Chapter 14 Reinforcement Learning

Artificial intelligence = deep learning + reinforcement learning

− David Silver

Reinforcement learning is another field of machine learning besides supervised learning and unsupervised learning. It mainly uses agents to interact with the environment in order to learn strategies that can achieve good results. Different from supervised learning, the action of reinforcement learning does not have clear label information. It only has the reward information from the feedback of the environment. It usually has a certain lag and is used to reflect the "good and bad" of the action.

With the rise of deep neural networks, the field of reinforcement learning has also developed vigorously. In 2015, the British company DeepMind proposed a deep neural network-based reinforcement learning algorithm DQN, which achieved a human level performance in 49 Atari games such as space invaders, bricks, and table tennis [1]. In 2017, the AlphaGo program proposed by DeepMind defeated Ke Jie, the No. 1 Go player at the time by a score of 3:0. In the same year, the new version of AlphaGo, AlphaGo Zero, used self-play training without any human knowledge defeated AlphaGo at 100:0 [3]. In 2019, the OpenAI Five program defeated the Dota2 world champion OG team 2:0. Although the game rules of this game are restricted, it requires a super individual intelligence level for Dota2. With a good teamwork game, this victory undoubtedly strengthened the belief of mankind in AGI.

In this chapter, we will introduce the mainstream algorithms in reinforcement learning, including the DQN algorithm for achieving human-like level in games such as Space Invaders, and the PPO algorithm for winning Dota2.

14.1 See it soon

The design of reinforcement learning algorithm is different from traditional supervised learning, and contains a large number of new mathematical formula derivations. Before entering the learning process of reinforcement learning algorithms, let us first experience the charm of reinforcement learning algorithms through a simple example.

In this section, you don’t need to master every detail, but should focus on intuitive experience and get the first impression.

14.1.1 Balance bar game

The balance bar game system contains three objects: sliding rail, trolley and pole. As shown in Figure 14.1, the trolley can move freely on the slide rail, and one side of the rod is fixed on the trolley through a bearing. In the initial state, the trolley is located in the center of the slide rail and

the rod stands on the trolley. The agent controls the balance of the rod by controlling the left and right movement of the trolley. When the angle between the rod and the vertical is greater than a certain angle or the trolley deviates from the center of the slide rail after a certain distance, the game is deemed to be over. The longer the game time, the more rewards the game will give, and the higher the control level of the agent.

In order to simplify the representation of the environment, we directly take the high-level environment feature vector s as the input of the agent. It contains a total of four high-level features, namely: car position, car speed, rod angle, and rod speed. The output action a of the agent is to move to the left or to the right. The action applied to the balance bar system will generate a new state, and the system will also return a reward value. This reward value can be simply recorded as 1, which is instantaneously adding 1 unit time. At each time stamp t, the agent generates an action a_t by observing the environment state s_t. After the environment receives the action, the state changes to s_{t+1} and returns the reward r_t.

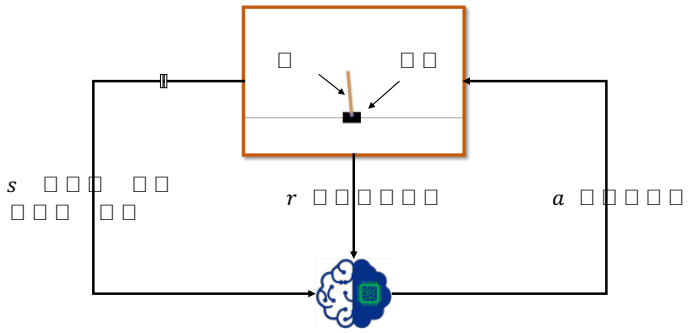


Figure 0.1 Balance bar game system

14.1.2 Gym platform

In reinforcement learning, the robot can directly interact with the real environment, and the updated environment state and rewards can be obtained through sensors. However, considering the complexity of the real environment and the cost of experiments, it is generally preferred to test algorithms in a virtual software environment, and then consider migrating to the real environment.

Reinforcement learning algorithms can be tested through a large number of virtual game environments. In order to facilitate researchers to debug and evaluate algorithm models, OpenAI has developed a Gym game interactive platform. Users can use Python language to complete game creation and interaction with only a small amount of code. It's very convenient.

The OpenAI Gym environment includes many simple and classic control games, such as balance bar and roller coaster (Figure 14.2). It can also call the Atari game environment and the complex MuJoco physical environment simulator (Figure 14.4). In the Atari game environment, there are familiar mini-games, such as Space Invaders, Brick Breaker (Figure 14.3), and racing. Although these games are small in scale, they require high decision-making capabilities and are very suitable for evaluating the intelligence of algorithms .

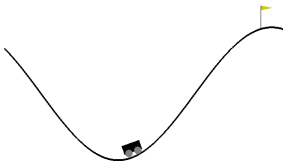


Figure 0.2 Roller coaster

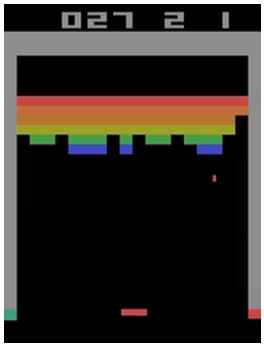


Figure 0.3 Break Breaker

Figure 0.4 Walking robot

At present, you may encounter some problems when installing the Gym environment on the Windows platform, because some of the software libraries are not friendly to the Windows platform. It is recommended that you use a Linux system for installation. The balance bar game environment used in this chapter can be used perfectly on the Windows platform, but other complex game environments are not necessarily.

Running the pip install gym command will only install the basic library of the Gym environment, and the balance bar game is already included in the basic library. If you need to use Atari or Mujoco emulators, additional installation steps are required. Let's take the installation of the Atari emulator as an example:

```
git clone https://github.com/openai/gym.git # Pull the code
cd gym # Go to directory
pip install -e '.[all]' # Install Gym
```

Generally speaking, creating a game and interacting in the Gym environment mainly consists of 5 steps:

1. Create a game. Through gym.make(name), you can create a game with the specified name and return the game object env.

2. Reset the game state. Generally, the game environment has an initial state. You can reset the game state by calling env.reset() and return to the initial state observation of the game.

3. Display the game screen. The game screen of each time stamp can be displayed by calling env.render(), which is generally used for testing. Rendering images during training will introduce a certain computational cost, so images may not be displayed during training.

4. Interact with the game environment. The action can be executed through env.step(action), and the system can return the new state observation, current reward, the game ending flag done and the additional information carrier. By looping this step, you can continue to interact with the environment until the end of the game.

5. Destroy the game. Just call env.close().

The following demonstrates a piece of interactive code for the balance bar game CartPole-v1. During each interaction, an action is randomly sampled in the action space: {left, right}, interact with the environment until the end of the game.

```
import gym # Import gym library
env = gym.make("CartPole-v1") # Create game environment
```

```
observation = env.reset() # Reset game state
for _ in range(1000): # Loop 1000 times
  env.render() # Render game image
  action = env.action_space.sample() # Randomly select an action
  # Interact with the environment, return new status, reward, end flag, other
information
  observation, reward, done, info = env.step(action)
  if done:# End of game round, reset state
    observation = env.reset()
env.close() # End game environment
```

14.1.3 Policy Network

Let's discuss the most critical link in reinforcement learning: how to judge and make decisions? We call judgment and decision-making Policy. The input of the Policy is the state s, and the output is a specific action a or the distribution of the action $\pi_\theta(a|s)$, where θ is the parameter of the strategy function π, and the π_θ function can be parameterized using neural networks, as shown in Figure 14.5. The input of the neural network π_θ is the state s of the balance bar system, that is, a vector of length 4, and the output is the probability of all actions $\pi_\theta(a|s)$: the probability to the left $P(\text{to left}|s)$ and the probability to the right $P(\text{to right}|s)$. The sum of all action probabilities is 1:

$$\sum_{a\in A}\pi_\theta(a|s)=1$$

where A is the set of all actions. The π_θ network represents the policy of the agent and is called the Policy network. Naturally, we can embody the policy function as a neural network with 4 input nodes, multiple fully connected hidden layers in the middle, and 2 output nodes in the output layer, which represents the probability distribution of these two actions. When interacting, choose the action with the highest probability

$$a_t=\underset{a}{\text{argmax}}\,\pi_\theta(a|s_t)$$

As a result of the decision, it acts in the environment and gets a new state s_{t+1} and reward r_t, and so on, until the end of the game.

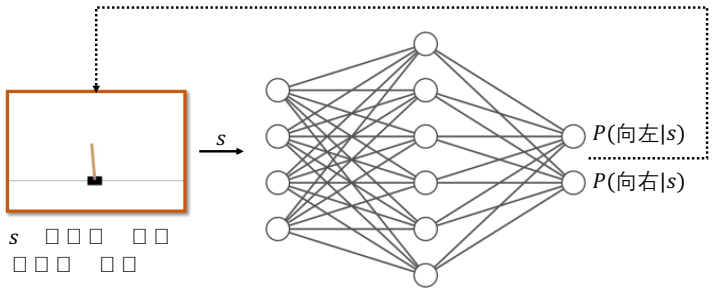


Figure 0.5 Strategy network

We implement the policy network as a 2-layer fully connected network. The first layer

converts a vector of length 4 to a vector of length 128, and the second layer converts a vector of 128 to a vector of 2, which is the probability distribution of actions. Just like the creation process of a normal neural network, the code is as follows:

```
class Policy(keras.Model):
    # Policy network, generating probability distribution of actions
    def __init__(self):
        super(Policy, self).__init__()
        self.data = [] # Store track
        # The input is a vector of length 4, and the output is two actions -
left and right, specifying the initialization scheme of the W tensor
        self.fc1 = layers.Dense(128, kernel_initializer='he_normal')
        self.fc2 = layers.Dense(2, kernel_initializer='he_normal')
        # Network optimizer
        self.optimizer = optimizers.Adam(lr=learning_rate)

    def call(self, inputs, training=None):
        # The shape of the state input s is a vector: [4]
        x = tf.nn.relu(self.fc1(inputs))
        x = tf.nn.softmax(self.fc2(x), axis=1) # Get the probability
distribution of the action
        return x
```

During the interaction, we record the state input s_t at each timestamp, the action distribution output a_t, the environment reward r_t and the new state s_{t+1} as a 4-tuple item for training the policy network.

```
    def put_data(self, item):
        # Record r,log_P(a|s)
        self.data.append(item)
```

14.1.4 Gradient update

If you need to use the gradient descent algorithm to optimize the network, you need to know the label information a_t of each input s_t and ensure that the loss value is continuously differentiable from the input to the loss. However, reinforcement learning is not the same as traditional supervised learning, which is mainly reflected in the fact that the action a_t of reinforcement learning at each timestamp t does not have a clear standard for good and bad. The reward r_t can reflect the quality of the action to a certain extent, but it cannot directly determine the quality of the action. Even some game interaction processes only have a final reward r_t signal representing the game result, such as Go. So is it feasible to define an optimal action a_t^* for each state as the label of the neural network input s_t? The first is that the total number of states in the game is usually huge. For example, the total number of states in Go is about 10^{170}. Furthermore, it is difficult to define an optimal action for each state. Although some actions have low short-term returns, long-term returns are better, and sometimes even humans do not know which action is the best.

Therefore, the optimization goal of the strategy should not be to make the output of the input s_t as close as possible to the labeling action, but to maximize the expected value of the total return. The total reward can be defined as the sum of incentives $\sum r_t$ from the beginning of the game to the end of the game. A good strategy should be able to obtain the highest expected value of total return $J(\pi_\theta)$ in the environment. According to the principle of the gradient ascent algorithm, if we can find $\frac{\partial J(\theta)}{\partial \theta}$, then the policy network only needs to follow

$$\theta' = \theta + \eta \cdot \frac{\partial J(\theta)}{\partial \theta}$$

to update the network parameters in order to maximize the expectation reward.

Unfortunately, the total return expectation $J(\pi_\theta)$ is given by the game environment. If the environment model is not known, then $\frac{\partial J(\theta)}{\partial \theta}$ cannot be calculated by automatic differentiation. So even if the expression of $J(\pi_\theta)$ is unknown, can the partial derivative $\frac{\partial J(\theta)}{\partial \theta}$ be solved directly?

