Hands on Multi-Agent Reinforcement Learning

January 4, 2024

Contents

1 Chapter 2: Deep Reinforcement Learning

Having learned the framework of RL, we now begin our journey into deep reinforcement learning, which is a field that combines reinforcement learning and deep learning. As you can learn in previous chapter, RL is field using MDP framework to model sequence decision problems and the goal is to find best policy, but in large state space or continuous state space, it can't get satisfying performance until researchers incorporate deep learning into the policy, the milestone event is the proposal of deep Q-learning which get an amazing performance on Atari games. and since then, deep reinforcement learning gets a rapid development and becomes the main method to solve various problems.

Deep learning is a subfield of machine learning that uses neural network and backpropagation algorithms to solve complex and large data problems, it's a process of transforming input data to output data through neural network, which includes feature extraction and model training and other estimation processes. It's a field of importance across many industries and technical domains. Applications range from computer vision to natural language processing to speech recognition.

As you can learn in the previous chapter, RL is a field that uses the MDP framework to model sequential decision problems and the goal is to find the best policy. So the difference between RL and DL including learning objective, learning paradigm and training process and task types. In this book, we focus on RL, and the role of neural network in it is to get the state value, so we can deal with large state space or continue state space, which means using neural network to adjust the value function, so in this chapter, we will focus on neural network. About how to use neural network in reinforcement learning will be discussed in specific algorithms in DRL including DQN and related variants.

In the first step, we will introduce cost functions and gradient descent, which are basic concepts that will serve as the foundation for the subsequent discussions.

We'll then explore the architecture of neural models, drawing inspiration from biological neural mechanisms. We'll understand how to construct a neural network by connecting groups of neurons and how the inner workings of these networks. To enhance our understanding, we'll learn a comprehensive example of planar data classification and gain hands-on experience implementing neural networks using Python.

With this foundation in place, we'll move on to more complicated operations within neural networks that will enable us to tackle more complex challenges. This includes integrating convolutional operations into neural networks to handle grid-based data such as images. In addition, we'll delve into recurrent constructions, allowing us to build recurrent neural networks for processing sequential data, such as audio clips and text sequences. Along with these discussions, we'll provide practical Python implementations for both types of neural networks, ensuring a comprehensive understanding of their basic components and core principles.

After learning the basic concepts about neural networks, we can integrate them with reinforcement learning, giving rise to Deep Reinforcement Learning. In this journey, we'll delve into the foundational structure known as Deep Q-learning (DQN). DQN utilizes a deep neural network as an approximator for the value function. We'll also explore value function approximation in detail to establish a solid foundation for understanding DQN. Subsequently, we will explore two variants of DQN—namely, double DQN and Dueling DQN. We'll discuss their advantages and how they address the drawbacks of the original DQN. Finally, we'll explore Rainbow, a comprehensive combination of six DQN variants. The remaining four variants will be introduced more briefly, including their main ideas and advantages.

This process will lead us to a deeper and more comprehensive understanding of deep reinforcement learning, laying a solid foundation for the subsequent content.

2.1 Cost Function and Gradient Descent

There are three broad approaches in machine learning, which are supervised learning, unsupervised learning, and reinforcement learning. Supervised learning is learning the general pattern between input and output, there is a correction answer in the data set called label, and the goal of supervised learning is to output the desired answer. Supervised learning includes regression and classification algorithms, regression algorithm is used to predict continuous values within a given range, while a classification algorithm, as its name suggests, is used to categorize data. For example, predicting the click rate based on certain characteristics is a regression problem, while determining whether an email is spam is a classification problem. Unsupervised learning has no label in the dataset, it has to learn the internal pattern hidden in the dataset, it can be used in clustering and dimensionality reduction problems.

Cost function and gradient descent algorithm is used in these areas, it's necessary to handle them. So in this section, we will use a linear regression example to learn the core concept of cost function, which is to predict the click rate of online products. Then we'll learn its mathematical representation. The next question is how to minimize the value of the cost function. This leads us to the gradient descent algorithm, where we'll explore its basic concept and the mathematical techniques for solving it.

