Chapter 10
Deep Reinforcement Learning

This chapter starts by covering the basic concepts involved in reinforcement learning and then describes how to solve reinforcement learning tasks by using basic and deep learning-based solutions. It also provides a brief overview of the typical algorithms central to the deep learning-based solutions, namely DQN, DDPG, and A3C.

10.1 Basic Concepts of Reinforcement Learning

Deep reinforcement learning, as its name implies, combines the principles used in deep learning and reinforcement learning, covering a wide range of knowledge and, in particular, mathematical knowledge such as Markov property, Markov decision process, Bellman equation, and optimal control. This section therefore focuses on the basic mathematical concepts involved in reinforcement learning in order to provide a greater understanding of deep reinforcement learning.

10.1.1 Basic Concepts and Theories

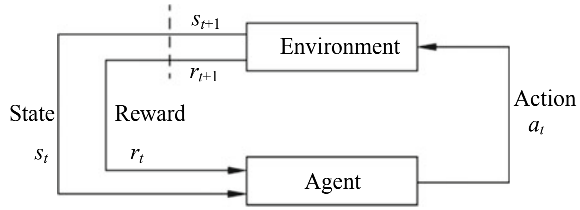
As mentioned earlier, we are better able to understand deep reinforcement learning once we have a firm grasp of the basic concepts and mathematical theories behind reinforcement learning. In this section, we focus on the concepts of policy π, action a, value v, reward r, and the relationships between them.

1. **Fundamental theory**
   We can understand reinforcement learning through the concepts of agents, environments, states, rewards, and actions, as shown in Fig. 10.1. The agent interacts with the environment, performing actions that are rewarded (or punished). In the figure, the state of the agent at time t is s_t. According to this state, the agent performs action a_t in the environment. The environment moves to a new state

© Tsinghua University Press 2021

L. Chen, *Deep Learning and Practice with MindSpore*, Cognitive Intelligence and Robotics, https://doi.org/10.1007/978-981-16-2233-5_10

Fig. 10.1 Basic architecture for reinforcement learning



s_{t+1} and gives the reward r_{t+1} corresponding to the new state after sensing the new action of the agent. The agent interacts with the environment repeatedly according to this process, which is central to reinforcement learning.

2. **Reward**
Reward is a mechanism by which to define the agent's learning objective. Each time the agent interacts with the environment, the environment feeds back the reward signal r to the agent. The reward indicates the quality of action a and can be regarded as a reward or punishment mechanism to evaluate the success or failure of the agent's actions.
The goal of reinforcement learning is not to maximize the current reward but rather to maximize cumulative reward R, the sum of all rewards, as shown in Formula (10.1):

$$R = r_1 + r_2 + \ldots + r_n \tag{10.1}$$

Because the environment is generally either stochastic or unknown, the next state s may be stochastic. This means that we are unable to determine the agent's next-step action nor its reward. As the agent explores further into the future, the results of the agent's actions become less certain. In order to account for this fact, we can replace the cumulative future reward with the cumulative future discounted reward G_t:

$$G_t = R_t + \gamma R_{t+1} + \ldots + \gamma^{n-t} R_n, \gamma \in [0, 1] \tag{10.2}$$

By multiplying the reward with the discount factor γ, this formula ensures that future rewards have less impact on the cumulative future discounted reward the further away they are from the current time step t. The cumulative future discounted reward G_t for the time step t can be represented by using the cumulative future discounted reward G_{t+1} for the time step $t + 1$:

$$\begin{aligned} G_t &= R_t + \gamma\left[R_{t+1} + \gamma(R_{t+2} + \ldots)\right] \\ &= R_t + \gamma G_{t+1} \end{aligned} \tag{10.3}$$

In conclusion, the ultimate goal of reinforcement learning is to ensure that the agent selects a policy that maximizes the cumulative future discounted reward G_t.

3. **Policy**

Policy, defined as π, refers to the rules that determine how the agent selects actions according to the state s_t and reward r_t, where the value function v is the expected cumulative reward E (G). In reinforcement learning, the optimal policy π^* is obtained through trial-and-error learning that utilizes feedback from the environment and corresponding state adjustments.

In general terms, we can classify policy π as either a deterministic or a stochastic policy.

(1) Deterministic policy: The action a is selected according to the state s, that is, $a = \pi(s)$. The selection process is deterministic, with no probability, and does not require the agent to make a selection.

(2) Stochastic policy: An action is selected according to the random probability P_{sa}, as defined by $\pi(a|s) = P[a_t = a|s_t= s]$ (note that $\Sigma\pi(a|s) = 1$). In state s, the agent selects the probability of action a according to the probability of each action $\pi(a|s)$.

4. **Value**

The value function is used to evaluate the state s_t of the agent at time step t, that is, to evaluate the immediate reward r for a given interaction. It is called the state value function $v(s)$ when the input of the algorithm is the state s; the action value function $q(s,a)$ when the input is the state-action pair $< s,a >$; and the value function v when the input is not distinguished.

The state value function $v(s)$ is a prediction of the cumulative future discounted reward, representing the expected reward that the agent can obtain by performing the action a in the state s.

$$v(s) = E[G_t|s_t = s] \tag{10.4}$$

The action value function $q(s,a)$ is typically used to evaluate the quality of the action a selected by the agent in the state s. This function and the state value function $v(s)$ are similar, but the difference between them is that the former one considers the effect of performing action a at the current time step t.

$$q(s, a) = E[G_t|s_t = s, a_t = a] \tag{10.5}$$

Formula (10.5) shows that the result of the action value function $q(s,a)$ is the mathematical expectation value, that is, the expected cumulative discounted reward.

10.1.2 Markov Decision Process

The Markov decision process (MDP) is a central aspect of deep reinforcement learning. In this section, we describe not only the MDP itself, but also the Markov property (MP), in order to lay a solid foundation for understanding deep reinforcement learning.

1. **Markov property**
   The Markov property indicates that a process is memoryless. Specifically, after the action a_t is performed in state s_t, the state s_{t+1} and reward r_{t+1} of the next time step are associated with only the current state s_t and action a_t. They are associated with neither the state of a historical time step nor an earlier time step. This means that the next state of the system is related to only the current state and not to the previous or earlier state—in essence, the Markov property has no aftereffect. However, in real-world applications, the feedback of the time step $t + 1$ does not necessarily depend on only the state and action of the time step t. Consequently, the task that the agent needs to accomplish does not completely satisfy the Markov property. But, to simplify the process of solving a reinforcement learning task, it is assumed that the task does indeed satisfy the Markov property—this is achieved by restricting the state of the environment.
2. **Markov decision process**
   The memoryless characteristic of the Markov property, as mentioned earlier, simplifies the Markov decision process considerably. This process is represented as a four-tuple:

$$\text{MDP} = (S, A, P, R) \tag{10.6}$$

where S is a set of states called the state space, and $S = \{s_1, s_2, \ldots, s_n\}$. Here, s_i denotes the state of the agent's environment at the time step i;

A is a set of actions called the action space, where $A = \{a_1, a_2, \ldots, a_n\}$. Here, a_i denotes the action performed by the agent at the time step i;

P is the state transition probability. It is the probability that action a in state s at time t will lead to state s' at time t+1 and is denoted as $p(s'|s,a)$. If the feedback signal r of the environment is received, the state transition probability is denoted as $p(s',r|s,a)$;

R is the reward function. It is the reward r received by the agent after transitioning from state s to state s' upon performing action a, where $r = R(s,a)$.

The Markov decision process is central to learning and solving most—if not all—reinforcement learning tasks. Because the process takes into account both the action a and state s, the next state in a reinforcement learning task is related to both the current state s and action a.