The answer is yes. We directly give the derivation result of $\frac{\partial J(\theta)}{\partial \theta}$ here. The specific derivation process will be introduced in detail in 14.3:

$$\frac{\partial J(\theta)}{\partial \theta} = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\left(\sum_{t=1}^{T} \frac{\partial}{\partial \theta} \log \pi_\theta (a_t | s_t) \right) R(\tau) \right]$$

Using the above formula, you only need to calculate $\frac{\partial}{\partial \theta} \log \pi_\theta (a_t | s_t)$, and multiply it by $R(\tau)$ to update and calculate $\frac{\partial J(\theta)}{\partial \theta}$. According to $\theta' = \theta - \eta \cdot \frac{\partial \mathcal{L}(\theta)}{\partial \theta}$, the policy network can be updated to maximize the $J(\theta)$ function, where $R(\tau)$ is the total return of a certain interaction, τ is the interaction trajectory $s_1, a_1, r_1, s_2, a_2, r_2, \cdots, s_T$, T is the number of timestamps or steps of the interaction, $\log \pi_\theta (a_t | s_t)$ is the log function of the probability value of the a_t action in the output of the policy network. $\frac{\partial}{\partial \theta} \log \pi_\theta (a_t | s_t)$ can be solved by TensorFlow automatic differentiation. The code of the loss function is implemented as:

```
        for r, log_prob in self.data[::-1]:# Get trajectory data in reverse
order
            R = r + gamma * R # Accumulate the return on each time stamp
            # The gradient is calculated once for each timestamp
            # grad_R=-log_P*R*grad_theta
            loss = -log_prob * R
```

The whole training and updating code is as follow:

```
    def train_net(self, tape):
        # Calculate the gradient and update the policy network parameters.
tape is a gradient recorder
        R = 0 # The initial return of the end state is 0
```

```
        for r, log_prob in self.data[::-1]:# Reverse order
            R = r + gamma * R # Accumulate the return on each time stamp
            # The gradient is calculated once for each timestamp
            # grad_R=-log_P*R*grad_theta
            loss = -log_prob * R
            with tape.stop_recording():
                # Optimize strategy network
                grads = tape.gradient(loss, self.trainable_variables)
                # print(grads)
                self.optimizer.apply_gradients(zip(grads,
self.trainable_variables))
        self.data = [] # Clear track
```

14.1.5 Hands-on balance bar game

We train for a total of 400 rounds. At the beginning of the round, we reset the game state, sample actions by sending input states, interact with the environment, and record the information of each time stamp until the end of the game.

The interactive and training part of the code is as follows:

```
    for n_epi in range(10000):
        s = env.reset() # Back to the initial state of the game, return to s0
        with tf.GradientTape(persistent=True) as tape:
            for t in range(501): # CartPole-v1 forced to terminates at 500
step.
                # Send the state vector to get the strategy
                s = tf.constant(s,dtype=tf.float32)
                # s: [4] => [1,4]
                s = tf.expand_dims(s, axis=0)
                prob = pi(s) # Action distribution: [1,2]
                # Sample 1 action from the category distribution, shape: [1]
                a = tf.random.categorical(tf.math.log(prob), 1)[0]
                a = int(a) # Tensor to integer
                s_prime, r, done, info = env.step(a) # Interact with the
environment
                # Record action a and the reward r generated by the action
                # prob shape:[1,2]
                pi.put_data((r, tf.math.log(prob[0][a])))
                s = s_prime # Refresh status
                score += r # Cumulative reward

                if done:  # The current episode is terminated
                    break
            # After the episode is terminated, train the network once
```

```
        pi.train_net(tape)
    del tape
```

The training process of the model is shown in Figure 14.6. The horizontal axis is the number of training rounds, and the vertical axis is the average return value of the rounds. It can be seen that as the training progresses, the average return obtained by the network is getting higher and higher, and the strategy is getting better and better. In fact, reinforcement learning algorithms are extremely sensitive to parameters, and modifying the random seed will result in completely different performance. In the process of implementation, it is necessary to carefully select parameters to realize the potential of the algorithm.

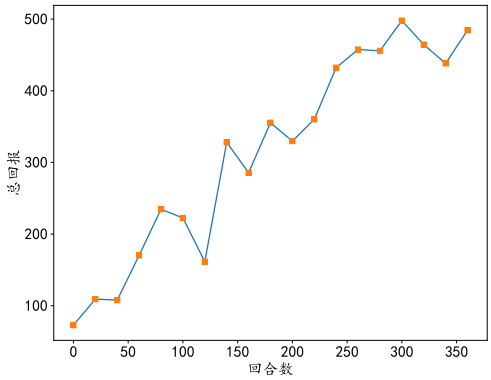


Figure 0.6 Balance bar game training process

Through this example, we have a preliminary impression and understanding of the interaction process between reinforcement learning algorithms and reinforcement learning, and then we will formally describe the reinforcement learning problem.

14.2 Reinforcement learning problems

In the reinforcement learning problem, the object with perception and decision-making capabilities is called an agent, which can be a piece of algorithm code, or a robotic software and hardware system with a mechanical structure. The agent completes a certain task by interacting with the external environment. The environment here refers to the sum of the external environment that can be affected by the action of the agent and gives corresponding feedback. For the agent, it generates decision-making actions (Action) by sensing the state of the environment (State). For the environment, it starts from an initial state s_1, and dynamically changes its state by accepting the actions of the agent , and give the corresponding reward signal (Reward).

We describe the reinforcement learning process from a probabilistic perspective. It contains the following five basic objects:

- State s reflects the state characteristics of the environment. The state on the time stamp t is marked as s_t. It can be the original visual image, voice waveform and other signals, or it can be the features after high-level abstraction, such as the speed and position of the car. All (finite) states constitute the state space S.
- Action a is the action taken by the agent. The state on the timestamp t is recorded as a_t,

which can be discrete actions such as leftward and rightward, or continuous actions such as strength and position. All (finite) actions constitute action space A.

- Policy $\pi(a|s)$ represents the decision model of the agent. It accepts the input as the state s and gives the probability distribution $p(a|s)$ of the action executed after the decision, which satisfies

$$\sum_{a \in A} \pi(a|s) = 1$$

This kind of action probability output with a certain randomness is called a Stochastic Policy. In particular, when the policy model always outputs a certain action with a probability of 1 and others at 0, this kind of policy model is called a Deterministic Policy, namely

$$a = \pi(s)$$

- Reward $r(s, a)$ expresses the feedback signal given by the environment after accepting action a in state s. It is generally a scalar value, which reflects the good or bad of the action to a certain extent. The reward obtained at the timestamp t is recorded as r_t (in some materials, it is recorded as r_{t+1}, because the reward often has a certain hysteresis)
- The state transition probability $p(s'|s, a)$ expresses the changing law of the state of the environment model, that is, after the environment of the current state s accepts the action a, the probability distribution that the state changes to s' satisfies

$$\sum_{s' \in S} p(s'|s, a) = 1$$

The interaction process between the agent and the environment can be represented by Figure 14.7.

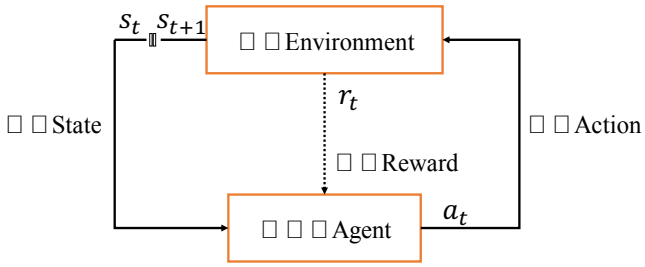


Figure 0.7 The interaction process between the agent and the environment

14.2.1 Markov decision process

The agent starts from the initial state s_1 of the environment, and executes a specific action a_1 through the policy model $\pi(a|s)$. The environment is affected by the action a_1, and the state s_1 changes to s_2 according to the internal state transition model $p(s'|s, a)$. In the meantime, it gives the feedback signal of the agent: the reward r_1, which is generated by the reward function $r(s_1, a_1)$. This cycle of interaction continues until the game reaches termination state s_T. This process produces a series of ordered data:

$$\tau = s_1, a_1, r_1, s_2, a_2, r_2, \cdots, s_T$$

This sequence represents an exchange process between the agent and the environment, called Trajectory, denoted as τ. An interaction process is called an Episode, and T represents the timestamp (or number of steps). Some environments have a clear Terminal State. For example, the game ends when a small plane in the space invaders is hit; while some environments do not have a clear termination mark. For example, some games can be played indefinitely as long as they remain healthy. At this time, T represents ∞.

The conditional probability $P(s_{t+1}|s_1, s_2, \ldots, s_t)$ is very important, but it requires multiple historical state, which is very complicated to calculate. For simplicity, we assume that the state s_{t+1} on the next time stamp is only affected by the current time stamp s_t, and has nothing to do with other historical states $s_1, s_2, \ldots, s_{t-1}$, that is :

$$P(s_{t+1}|s_1, s_2, \ldots, s_t) = P(s_{t+1}|s_t)$$

The property that next state s_{t+1} is only related to the current state s_t is called Markov Property, and the sequence $s_1, s_2, \ldots, s_T$ with Markov property is called Markov Process.

If the action a is also taken into consideration of the state transition probability, the Markov hypothesis is also applied: the state s_{t+1} of the next time stamp is only related to the current state s_t and the action a_t performed on the current state, then the condition probability becomes:

$$P(s_{t+1}|s_1, a_1, \ldots, s_t, a_t) = P(s_{t+1}|s_t, a_t)$$

We call the sequence of states and actions $s_1, a_1, \ldots, s_T$ the Markov Decision Process (MDP). In some scenarios, the agent can only observe part of the state of the environment, which is called Partially Observable Markov Decision Process (POMDP). Although the Markovian hypothesis does not necessarily correspond to the actual situation, it is the cornerstone of a large number of theoretical derivations in reinforcement learning. We will see the application of Markovianness in subsequent derivations.

Now let's consider a certain trajectory

$$\tau = s_1, a_1, r_1, s_2, a_2, r_2, \cdots, s_T$$

It's probability of occurrence $P(\tau)$:

$$P(\tau) = P(s_1, a_1, s_2, a_2, \cdots, s_T)$$

$$= P(s_1)\pi(a_1|s_1)P(s_2|s_1, a_1)\pi(a_2|s_2)P(s_3|s_1, a_1, s_2, a_2)\cdots$$

$$= P(s_1)\prod_{t=1}^{T-1}\pi(a_t|s_t)p(s_{t+1}|s_1, a_1, \ldots, s_t, a_t)$$

After applying Markovianity, we simplify the above expression to:

$$P(\tau) = P(s_1)\prod_{t=1}^{T-1}\pi(a_t|s_t)p(s_{t+1}|s_t, a_t)$$

The diagram of Markov decision process is shown in Figure 14.8.

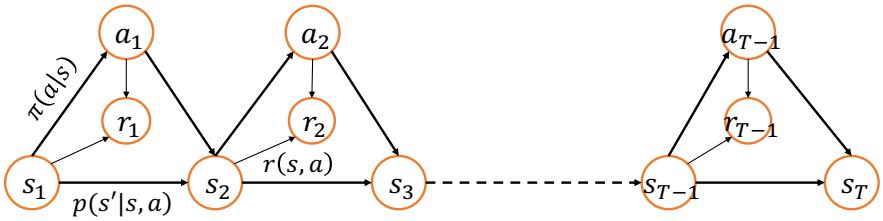


Figure 0.8 Markov decision process

If the state transition probability $p(s'|s,a)$ and the reward function $r(s,a)$ of the environment can be obtained, the value function can be directly calculated iteratively. This method of known environmental models is collectively called Model-based Reinforcement Learning. However, environmental models in the real world are mostly complex and unknown. Such methods with unknown models are collectively called Model-free Reinforcement Learning. Next, we will mainly introduce Model-free Reinforcement Learning algorithms.

14.2.2 Objective function

Each time the agent interacts with the environment, it will get a (lagging) reward signal:

$$r_t = r(s_t, a_t)$$

The cumulative reward of one interaction trajectory τ is called total return:

$$R(\tau) = \sum_{t=1}^{T-1} r_t$$

where T is the number of steps in the trajectory. If we only consider the cumulative return of $s_t, s_{t+1}, \ldots, s_T$ starting from the intermediate state s_t of the trajectory, it can be recorded as:

$$R(s_t) = \sum_{k=1}^{T-t-1} r_{t+k}$$

In some environments, the stimulus signal is very sparse, such as Go, the stimulus of the previous move is 0, and only at the end of the game will there be a reward signal representing the win or loss.

Therefore, in order to weigh the importance of short-term and long-term rewards, discounted returns that decay over time (Discounted Return) can be used:

$$R(\tau) = \sum_{t=1}^{T-1} \gamma^{t-1} r_t$$

where $\gamma \in [0,1]$ is called the discount rate. It can be seen that the recent incentive r_1 is all used for total return, while the long-term incentive r_{T-1} can be used to contribute to the total return $R(\tau)$ after attenuating γ^{T-2}. When $\gamma \approx 1$, the short-term and long-term reward weights are approximately the same, and the algorithm is more forward-looking; when $\gamma \approx 0$, the later long-term reward decays close to 0, short-term reward becomes more important. For an environment

with no termination state, that is, $T = \infty$, the discounted return becomes very important, because $\sum_{t=1}^{\infty} \gamma^{t-1} r_t$ may increase to infinity, and the discounted return can be approximately ignored for long-term rewards to facilitate algorithm implementation.

We hope to find a policy $\pi(a|s)$ model so that the higher the total return $R(\tau)$ of the trajectory τ generated by the interaction between the agent and the environment under the control of the policy $\pi(a|s)$, the better. Due to the randomness of environment state transition and policy, the same policy model acting on the same environment with the same initial state may also produce completely different trajectory sequence τ. Therefore, the goal of reinforcement learning is to maximize the Expected Return:

$$J(\pi_\theta) = \mathbb{E}_{\tau \sim p(\tau)}[R(\tau)] = \mathbb{E}_{\tau \sim p(\tau)}\left[\sum_{t=1}^{T-1} \gamma^{t-1} r_t\right]$$

The goal of training is to find a policy network π_θ represented by a set of parameters θ, so that $J(\pi_\theta)$is the largest:

$$\theta^* = \arg\max_\theta \mathbb{E}_{\tau \sim p(\tau)}[R(\tau)]$$

where $p(\tau)$ represents the distribution of trajectory τ, which is jointly determined by the state transition probability $p(s'|s,a)$ and the strategy $\pi(a|s)$. The quality of strategy π can be measured by $J(\pi_\theta)$. The greater the expected return, the better the policy; otherwise, the worse the strategy.