2.1.1 Cost Function

We will learn the concept of cost function by solving a prediction problem. The problem is from the online shopping website, As is commonly observed, higher prices for a product lead to fewer clicks, resulting in a lower click rate. Our objective is to construct a model that predicts a product's click rate based on its price. Given a dataset containing various commodities' prices and their corresponding click rates, we can visualize this information in the figure below, with 'x' representing price and 'y' representing click rate.

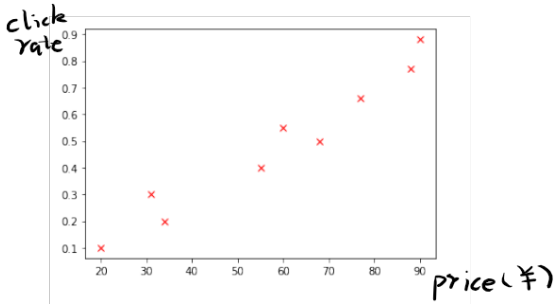

Figure 1: product's Price And Click Rate

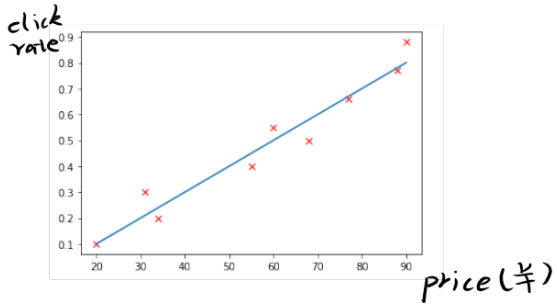

Figure 2: Product's Price And Click Rate

It's important to note that there are several other factors that influence click rates. In practical scenarios, datasets tend to be more intricate. For the sake of clarity, we present a simplified example to facilitate a better understanding of the underlying concept.

From Figure 1, we intuitively gather that perhaps a straight line could aptly represent the relationship between x and y as in Figure 2.Hence, we hypothesize that the data pattern follows a linear function, specifically $h_\theta(x) = \theta_1 * x + \theta_0$. In formal terms, theta represents the parameters of this function, which determines accuracy of the model. So the next step is to choose values for the parameters so that the straight line can best fit the data.

For a clearer understanding, consider one example (x_i, y_i), if the predicted value $y = f(x_i)$ closer to the actual value y_i, the model is considered more accurate. The gap between the predicted value and the true value is referred to as the error, and we need to measure the error over the entire data set. Therefore, we sum the errors of each data point and square the sum. However, merely summing these errors may result in an inflation of errors with more data points. To counter this, we compute the average error, and for later calculate conveniency, we add divide by 2.

So in the price and click rate problem, we define the hypothesis as linear function and cost function as square error function as shown below. The role of the cost function is to quantify the overall discrepancy between our hypothesis and the true values across the entire dataset. The ability to measure this error is critical to evaluating the effectiveness of our hypothesis and subsequently refining it. While squared error functions are widely used, it's important to note that in real-world scenarios, the design of hypotheses and cost functions can be adapted to different structures based on specific conditions.

Hypothesis:

$$h_\theta(x) = \theta_1 * x + \theta_0 \tag{1}$$

Cost Function:

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right)^2 \tag{2}$$

Our next goal is to find appropriate parameters that minimize the cost function J, making it as small as possible. In the next section, we will discuss how to address this challenge.

2.1.2 Gradient Descent

To solve the problem of minimizing the cost function your initial instinct might be to employ differentiation and set the result to zero, a method often used to find minimum or maximum of functions. However, this approach isn't always feasible, especially when dealing with a large number of parameters. Thus, we require a more systematic and versatile approach to identify parameters that minimize the cost function. This led to the discovery of the gradient descent algorithm, a highly effective method. It can be applied to a wide range of functions across various algorithms, making it a cornerstone in the field of machine learning.

Let me first introduce the core idea and procedure of gradient descent, followed by presenting its mathematical formulation.