By converting reinforcement learning tasks into the Markov decision process, we can significantly reduce the difficulty and complexity and thereby increase the efficiency and accuracy, in solving such tasks.

10.1.3 Bellman Equation

Solving a deep reinforcement learning task is, to some extent, equivalent to optimizing the Bellman equation.

Because the Bellman equation represents the relationship between the current time t and the next time $t + 1$ (specifically, the values of their states), we can use it to represent both the state value function $v(s)$ and the action value function $q(s,a)$.

For both of these functions, the substitution of Formula (10.2) includes two parts: the immediate reward r_t and the discount value $\gamma v(s_{t+1})$ of a future state. An example of the state value function $v(s)$ represented by the Bellman equation is as follows:

$$v(s) = E[G_t|s_t = s] = E[r_t + \gamma v(s_{t+1})|s_t = s] \tag{10.7}$$

We can therefore express the Bellman equation of the state value function $v(s)$ as follows:

$$v(s) = R_s + \gamma \sum_{s' \in S} P_{ss'} v(s') \tag{10.8}$$

Using Formula (10.8), we can obtain the value function of the current state s by adding the reward R_s of the current state to the product of the state transition probability $P_{ss'}$ and the state value function $v(s')$ of the next state, where γ is the future discount factor. We can also use linear algebra to reduce Formula (10.8) as follows:

$$v = \mathbf{R} + \gamma \mathbf{P} v' \tag{10.9}$$

10.2 Basic Solution Method

Section 10.1 covered the basic concepts of reinforcement learning that allow us to abstract reinforcement learning tasks through the Markov decision process and express such tasks through the Bellman equation. Building on that understanding, this section describes the solution methods of reinforcement learning.

We can consider the process of solving a reinforcement learning task to be the same as finding the optimal policy, which we can obtain by first solving the optimal value function. This means that the solution of the optimal value function and, in turn, the reinforcement learning task is optimization of the Bellman equation.

For a small-scale Markov decision process, we can solve the value function directly; but for a large-scale process, we need to optimize the Bellman equation by using methods such as dynamic programming, Monte Carlo, or temporal difference.

These methods enable us to solve the Bellman equation according to the Markov decision process, allowing us to obtain the reinforcement learning model.

10.2.1 Dynamic Programming Method

The dynamic programming method splits a complex problem into several subproblems that can be solved individually. “Dynamic” means that the problem includes a series of states and can change gradually over time, whereas “programming” means that each subproblem can be optimized.

With this method, we use the value function of all subsequent states (denoted as *s’)* of the current state *s* to calculate the value function and calculate the subsequent state value function according to $p(s'|s,a)$ of the MDP in the environment model. The formula for calculating the value function is as follows:

$$v(s) \leftarrow \sum \pi(a|s) \sum p(s'|s,a)[r(s'|s,a) + \gamma v(s')] \tag{10.10}$$

1. **Policy evaluation**
   Given a known environment model, we can use policy evaluation to estimate the expected cumulative reward of a policy and accurately measure the policy. Policy evaluation enables us to evaluate the policy π by calculating the state value function $v_\pi(s)$ corresponding to this policy. In other words, given a policy π, we can calculate the expected state value $v(s)$ of each state s under the policy and subsequently evaluate the policy according to the obtained values.
2. **Policy improvement**
   Policy improvement allows us to act on the results we obtain through policy evaluation, namely to find a better policy. After we calculate the state value $v(s)$ of the current policy by using policy evaluation, we then use the policy improvement algorithm to further solve the calculated state value $v(s)$ in order to find a better policy.
3. **Policy iteration**
   Policy iteration encompasses both policy evaluation and policy improvement. In order to describe the policy iteration process, let us take policy π_0 as an example. We first use policy evaluation to obtain the state value function $v_{\pi_0}(s)$ of the policy and then use policy improvement to obtain a better policy π_1. For this new policy, we again use policy evaluation to obtain the corresponding state value function v_{π_0} (s), and then, through policy improvement, we again obtain a better policy π_2. By performing these steps repeatedly, the policy iteration algorithm nears the optimal state value $v(s)$, enabling us to eventually obtain the optimal policy π^* and its corresponding state value function v_{π^*} (s).
4. **Value iteration**

The value iteration algorithm, in essence, is a more efficient version of the policy iteration algorithm we just described. With this efficiency-optimized algorithm, all states *s* are updated in each iteration according to the following formula:

$$v_{k+1}(s) = \max_{a} \sum_{s', r} p(s', r|s, a)[r + \gamma v_k(s')] \tag{10.11}$$

where *p*(*s'*, *r* | *s*, *a*) is the probability that the environment transitions to the state *s'* and the reward *r* is obtained when the action *a* is performed in the state *s*.

The objective of value iteration is to maximize the probability of the state value. After $k + 1$ rounds of iterations are performed, the maximum state value *v*(*s*) can be assigned to $v_{k+1}(_s)$ through the value iteration until the algorithm ends. We can then obtain the optimal policy based on the state value *v*.

By using Formula (10.11), we are able to obtain the local optimal state value after iteration of all states and subsequently obtain the local optimal policy based on the local optimal state value. This iterative process continues until the local optimal state value converges to the global optimal state value.

10.2.2 Monte Carlo Method

The Monte Carlo method is suitable for model-free tasks because it needs only to collect the experience episode from the environment rather than requiring complete knowledge of the environment. With this method, we use the calculated data of the experience episode in order to obtain the optimal policy. Specifically, the Monte Carlo method estimates the state value function based on the experience episode mean, which refers to the cumulative discount return value *G* at the state *s* in a single experience episode. Its value function is calculated according to the following formula:

$$v(s_t) \leftarrow v(s_t) + a[G_t - v(s_t)] \tag{10.12}$$

The Monte Carlo method is notable for the following four characteristics:

(1) It can learn the experience episode directly from the environment, that is, the sampling process.
(2) It does not need to know the state transition probability *P* of the MDP in advance, making this method suitable for model-free tasks.
(3) It uses a complete experience episode for learning and is an offline learning method.
(4) It uses a simpler process to solve model-free reinforcement learning tasks based on the assumption that the expected state value is equal to the average reward of multiple rounds of sampling.

10.2.3 Temporal Difference Method

The temporal difference method is mainly based on the difference data of time series and includes an on-policy, represented by the Sarsa algorithm, and an off-policy, represented by the Q-Learning algorithm.

1. **Sarsa algorithm**
   The Sarsa algorithm is used to estimate the action value function $q(s,a)$. Specifically, it estimates the action value function $q_\pi(s,a)$ for all possible actions in any state s in policy π. The expression for this function is as follows:

$$q(s_t, a_t) \leftarrow q(s_t, a_t) + a\underbrace{[\underbrace{r_{t+1} + \gamma q(s_{t+1}, a_{t+1})}_{\text{Target}} - q(s_t, a_t)]}_{\text{Error}} \tag{10.13}$$

where $\theta = r_{t+1} + \gamma q(s_{t+1}, a_{t+1})$ is the temporal difference target, and.

θ-$q(s_t, a_t)$ is the temporal difference error.

The name Sarsa is derived from the five variables needed for each update of the action value function: current state *s*, current action *a*, environment feedback reward *r*, state *s'* of the next time step, and action *a'* of the next time step. The algorithm flow is as follows:

Algorithm 10.1 Sarsa algorithm

Input: Random state *s*

Output: Action value function $q(s,a)$

(1) **Initialization:**

(2) Set $q(s,a)$ to any value for any state *s*

(3) **Repeat** the experience episode

(4) Initialize the state *s*

(5) Perform the action *a* in state *s* according to the action value *q*

(6) **Repeat** the time step *t* in the experience episode

(7) Perform the action *a* in state *s* according to the action value *q*

(8) Update the action value function: $q(s,a) \leftarrow q(s,a) + \alpha\, [r + \gamma q\, (s', a') - q(s,a)]$

(9) Record the new state and the new action: $s \leftarrow s'$, $a \leftarrow a'$

(10) **Proceed until** the state *s* ends

(11) **Output** the action value function $q(s,a)$.