14.3 Policy gradient method

Since the goal of reinforcement learning is to find an optimal policy $\pi_\theta(a|s)$ that maximizes the expected return $J(\theta)$, this type of optimization problem is similar to supervised learning. It is necessary to solve the partial derivative of the expected return with the network parameters $\frac{\partial J}{\partial \theta}$, and use gradient ascent algorithm to update network parameters:

$$\theta' = \theta + \eta \cdot \frac{\partial J}{\partial \theta}$$

That is, where η is the learning rate.

The policy model $\pi_\theta(a|s)$ can use a multilayer neural network to parameterize $\pi_\theta(a|s)$. The input of the network is the state s, and the output is the probability distribution of the action a. This kind of network is called a policy network.

To optimize this network, you only need to obtain the partial derivative of each parameter $\frac{\partial J}{\partial \theta}$. Now we come to derive the expression of $\frac{\partial J}{\partial \theta}$. First, expand it by trajectory distribution:

$$\frac{\partial J}{\partial \theta} = \frac{\partial}{\partial \theta} \int \pi_\theta(\tau) R(\tau) d\tau$$

Move the derivative symbol to the integral symbol:

$$= \int \left(\frac{\partial}{\partial \theta} \pi_\theta(\tau) \right) R(\tau) d\tau$$

Adding $\pi_\theta(\tau) \cdot \frac{1}{\pi_\theta(\tau)}$ does not change the result：

$$= \int \pi_\theta(\tau) \left(\frac{1}{\pi_\theta(\tau)} \frac{\partial}{\partial \theta} \pi_\theta(\tau) \right) R(\tau) d\,\tau$$

Considering:

$$\frac{\mathrm{d}\log\big(f(x)\big)}{\mathrm{d}x} = \frac{1}{f(x)} \frac{\mathrm{d}f(x)}{\mathrm{d}x}$$

So:

$$\frac{1}{\pi_\theta(\tau)} \frac{\partial}{\partial \theta} \pi_\theta(\tau) = \frac{\partial}{\partial \theta} \log \pi_\theta\,(\tau)$$

We can get:

$$= \int \pi_\theta(\tau) \left(\frac{\partial}{\partial \theta} \log \pi_\theta\,(\tau) \right) R(\tau) d\tau$$

i.e.：

$$\frac{\partial J}{\partial \theta} = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\frac{\partial}{\partial \theta} \log \pi_\theta\,(\tau) R(\tau) \right]$$

where $\log \pi_\theta\,(\tau)$ represents the log probability value of trajectory $\tau = s_1, a_1, s_2, a_2, \cdots, s_T$.

Considering that $R(\tau)$ can be obtained by sampling, the key becomes to solve $\frac{\partial}{\partial \theta} \log \pi_\theta\,(\tau)$, we can decompose $\pi_\theta(\tau)$ to get:

$$\frac{\partial}{\partial \theta} \log \pi_\theta\,(\tau) = \frac{\partial}{\partial \theta} \log \left(p(s_1) \prod_{t=1}^{T-1} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t) \right)$$

Convert $\log \prod \cdot$ to $\sum \log(\cdot)$：

$$= \frac{\partial}{\partial \theta} \left(\log p\,(s_1) + \sum_{t=1}^{T-1} \log \pi_\theta\,(a_t|s_t) + \log p\,(s_{t+1}|s_t, a_t) \right)$$

Considering that both $\log p\,(s_{t+1}|s_t, a_t)$ and $\log p\,(s_1)$ are not related to θ，the above formula becomes:

$$\frac{\partial}{\partial \theta} \log \pi_\theta\,(\tau) = \sum_{t=1}^{T-1} \frac{\partial}{\partial \theta} \log \pi_\theta\,(a_t|s_t)$$

It can be seen that the partial derivative $\frac{\partial}{\partial \theta} \log \pi_\theta\,(\tau)$ can finally be converted to $\log \pi_\theta\,(a_t|s_t)$ which is the derivative of the policy network output to the network parameter θ. It has nothing to

do with the state probability transition $p(s'|s,a)$, that is, it can be solved without knowing the environment model $\frac{\partial}{\partial\theta}\log p_\theta(\tau)$.

Put it into $\frac{\partial J}{\partial\theta}$:

$$\frac{\partial J(\theta)}{\partial\theta} = \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\frac{\partial}{\partial\theta}\log\pi_\theta(\tau)R(\tau)\right]$$

$$= \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\left(\sum_{t=1}^{T-1}\frac{\partial}{\partial\theta}\log\pi_\theta(a_t|s_t)\right)R(\tau)\right]$$

Let us intuitively understand the above formula. When the total return of a certain round $R(\tau)>0$, $\frac{\partial J(\theta)}{\partial\theta}$ and $\frac{\partial}{\partial\theta}\log\pi_\theta(\tau)$are in the same direction. According to the gradient ascent algorithm, the θ parameter is updated toward the direction of increasing $J(\theta)$, and also in the direction of increasing $\log\pi_\theta(a_t|s_t)$, which encourages the generation of more such trajectories τ. When the total return $R(\tau)<0$, $\frac{\partial J(\theta)}{\partial\theta}$ and $\frac{\partial}{\partial\theta}\log\pi_\theta(\tau)$ are reversed, so when the θ parameter is updated according to the gradient ascent algorithm. It is updated toward the direction of increasing $J(\theta)$ and decreasing $\log\pi_\theta(a_t|s_t)$, that is, to avoid generating more such trajectories τ. Through this, it is possible to intuitively understand how the network adjusts itself to achieve greater expected return.

With the above expression of $\frac{\partial J}{\partial\theta}$, we can easily solve $\frac{\partial}{\partial\theta}\log\pi_\theta(a_t|s_t)$ through the automatic differentiation tool of TensorFlow to calculate $\frac{\partial J}{\partial\theta}$. Finally, we can use the gradient ascent algorithm to update the parameters. The general flow of the policy gradient algorithm is shown in Figure 14.9.

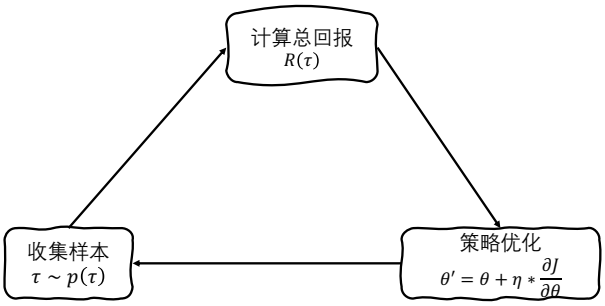


Figure 0.9 Policy gradient method training process

14.3.1 Reinforce algorithm

According to the law of large numbers, write the expectation as the mean value of multiple sampling trajectories $\tau^n, n\in[1,N]$:

$$\frac{\partial J(\theta)}{\partial \theta} \approx \frac{1}{N}\sum_{n=1}^{N}\left(\left(\sum_{t=1}^{T-1}\frac{\partial}{\partial \theta}\log \pi_\theta\left(a_t^{(n)}\middle|s_t^{(n)}\right)\right)R\left(\tau^{(n)}\right)\right)$$

where N is the number of trajectories, and $a_t^{(n)}$ and $s_t^{(n)}$ represent the actions and input states of the t-th time stamp of the n-th trajectory τ^n. Then update the θ parameters through gradient ascent algorithm. This algorithm is called the REINFORCE algorithm [4], which is also the earliest algorithm that uses the policy gradient idea.

Algorithm 1：REINFORCE Algorithm

Randomly initialize $\boldsymbol{\theta}$

repeat

Interact with environment according to policy $\pi_\theta(a_t|s_t)$ and generate multiple trajectories $\{\boldsymbol{\tau^{(n)}}\}$

Calculate $\boldsymbol{R\left(\tau^{(n)}\right)}$

Calculate $\boldsymbol{\frac{\partial J(\theta)}{\partial \theta} \approx \frac{1}{N}\sum_{n=1}^{N}\left(\left(\sum_{t=1}^{T-1}\frac{\partial}{\partial \theta}\log \pi_\theta\left(a_t^{(n)}\middle|s_t^{(n)}\right)\right)R\left(\tau^{(n)}\right)\right)}$

Update parameter $\boldsymbol{\theta' \leftarrow \theta + \eta \cdot \frac{\partial J}{\partial \theta}}$

until reach certain training times

Output: policy network $\boldsymbol{\pi_\theta(a_t|s_t)}$

14.3.2 Improvement of the original policy gradient method

Because the original REINFORCE algorithm has a large variance between the optimized trajectories, the convergence speed is slow, and the training process is not smooth enough. We can use the idea of Variance Reduction to make improvements from the perspectives of causality and baseline.

Causality Considering the partial derivative expression of $\frac{\partial J(\theta)}{\partial \theta}$, for the action a_t with a time stamp of t, it has no effect on $\tau_{1:t-1}$, but only has an effect on the subsequent trajectory $\tau_{t:T}$. So for $\pi_\theta(a_t|s_t)$, we only consider the cumulative return $R(\tau_{t:T})$ starting from the timestamp t. The expression of $\frac{\partial J(\theta)}{\partial \theta}$ is given by

$$\frac{\partial J(\theta)}{\partial \theta} = \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\left(\sum_{t=1}^{T-1}\frac{\partial}{\partial \theta}\log \pi_\theta\left(a_t|s_t\right)\right)R(\tau_{1:T})\right]$$

It can be written as：

$$\frac{\partial J(\theta)}{\partial \theta} = \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\sum_{t=1}^{T-1}\left(\frac{\partial}{\partial \theta}\log \pi_\theta\left(a_t|s_t\right)R(\tau_{t:T})\right)\right]$$

$$= \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{t=1}^{T-1} \left(\frac{\partial}{\partial \theta} \log \pi_\theta (a_t|s_t) \hat{Q}(s_t, a_t) \right) \right]$$

where $\hat{Q}(s_t, a_t)$ function represents the estimated reward value of π_θ after the a_t action is executed from the state s_t. The definition of the Q function will also be introduced in Section 14.4. Since only the trajectory $\tau_{t:T}$ starting from a_t is considered, the variance of $R(\tau_{t:T})$ becomes smaller.

Bias The reward r_t in the real environment is not distributed around 0. The rewards of many games are all positive, so that $R(\tau)$ is always greater than 0. The network tends to increase the probability of all sampled actions. The probability of unsampled action is relatively reduced. This is not what we want. We hope that $R(\tau)$ can be distributed around 0, so we introduce a bias variable b, called the baseline, which represents the average level of return $R(\tau)$. The expression of $\frac{\partial J(\theta)}{\partial \theta}$ is converted to:

$$\frac{\partial J(\theta)}{\partial \theta} = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{t=1}^{T-1} \frac{\partial}{\partial \theta} \log \pi_\theta (a_t|s_t) (R(\tau) - b) \right]$$

Considering causality，$\frac{\partial J(\theta)}{\partial \theta}$ can be written as：

$$\frac{\partial J(\theta)}{\partial \theta} = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{t=1}^{T-1} \left(\frac{\partial}{\partial \theta} \log \pi_\theta (a_t|s_t) \big(\hat{Q}(s_t, a_t) - b \big) \right) \right]$$

where $\delta = R(\tau) - b$ is called the advantage function, which represents the advantage of the current action sequence relative to the average return.

After adding bias b，will the value of $\frac{\partial J(\theta)}{\partial \theta}$ change？To answer the question, we only need to consider whether $\mathbb{E}_{\tau \sim p_\theta(\tau)}[\nabla_\theta \log \pi_\theta (\tau) \cdot b]$ can be 0. If it's 0，then the value of $\frac{\partial J(\theta)}{\partial \theta}$ will not change. Expand $\mathbb{E}_{\tau \sim p_\theta(\tau)}[\nabla_\theta \log \pi_\theta (\tau) \cdot b]$ to ：

$$\mathbb{E}_{\tau \sim p_\theta(\tau)}[\nabla_\theta \log \pi_\theta (\tau) \cdot b] = \int \pi_\theta(\tau) \nabla_\theta \log \pi_\theta (\tau) \cdot b \, \mathrm{d}\tau$$

Because

$$\pi_\theta(\tau) \nabla_\theta \, log \, \pi_\theta (\tau) = \nabla_\theta \pi_\theta(\tau)$$

We have

$$\mathbb{E}_{\tau \sim p_\theta(\tau)}[\nabla_\theta \log \pi_\theta (\tau) \cdot b] = \int \nabla_\theta \pi_\theta(\tau) b \, \mathrm{d}\tau$$

$$= b \nabla_\theta \int \pi_\theta(\tau) \mathrm{d}\tau$$

Consider $\int \pi_\theta(\tau) \mathrm{d}\tau = 1$,

$$\mathbb{E}_{\tau \sim p_\theta(\tau)}[\nabla_\theta \log \pi_\theta (\tau) \cdot b] = b \nabla_\theta 1 = 0$$

Therefore, adding bias b doesn't change the value of $\frac{\partial J(\theta)}{\partial \theta}$, but it indeed reduces the variance of $\sum_{t=1}^{T-1}\left(\frac{\partial}{\partial \theta}\log \pi_\theta\left(a_t|s_t\right)\left(\hat{Q}(s_t,a_t)-b\right)\right)$.