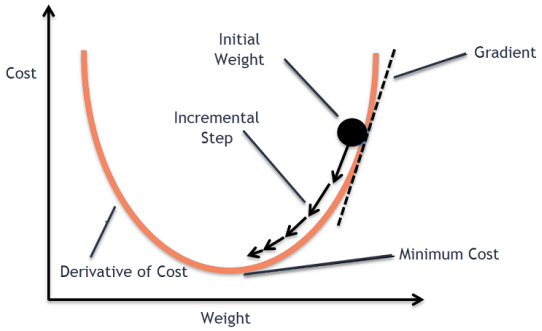

Figure 3: Gradient Descent

To get a more intuitive understanding of gradient descent, referring to the earlier example of commodity price and click rate, suppose the curve in Figure 3 is the cost function, we initialize the parameters in the hypothesis function, which we get a cost value shown as the starting point in Figure 3, then calculate the gradient of the cost function, which is the tangent of the starting point in the curve, and update the parameters along the gradient direction.

We start by setting initial values for parameters, which can be arbitrary. Subsequently, we continuously adjust these parameters with the hope of achieving improvement, iterating this process. We need to concern the direction of modification and determine when to stop the process.

The whole process is like standing at the summit of a mountain, aiming to descend as swiftly as possible. Given our limited view, we pivot 360 degrees, identifying the steepest descent direction. We repeat this procedure, advancing one step at a time, until we reach the bottom of the mountain. The gradient descent algorithm employs a similar strategy to pinpoint the location that minimizes the cost function. Mathematically, this is the direction of steepest descent. As for determining when to stop the process, you can set a threshold based on the gap between predicted and true values, or specify a set number of iterations.

The mathematical expression for the gradient descent algorithm is presented below:

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \qquad \begin{array}{c}(\text{simultaneously update} \\ j = 0 \text{ and } j = 1)\end{array} \tag{3}$$

For each parameter in each update step, calculated as the original value minus alpha times the derivative value. Here, alpha is the learning rate, which determines the size of each step in the descent. It typically ranges from 0 to 1. A larger alpha implies a more aggressive algorithm, taking larger steps and possibly overshooting the minimum. Conversely, a smaller alpha leads to smaller steps, reducing the chance of missing the minimum, but at the cost of slower convergence. Determining the appropriate learning rate requires several experimentation and experience. The derivative value indicates the steepest descent. Gradient Descent continues this process until the algorithm converges. Convergence means that the parameters have reached a local minimum of the cost function and no further adjustments are required. This implies that the error between predicted and true values is locally minimized.

It is important to note that the parameters must be updated simultaneously. This can be succinctly expressed by the formula:

$$\begin{aligned} \text{temp0} &\leftarrow \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) \\ \text{temp1} &\leftarrow \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) \\ \theta_0 &\leftarrow \text{temp0} \\ \theta_1 &\leftarrow \text{temp1} \end{aligned} \tag{4}$$

It's worth noting that some functions may exhibit complex shapes with multiple low points, as depicted in the Figure 4. Consequently, they may possess more than one potential minimum. Depending on the initial values chosen, the gradient descent algorithm will yield different results, known as local minima. This is a distinctive characteristic of the gradient descent algorithm. In certain scenarios, some algorithms may be need to ascertain the global minimum among all local minima.

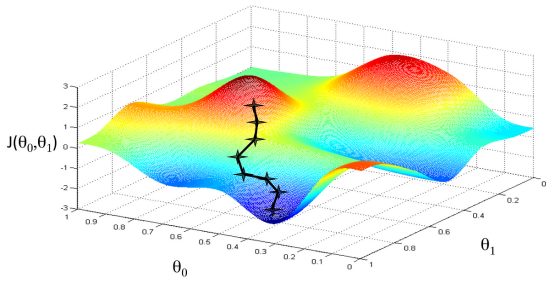

Figure 4: Multi Local Minimum Function

2.2 Neural Network

In the last section, we use a straight line to fit the relationship between price and click rate, but there may be multiple factors or features and high dimensions in practice scenarios, then the relationship will be more complex. So researchers proposed artificial neural network to solve these complex problems. This need to start with artificial intelligence.