The Sarsa algorithm starts by initializing the action value function *q* with a random value and then samples the experience episodes iteratively. In collecting an experience episode, the agent first selects and performs the action *a* in state *s* according to the greedy policy. It then learns the environment and updates the action value function $q(s,a)$ until the algorithm ends.

2. Q-Learning algorithm

In updating the action value function $q(s,a)$, the Q-Learning algorithm adopts a policy that differs from one used for selecting an action. The action value function $q(s,a)$ is updated as follows:

$$q(s_t, a_t) \leftarrow q(s_t, a_t) + a\underbrace{[\underbrace{r_{t+1} + \gamma \max_a q(s_{t+1}, a_t)}_{\text{Target}} - q(s_t, a_t)]}_{\text{Error}} \tag{10.14}$$

When the Q-Learning algorithm updates the Q value, the temporal difference target is the maximum value $\max_a q(s_{t+1}, a_t$ of the action value function, which is independent of the policy used in the current selected action. This approach is different from the Sarsa algorithm and means that the action value Q is usually optimal.

In many aspects, the Q-Learning algorithm is similar to Sarsa. However, once the Q-Learning algorithm enters the cycle of repeating the experience episode and initializes the state s, it directly enters the iteration phase of the experience episode and then selects the action a' in the state s' according to the greedy policy. The algorithm flow is shown in Algorithm 10.2.

Algorithm 10.2 Q-Learning algorithm

Input: Random state s

Output: Action value function $q(s,a)$

(1) **Initialization:**

(2) Set $q(s,a)$ to any value for any state s

(3) **Repeat** the experience episode

(4) Initialize the state s

(5) **Repeat** the time step t in the experience episode

(6) Perform the action a in state s according to the action value q

(7) Perform action a to obtain the reward and the state s' of the next time step

(8) Update the action value function: $q(s,a) \leftarrow q(s,a) + a[r + \gamma \max_a q(s',a) - q(s,a)]$·

(9) Record the new state: $s \leftarrow s'$

(10) **Proceed until** the time step T_s ends

(11) **Output** the action value function $q(s,a)$.

Although the dynamic programming method can adequately represent the Bellman equation, most reinforcement learning tasks in real-world applications are model-free tasks, that is, only limited environmental knowledge is provided. The Monte Carlo method based on sampling can solve the problem of reinforcement learning tasks to some extent, and both the Monte Carlo and temporal difference methods are similar in that they estimate the current value function based on the sampled data. The difference between them lies in how they calculate the current

value function: The former performs calculation only after the sampling is completed, whereas the latter uses the boosting algorithm in the dynamic programming method for calculation.

10.3 Deep Reinforcement Learning Algorithm

In order to improve the performance of reinforcement learning agents in practical tasks, we need to employ all of the advantages deep learning offers, especially its strong representation ability. As mentioned earlier, deep reinforcement learning is the combination of both deep learning and reinforcement learning, and this combination lends itself to ensuring the agent has strong perception and decision-making abilities.

10.3.1 DQN Algorithm

The deep Q-Learning network (DQN) algorithm, proposed by Minh et al. at Google DeepMind in 2013, was the first deep reinforcement learning algorithm to achieve human-level performance playing a number of classic Atari 2600 games. For some games, the DQN algorithm even exceeded human performance. Figure 10.2 compares the effects of the DQN algorithm, which spawned a surge in research into deep reinforcement learning.

The DQN algorithm introduces three key technologies: objective function, target network, and experience replay. Through these technologies, the DQN algorithm can implicitly learn the optimal policy based on the action value function.

1. **Objective function**
   The DQN algorithm introduces a DNN, which is used to approximate the action value function $q(s,a)$ in high-dimensional and continuous state. However, before using this approximation, we need to define the optimization objective of the network (that is, the objective function of optimization, also called the loss function). We also need to update the weight parameter of the model by using other parameter learning methods.
   In order to obtain an objective function that the DNN model can learn, the DQN algorithm uses the Q-Learning algorithm to construct the loss function that the network model can optimize. Using formula (10.14) as the basis, the loss function of the DQN algorithm is as follows:

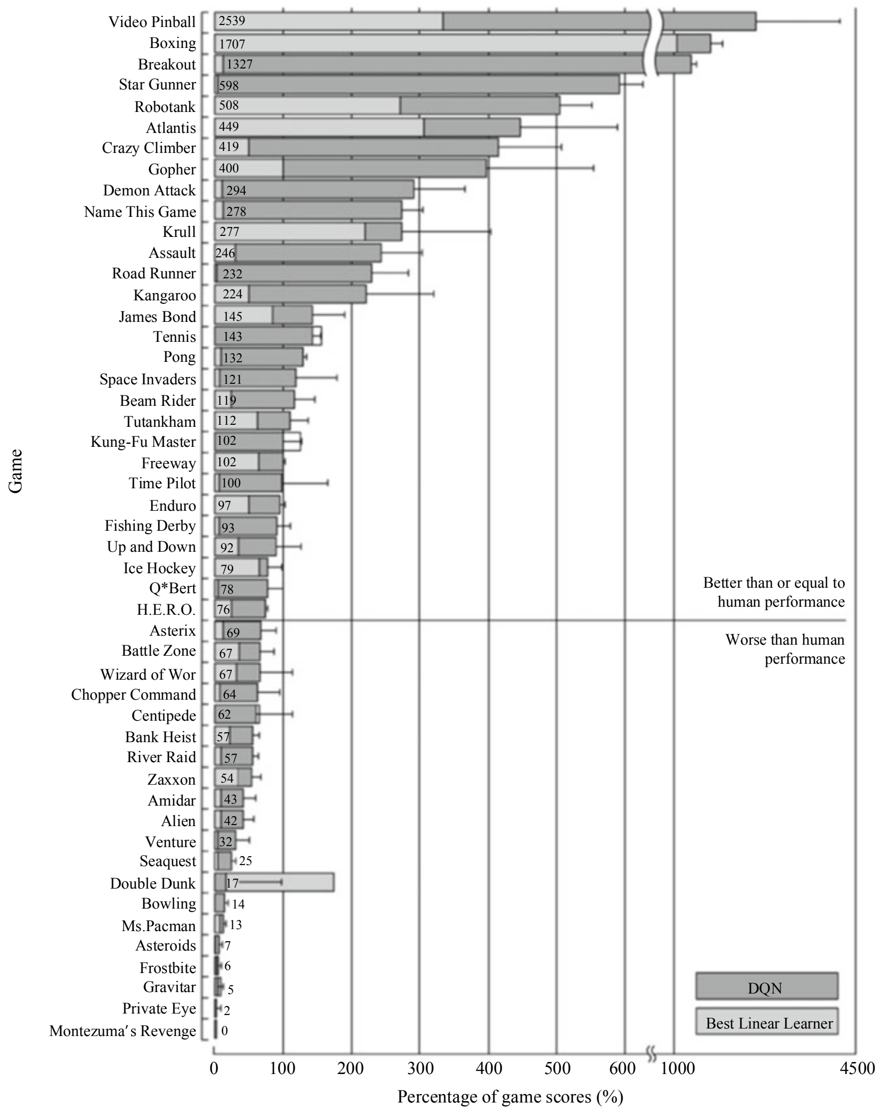


Fig. 10.2 Comparison of the effects of the DQN algorithm [1]

$$L(\boldsymbol{\theta}) = E\left[(\text{Target}\,\boldsymbol{Q} - q(s, a, \boldsymbol{\theta}))^2\right] \tag{10.15}$$

where $\boldsymbol{\theta}$ is the weight parameter of the DNN model; and target $\boldsymbol{Q}$ is the target action value, where

$$\text{Target } \boldsymbol{Q} = r + \gamma \max_{a'} q(s', a', \boldsymbol{\theta}) \tag{10.16}$$

Because the loss function in the DQN algorithm is determined based on the Q-Learning algorithm update, the effect of Formula (10.14) is the same as that of Formula (10.15); specifically, both formulas approximate the target value based on the current prediction value.
After obtaining the loss function of the DQN algorithm, we can solve the weight parameter $\boldsymbol{\theta}$ of the DNN model's function $L(\boldsymbol{\theta})$ by using the gradient descent algorithm in deep learning.

