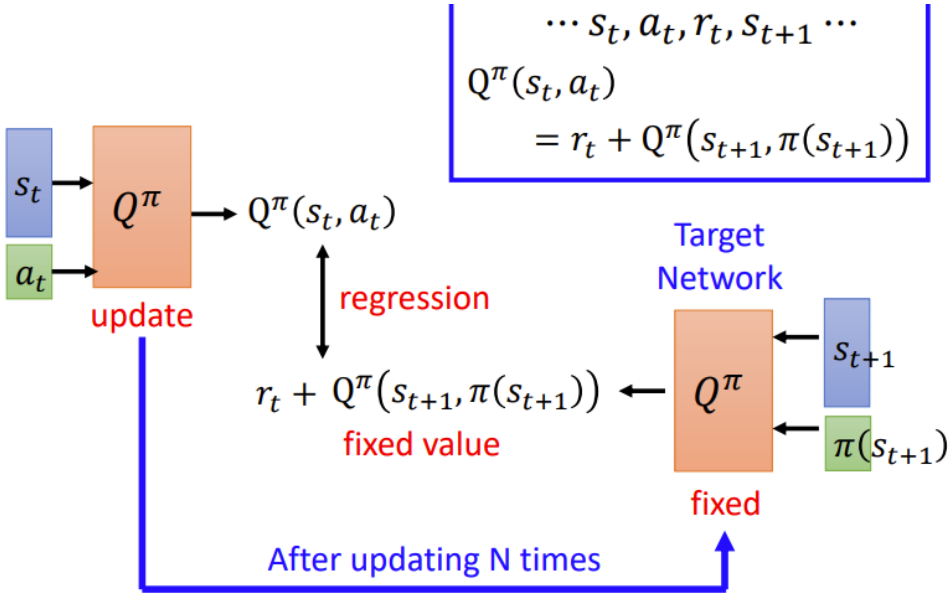
Because $V^\pi(s) \leq Q^\pi\big(s, \pi'(s)\big)$, we know

$$V^\pi(s_{t+1}) \leq Q^\pi\big(s_{t+1}, \pi'(s_{t+1})\big)$$

Compared with taking action $\pi(\cdot)$ for all states, if you take the action $\pi'(\cdot)$ at one timestamp $t+1$, you will receive larger expected rewards.

$$Q^\pi\big(s, \pi'(s)\big) \leq E\big[r_{t+1} + r_{t+2} + Q^\pi\big(s_{t+2}, \pi'(s_{t+2})\big) | \dots\big] \dots \leq V^{\pi'}(s)$$

8.4.2 Target Network



Description

At state s_t you take action state a_t and receive reward state r_t, then the environment switches to state s_{t+1}. Then Q function tells you the difference between the immediate reward at state t and the expected reward at state $t+1$ is r_t (i.e. $R(s_t, a_t)$ in previous sections).

The training process is a challenge. Assume $Q^\pi(s_t, a_t)$ is a regression problem, you will find your objective function to be stochastic. The method to resolve the issue is to fix $Q^\pi(s_{t+1}, \pi(s_{t+1}))$ during the training of $Q^\pi(s_t, a_t)$ and do not update its parameter values. $Q^\pi(s_{t+1}, \pi(s_{t+1}))$ is thus the "Target Network." The training updates parameters of $Q^\pi(s_t, a_t)$ alone. In this sense, we minimize the root mean squared error between $Q^\pi(s_t, a_t)$ and $Q^\pi(s_{t+1}, \pi(s_{t+1}))$. We update $Q^\pi(s_t, a_t)$ for N times (using gradient descent), and then we replace the parameter values in the target network using the estimates. Once you replace the parameter values in the target network, the target value changes. We then train the $Q^\pi(s_t, a_t)$ again.

8.4.3 Exploration

Description: The policy is based on Q-function. Given the state s, we enumerate all actions and find the action that leads to the maximum expected rewards. The action is a.

$$a = arg \max_a Q^\pi(s, a)$$

This is different from policy gradient. In policy gradient, the output a is stochastic. What we receive is a distribution of the action. We then sample from the distribution. We take a different action every time we sample the data.

The above method is not a good way for data collection.

$$s = \begin{cases} a_1, Q(s, a_1) = 0 \\ a_2, Q(s, a_2) = 1 \\ a_3, Q(s, a_3) = 0 \end{cases}$$

You can only know the Q value when you take the action a_i at the state s. You won't be able to know the Q value if you never take the action. If the Q function is a deep network, the problem may be resolvable. If Q is a table, the Q function will never return the value for a state-action pair it has never seen. When you sample $Q(s, a_2)$ and know its reward is 1 (better than 0). You choose a_2 for the state s. Then every time you sampled the state s, you always take a_2. Other actions are never sampled.

Solution: Allow for sampling a_1 and a_3 occasionally.

Method 1 Epsilon Greedy

$$a = \begin{cases} \arg\max_a Q(s,a) & with\ probability\ 1-\varepsilon \\ random, & otherwise \end{cases}$$

Let $\varepsilon = 10\%$, then 90% of time we determine the action using Q function, but the rest 10% our action is all random. It is often to make ε decay during the training process.