The original concept of artificial intelligence was to create a machine with capabilities similar to the human brain. When we look at the human brain, even that of a ten-year-old child, we see a miracle. Not only does it process visual information and understand language and symbols, but it also formulates complex concepts and strategies. It also has a sense of logic and an understanding of time and space. This formidable organ is the driving force behind the progress of human civilization. Given this, if our goal is to construct an intelligent machine, why not draw inspiration from the human brain? Although a baby's brain lacks complete abilities, it is capable of performing tasks as complex as those of an adult brain through training. So perhaps we can design a machine that emulates the infant brain, coupled with a learning algorithm to teach it. Admittedly, our understanding of how the human brain works is also incomplet. The first hypothesis is that we would need to develop thousands of different algorithms to mimic different part of the brain. A second hypothesis suggesting that the brain functions as a vast neural network governed by a single learning algorithm. Neural rewiring experiments showed that a single piece of biological brain tissue could process visual, auditory, and tactile information. For example, both the auditory cortex and the somatosensory cortex showed the ability to learn to see. This suggested the existence of a universal learning algorithm capable of processing different types of data and autonomously acquiring a variety of skills. It was on this second hypothesis that the neural network was founded and eventually became widely accepted.

Currently, neural networks are widely used in various industries, ranging from image object recognition, recommendation systems, speech recognition, and more.

2.2.1 Neuron Model

Biological Neuron
Neurons are cells in the brain. Each neuron has dendrites, which serve to receive signals from other neurons. In structure and function, dendrites can be compared to the input wires of a system. They transmit information to the cell body, where it is processed. The processed information is then sent to other neurons through the axon, which is like an output wire. This communication between neurons occurs through pulses of electricity known as spikes, which consume little electricity. The axon connects to the dendrites of other neurons, allowing the transmission of spikes to neighboring cells. The message sent by the neuron can be either the processed result or the same information it originally received.

In simple terms, this is how the human body works. Our sensory organs, such as the eyes and ears, gather information from the external environment. This information is then transmitted to the brain through the interconnected network of neurons using electrical impulses. The brain then processes this information and sends the calculated results to the muscles and other organs.

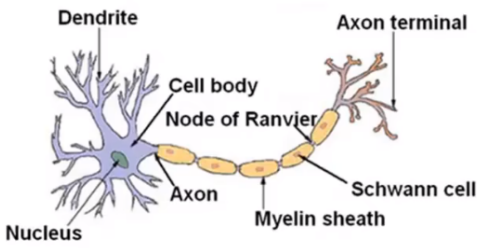

Figure 5: Biologic Neural

Neuron Model
We can think of a neuron as a computational unit. It receives information from an input structure, performs computations, and transmits the results to other nodes through an output structure.

The following diagram illustrates our neuron model. The circle represents a neuron that receives input through input wires, performs computations, and then sends output values through output wires. These output values always pass through an activation function called $f(x)$, which we will discuss later.

As in Figure 6, the parameters of the model are referred to as θ or weights w, and in most cases x denotes input values, x_0 means bias to get a shift or offset in the prediction, for example in the expression $y = mx + b$, b is the bias term. The result of the calculation between input variables and weight is sent to the activation function, then get the last result of the neural unit. It's worth noting that both θ and x are typically represented as vectors.

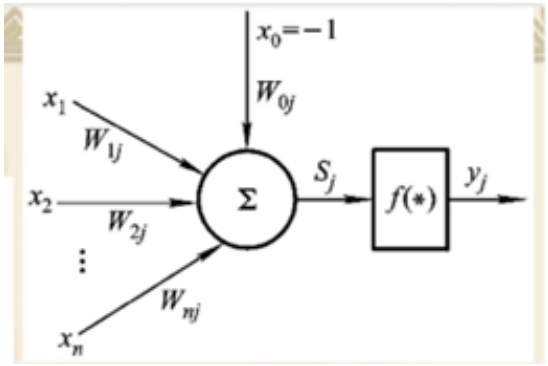

Figure 6: Neural Model

Activation Function

In the neuron model, the neuron is only capable of linear computation. To import non-linear factors, we include an activation function. This allows the model to accommodate non-linear functions and execute more complex tasks.