2. **Target network**
As we can see from Formula (10.15), the prediction value and target value use the same parameter model in the Q-Learning algorithm. This means that the prediction value increases with the target value, increasing the probability of model oscillation and divergence.
In order to solve this problem, the DQN algorithm uses the historical network parameter $\boldsymbol{\theta}^-$ to evaluate the value of the next time step of an experience sample. It updates the historical network parameter $\boldsymbol{\theta}^-$ only in the offline multi-step time interval to provide a stable training target for the network to be evaluated and allows sufficient training time to mitigate any estimation errors.
Furthermore, the DQN algorithm uses two CNNs for learning: a prediction network and a target network.

   (1) Prediction network $q(s, a, \boldsymbol{\theta}_i)$: used to evaluate the value function of the current state-action pair.
   (2) Target network $q(s, a, \boldsymbol{\theta}_i^-)$: used to generate the target action value (target Q). The algorithm updates parameter $\boldsymbol{\theta}$ belonging to the prediction network according to the loss function in Formula (10.15). Then, after N rounds of iterations, the algorithm copies this parameter to the parameter $\boldsymbol{\theta}^-$ in the target network

By introducing the target network, the DQN algorithm ensures that the target value remains unchanged for a certain period of time. It also reduces the correlation between the prediction and target values to a certain extent and reduces the possibility of oscillation and divergence occurring in the loss value during training, thereby improving algorithm stability.

3. **Experience replay**
In deep learning tasks, each input sample data is independent of other samples, meaning that there is no direct relationship between them. For example, images A and B used as input in an image classification model are not directly related. Conversely, in reinforcement learning tasks, the samples are often strongly correlated and non-static. If we were to use correlated data directly to train the DNN, the loss value would fluctuate continuously during training, making it difficult to achieve model convergence.

To solve this problem, the DQN algorithm introduces the experience replay mechanism. This mechanism allows experience sample data—obtained through interaction between the agent and environment at each time step—to be stored in an experience pool. Subsequently, when network training needs to be performed, a mini-batch of data is selected from the experience pool for retraining. Such a mechanism offers a number of advantages: (1) facilitates the backing up of reward data; (2) helps remove the correlation and dependence between samples, because mini-batches of random samples can be obtained; (3) reduces the deviation of value function estimation after function approximation; (4) solves the problems of data correlation and non-static distribution; and (5) facilitates network model convergence.

Due to the use of this mechanism, the DQN algorithm stores a large amount of historical experience sample data, using the following quintuple format for storage:

$$\left(s, a, r, s', T\right) \tag{10.17}$$

This quintuple indicates that the agent performs action a in state s, enters the new state s', and obtains the corresponding reward r. Here, T is a Boolean value that indicates whether the new state s' is the end state.

After performing any time step in the environment, the agent stores the obtained experience data in the experience pool. Subsequently, the agent randomly selects a mini-batch of experience sample data from this pool after performing N steps. Based on the experience sample data, the DQN Formula (10.15) is executed to update the Q function.

The experience replay mechanism, despite its simplicity, effectively removes the correlation and dependence between samples. In so doing, it allows the DNN to obtain an accurate value function in reinforcement learning tasks.

4. **Algorithm flow**

The DQN algorithm uses a deep CNN with the weight parameter $\boldsymbol{\theta}$ as the network model of the action value function and simulates the action value function $q_\pi(s, a)$ by using the CNN model $q(s, a, \boldsymbol{\theta})$, that is,

$$q(s, a, \boldsymbol{\theta}) \approx q_\pi(s, a) \tag{10.18}$$

This means that we can define the object function, based on the mean square error, as the loss function of the deep CNN.

$$L_i(\boldsymbol{\theta_i}) = E[(r + \gamma \max_{a'} q(s', a', \boldsymbol{\theta}_i) - \max_{a} q(s, a, \boldsymbol{\theta}_i)^2] \tag{10.19}$$

where a' is the action value of the next time step, and
s' is the state value of the next time step.

From Formula (10.19), we can see that the Q-Learning algorithm mainly uses the updated Q value as the target value for training. Conversely, in the DQN

algorithm, the target Q value is predicted by the target network, and the current Q value is predicted by the prediction network. Subsequently, the mean square error algorithm is used to calculate the temporal difference error in the Q-Learning algorithm.

Using Formula (10.19) as the basis, we calculate the gradient of the deep CNN model parameter $\boldsymbol{\theta}$ as follows:

$$\nabla_{\boldsymbol{\theta}_i} L_i(\boldsymbol{\theta}_i) = E[(r + \gamma \max_{a'} q(s', a', \boldsymbol{\theta}_i) - \max_{a} q(s, a, \boldsymbol{\theta}_i))\nabla_{\theta_i}(s, a, \boldsymbol{\theta}_i)] \tag{10.20}$$

To optimize the objective function, we implement the CNN model by using the mini-batch stochastic gradient descent algorithm. The CNN then calculates $\nabla_{\boldsymbol{\theta}_i}$ $(s, a, \boldsymbol{\theta}_i)$ in order to obtain the optimal action value (Q value).

The application scope of the DQN algorithm is limited, as it can deal with only discrete-control reinforcement learning tasks. In order to overcome these limitations, a number of DQN algorithm variants, such as Double DQN and Dueling DQN, have been proposed.

The specific flow of DQN algorithm version 2015 is described below.

Two CNNs are used: The prediction network $q(s, a, \boldsymbol{\theta}_i)$ is used to evaluate the current action function, and the target network $q(s, a, \boldsymbol{\theta}_i^-)$ is used to calculate the target value. The DQN algorithm updates the parameters of the target network according to the loss function and after C rounds of iterations assigns the relevant parameters of the prediction network to the target network.

Algorithm 10.3 DQN algorithm version 2015

Input: Prediction network and target network

Output: Target network

(1) **Initialize** the experience pool $\mathcal{D}$, which stores a maximum of $\boldsymbol{N}$ experience samples

(2) **Initialize** the prediction network with the weight parameter $\boldsymbol{\theta}$

(3) **Initialize** the target network with the weight parameter $\boldsymbol{\theta}^- = \boldsymbol{\theta}$

(4) **Repeat** the experience episodes 1 to M times

(5) Initialize the state s_1, and calculate the input sequence $\phi_1 = \phi(s_1)$

Repeat the time steps from 1 to T in the experience episode

(6) Select the random action a_t based on the probability ε

Select the action a_t based on the probability 1-ε according to $a_t = \max_a q^*(\phi(s_1), a, \boldsymbol{\theta})$

(7) Perform action a_t to obtain the reward r_t and the state image frame x_{t+1}

(8) Let $s_{t+1} = s_t, x_{t+1}$, and calculate the input sequence for the next time step:

$$\phi_{t+1} = \phi(s_{t+1}) \tag{1.21}$$

(9) Store the experience samples ($\phi_t, a_t, r_t, \phi_{t+1}$) in the experience pool $\mathcal{D}$

(10) Obtain a mini-batch of random sample data ($\phi_t, a_t, r_t, \phi_{t+1}$) from the experience pool $\mathcal{D}$

(11) Set y_i:

$$\mathrm{y_i} = \begin{cases} r_j \\ r_j + \gamma \max_{a'} q^-(\phi_{j+1}, a', \boldsymbol{\theta}^-) \end{cases} \tag{1.22}$$

(12) Update the network parameter $\boldsymbol{\theta}$ in the loss function $(y_i - q\ (\phi_j, a_j, \boldsymbol{\theta}))^2$ by using the gradient descent algorithm

(13) Re-assign $q^- = q$ every C steps.