14.3.3 REINFORCE algorithm with bias

Bias b can be estimated using Monte Carlo method:

$$b=\frac{1}{N}\sum_{n=1}^{N}R\left(\tau^{(n)}\right)$$

If causality is considered，then：

$$b=\frac{1}{N}\sum_{n=1}^{N}R\left(\tau_{t:T}^{(n)}\right)$$

Bias b can also be estimated using another neural network, which is also the Actor-Critic method introduced in Section 14.5. In fact, many policy gradient algorithms often use neural networks to estimate bias b. The algorithm can be flexibly adjusted, and it is most important to master the algorithm idea. The REINFORCE algorithm flow with bias is shown in Algorithm 2.

Algorithm 2: REINFORCE algorithm flow with bias

Randomly initialize $\boldsymbol{\theta}$

repeat

  Interact with environment according to policy $\boldsymbol{\pi_\theta(a_t|s_t)}$, generate multiple trajectory $\{\boldsymbol{\tau^n}\}$

  Calculate $\widehat{\boldsymbol{Q}}(\boldsymbol{s_t},\boldsymbol{a_t})$

  Estimate bias $\boldsymbol{b}$ through Monte Carlo method

  Calculate $\frac{\partial J(\theta)}{\partial \theta}\approx\frac{1}{N}\sum_{n=1}^{N}\left(\left(\sum_{t=1}^{T-1}\frac{\partial}{\partial \theta}\log \pi_\theta\left(a_t^{(n)}\middle|s_t^{(n)}\right)\right)\left(\widehat{Q}(s_t,a_t)-b\right)\right)$

  Update parameter $\boldsymbol{\theta'\leftarrow\theta+\eta\cdot\frac{\partial J}{\partial \theta}}$

until reach training times

Output: policy network $\boldsymbol{\pi_\theta(a_t|s_t)}$

14.3.4 Importance sampling

After updating the network parameters using the policy gradient method, the policy network $\pi_\theta(a|s)$ has also changed, and the new policy network must be used for sampling. As a result, the previous historical trajectory data cannot be reused, and the sampling efficiency is very low. How to improve the sampling efficiency and reuse the trajectory data generated by the old policy?

In statistics, importance sampling techniques can estimate the expectation of the original distribution p from another distribution q. Considering that the trajectory τ is sampled from the

original distribution p, we hope to estimate the expectation $\mathbb{E}_{\tau\sim p}[f(\tau)]$ of the trajectory $\tau\sim p$ function.

$$\mathbb{E}_{\tau\sim p}[f(\tau)] = \int p(\tau)f(\tau)\mathrm{d}\tau$$

$$= \int \frac{p(\tau)}{q(\tau)}q(\tau)f(\tau)\mathrm{d}\tau$$

$$= \mathbb{E}_{\tau\sim q}\left[\frac{p(\tau)}{q(\tau)}f(\tau)\right]$$

Through derivation, we find that the expectation of $f(\tau)$ can be sampled not from the original distribution p, but from another distribution q, which only needs to be multiplied by the ratio $\frac{p(\tau)}{q(\tau)}$. This is called Importance Sampling in statistics.

Let the target policy distribution be $p_\theta(\tau)$, and a certain historical policy distribution is $p_{\bar{\theta}}(\tau)$, we hope to use the historical sampling trajectory $\tau \sim p_{\bar{\theta}}(\tau)$ to estimate the expected return of the target policy network:

$$J(\theta) = \mathbb{E}_{\tau\sim p_\theta(\tau)}[R(\tau)]$$

$$= \sum_{t=1}^{T-1}\mathbb{E}_{(s_t,a_t)\sim p_\theta(s_t,a_t)}[r(s_t,a_t)]$$

$$= \sum_{t=1}^{T-1}\mathbb{E}_{s_t\sim p_\theta(s_t)}\mathbb{E}_{a_t\sim\pi_\theta(a_t|s_t)}[r(s_t,a_t)]$$

Applying importance sampling technique, we can get:

$$J_{\bar{\theta}}(\theta) = \sum_{t=1}^{T-1}\mathbb{E}_{s_t\sim p_{\bar{\theta}}(s_t)}\left[\frac{p_\theta(s_t)}{p_{\bar{\theta}}(s_t)}\mathbb{E}_{a_t\sim\pi_{\bar{\theta}}(a_t|s_t)}\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\bar{\theta}}(a_t|s_t)}r(s_t,a_t)\right]\right]$$

where $J_{\bar{\theta}}(\theta)$ represents the value of $J(\theta)$ for the original distribution $p_\theta(\tau)$ estimated through the distribution $p_{\bar{\theta}}(\tau)$. Under the assumption of approximately ignoring the terms $\frac{p_\theta(s_t)}{p_{\bar{\theta}}(s_t)}$, it is considered that the probability of state s_t appearing under different policies is approximately equal, i.e. $\frac{p_\theta(s_t)}{p_{\bar{\theta}}(s_t)} \approx 1$, so

$$J_{\bar{\theta}}(\theta) = \sum_{t=1}^{T-1}\mathbb{E}_{s_t\sim p_{\bar{\theta}}(s_t)}\left[\mathbb{E}_{a_t\sim\pi_{\bar{\theta}}(a_t|s_t)}\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\bar{\theta}}(a_t|s_t)}r(s_t,a_t)\right]\right]$$

$$= \sum_{t=1}^{T-1}\mathbb{E}_{(s_t,a_t)\sim p_{\bar{\theta}}(s_t,a_t)}\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\bar{\theta}}(a_t|s_t)}r(s_t,a_t)\right]$$

The method in which the sampling policy $p_{\bar{\theta}}(\tau)$and the target policy $p_\theta(\tau)$ to be optimized are not the same is called the Off-Policy method. Conversely, the method in which the sampling policy and the target policy to be optimized are the same policy is called On-Policy method.

REINFORCE algorithm belongs to the On-Policy method category. The Off-Policy method can use historical sampling data to optimize the current policy network, which greatly improves data utilization, but also introduces computational complexity. In particular, when importance sampling is implemented by Monte Carlo sampling method, if the difference between the distributions p and q is too large, the expectation estimation will have a large deviation. Therefore, the implementation needs to ensure that the distributions p and q are as similar as possible, such as adding KL divergence constrain to limit the difference between p and q.

We also call the training objective function of the original policy gradient method $\mathcal{L}^{PG}(\theta)$:

$$\mathcal{L}^{PG}(\theta) = \hat{\mathbb{E}}_t\left[\log \pi_\theta\left(a_t|s_t\right)\hat{A}_t\right]$$

where PG stands for Policy Gradient, and $\hat{\mathbb{E}}_t$ and $\hat{A}_t$ represent empirical estimates. The objective function based on importance sampling is called $\mathcal{L}^{IS}_{\bar{\theta}}(\theta)$:

$$\mathcal{L}^{IS}_{\bar{\theta}}(\theta) = \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\bar{\theta}}(a_t|s_t)}\hat{A}_t\right]$$

where IS stands for Importance Sampling, θ stands for the target policy distribution p_θ, and $\bar{\theta}$stands for the sampling policy distribution $p_{\bar{\theta}}$.

14.3.5 PPO algorithm

After applying importance sampling, the policy gradient algorithm greatly improves the data utilization rate, which greatly improves the performance and training stability. The more popular Off-Policy gradient algorithms include TRPO algorithm and PPO algorithm, among which TRPO is the predecessor of PPO algorithm, and PPO algorithm can be regarded as an approximate simplified version of TRPO algorithm.

TRPO algorithm In order to constrain the distance between the target policy $\pi_\theta(\cdot|s_t)$ and the sampling policy $\pi_{\bar{\theta}}(\cdot|s_t)$, the TRPO algorithm uses KL divergence to calculate the distance expectation between $\pi_\theta(\cdot|s_t)$ and $\pi_{\bar{\theta}}(\cdot|s_t)$. The distance expectation is used as the constraint term of the optimization problem. The implementation of TRPO algorithm is more complicated and computationally expensive. The optimization objective of the TRPO algorithm is:

$$\theta^* = \arg\max_\theta \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\bar{\theta}}(a_t|s_t)}\hat{A}_t\right]$$

$$s.t.\ \hat{\mathbb{E}}_t\left[D_{KL}\left(\pi_\theta(\cdot|s_t)||\pi_{\bar{\theta}}(\cdot|s_t)\right)\right] \le \delta$$

PPO algorithm In order to solve the disadvantage of high TRPO calculation cost, the PPO algorithm adds the KL divergence constraint as a penalty item to the loss function. The optimization goal is:

$$\theta^* = \arg\max_\theta \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\bar{\theta}}(a_t|s_t)}\hat{A}_t\right] - \beta\hat{\mathbb{E}}_t\left[D_{KL}\left(\pi_\theta(\cdot|s_t)||\pi_{\bar{\theta}}(\cdot|s_t)\right)\right]$$

where $D_{KL}\left(\pi_\theta(\cdot|s_t)||\pi_{\bar{\theta}}(\cdot|s_t)\right)$ refers to the distance between the policy distribution $\pi_\theta(\cdot|s_t)$ and $\pi_{\bar{\theta}}(\cdot|s_t)$, and the hyperparameter β is used to balance the original loss term and the KL divergence penalty term.

Adaptive KL Penalty algorithm The hyperparameter β is dynamically adjusted by setting the threshold $\mathrm{KL}_{\max}$ of KL divergence. The adjustment rules are as follows: if

$\widehat{\mathbb{E}}_t\left[D_{KL}(\pi_\theta(\cdot|s_t)||\pi_{\bar{\theta}}(\cdot|s_t))\right] > \mathrm{KL}_{\max}$, increase β; if $\widehat{\mathbb{E}}_t\left[D_{KL}(\pi_\theta(\cdot|s_t)||\pi_{\bar{\theta}}(\cdot|s_t))\right] < \mathrm{KL}_{\max}$, then decrease β.

PPO2 algorithm Based on the PPO algorithm, the PPO2 algorithm adjusts the loss function:

$$\mathcal{L}_{\bar{\theta}}^{CLIP}(\theta) = \widehat{\mathbb{E}}_t\left[\min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\bar{\theta}}(a_t|s_t)}\hat{A}_t, \text{clip}\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\bar{\theta}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon\right)\hat{A}_t\right)\right]$$

The schematic diagram of the error function is shown in Figure 14.10.

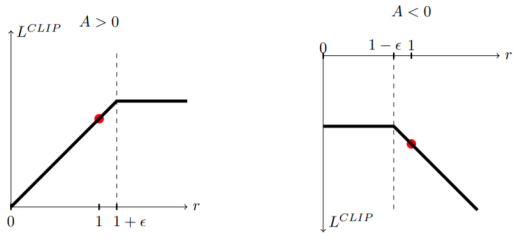


Figure 0.10 Schematic diagram of PPO2 algorithm error function

14.3.6 Hands-on PPO

In this section, we implement the PPO algorithm based on importance sampling technology, and test the performance of the PPO algorithm in the balance bar game environment.

Policy network The policy network is also called the Actor network. The input of the policy network is the state s_t, 4 input nodes, and the output is the probability distribution $\pi_\theta(a_t|s_t)$ of the action a_t, which is implemented by a 2-layer fully connected network.

```
class Actor(keras.Model):
    def __init__(self):
        super(Actor, self).__init__()
        # The policy network is also called the Actor network. Output
probability p(a|s)
        self.fc1 = layers.Dense(100, kernel_initializer='he_normal')
        self.fc2 = layers.Dense(2, kernel_initializer='he_normal')

    def call(self, inputs):
        # Forward propagation
        x = tf.nn.relu(self.fc1(inputs))
        x = self.fc2(x)
        # Output action probability
        x = tf.nn.softmax(x, axis=1) # Convert to probability
        return x
```

Bias b network Bias b network is also called Critic network, or V-value function network. The input of the network is the state s_t, 4 input nodes, and the output is the scalar value b. A 2-layer fully connected network is used to estimate b. The code is implemented as follows:

```
class Critic(keras.Model):
    def __init__(self):
```

```
        super(Critic, self).__init__()
        # Bias b network is also called Critic network, output is v(s)
        self.fc1 = layers.Dense(100, kernel_initializer='he_normal')
        self.fc2 = layers.Dense(1, kernel_initializer='he_normal')

    def call(self, inputs):
        x = tf.nn.relu(self.fc1(inputs))
        x = self.fc2(x)  # Output b's estimate
        return x
```

接下来完成策略网络、值函数网络的创建工作，同时分别创建两个优化器，用于优化策略网络和值函数网络的参数，我们创建在 PPO 算法主体类的初始化方法中。代码如下：

Next, complete the creation of the strategy network and the value function network, and create two optimizers respectively to optimize the parameters of the strategy network and the value function network. We create it in the initialization method of the main class of the PPO algorithm. code show as below:

```
class PPO():
    # PPO algorithm
    def __init__(self):
        super(PPO, self).__init__()
        self.actor = Actor() # Create Actor network
        self.critic = Critic() # Create Critic network
        self.buffer = [] # Data buffer
        self.actor_optimizer = optimizers.Adam(1e-3) # Actor optimizer
        self.critic_optimizer = optimizers.Adam(3e-3) # Critic optimizer
```