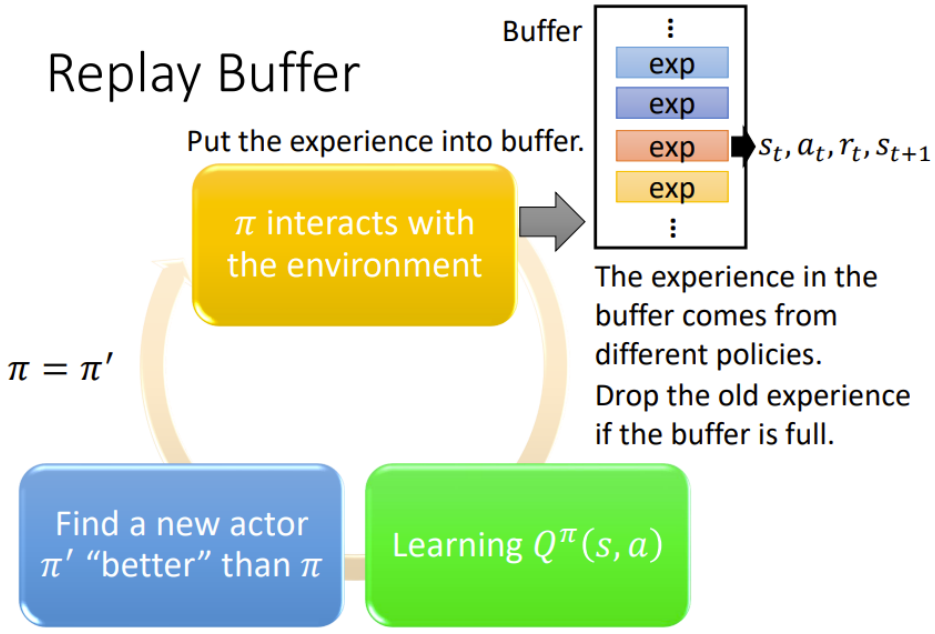
Method 2 Boltzmann Exploration

$$p = \frac{\exp[Q(s,a)]}{\sum_a^A \exp[Q(s,a)]}$$

This is in similar spirit as the policy gradient. Policy gradient chooses an action based on the probability distribution from the expected action space. Here, the higher the Q value, the larger the probability an action will be sampled. The smaller the Q value, the smaller the probability an action will be sampled.

In practice, $Q(s, a)$ is specified as a deep neural network. All actions may leads to similar Q values. The distribution of Q function may be close to uniform distribution.

8.4.4 Relay Buffer



Assume you have a Buffer size of 50,000. For each policy π, you allow it to interact with the environment for 10,000 times and record the experience $(s_t. a_t, r_t, s_{t+1})$ into buffer. The buffer now may have path sampled from five different policies. Then for each policy π, you are ready to learn its Q function.

There are two advantages of using buffers.

- First, the most time-consuming step in reinforcement learning is to let the policy interact with the environment. Using replay buffer could significantly reduce the number of times you interact with the environment. Now your experience is no longer from a particular policy alone. You could also recycle experiences from other policies that are saved in the buffer.
- Second, in order to train network successfully, we want the data saved in one batch the more diverse the better. When experiences in batch are from different policy,

You first sample from the batch and obtain a batch of experience: $(s_i. a_i, r_i, s_{i+1})$ for $i = 1 \dots T$. Then calculate target network $\hat{Q}$ using the sampled batch. In particular, we first choose action a (given the state s_{i+1}) that lead to the maximum value of target network $\hat{Q}$ using the sampled batch. We then update the parameters in $Q(\cdot)$ to make $Q(s_i, a_i)$ stay close to the target. We can update the parameters in $Q(\cdot)$ for $C = 100$ times, we then reset the target network to be $Q(\cdot)$ to repeat the above process.

8.5 Summary: Q-Learning Algorithm

Initialize Q-function Q and target Q-function. You set the network to be trained and the target network to be the same: $Q = \hat{Q}$.

In each episode

For each time step t

- Given state s_t, take action a_t based on Q (e.g. epsilon greedy)
- Obtain reward r_t, and reach new state s_{t+1}
- Store the experience $(s_t. a_t, r_t, s_{t+1})$ into buffer.
- Sample $(s_i. a_i, r_i, s_{i+1})$ from buffer (usually a batch)
- Define the target network $y = r_i + \max_a \hat{Q}(s_{i+1}, a)$
- Update the parameters of Q to make $Q(s_i, a_i)$ close to the target network y (regression)
- For every C sweeps (for example, $C = 100$), reset $\hat{Q} = Q$

Reference

https://www.mlq.ai/deep-reinforcement-learning-twin-delayed-ddpg-algorithm/

https://www.mlq.ai/deep-reinforcement-learning-q-learning/

https://www.mlq.ai/deep-reinforcement-learning-for-trading-with-tensorflow-2-0/

https://speech.ee.ntu.edu.tw/~tlkagk/courses_MLDS18.html

https://gym-trading-env.readthedocs.io/en/latest/