10.3.2 DDPG Algorithm

The deep deterministic policy gradient (DDPG) [2] algorithm, proposed by Lillicrap et al. in 2015, widens the application scope of the DQN algorithm, allowing it to be used for reinforcement learning tasks with continuous action spaces rather than only those tasks with discrete actions. To better understand the DDPG algorithm, we must first explore its constituent parts.

1. **Policy gradient algorithm** [3]
The policy gradient (PG) algorithm explicitly expresses the optimal policy of each time step by using the policy gradient probability distribution function $\pi_{\boldsymbol{\theta}}(s_t \mid \boldsymbol{\theta}^{\pi})$. To obtain the optimal action value a_t^* of the current time step,

the agent samples the action at each time step t according to the probability distribution.

$$a_t^* \sim \pi_{\boldsymbol{\theta}}(s_t|\boldsymbol{\theta}^{\pi}) \tag{10.23}$$

The optimal action is generated through a random process, meaning that the policy distribution function $\pi_{\boldsymbol{\theta}}$ $(s_t \mid \boldsymbol{\theta}^{\pi})$ learned by the policy gradient algorithm is a stochastic policy.

2. **Deterministic policy gradient algorithm** [4]
One of the major weaknesses in the policy gradient algorithm is its inefficiency in terms of policy evaluation. Once the algorithm learns the stochastic policy, the agent can obtain a specific action value only after sampling actions at each time step according to the optimal policy probability distribution function. This consumes a great deal of computing resources, because the agent performs sampling in high-dimensional action space at each time step.
To address this inefficiency, David Silver—in 2014—explored the possibility of using the deterministic policy gradient (DPG) algorithm to quickly and efficiently solve reinforcement learning tasks with continuous actions. For the action of each time step t, the action value is determined by the function μ.

$$a_t^* \sim \mu_{\boldsymbol{\theta}}(s_t|\boldsymbol{\theta}^{\mu}) \tag{10.24}$$

where μ is the optimal action policy, which is a stochastic policy obtained without sampling.

3. **Deep deterministic policy gradient algorithm**
In 2016, Lillicrap et al. pointed out that the DDPG algorithm fuses the DNN with the DPG algorithm and uses the actor-critic algorithm as the basic architecture of the algorithm. The DDPG algorithm makes the following improvements over the DPG algorithm:

    (1) Uses the DNN as a function approximation
    The DNN is used as an approximation of the policy function $\mu(s; \boldsymbol{\theta}^{\mu})$ and of the action value function $q(s, a; \boldsymbol{\theta}^{q})$. To train the parameters in these two neural network models, the stochastic gradient descent algorithm is used. With the accuracy, efficiency, and convergence of nonlinear approximation policy functions, deep reinforcement learning can deal with deterministic policy problems.
    (2) Introduces an experience replay mechanism
    When actors interact with the environment, the resulting state transition sample data is chronologically correlated. With the experience replay mechanism of the DQN algorithm, the correlation and dependence between samples are removed. Furthermore, the deviation of value function estimation is reduced after the approximation of function. This effectively solves the problem of independently identically distribution (i. i. d) and allows the algorithm to converge more easily.

(3) Uses a dual-network architecture

For both the policy function and value function, the dual-DNN architecture is used, which includes the policy target network $\mu'(s;\boldsymbol{\theta}^{\mu'})$, policy online network $\mu(s;\boldsymbol{\theta}^{\mu})$, value online network $q(s;\boldsymbol{\theta}^{q})$, and value target network $q'(s;\boldsymbol{\theta}^{q'})$. Such an architecture speeds up the algorithm's convergence and achieves a stable learning process.

10.3.3 A3C Algorithm

Based on the idea of asynchronous reinforcement learning, Minh et al. proposed a lightweight deep reinforcement learning framework called asynchronous advantage actor-critic (A3C) [5]. This framework uses the asynchronous gradient descent algorithm to optimize the deep network model and employs several reinforcement learning algorithms in order to perform fast, CPU-based policy learning during deep reinforcement learning. The A3C algorithm is a combination of the advantage actor-critic algorithm and the asynchronous algorithm, both of which are described as follows.

1. **Advantage actor-critic algorithm**
   The actor-critic algorithm combines the advantages of two algorithms: value-based reinforcement learning, which is used as a critic, and policy-based reinforcement learning, which is used as an actor. When the critic network is updated, the concept of advantage function is introduced to evaluate the output action of the network model. In so doing, deviation in the evaluation of the policy gradient is reduced.
   The A3C algorithm combines the advantage function and the actor-critic algorithm and uses two network models: one to approximate the value function $v(s)$, used to judge the quality of a state; the other to approximate the policy function $\pi(s)$, used to estimate the probability of a set of output actions.

(1) Value-based learning—critic
In reinforcement learning based on value function approximation, the DNN can be used as the approximation function of the value function, where $\boldsymbol{w}$ is the weight parameter of the network model.

$$q(s, a) \approx q(s, a; w) \tag{10.25}$$

The loss function of the DQN algorithm is as follows:

$$L(w_i) = E[(\text{Target}Q - q(s, a; w_i))^2] \tag{10.26}$$

where target Q is the target action value:

$$\text{Target } \boldsymbol{Q} = r + \gamma \max_{a'} \boldsymbol{Q}(s', a'; w_i^-) \tag{10.27}$$

The loss function in Formula (10.25) is based on the single-step Q-Learning algorithm. This means that only the state of the next time step is considered during calculation of the target action value, having an adverse effect directly or indirectly. Specifically, it only directly affects the value of the state-action pair that produces the reward r and can only indirectly affect other state-action pairs based on the action value function. The result of this is a decrease in the algorithm's learning rate.
In order to quickly propagate the reward, we can use the multi-step Q-Learning algorithm, where "multi-step" refers to the states of the subsequent n steps.

$$\text{Target } \boldsymbol{Q} = r_t + \gamma r_{t+1} + \ldots + \gamma^{n-1} r_{t+n-1} + \gamma^n \max_a \boldsymbol{Q}(s_{t+n}, a)$$

The main advantages of this approach are that the previous n state-action pairs can be directly affected by the reward r, the historical experience can be better simulated, and the learning efficiency of the algorithm can be significantly improved.