Action sampling The select_action function can calculate the action distribution $\pi_\theta(a_t|s_t)$ of the current state, and randomly sample actions according to the probability, and return the action and its probability.

```
    def select_action(self, s):
        # Send the state vector to get the strategy: [4]
        s = tf.constant(s, dtype=tf.float32)
        # s: [4] => [1,4]
        s = tf.expand_dims(s, axis=0)
        # Get strategy distribution: [1, 2]
        prob = self.actor(s)
        # Sample 1 action from the category distribution, shape: [1]
        a = tf.random.categorical(tf.math.log(prob), 1)[0]
        a = int(a)  # Tensor to integer
        return a, float(prob[0][a]) # Return action and its probability
```

Environment interaction In the main function, interact with the environment for 500 rounds. In each round, the policy is sampled by the select_action function and saved in the buffer pool. The agent.optimizer() function is called to optimize the policy at intervals.

```
def main():
```

```
    agent = PPO()
    returns = [] # total return
    total = 0 #  Average return over time
    for i_epoch in range(500): # Number of training rounds
        state = env.reset() # Reset environment
        for t in range(500): # at most 500 rounds
            # Interact with environment with new policy
            action, action_prob = agent.select_action(state)
            next_state, reward, done, _ = env.step(action)
            # Create and store samples
            trans = Transition(state, action, action_prob, reward, next_stat
e)
            agent.store_transition(trans)
            state = next_state # Update state
            total += reward # Accumulate rewards
            if done: # Train network
                if len(agent.buffer) >= batch_size:
                    agent.optimize() # Optimize
                break
```

Network optimization When the buffer pool reaches a certain capacity, the error of the policy network and the error of the value network are constructed through optimizer() function to optimize the parameters of the network. First, the data is converted to the Tensor type according to the category, and then the cumulative return $R(\tau_{t:T})$ is calculated by the MC method.

```
    def optimize(self):
        # Optimize the main network function
        # Take sample data from the cache and convert it into tensor
        state = tf.constant([t.state for t in self.buffer], dtype=tf.float32
)
        action = tf.constant([t.action for t in self.buffer], dtype=tf.int32
)
        action = tf.reshape(action,[-1,1])
        reward = [t.reward for t in self.buffer]
        old_action_log_prob = tf.constant([t.a_log_prob for t in self.buffer
], dtype=tf.float32)
        old_action_log_prob = tf.reshape(old_action_log_prob, [-1,1])
        # Calculate R(st) using MC method
        R = 0
        Rs = []
        for r in reward[::-1]:
            R = r + gamma * R
            Rs.insert(0, R)
        Rs = tf.constant(Rs, dtype=tf.float32)
        …
```

Then the data in the buffer pool is taken out according to the Batch Size. Train the network iteratively 10 times. For the policy network, $\mathcal{L}_{\bar{\theta}}^{CLIP}(\theta)$ is calculated according to the error function of the PPO2 algorithm. For the value network, the distance between the prediction of the value network and $R(\tau_{t:T})$ is calculated through the mean square error, so that the value of the network estimation is getting more and more accurate.

```
    def optimize(self):
        …
        # Iterate roughly 10 times on the buffer pool data
        for _ in range(round(10*len(self.buffer)/batch_size)):
            # Randomly sample batch size samples from the buffer pool
            index = np.random.choice(np.arange(len(self.buffer)), batch_size
, replace=False)
            # Build a gradient tracking environment
            with tf.GradientTape() as tape1, tf.GradientTape() as tape2:
                # Get R(st), [b,1]
                v_target = tf.expand_dims(tf.gather(Rs, index, axis=0), axis
=1)
                # Calculate the predicted value of v(s), which is the bias
b, we will introduce why it is written as v later
                v = self.critic(tf.gather(state, index, axis=0))
                delta = v_target - v # Calculating advantage value
                advantage = tf.stop_gradient(delta) # Disconnect the
gradient
                #  Because TF's gather_nd and pytorch's gather function are
different, it needs to be constructed
              # Coordinate parameters required by gather_nd need to be
constructed, indices:[b, 2]
                # pi_a = pi.gather(1, a) # pytorch only need oneline
implementation
                a = tf.gather(action, index, axis=0) # Take out the action
                # batch's action distribution pi(a|st)
                pi = self.actor(tf.gather(state, index, axis=0))
                indices = tf.expand_dims(tf.range(a.shape[0]), axis=1)
                indices = tf.concat([indices, a], axis=1)
                pi_a = tf.gather_nd(pi, indices)  # The probability of
action, pi(at|st), [b]
                pi_a = tf.expand_dims(pi_a, axis=1)  # [b]=> [b,1]
                # Importance sampling
                ratio = (pi_a / tf.gather(old_action_log_prob, index, axis=0
))
                surr1 = ratio * advantage
                surr2 = tf.clip_by_value(ratio, 1 - epsilon, 1 + epsilon) *
advantage
                # PPO error function
```

```
                policy_loss = -tf.reduce_mean(tf.minimum(surr1, surr2))
                # For the bias v, it is hoped that the R(st) estimated by MC
is as close as possible
                value_loss = losses.MSE(v_target, v)
            # Optimize policy network
            grads = tape1.gradient(policy_loss, self.actor.trainable_variabl
es)
            self.actor_optimizer.apply_gradients(zip(grads, self.actor.train
able_variables))
            # Optimize bias network
            grads = tape2.gradient(value_loss, self.critic.trainable_variabl
es)
            self.critic_optimizer.apply_gradients(zip(grads, self.critic.tra
inable_variables))

        self.buffer = []  # Empty trained data
```

Training results After 500 rounds of training, we draw the total return curve, as shown in Figure 14.11, we can see that for a simple game such as a balance bar, the PPO algorithm appears to be easy to use.

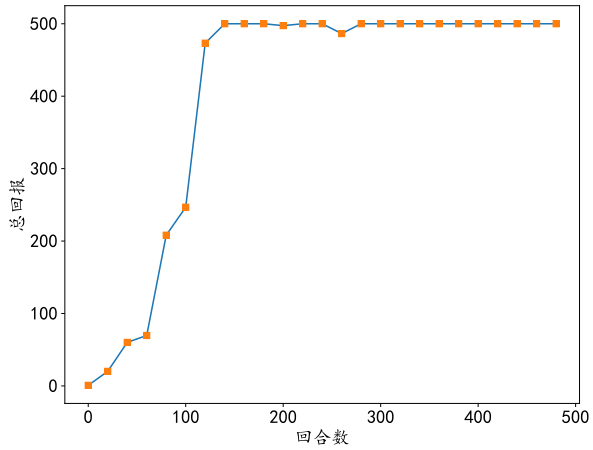


Figure 0.11 Return curve of PPO algorithm

14.4 Value function method

A better policy model can be obtained using the policy gradient method by directly optimizing the policy network parameters. In the field of reinforcement learning, in addition to the policy gradient method, there is another type of method that indirectly obtains the policy by modeling the value function, which we collectively call the value function method.

Next, we will introduce the definition of common value functions, how to estimate value functions, and how value functions help generate policies.

14.4.1 Value function

In reinforcement learning, there are two types of value functions: state value function and state-action value function, both of which represent the definition of the starting point of the expected return trajectory is different under the strategy π.

State Value Function (V function for short), which is defined as the expected return value that can be obtained from the state s_t under the control of the strategy π:

$$V^{\pi}(s_t) = \mathbb{E}_{\tau \sim p(\tau)}\left[R(\tau_{t:T}) | \tau_{s_t} = s_t\right]$$

Expand $R(\tau_{t:T})$ as：

$$\begin{aligned} R\big((\tau_{t:T})\big) &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots \\ &= r_t + \gamma(r_{t+1} + \gamma^1 r_{t+2} + \cdots) \\ &= r_t + \gamma R\big((\tau_{t+1:T})\big) \end{aligned}$$

So：

$$\begin{aligned} V^{\pi}(s_t) &= \mathbb{E}_{\tau \sim p(\tau)}[r_t + \gamma R(\tau_{t+1:T})] \\ &= \mathbb{E}_{\tau \sim p(\tau)}[r_t + \gamma V^{\pi}(s_{t+1})] \end{aligned}$$

This is also called the Bellman equation of the state value function. Among all policies, the optimal policy π^* refers to the policy that can obtain the maximum value of $V^{\pi}(s)$, namely

$$\pi^* = \arg\max_{\pi} V^{\pi}(s) \ \forall s \in S$$

At this time, the state value function achieves the maximum value

$$V^*(s) = \max_{\pi} V^{\pi}(s) \ \forall s \in S$$

For the optimal policy, Bellman's equation is also satisfied:

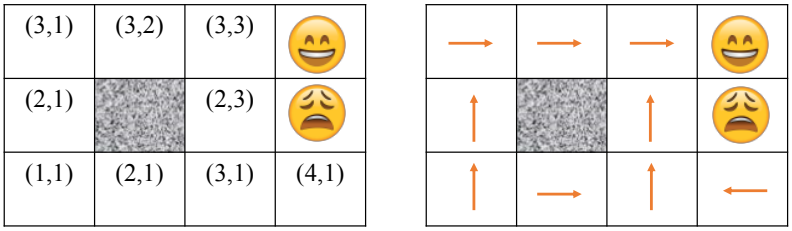
$$V^*(s_t) = \mathbb{E}_{\tau \sim p(\tau)}[r_t + \gamma V^*(s_{t+1})]$$

which is called Bellman optimal equation of the state value function.

Consider the maze problem in Figure 14.12. In the 3×4 grid, the grid with coordinates (2,2) is impassable, and the grid with coordinates (4,2) has a reward of -10 and the grid with coordinates (4,3) has a reward of is 10. The agent can start from any position, and the reward is -1 for every additional step. The goal of the game is to maximize the return. For this simple maze, the optimal vector for each position can be drawn directly, that is, at any starting point, the optimal strategy $\pi^*(a|s)$ is a deterministic policy, and the actions are marked in Figure 14.12(b) . Let $\gamma = 0.9$, then:

- Starting from $s_{(4,3)}$， i.e. coordinates $(4,3)$， the optimal policy is $V^*\big(s_{(4,3)}\big) = 10$
- Starting from $s_{(3,3)}$， $V^*\big(s_{(4,3)}\big) = -1 + 0.9 \cdot 10 = 8$
- Starting from $s_{(2,1)}$， $V^*\big(s_{(2,1)}\big) = -1 - 0.9 \cdot 1 - 0.9^2 \cdot 1 - 0.9^3 \cdot 1 + 0.9^4 \cdot 10 = 3.122$

It should be noted that the premise of the state value function is that under a certain strategy π, all the above calculations are to calculate the state value function under the optimal strategy.



(a)□□□□□□ (b)□□□□□□□□

Figure 0.12 Maze problem-V function

The value of the state value function reflects the quality of the state under the current policy. The larger $V^{\pi}(s_t)$, the greater the total return expectation of the current state. Take the space invader game that is more in line with the actual situation as an example. The agent needs to fire at the flying saucers, squids, crabs, octopuses and other objects, and score points when it hit them. At the same time, it must avoid being concentrated by these objects. A red shield can protect the agent, but the shield can be gradually destroyed by hits. In Figure 14.13, in the initial state of the game, there are many objects in the figure. Under a good policy π, a larger $V^{\pi}(s)$ value should be obtained. In Figure 14.14, there are fewer objects. No matter how good the policy is, it is impossible to obtain a larger value of $V^{\pi}(s)$. The quality of the policy will also affect the value of $V^{\pi}(s)$. As shown in Figure 14.15, a bad policy (such as moving to the right) will cause the agent to be hit. Therefore, $V^{\pi}(s)$=0. A good policy can shoot down the objects in the picture and obtain a certain reward.

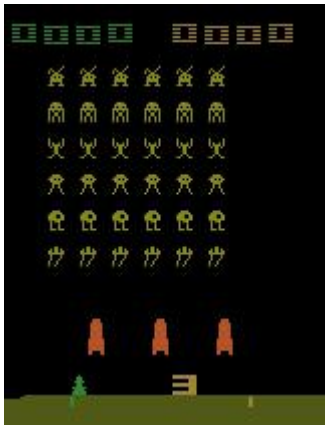


Figure 0.13 $V^{\pi}(s)$ may be larger under the policy π

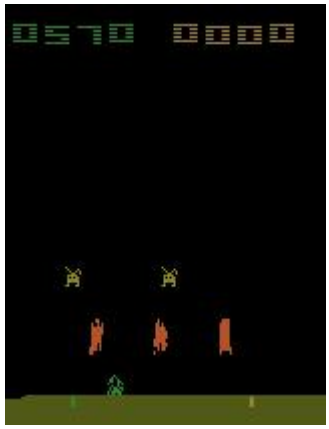


Figure 0.14 $V^{\pi}(s)$ is small under any policy π

Figure 0.15 Bad policy (such as to the right) will end the game $V^{\pi}(s) = 0$, good policy can still get a small return

State-Action Value Function (Q function for short), which is defined as the expected return value that can be obtained under the control of strategy π from the dual setting of state s_t and execution of action a_t:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{\tau \sim p(\tau)}\left[R(\tau_{t:T}) | \tau_{a_t} = a_t, \tau_{s_t} = s_t\right]$$

Although both the Q function and the V function are expected return values, the action a_t of the Q function is a prerequisite, which is different from the definition of the V function. Expand the Q

function to:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{\tau \sim p(\tau)}[r(s_t, a_t) + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots]$$
$$= \mathbb{E}_{\tau \sim p(\tau)}[r(s_t, a_t) + r_t + \gamma(r_{t+1} + \gamma^1 r_{t+2} + \cdots)]$$

So：

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{\tau \sim p(\tau)}[r(s_t, a_t) + \gamma V^{\pi}(s_{t+1})]$$

Because s_t and a_t are fixed，$r(s_t, a_t)$ is also fixed.