Example of activation function

Sigmoid Function: $f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$

Tanh Function: $f(x) = \tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$

Relu Function: $f(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$

Softmax Function: $y_i = \text{soft}\max(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}}$

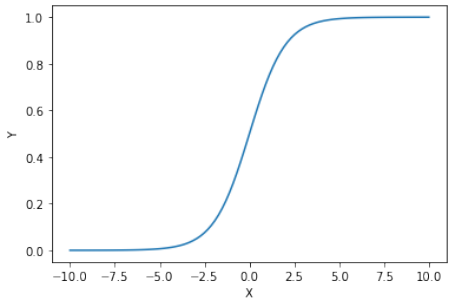

Figure 7: Sigmoid Function

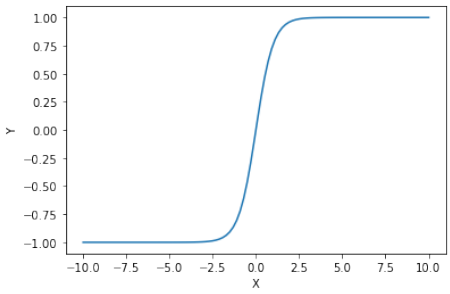

Figure 8: Tanh Function

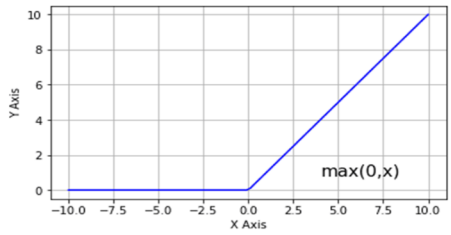

Figure 9: Relu Function

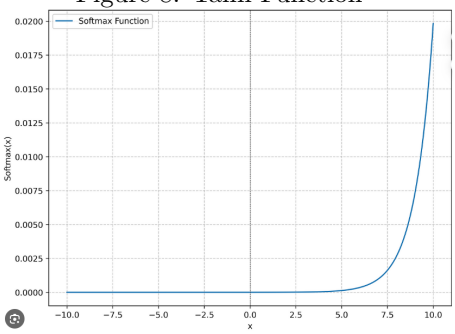

Figure 10: Softmax Function

2.2.2 Feedforward Neural Network

A number of neurals connected together form a neural network like in Figure 11.

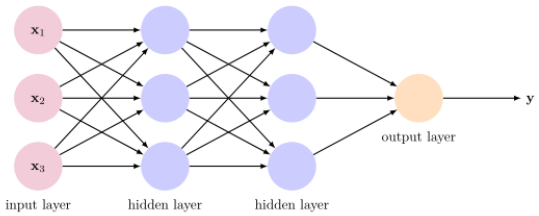

Figure 11: Neural Network

The first layer is called the input layer, and the last layer is called the output layer. The layers between the input and output layers are referred to as hidden layers.

Now we have network model, it's important to understand the mathematical underpinnings of neural networks. Let me explain the notations used in the following formula. As shown in Figure 12, $a_i^{(j)}$ denotes the activation of the neuron or unit i in layer j. This activation is both computed and output by the unit. For example, $a_1^{(2)}$ denotes the activation of the first unit in layer 2. The symbol θ represents the parameters of the neural network, and it's a matrix. $\theta^{(j)}$ is a matrix that governs the functional mapping from the $(j-1)$th layer to the jth layer. Note that the bias term is ignored in the diagram, bias is a constant term used as x_0, a_0, representing a constant factor in functions. Whether to include a bias depends on the specific problem in practice. In the following example, we will explore how to include the bias term.

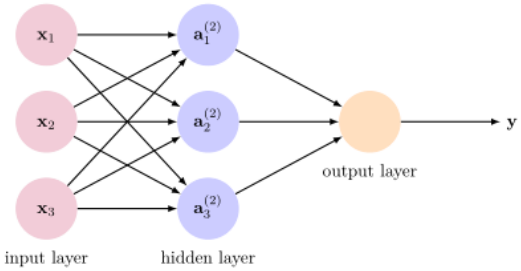

Figure 12: Neural Network

The hidden unit is activated by the function $g(x)$. Consequently, its value $a_i^{(j)}$ can be calculated by applying the activation function $g(x)$ to a linear

combination of the previous layer. In detail, the calculation is as follows:

$$\begin{aligned} a_1^{(2)} &= g\left(\Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3\right) \\ a_2^{(2)} &= g\left(\Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3\right) \\ a_3^{(2)} &= g\left(\Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3\right) \\ h_\Theta(x) = a_1^{(3)} &= g\left(\Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)}\right) \end{aligned} \tag{5}$$

Vectorized implementation

Vector equations give us a more efficient way to perform calculations.