(2) Policy-based learning—actor
In policy-based reinforcement learning, the DNN is used as the approximation function of the policy function, where $\boldsymbol{\theta}$ is the weight parameter of the policy network model.

$$\pi(s, a) \approx \pi(a|s; \theta) \tag{10.29}$$

The A3C algorithm uses policy iteration to update the weight parameter $\boldsymbol{\theta}$ in the network. Because the goal of the policy function is to maximize the reward, we can calculate the expected reward by using the gradient ascent algorithm. The formula for updating the policy gradient is as follows:

$$\nabla_\theta \mathrm{E}[r_t] = \nabla_\theta \log\pi(a_t|s_t; \boldsymbol{\theta}) r_t \tag{10.30}$$

where $\pi(a_t \mid s_t; \boldsymbol{\theta})$ is the probability of selecting action a_t in state s_t; and $\nabla_{\boldsymbol{\theta}} \log\pi(a_t \mid s_t; \boldsymbol{\theta}) r_t$ means that the logarithm of the probability is multiplied by the reward r_t of the action, and the weight parameter $\boldsymbol{\theta}$ is updated by using the gradient ascent algorithm.
Formula (10.28) indicates that an action with a higher reward expectation is more likely to be selected. However, assuming that each action has a positive reward, the probability of outputting each action will increase continuously with the gradient ascent algorithm, leading to a significant reduction in the learning rate while increasing the gradient variance. In order to reduce the gradient variance, we can standardize Formula (10.30) as follows:

$$\nabla_\theta \log\pi(a_t|s_t; \theta)(r_t - b_t(s_t)) \tag{10.31}$$

where $b_t(s_t)$ is a baseline function, which is set to the estimated reward r_t. The gradient is calculated to update the parameter $\boldsymbol{\theta}$. When the total reward exceeds the baseline, the probability of the action increases; conversely, the probability decreases when the reward falls below the baseline. In both cases, the gradient variance will be reduced.

The estimated variance can be reduced and kept unbiased by deducting the baseline function $b_t(s_t)$ from the reward r_t in order to learn the policy function.

(3) Advantage function.

The advantage function is modified based on the loss function of the actor-critic algorithm, allowing it to estimate the action value more accurately according to the reward.

During a policy gradient update, the agent learns which actions are good and which ones are bad based on the discounted reward r_t used in the update rule. Subsequently, the network is updated to determine the quality of the action. The function is the advantage function $A(s_t, a_t)$:

$$A(s_t, a_t) = q(s_t, a_t) - v(s_t) \tag{10.32}$$

Referring back to Formula (10.31), we can regard the discounted reward r_t as an estimate of the action value function $q(s_t, a_t)$ and regard the baseline function $b_t(s_t)$ as an estimate of the state value function (s_t). This means that we can replace $r_t - b_t(s_t)$ with an action advantage function, as follows:

$$r_t \approx q^{\pi}(s_t, a_t) \tag{10.33}$$

$$b_t(s_t) \approx v^{\pi}(s_t) \tag{10.34}$$

From Formula (10.32), we can use $q(s_t, a_t) - v(s_t)$ to evaluate the value of the current action value function relative to the mean value. This is because the state value function $v(s_t)$ is the expected action probability of all the action value functions in the state of the time step t, and the action value function $q(s_t, a_t)$ is the value corresponding to a single action.

Although the A3C algorithm does not directly determine the action value Q, it uses the discounted cumulative reward R as the estimate of the action value Q. As a result, we are able to obtain the advantage function, as follows:

$$A(s_t, a_t) = R(s_t, a_t) - v(s_t) \tag{10.35}$$

2. **Asynchronous algorithm**

The DQN algorithm uses an agent, represented by a single DNN, to interact with the environment, whereas the A3C algorithm uses multiple agents to interact with the environment and thereby achieve greater learning efficiency. As shown in

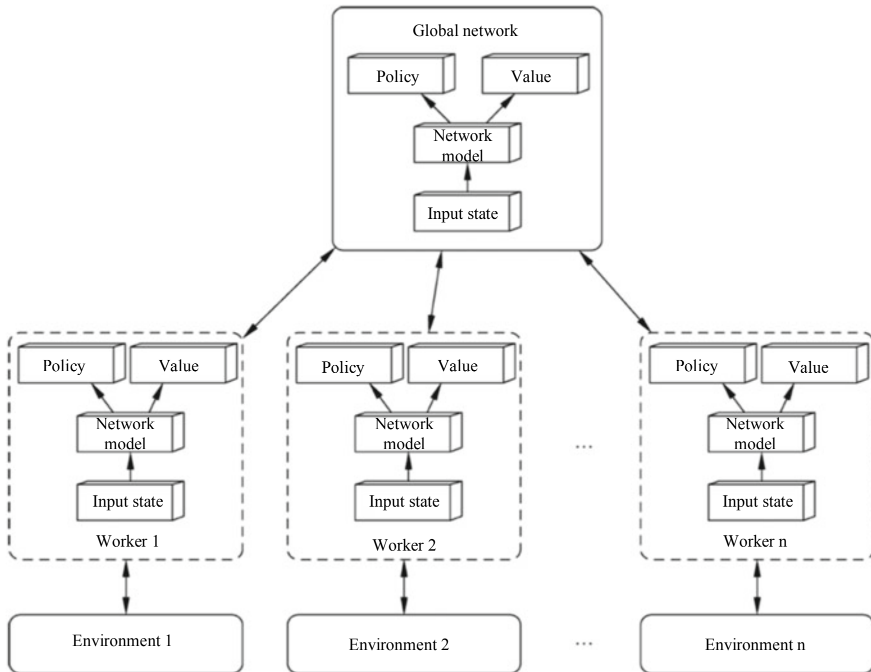


Fig. 10.3 A3C asynchronous architecture

Fig. 10.3, the main components of the A3C asynchronous architecture are environments, workers, and a global network, where each worker functions as an agent to interact with an independent environment and uses its own DNN model. In this architecture, different workers simultaneously interact with the environment, and executed policies and learned experiences differ among the workers. As a result, this multi-agent asynchronous approach offers faster and more effective operation along with greater diversity than a single worker approach.

The flow of the A3C algorithm is asynchronous, as shown in Fig. 10.4. From the figure, we can see that each worker replicates the global network as a parameter of its own DNN model (1). Then, each agent uses multiple CPU threads to allocate tasks, and different workers use the greedy policy with different parameters in order to ensure that they obtain different experiences (2). Next, each worker calculates its own value and policy loss (3). Based on the calculation results, each worker then calculates the gradient by using the loss function (4). Finally, each worker updates the parameters of the global network, that is, each thread updates the learned parameters to the global network (5). This flow is repeated until the ideal network parameters are learned.

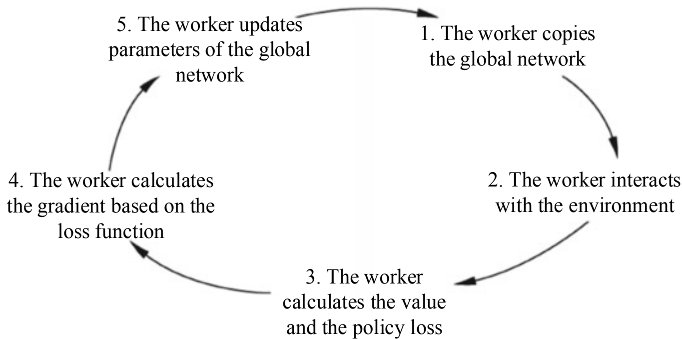


Fig. 10.4 Flow of the asynchronous algorithm

10.4 Latest Applications

10.4.1 Recommendation System

In recent years, Internet companies such as Facebook, Alibaba, JD.com [6], and Tencent have begun to explore how they can use and implement deep reinforcement learning in their recommendation systems. For example, Zheng et al. [7] used deep reinforcement learning to address the recommendation variability problem in the news field, and Chen et al. [8] from Alibaba used model-based deep learning in recommendation systems, the latter of which we discuss below.

Those who implement recommendation systems typically do so using a loss function to evaluate a model that can minimize the difference between the model's prediction result and the user's immediate response. In other words, the typical recommendation system model does not explicitly consider users' long-term interests, which may vary over time depending on what they see. Furthermore, such changes may have a significant influence on the behavior of the recommenders.

Chen pointed out that, in the recommendation system, solving high sample complexity of a model-free task is performed more reliably by using model-based deep reinforcement learning. As shown in Fig. 10.5, the recommendation system framework uses a unified minimax framework to learn a user behavior model and related reward functions and subsequently uses this model to learn the policy of deep reinforcement learning.