The Q function and the V function have the following relationship:

$$V^{\pi}(s_t) = \mathbb{E}_{a_t \sim \pi(a_t|s_t)}[Q^{\pi}(s_t, a_t)]$$

That is, when a_t is sampled from policy $\pi(a_t|s_t)$, the expected value of $Q^{\pi}(s_t, a_t)$ is equal to $V^{\pi}(s_t)$. Under the optimal policy $\pi^*(a|s)$, there is the following relationship:

$$Q^*(s_t, a_t) = \max_{\pi} Q^{\pi}(s_t, a_t)$$

$$\pi^* = \arg\max_{a_t} Q^*(s_t, a_t)$$

It also means：

$$V^*(s_t) = \max_{a_t} Q^*(s_t, a_t)$$

At this time：

$$Q^*(s_t, a_t) = \mathbb{E}_{\tau \sim p(\tau)}[r(s_t, a_t) + \gamma V^*(s_{t+1})]$$

$$= \mathbb{E}_{\tau \sim p(\tau)}\left[r(s_t, a_t) + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})\right]$$

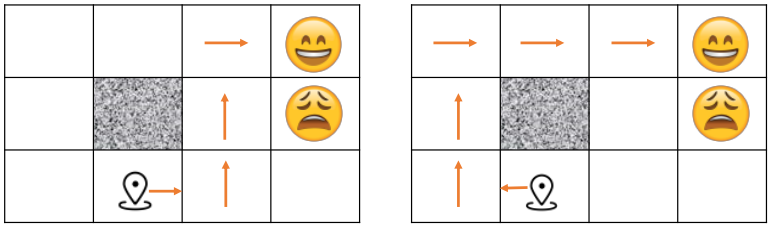
The above formula is called the Bellman optimal equation of the Q function.

We define the difference between $Q^{\pi}(s_t, a_t)$ and $V^{\pi}(s)$ as the advantage value function:

$$A^{\pi}(s, a) \triangleq Q^{\pi}(s, a) - V^{\pi}(s)$$

It shows the degree of advantage of taking action a in state s over the average level: $A^{\pi}(s, a) > 0$ indicates that taking action a is better than the average level; otherwise, it is worse than the average level. In fact, we have already applied the idea of advantage value function in the section of REINFORCE algorithm with bias.

Continuing to consider the example of the maze, let the initial state be $s_{(2,1)}$，a_t can be right or left. For function $Q^*(s_t, a_t)$, $Q^*(s_{(2,1)}, \text{right}) = -1 - 0.9 \cdot 1 - 0.9^2 \cdot 1 - 0.9^3 \cdot 1 + 0.9^4 \cdot 10 = 3.122$，$Q^*(s_{(2,1)}, \text{left}) = -1 - 0.9 \cdot 1 - 0.9^2 \cdot 1 - 0.9^3 \cdot 1 - 0.9^4 \cdot 1 - 0.9^5 \cdot 1 + 0.9^6 \cdot 10 = 0.629$. We have calculated $V^*(s_{(2,1)}) = 3.122$，and we can intuitively see that they satisfy $V^*(s_t) = \max_{a_t} Q^*(s_t, a_t)$.



(b)$Q^*(s_{2,1}, 右)$ (b) $Q^*(s_{2,1}, 左)$

Figure 0.16 Maze problem-Q function

Take the space invader game as an example to intuitively understand the concept of the Q function. In Figure 14.17, the agent in the figure is under the protective cover. If you choose to fire at this time, it is generally considered a bad action. Therefore, under a good policy π, $Q^{\pi}(s, \text{no fire}) > Q^{\pi}(s, \text{fire})$. If you choose to move to the left at this time in Figure 14.18, you may miss the object on the right due to insufficient time, so $Q^{\pi}(s, \text{left})$may be small. If the agent moves to the right and fires in Figure 14.19, $Q^{\pi}(s, \text{right})$will be larger.

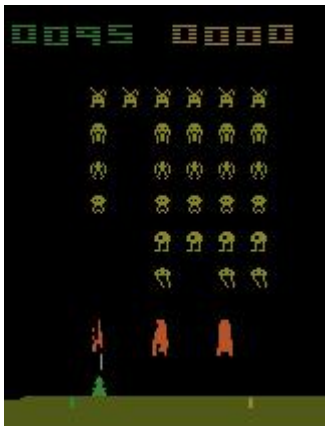


Figure 0.17 $Q^{\pi}(s, \text{no fire})$ may be larger than $Q^{\pi}(s, \text{fire})$

Figure 0.18 $Q^{\pi}(s, \text{left})$ may be smaller

Figure 0.19 Under a good policy π, $Q^{\pi}(s, \text{right})$ can still get some rewards.

After introducing the definition of the Q function and the V function, we will mainly answer the following two questions:

- ❑ How is the value function estimated?
- ❑ How to derive the policy from the value function?

14.4.2 Value function estimation

The estimation of value function mainly includes Monte Carlo method and Temporal-Difference method.

Monte Carlo method

The Monte Carlo method is actually to estimate the V function and the Q function through

multiple trajectories $\{\tau^{(n)}\}$ generated by the sampling policy $\pi(a|s)$. Consider the definition of the Q function:

$$Q^{\pi}(s,a) = \mathbb{E}_{\tau\sim p(\tau)}\left[R\left(\tau_{s_0=s,a_0=a}\right)\right]$$

According to the law of large numbers, it can be estimated by sampling

$$Q^{\pi}(s,a) \approx \hat{Q}^{\pi}(s,a) = \frac{1}{N}\sum_{n=1}^{N} R\left(\tau_{s_0=s,a_0=a}^{(n)}\right)$$

where $\tau_{s_0=s,a_0=a}^{(n)}$ represents the n-th sampled trajectory, $n \in [1,N]$. The actual state of each sampled trajectory is s, the initial action is a, and N is the total number of trajectories. The V function can be estimated according to the same method:

$$V^{\pi}(s) \approx \hat{V}^{\pi}(s) = \frac{1}{N}\sum_{n=1}^{N} R\left(\tau_{s_0=s}^{(n)}\right)$$

This method of estimating the expected return by sampling the total return of the trajectory is called the Monte Carlo method (MC method for short).

When the Q function or V function is parameterized through a neural network, the output of the network is recorded as $Q^{\pi}(s,a)$ or $V^{\pi}(s)$, and its true label is recorded as the Monte Carlo estimate $\hat{Q}^{\pi}(s,a)$ or $\hat{V}^{\pi}(s)$, the direct error between the network output value and the estimated value can be calculated through an error function such as the mean square error. The gradient descent algorithm is used to optimize the neural network. From this perspective, the estimation of the value function can be understood as a regression problem. The Monte Carlo method is simple and easy to implement, but it needs to obtain the complete trajectory, so the calculation efficiency is low, and there is no clear end state in some environments.

Temporal-Difference

Temporal-Difference (TD method for short) utilizes the Bellman equation properties of the value function. In the calculation formula, only one or more steps are required to obtain the error of the value function and optimize the update value function network. The Carlo method is more computationally efficient.

Recall the Bellman equation of the V function:

$$V^{\pi}(s_t) = \mathbb{E}_{\tau\sim p(\tau)}[r_t + \gamma V^{\pi}(s_{t+1})]$$

Therefore, the TD error term $\delta = r_t + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t)$ is constructed and updated as follows：

$$V^{\pi}(s_t) \leftarrow V^{\pi}(s_t) + \alpha\left(r_t + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t)\right)$$

where $\alpha \in [0,1]$ is the update step.

The Bellman optimal equation of the Q function is:

$$Q^{*}(s_t,a_t) = \mathbb{E}_{\tau\sim p(\tau)}\left[r(s_t,a_t) + \gamma \max_{a_{t+1}} Q^{*}(s_{t+1},a_{t+1})\right]$$

Similarly, construct TD error term $\delta = r(s_t, a_t) + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}) - Q^*(s_t, a_t)$, and use the following equation to update:

$$Q^*(s_t, a_t) \leftarrow Q^*(s_t, a_t) + \alpha \left(r(s_t, a_t) + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}) - Q^*(s_t, a_t) \right)$$

14.4.3 Policy improvement

The value function estimation method can obtain a more accurate value function estimation, but the policy model is not directly given. Therefore, the policy model needs to be derived indirectly based on the value function.

First, look at how to derive the policy model from the V function:

$$\pi^* = \arg\max_{\pi} V^{\pi}(s) \ \forall s \in S$$

Considering that the state space S and the action space A are usually huge, this way of traversing to obtain the optimal policy is not feasible. So can the policy model be derived from the Q function? Consider

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

In this way, an action can be selected by traversing the discrete action space A in any state s. This strategy $\pi'(s)$ is a deterministic policy. Because

$$V^{\pi}(s_t) = \mathbb{E}_{a_t \sim \pi(a_t|s_t)}[Q^{\pi}(s_t, a_t)]$$

So

$$V^{\pi'}(s_t) \geq V^{\pi}(s_t)$$

That is, the strategy π' is always better than or equal to the strategy π, thus achieving policy improvement.

The deterministic policy produces the same action in the same state, so the trajectory produced by each interaction may be similar. The policy model always tends to Exploitation but lacks Exploration, thus making the policy model limited to a local area, lack of understanding of global status and actions. In order to be able to add exploration capabilities to the $\pi'(s)$ deterministic policy, we can make the $\pi'(s)$ policy have a small probability ϵ to adopt a random policy to explore unknown actions and states.

$$\pi^{\epsilon}(s_t) = \begin{cases} \arg\max_{a} Q^{\pi}(s, a), & \text{probability of } 1 - \epsilon \\ \text{random action}, & \text{probability of } \epsilon \end{cases}$$

This policy is called ϵ-greedy method. It makes a small amount of modification on the basis of the original policy, and can balance utilization and exploration by controlling the hyperparameter ϵ, achieving simple and efficient.

The training process of the value function is shown in Figure 14.20.

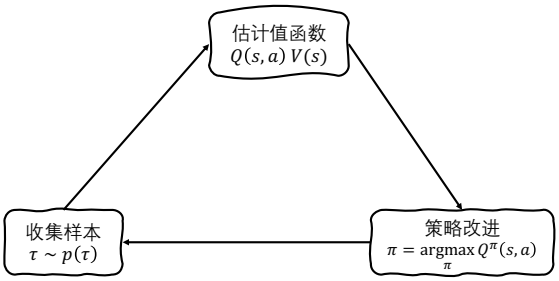


Figure 0.20 Value function method training process

14.4.4 SARSA algorithm

SARSA algorithm [5] uses

$$Q^{\pi}(s_t,a_t) \leftarrow Q^{\pi}(s_t,a_t) + \alpha\big(r(s_t,a_t) + \gamma Q^{\pi}(s_{t+1},a_{t+1}) - Q^{\pi}(s_t,a_t)\big)$$

method to estimate the Q function, at each step of the trajectory, only s_t, a_t, r_t, s_{t+1}, and a_{t+1}data can be used to update the Q network once, so it is called SARSA (State Action Reward State Action) algorithm. The s_t, a_t, r_t, s_{t+1}, and a_{t+1} of the SARSA algorithm come from the same policy $\pi^{\epsilon}(s_t)$, so they belong to the On-Policy algorithm.

14.4.5 DQN algorithm

In 2015, DeepMind proposed the Q Learning [4] algorithm implemented using deep neural networks, published in Nature [1], and trained and learned on 49 mini games in the Atari game environment, achieving a human level equivalent or even superior. The performance of human level has aroused the strong interest of the industry and the public in the research of reinforcement learning.

Q Learning algorithm uses

$$Q^{*}(s_t,a_t) \leftarrow Q^{*}(s_t,a_t) + \alpha\left(r(s_t,a_t) + \gamma \max_{a_{t+1}} Q^{*}(s_{t+1},a_{t+1}) - Q^{*}(s_t,a_t)\right)$$

to estimate the $Q^{*}(s_t,a_t)$ function, and use the $\pi^{\epsilon}(s_t)$ policy to obtain policy improvement. The Deep Q Network (DQN) uses a deep neural network to parameterize the $Q^{*}(s_t,a_t)$ function, and uses the gradient descent algorithm to update the Q network. The loss function is:

$$\mathcal{L} = \left(r_t + \gamma \max_{a} Q_{\theta}(s_{t+1},a) - Q_{\theta}(s_t,a_t)\right)^2$$

Since both the training target value $r_t + \gamma \max_{a} Q_{\theta}(s_{t+1},a)$ and the predicted value $Q_{\theta}(s_t,a_t)$ come from the same network, and the training data has a strong correlation, [1] proposed two measures to solve the problem: by adding Experience Relay Buffer to reduce the strong correlation of the data; by Freezing Target Network technology to fix the target estimation network and stabilize the training process.

The replay buffer pool is equivalent to a large data sample buffer pool. During each training,

the data pair (s, a, r, s') generated by the latest policy is stored in the replay buffer pool, and then multiple data pairs (s, a, r, s') are randomly sampled from the pool for training. In this way, the strong correlation of the training data can be reduced. It can also be found that the DQN algorithm is an Off-Policy algorithm with high sampling efficiency.