Specifically, this framework uses the generative adversarial learning network to simulate the dynamic behavior of users and learn its reward function. User behavior and rewards can be evaluated using the minimax algorithm. As a result, we are able to obtain a more accurate user model, as well as a method for learning a reward function that is consistent with the user model. Such a reward function can also strengthen the learning task compared with an artificially designed one. Furthermore, this user model enables researchers to execute model-based reinforcement learning tasks for new users, thereby achieving better recommendation results.

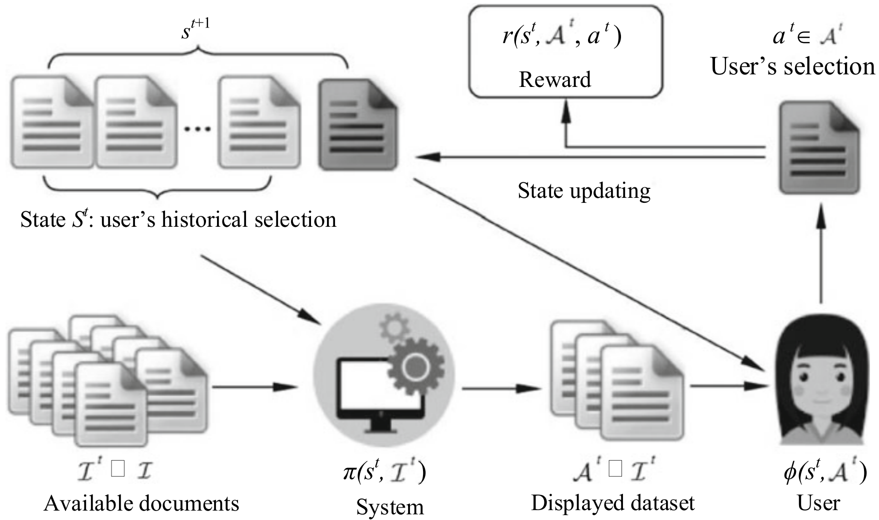


Fig. 10.5 Interaction between a user and a recommendation system

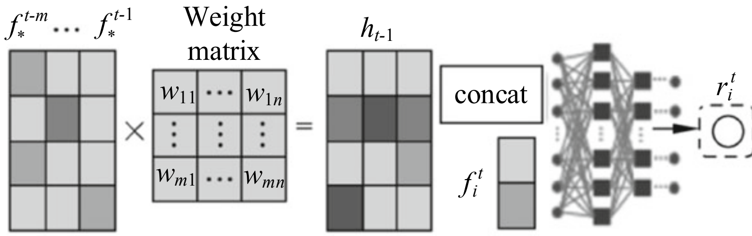
To obtain a combined recommendation policy, researchers developed a cascading DQN algorithm that not only identifies the optimal subset of objects from many potential candidates, by using the cascading design of the action function, but also significantly reduces the computational difficulty because the time complexity of the cascading function is linearly related to the number of candidates. Figure 10.6 compares such effects of the DQN algorithm.

In terms of held-out likelihood and click prediction, experimental results show that the GAN model is a better fit for user behavior. Based on the user model and the reward learned, researchers found that evaluating the recommendation policy provides users with better long-term cumulative rewards. Furthermore, in the case of model mismatch, the model-based policy can quickly adapt to the dynamic changes of user interests.

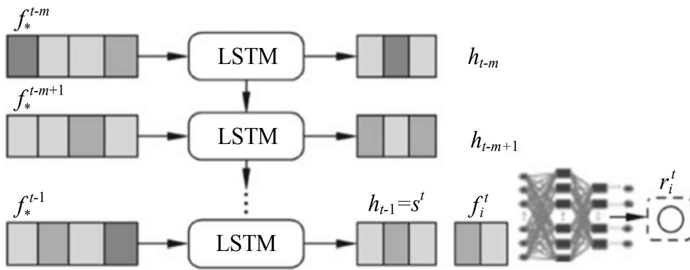
10.4.2 Gambling Game

In 2018, Wu et al. [9] from Tencent released the Honor of Kings AI based on reinforcement learning for group gaming. The algorithm used five independent Honor of Kings agents, pitted against five human players, to play 250 games, 48% of which the algorithm won.

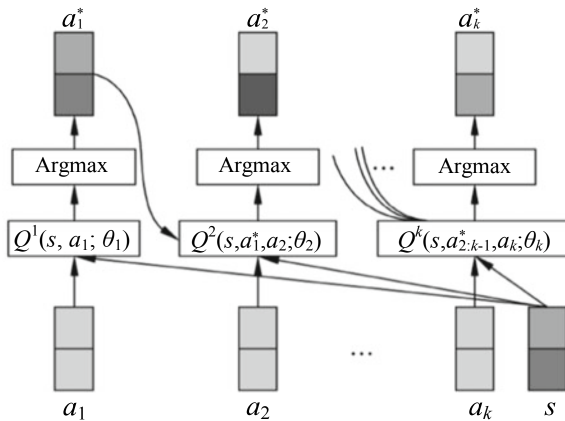
StarCraft is another game that also uses reinforcement learning. Because it involves a large observation space, huge action space, partial observations, simultaneous multi-player moves, and long-term local decision-making, researchers have found this game to be of significant interest. For example, Sun et al. [10] in Tencent



(a) Position weights



(b) LSTM parameterization model architecture



(c) Cascading Q network

Fig. 10.6 Comparison of the effects of the DQN algorithm

AI Lab developed an AI agent capable of defeating the built-in AI agents in the full game of StarCraft II, including the cheating agents at levels 8, 9, and 10, by using reinforcement deep learning. Of note is that the built-in AI at level 10 is estimated to be equivalent to the top 30–50% of human players according to the Battle.net League

ranking system. DeepMind has also explored StarCraft, using the meta-reinforcement learning (Meta-RL) [11] algorithm to play the game and achieving excellent results.

Against this backdrop, we can see that deep reinforcement learning has become increasingly popular in gaming over the past few years. Taking AlphaGo Zero as an example, we proceed to describe the use of deep reinforcement learning in the board game Go.

AlphaGo [12], the predecessor to AlphaGo Zero, learns how to play by observing human play. AlphaGo Zero [13] skips this step. Instead, it learns by playing itself, starting from scratch. The AlphaGo Zero algorithm begins with a DNN that, except for the game rules, lacks any understanding of how to play Go. It proceeds to learn the game by combining the DNN with the search algorithm to play against itself, during which the DNN continuously adjusts and upgrades its parameters in order to predict the probability of each move as well as the final winner.

As mentioned already, AlphaGo Zero has no background knowledge of Go, except the rules. It uses only one neural network, differing from AlphaGo, which includes two DNNs: One, called the policy network, evaluates the possibility of the next move based on numerous human games, and the other, called the value network, evaluates the current situation.

AlphaGo Zero uses the Monte Carlo tree search algorithm to generate games that it uses as training data for the DNN. A 19 × 19 Go board is used as the input in the DNN, while the output is based on the probability of the next move and the win rate (the difference between the two is the loss). Through ongoing execution of the Monte Carlo tree search algorithm, the probability of the move and the win rate will eventually stabilize, and the accuracy will increase as training continues. Next, we will describe the AlphaGo Zero algorithm.

1. **Reinforcement learning process**

Figure 10.7 shows the self-play process of the algorithm for the time steps s_1 to s_T. In each state s_t, the last network f_θ is used to execute the Monte Carlo tree search algorithm once in order to obtain the corresponding action a_θ. The action is selected according to the search probability calculated by using the Monte Carlo tree search

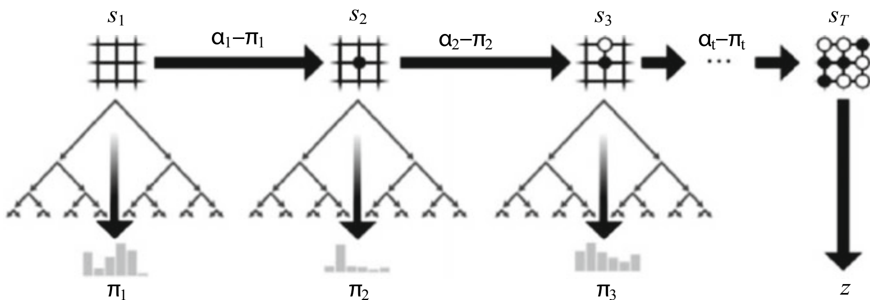


Fig. 10.7 Self-play process of AlphaGo Zero

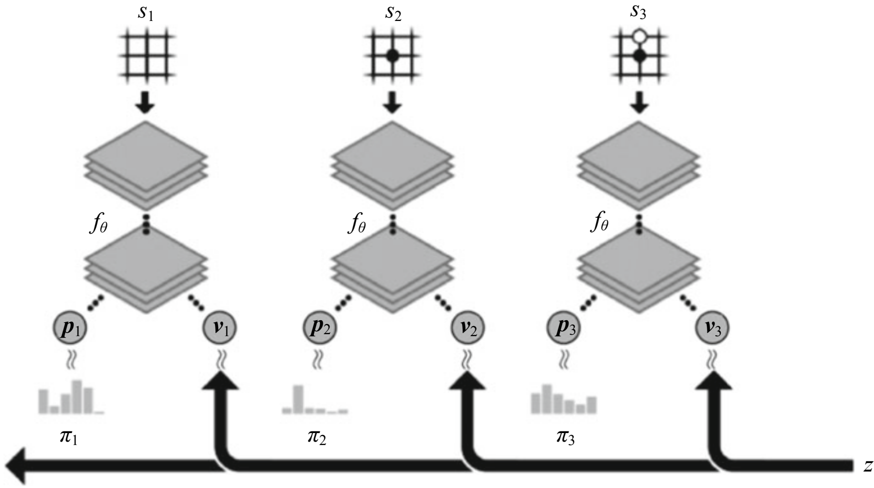


Fig. 10.8 Neural network training process of AlphaGo Zero

algorithm, that is, a_t to π_t. After reaching the end state s_T, the algorithm determines the winner z and the reward according to the rules of the game.

Figure 10.8 shows the neural network training process of AlphaGo Zero. The network uses the original Go board state s_t as the input and outputs two channels of data through multiple convolution operations. One of these channels is vector $\boldsymbol{p}_t$, which indicates the probability distribution of the moves in the Go game; the other channel is scalar v_t, which represents the win rate of the player in the current game s_t. The error between the predicted winner v_t and the actual winner z is calculated based on the similarity between the maximization policy vector $\boldsymbol{p}_t$ and the search probability π_t. The neural network model parameter $\boldsymbol{\theta}$ is updated automatically, and the new parameters will be applied to the next round of self-play.

2. **Monte Carlo tree search process**

As already mentioned, AlphaGo Zero uses the Monte Carlo tree search algorithm. Figure 10.9 shows the algorithm process, which is subsequently described.

(1) For each branch selected through simulation, the largest $Q + U$ is selected, where Q is the action value, and U is the upper confidence limit. U depends on the priority probability $\boldsymbol{p}$ stored on the branch and the number of access times N to the branch of the search tree.
(2) The leaf node is extended, the neural network $(\boldsymbol{p}, \boldsymbol{v}) = f_\theta$ is used to evaluate the state s, and the value of vector $\boldsymbol{p}$ is stored on the extended edge corresponding to the state s.
(3) The action value Q is updated based on the value v and reflects the mean value of the subtrees of all actions.

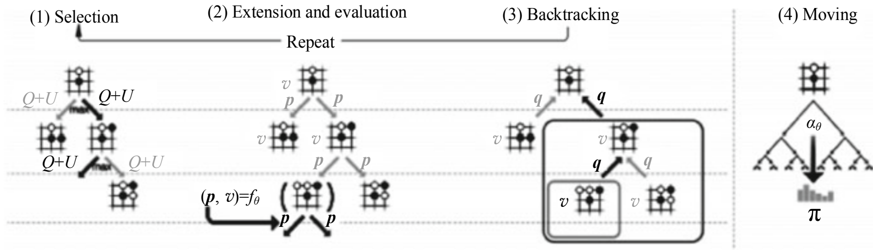


Fig. 10.9 Monte Carlo tree search process of AlphaGo Zero

(4) Once the search is completed, the search probability π is returned. The number of remaining searches is directly proportional to $N^{1/\tau}$, where N is the number of access times to each branch, and τ is the control parameter.

10.5 Implementing DQN-Based Game Using MindSpore

The interfaces and processes of MindSpore may constantly change due to iterative development. For all runnable code, see the code in corresponding chapters at https://mindspore.cn/resource. You can scan the QR code on the right to access relevant resources.

References

1. V. Mnih, K. Kavukcuoglu, D. Silver et al., Playing atari with deep reinforcement learning, (2013). [2019–11–10] https://arxiv.org/pdf/1312.5602.pdf
2. T.P.Lillicrap, J.J. Hunt, A. Pritzel et al., Continuous control with deep reinforcement learning, (2015). [2019–11–10] https://arxiv.org/pdf/1509.02971.pdf
3. R.S. Sutton, D.A. McAllester, S.P.Singh et al., Policy gradient methods for reinforcement learning with function approximation. in *Advances in Neural Information Processing Systems* (2000), pp. 1057–1063
4. D. Silver, G. Lever, N. Heess et al., Deterministic policy gradient algorithms, (2014). [2019–11–10] http://xueshu.baidu.com/usercenter/paper/show?paperid=43a8642b81092513eb6bad1f3f5231e2&site=xueshu_se
5. V. Mnih, A.P. Badia, M. Mirza et al., Asynchronous methods for deep reinforcement learning. in *International Conference on Machine Learning* (2016), pp. 1928–1937

6. X. Zhao, L. Zhang, Z. Ding et al., Deep reinforcement learning for list-wise recommendations, (2017). [2019–11–10] https://arxiv.org/pdf/1801.00209.pdf
7. G. Zheng, F. Zhang, Z. Zheng et al., DRN: a deep reinforcement learning framework for news recommendation. in *Proceedings of the 2018 World Wide Web Conference. International World Wide Web Conferences Steering Committee*, (2018), pp. 167–176
8. X. Chen, S. Li, H. Li et al., Generative adversarial user model for reinforcement learning based recommendation system. in *International Conference on Machine Learning* (2019), pp. 1052–1061
9. B. Wu, Q. Fu, J. Liang et al., Hierarchical macro strategy model for MOBA game AI, (2018) [2019–11–10] https://arxiv.org/pdf/1812.07887.pdf
10. P. Sun, X. Sun, L. Han et al., TStarBots: defeating the cheating level builtin AI in starCraft II in the full game, (2018). [2019–11–10] https://arxiv.org/pdf/1809.07193.pdf
11. J.X. Wang, Z. Kurth-Nelson, D. Kumaran et al., Prefrontal cortex as a meta-reinforcement learning system. Nat. Neurosci. **21**(6), 860 (2018)
12. D. Silver, A. Huang, C.J. Maddison et al., Mastering the game of go with deep neural networks and tree search. Nature **529**(7587), 484 (2016)
13. D. Silver, J. Schrittwieser, K. Simonyan et al., Mastering the game of go without human knowledge. Nature **550**(7676), 354 (2017)