Freezing Target Network is a training technique. During training, the target network $Q_{\bar{\theta}}(s_{t+1}, a)$ and the prediction network $Q_\theta(s_t, a_t)$ come from the same network, but the update frequency of $Q_{\bar{\theta}}(s_{t+1}, a)$ network will be after $Q_\theta(s_t, a_t)$, which is equivalent to being in a frozen state when $Q_{\bar{\theta}}(s_{t+1}, a)$ is not updated, and then pull latest network parameters from $Q_\theta(s_t, a_t)$ after the freezing is over:

$$\mathcal{L} = \left(r_t + \gamma \max_a Q_{\bar{\theta}}(s_{t+1}, a) - Q_\theta(s_t, a_t) \right)^2$$

In this way, the training process can become more stable.

DQN algorithm is shown in Algorithm 3.

Algorithm 3: DQN algorithm

Randomly initialize $\boldsymbol{\theta}$

repeat

  Reset and get game initial state $\boldsymbol{s}$

  repeat

    Sample action $\boldsymbol{a} = \boldsymbol{\pi}^{\epsilon}(\boldsymbol{s})$

    Interact with environment and get reward $\boldsymbol{r}$ and state $\boldsymbol{s'}$

    Optimize Q network:

$$\boldsymbol{\nabla_\theta} \left(\boldsymbol{r}(\boldsymbol{s_t}, \boldsymbol{a_t}) + \boldsymbol{\gamma} \max_{\boldsymbol{a_{t+1}}} \boldsymbol{Q}^*(\boldsymbol{s_{t+1}}, \boldsymbol{a_{t+1}}) - \boldsymbol{Q}^*(\boldsymbol{s_t}, \boldsymbol{a_t}) \right)$$

    Update state $\boldsymbol{s} \leftarrow \boldsymbol{s'}$

  Until game ending

until reach required training times

Output: policy network $\boldsymbol{\pi_\theta}(\boldsymbol{a_t}|\boldsymbol{s_t})$

14.4.6 DQN variants

Although the DQN algorithm has made a huge breakthrough on the Atari game platform, follow-up studies have found that the Q value in DQN is often overestimated. In view of the defects of the DQN algorithm, some variant algorithms have been proposed.

Double DQN In [6], the Q network and estimated $\bar{Q}$ network of target $r_t + \gamma \bar{Q}\left(s_{t+1}, \max_a Q(s_{t+1}, a)\right)$ were separated and updated according to the loss function

$$\mathcal{L} = \left(r_t + \gamma \bar{Q}\left(s_{t+1}, \max_a Q(s_{t+1}, a)\right) - Q(s_t, a_t) \right)^2$$

Dueling DQN [7] separated the network output into $V(s)$ and $A(s,a)$, as shown in Figure 14.21. Then use

$$Q(s,a) = V(s) + A(s,a)$$

to generate Q function estimate $Q(s,a)$. The rest and DQN remain the same.

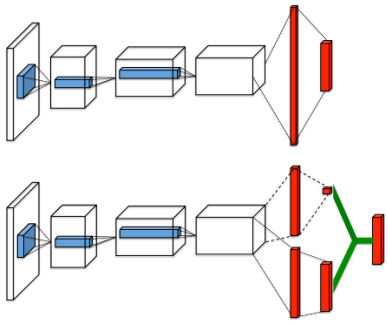


Figure 0.21 DQN network(upper) and Dueling DQN network(lower) [7]

14.4.7 Hands-on DQN

Here we continue to implement the DQN algorithm based on the balance bar game environment.

Q network The state of the balance bar game is a vector of length 4. Therefore, the input of the Q network is designed as 4 nodes. After a 256-256-2 fully connected layer, the distribution of the Q function estimation $Q(s,a)$ with the number of output nodes of 2 is obtained. The implementation of the network is as follows:

```
class Qnet(keras.Model):
    def __init__(self):
        # Create a Q network, the input is the state vector, and the output is
the Q value of the action
        super(Qnet, self).__init__()
        self.fc1 = layers.Dense(256, kernel_initializer='he_normal')
        self.fc2 = layers.Dense(256, kernel_initializer='he_normal')
        self.fc3 = layers.Dense(2, kernel_initializer='he_normal')

    def call(self, x, training=None):
        x = tf.nn.relu(self.fc1(x))
        x = tf.nn.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Replay buffer pool The replay buffer pool is used in the DQN algorithm to reduce the strong correlation between data. We use the Deque object in the ReplayBuffer class to implement the buffer pool function. During training, the latest data (s,a,r,s') is stored in the Deque object through the put (transition) method, and n data (s,a,r,s') are randomly sampled from the Deque object using sample(n) method. The implementation of the replay buffer pool is as follows:

```
class ReplayBuffer():
    # Replay buffer pool
    def __init__(self):
        # Deque
        self.buffer = collections.deque(maxlen=buffer_limit)

    def put(self, transition):
        self.buffer.append(transition)

    def sample(self, n):
        # Sample n samples
        mini_batch = random.sample(self.buffer, n)
        s_lst, a_lst, r_lst, s_prime_lst, done_mask_lst = [], [], [], [], []
        # Organize by category
        for transition in mini_batch:
            s, a, r, s_prime, done_mask = transition
            s_lst.append(s)
            a_lst.append([a])
            r_lst.append([r])
            s_prime_lst.append(s_prime)
            done_mask_lst.append([done_mask])
        # Convert to tensor
        return tf.constant(s_lst, dtype=tf.float32),\
                    tf.constant(a_lst, dtype=tf.int32), \
                    tf.constant(r_lst, dtype=tf.float32), \
                    tf.constant(s_prime_lst, dtype=tf.float32), \
                    tf.constant(done_mask_lst, dtype=tf.float32)
```

Policy improvement The ϵ-greedy method is implemented here. When sampling actions, there a probability of $1-\epsilon$ to choose $\arg\max_a Q^{\pi}(s,a)$, and a probability of ϵ to randomly choose an action.

```
    def sample_action(self, s, epsilon):
        # Send the state vector to get the strategy: [4]
        s = tf.constant(s, dtype=tf.float32)
        # s: [4] => [1,4]
        s = tf.expand_dims(s, axis=0)
        out = self(s)[0]
        coin = random.random()
        # Policy improvement: e-greedy way
        if coin < epsilon:
            # epsilon larger
            return random.randint(0, 1)
        else:  # Q value is larger
            return int(tf.argmax(out))
```

Network main process The network trains up to 10,000 rounds. At the beginning of the round, the game is first reset to get the initial state s, and an action is sampled from the current Q network to interact with the environment to obtain the data pair (s, a, r, s'), and stored in the replay buffer pool. If the number of samples in the current replay buffer pool is sufficient, sample a batch of data, and optimize the estimation of the Q network according to the TD error until the end of the game.

```
for n_epi in range(10000):  # Training times
        # The epsilon probability will also be attenuated by 8% to 1%. The
more you go, the more you use the action with the highest Q value.
        epsilon = max(0.01, 0.08 - 0.01 * (n_epi / 200))
        s = env.reset()  # Reset environment
        for t in range(600):  # Maximum timestamp of a round
            # if n_epi>1000:
            #     env.render()
            # According to the current Q network, extract and improve the
policy.
            a = q.sample_action(s, epsilon)
            # Use improved strategies to interact with the environment
            s_prime, r, done, info = env.step(a)
            done_mask = 0.0 if done else 1.0  # End flag mask
            # Save
            memory.put((s, a, r / 100.0, s_prime, done_mask))
            s = s_prime  # Update state
            score += r  # Record return
            if done:  # End round
                break
        if memory.size() > 2000:  # train if size is greater than 2000
            train(q, q_target, memory, optimizer)
        if n_epi % print_interval == 0 and n_epi != 0:
            for src, dest in zip(q.variables, q_target.variables):
                dest.assign(src)  # weights come from Q
```

During training, only the Q_θ network will be updated, while the $Q_{\bar{\theta}}$ network will be frozen. After the Q_θ network has been updated many times, use the following code to copy the latest parameters from Q_θ to $Q_{\bar{\theta}}$.

```
for src, dest in zip(q.variables, q_target.variables):
                dest.assign(src)  # weights come from Q
```

Optimize the Q network When optimizing the Q network, it will train and update 10 times at a time. Randomly sample from the replay buffer pool each time, and select the action $\max_a Q_{\bar{\theta}}(s_{t+1}, a)$ to construct the TD difference. Here we use the Smooth L1 error to construct the TD error:

$$\mathcal{L} = \begin{cases} 0.5 * (x-y)^2, & |x-y| < 1 \\ |x-y| - 0.5, & |x-y| \geq 1 \end{cases}$$

In TensorFlow, Smooth L1 error can be implemented using Huber error as follows:

```
def train(q, q_target, memory, optimizer):
    # Construct the error of Bellman equation through Q network and shadow
network.
    # And only update the Q network, the update of the shadow network will
lag behind the Q network
    huber = losses.Huber()
    for i in range(10):  # Train 10 times
        # Sample from buffer pool
        s, a, r, s_prime, done_mask = memory.sample(batch_size)
        with tf.GradientTape() as tape:
            # s: [b, 4]
            q_out = q(s)  # Get Q(s,a) distribution
            # Because TF's gather_nd is different from pytorch's gather, we
need to the coordinates of gather_nd, indices:[b, 2]
            # pi_a = pi.gather(1, a) # pytorch only needs one line.
            indices = tf.expand_dims(tf.range(a.shape[0]), axis=1)
            indices = tf.concat([indices, a], axis=1)
            q_a = tf.gather_nd(q_out, indices) # The probability of action,
[b]
            q_a = tf.expand_dims(q_a, axis=1) # [b]=> [b,1]
            # Get the maximum value of Q(s',a). It comes from the shadow
network!  [b,4]=>[b,2]=>[b,1]
            max_q_prime =
tf.reduce_max(q_target(s_prime),axis=1,keepdims=True)
            # Construct the target value of Q(s,a_t)
            target = r + gamma * max_q_prime * done_mask
            # Calcualte error between Q(s,a_t) and target
            loss = huber(q_a, target)
        # Update network
        grads = tape.gradient(loss, q.trainable_variables)
        optimizer.apply_gradients(zip(grads, q.trainable_variables))
```

14.5 Actor-Critic method

在介绍原始的策略梯度算法时，为了缩减方差，我们引入了基准线b机制：

When introducing the original policy gradient algorithm, in order to reduce the variance, we introduced the bias b mechanism:

$$\frac{\partial J(\theta)}{\partial \theta} = \mathbb{E}_{\tau \sim p_\theta(\tau)}\left[\sum_{t=1}^{T-1} \frac{\partial}{\partial \theta} \log \pi_\theta\left(a_t|s_t\right)(R(\tau) - b)\right]$$

where b can be estimated by Monte Carlo method $b = \frac{1}{N}\sum_{n=1}^{N} R\left(\tau^{(n)}\right)$. If $R(\tau)$ is understood

as the estimated value of $Q^{\pi}(s_t, a_t)$, the bias b is understood as the average level $V^{\pi}(s_t)$ of state s_t, then $R(\tau) - b$ is (approximately) the advantage value function $A^{\pi}(s, a)$. Among them, if the bias value function $V^{\pi}(s_t)$ is estimated using neural networks, it is the Actor-Critic method (AC method for short). The policy network $\pi_{\theta}(a_t|s_t)$ is called Actor, which is used to generate policies and interact with the environment. The $V_{\phi}^{\pi}(s_t)$ value network is called Critic, which is used to evaluate the current state. θ and ϕ are the parameters of the Actor network and the Critic network, respectively.

For the Actor network π_{θ}, the goal is to maximize the return expectation, and the parameter θ of the policy network is updated through the partial derivative of $\frac{\partial J(\theta)}{\partial \theta}$:

$$\theta' \leftarrow \theta + \eta \cdot \frac{\partial J}{\partial \theta}$$

For the Critic network V_{ϕ}^{π}, the goal is to obtain an accurate $V_{\phi}^{\pi}(s_t)$ value function estimate through the MC method or the TD method:

$$\phi = \underset{\phi}{\operatorname{argmin}} \operatorname{dist}(V_{\phi}^{\pi}(s_t), V_{\text{target}}^{\pi}(s_t))$$

where dist(a,b) is the distance measurer of a and b, such as Euclidean distance. $V_{\text{target}}^{\pi}(s_t)$ is the target value of $V_{\phi}^{\pi}(s_t)$. When estimated by the MC method,

$$V_{\text{target}}^{\pi}(s_t) = R(\tau_{t:T})$$

When estimated by the TD method,

$$V_{\text{target}}^{\pi}(s_t) = r_t + \gamma V^{\pi}(s_{t+1})$$

14.5.1 Advantage AC algorithm

The Actor Critic algorithm using the advantage value function $A^{\pi}(s, a)$ is called the Advantage Actor-Critic algorithm. It is currently one of the mainstream algorithms that use the Actor Critic idea. In fact, the Actor Critic series of algorithms do not have to use the advantage value function $A^{\pi}(s, a)$. There are other variants.

When the Advantage Actor-Critic algorithm is trained, the Actor obtains the action a_t according to the current state s_t and the policy π_{θ} sampling, and then interacts with the environment to obtain the next state s_{t+1} and reward r_t. The TD method can estimate the target value $V_{\text{target}}^{\pi}(s_t)$ of each step, thereby updating the Critic network so that the estimation of the value network is closer to the expected return of the real environment. $\hat{A}_t = r_t + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t)$ is used to estimate the advantage value of the current action, and the following equation is used to calculate the gradient info of the Actor network.

$$\mathcal{L}^{PG}(\theta) = \hat{\mathbb{E}}_t\left[\log \pi_{\theta}(a_t|s_t)\hat{A}_t\right]$$

By repeating this process, the Critic network will be more and more accurate, and the Actor network will also adjust its policy to make it better next time.

14.5.2 A3C algorithm

The full name of the A3C algorithm is the Asynchronous Advantage Actor-Critic algorithm. It is an asynchronous version proposed by DeepMind based on the Advantage Actor-Critic algorithm [8]. The Actor-Critic network is deployed in multiple threads for simultaneous training, and the parameters are synchronized through the global network. . This asynchronous training mode greatly improves the training efficiency, therefore, the training speed is faster and the algorithm performance is better.

As shown in Figure 14.22, the algorithm will create a new Global Network and M Worker threads. Global Network contains Actor and Critic networks, and each thread creates a new interactive environment, Actor and Critic networks. In the initialization phase, Global Network initializes parameters θ and ϕ randomly. The Actor-Critic network in Worker pulls parameters synchronously from Global Network to initialize the network. During training, the Actor-Critic network in the Worker first pulls the latest parameters from the Global Network, and then the latest policy $\pi_\theta(a_t|s_t)$ will sample actions to interact with the private environment, and calculate the gradients of parameters θ and ϕ according to the Advantage Actor-Critic algorithm. After completing the gradient calculation, each worker submits the gradient information to the Global Network, and uses the optimizer of the Global Network to complete the parameter update. In the algorithm testing phase, only Global Network interacts with the environment.

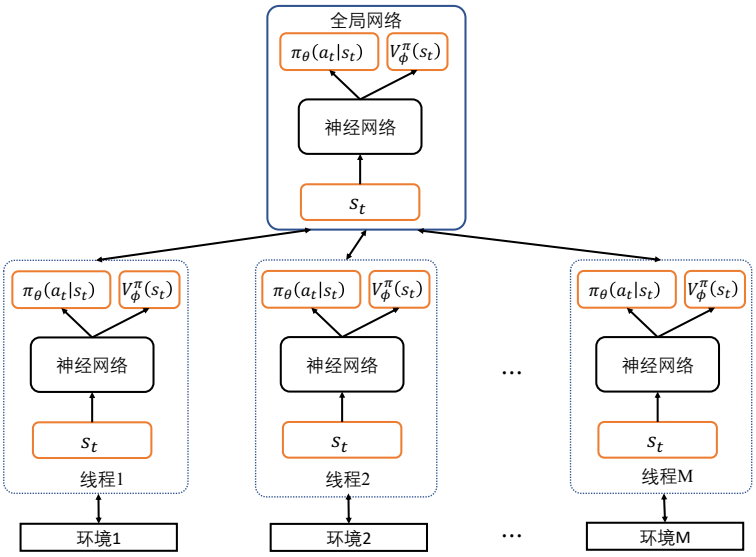


Figure 0.22 A3C algorithm

14.5.3 Hands-on A3C

Next we implement the asynchronous A3C algorithm. Like the ordinary Advantage AC algorithm, the ActorCritic network needs to be created. It contains an Actor sub-network and a Critic sub-network. Sometimes Actor and Critic will share the previous network layers to reduce the amount of network parameters. The balance bar game is relatively simple. We use a 2-layer fully connected network to parameterize the Actor network, and another 2-layer fully connected network to parameterize the Critic network.

The Actor-Critic network code is as follows:

```
class ActorCritic(keras.Model):
    # Actor-Critic model
    def __init__(self, state_size, action_size):
        super(ActorCritic, self).__init__()
        self.state_size = state_size # state vector length
        self.action_size = action_size # action size
        # Policy network Actor
        self.dense1 = layers.Dense(128, activation='relu')
        self.policy_logits = layers.Dense(action_size)
        # V network Critic
        self.dense2 = layers.Dense(128, activation='relu')
        self.values = layers.Dense(1)
```

The forward propagation process of Actor-Critic calculates the policy distribution $\pi_\theta(a_t|s_t)$ and the V function estimation $V^\pi(s_t)$ separately. The code is as follows:

```
    def call(self, inputs):
        # Get policy distribution Pi(a|s)
        x = self.dense1(inputs)
        logits = self.policy_logits(x)
        # Get v(s)
        v = self.dense2(inputs)
        values = self.values(v)
        return logits, values
```

Worker thread class In the Worker thread, the same calculation process as the Advantage AC algorithm is implemented, except that the gradient information of parameters θ and ϕ is not directly used to update the Actor-Critic network of the Worker, instead it is submitted to the Global Network for update. Specifically, in the initialization phase of the Worker class, the server object and the opt object represent the Global Network model and optimizer respectively, and create a private ActorCritic class client and interactive environment env.

```
class Worker(threading.Thread):
    # The variables created here belong to the class, not to the instance,
and are shared by all instances
    global_episode = 0 # Round count
    global_avg_return = 0 # Average return
    def __init__(self,  server, opt, result_queue, idx):
        super(Worker, self).__init__()
        self.result_queue = result_queue # Shared queue
        self.server = server # Central model
        self.opt = opt # Central optimizer
        self.client = ActorCritic(4, 2) # Thread private network
        self.worker_idx = idx # Thread id
        self.env = gym.make('CartPole-v0').unwrapped
        self.ep_loss = 0.0
```

In the thread running phase, each thread interacts with the environment for up to 400 rounds. At the beginning of the round, the client network sampling action is used to interact with the environment and saved to the Memory object. At the end of the round, train the Actor network and the Critic network to obtain the gradient information of the parameters θ and ϕ, and call the opt optimizer object to update the Global Network.

```
    def run(self):
        total_step = 1
        mem = Memory() # Each worker maintains a memory
        while Worker.global_episode < 400: # Maximum number of frames not
reached
            current_state = self.env.reset() # Reset client state
            mem.clear()
            ep_reward = 0.
            ep_steps = 0
            self.ep_loss = 0
            time_count = 0
            done = False
            while not done:
                # Get Pi(a|s),no softmax
                logits, _ = self.client(tf.constant(current_state[None, :],
                                        dtype=tf.float32))
                probs = tf.nn.softmax(logits)
                # Random sample action
                action = np.random.choice(2, p=probs.numpy()[0])
                new_state, reward, done, _ = self.env.step(action) # Interact
                if done:
                    reward = -1
                ep_reward += reward
                mem.store(current_state, action, reward) # Record

                if time_count == 20 or done:
                    # Calculate the error of current client
                    with tf.GradientTape() as tape:
                        total_loss = self.compute_loss(done, new_state, mem)
                    self.ep_loss += float(total_loss)
                    # Calculate error
                    grads = tape.gradient(total_loss,
self.client.trainable_weights)
                    # Submit gradient info to server, and update gradient
                    self.opt.apply_gradients(zip(grads,
                                                 self.server.trainable_weights))
                    # Pull latest gradient info from server
                    self.client.set_weights(self.server.get_weights())
                    mem.clear() # Clear Memory
```

```
                time_count = 0

                if done:  # Calcualte return
                    Worker.global_avg_return = \
                        record(Worker.global_episode, ep_reward,
self.worker_idx,
                               Worker.global_avg_return, self.result_queue,
                               self.ep_loss, ep_steps)
                    Worker.global_episode += 1
            ep_steps += 1
            time_count += 1
            current_state = new_state
            total_step += 1
        self.result_queue.put(None) # End thread
```

Actor-Critic error calculation When each Worker class is trained, the error calculation of Actor and Critic network is implemented as follows. Here we use the Monte Carlo method to estimate the target value $V^{\pi}_{\text{target}}(s_t)$, and use the distance between $V^{\pi}_{\text{target}}(s_t)$ and $V^{\pi}_{\phi}(s_t)$ the two as the error function value_loss of the Critic network. The policy loss function policy_loss of the Actor network comes from

$$-\mathcal{L}^{PG}(\theta) = -\hat{\mathbb{E}}_t\left[\log \pi_\theta\,(a_t|s_t)\hat{A}_t\right]$$

where $-\hat{\mathbb{E}}_t\left[\log \pi_\theta\,(a_t|s_t)\hat{A}_t\right]$ is implemented by TensorFlow's cross-entropy function. After the various loss functions are aggregated, the total loss function is formed and returned.

```
def compute_loss(self,
                 done,
                 new_state,
                 memory,
                 gamma=0.99):
        if done:
            reward_sum = 0.
        else:
            reward_sum = self.client(tf.constant(new_state[None, :],
                                     dtype=tf.float32))[-1].numpy()[0]
        # Calculate return
        discounted_rewards = []
        for reward in memory.rewards[::-1]:  # reverse buffer r
            reward_sum = reward + gamma * reward_sum
            discounted_rewards.append(reward_sum)
        discounted_rewards.reverse()
        # Get Pi(a|s) and v(s)
        logits, values = self.client(tf.constant(np.vstack(memory.states),
                                     dtype=tf.float32))
        # Calculate advantage = R() - v(s)
        advantage = tf.constant(np.array(discounted_rewards)[:, None],
```

```
                                dtype=tf.float32) - values
        # Critic network loss
        value_loss = advantage ** 2
        # Policy loss
        policy = tf.nn.softmax(logits)
        policy_loss = tf.nn.sparse_softmax_cross_entropy_with_logits(
                    labels=memory.actions, logits=logits)
        # When calculating the policy network loss, the V network is not
calculated
        policy_loss *= tf.stop_gradient(advantage)

        entropy = tf.nn.softmax_cross_entropy_with_logits(labels=policy,
                                                  logits=logits)
        policy_loss -= 0.01 * entropy
        # Aggregate each error
        total_loss = tf.reduce_mean((0.5 * value_loss + policy_loss))
        return total_loss
```

Agent The agent is responsible for the training of the entire A3C algorithm. In the initialization phase, the agent class creates a new Global Network object server and its optimizer object opt.

```
class Agent:
    # Agent, include server
    def __init__(self):
        # server optimizer, no client, pull parameters from server
        self.opt = optimizers.Adam(1e-3)
        # Sever model
        self.server = ActorCritic(4, 2) # State vector, action size
        self.server(tf.random.normal((2, 4)))
```

At the beginning of training, each worker thread object is created, and each thread object is started to interact with the environment. When each worker object interacts, it will pull the latest network parameters from the Global Network, and use the latest policy to interact with the environment and calculate its own loss. Finally, each worker submits the gradient information to the Global Network, and call the opt object to optimize the Global Network. The training code is as follows:

```
    def train(self):
        res_queue = Queue() # Shared queue
        # Create interactive environment
        workers = [Worker(self.server, self.opt, res_queue, i)
                   for i in range(multiprocessing.cpu_count())]
        for i, worker in enumerate(workers):
            print("Starting worker {}".format(i))
            worker.start()
        # Plot return curver
```

```
moving_average_rewards = []
while True:
    reward = res_queue.get()
    if reward is not None:
        moving_average_rewards.append(reward)
    else: # End
        break
[w.join() for w in workers] # Quit threads
```

14.6 Summary

This chapter introduces the problem setting and basic theory of reinforcement learning, and introduces two series of algorithms to solve reinforcement learning problems: policy gradient method and value function method. The policy gradient method directly optimizes the policy model, which is simple and direct, but the sampling efficiency is low. The sampling efficiency of the algorithm can be improved by the importance sampling technique. The value function method has high sampling efficiency and is easy to train, but the policy model needs to be derived indirectly from the value function. Finally, the Actor-Critic method combining the policy gradient method and the value function method is introduced. We also introduced the principles of several typical algorithms, and used the balance bar game environment for algorithm implementation and testing.

14.7 References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature,* 518, pp. 529-533, 2 2015.

[2] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature,* 529, pp. 484-503, 2016.

[3] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. Driessche, T. Graepel and D. Hassabis, "Mastering the game of Go without human knowledge," *Nature,* 550, pp. 354--, 10 2017.

[4] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning,* 8, pp. 229-256, 01 5 1992.

[5] G. A. Rummery and M. Niranjan, "On-Line Q-Learning Using Connectionist Systems," 1994.

[6] H. Hasselt, A. Guez and D. Silver, "Deep Reinforcement Learning with Double Q-learning," *CoRR,* abs/1509.06461, 2015.

[7] Z. Wang, N. Freitas and M. Lanctot, "Dueling Network Architectures for Deep Reinforcement Learning," *CoRR,* abs/1511.06581, 2015.

[8] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver and K. Kavukcuoglu, "Asynchronous Methods for Deep Reinforcement Learning," *CoRR,* abs/1602.01783, 2016.

[9] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, 1992.

[10] J. Schulman, S. Levine, P. Abbeel, M. Jordan and P. Moritz, "Trust Region Policy Optimization," *Proceedings of the 32nd International Conference on Machine Learning*, Lille, 2015.

[11] J. Schulman, F. Wolski, P. Dhariwal, A. Radford and O. Klimov, "Proximal Policy Optimization Algorithms," *CoRR,* abs/1707.06347, 2017.