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad z^{(2)} = \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{bmatrix} \tag{6}$$

$$\begin{aligned} z^{(2)} &= \Theta^{(1)}x \\ a^{(2)} &= g\left(z^{(2)}\right) \end{aligned} \tag{7}$$

$$\begin{aligned} z^{(3)} &= \Theta^{(2)}a^{(2)} \\ h_\Theta(x) &= a^{(3)} = g\left(z^{(3)}\right) \end{aligned} \tag{8}$$

It's obvious that the process called forward propagation is beacuase the flow of computation: it begins with the input, traverses through the hidden layers, and end on the output layer.

Need to pay attention, the bias term is ignored in the diagram, bias is a constant term used as x_0, a_0, representing a constant factor in functions. The decision to include a bias depends on the specific problem at hand. In the subsequent example, we will explore how to incorporate the bias term.

Through the computational steps outlined, we gain insight into how a neural network fits functions. By adjusting the parameters θ, we can generate different functions. These functions represent our hypotheses regarding a specific task. We will explore this further in the following examples.

The input variables are also called features. A neural network can be used to predict target variables or to classify data. This involves learning patterns from the input features, which are then applied to the prediction or classification task. In some cases, the task may be complex and the data may be large and complicated. How can a neural network extract distinctive and valuable patterns from such data? This is a critical question. Both feature design and network structure need to be considered. But even from our simplified structure above, it is clear that the architecture of a neural network plays a critical role

in answering this question. The neural network can reveal more complex and deeper patterns in the data due to the presence of hidden layers. If we were to remove these hidden layers, the structure would essentially be reduced to an activation function. The hidden layers have the ability to perform intricate and deep combinations of the output from the previous layers. By incorporating additional hidden layers, the neural network can go beyond the surface patterns in the data to reveal deeper and more substantial patterns. These are often elusive to human observers when dealing with large data sets.

2.2.3 Backpropagation

In the last section, we introduced a feedforward neural network model capable of producing an output in the final layer. However, a critical problem arises in determining the parameters for each layer. Randomly initializing them won't produce the desired output. Therefore, we must find a way to update these parameters with the goal of minimizing the divergence between predicted and true values across the entire data set. As discussed in the previous section, one solution is the gradient descent algorithm applicable to neural networks, called Backpropagation

The difference between neural networks and the linear regression discussed earlier is that the former has multiple layers, each containing numerous neurons. In a large network, there may be millions of parameters that need to be updated, and the backpropagation algorithm was designed to handle this challenge efficiently.

Backpropagation is an efficient application of the chain rule, allowing the gradient for each layer to be computed from the last layer, thus avoiding redundant computations in the intermediate layers.

Chain Rule

Formation 1:

$$\begin{aligned} y &= g(x) \\ z &= h(y) \end{aligned} \tag{9}$$

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} \tag{10}$$

Formation 2:

$$\begin{aligned} x &= g(s) \\ y &= h(s) \\ z &= k(x, y) \end{aligned} \tag{11}$$

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} \tag{12}$$

$$\frac{dz}{ds} = \frac{\partial z}{\partial x}\frac{dx}{ds} + \frac{\partial z}{\partial y}\frac{dy}{ds} \tag{13}$$

Our goal is to obtain the derivative of the cost function with respect to the training data. According to the chain rule, the derivative process can be decomposed as shown below:

$$\frac{dC}{dx} = \frac{dC}{da^L} \cdot \frac{da^L}{dz^L} \cdot \frac{dz^L}{da^{L-1}} \cdot \frac{da^{L-1}}{dz^{L-1}} \cdot \frac{dz^{L-1}}{da^{L-2}} \cdot \ldots \cdot \frac{da^1}{dz^1} \cdot \frac{dz^1}{dx} \tag{14}$$

The capital letter C represents the cost function, the first two terms represent

the derivative of the cost function and the activation function respectively, and the third term represents the matrices of the weights. And it doesn't matter what form the cost function and activation function take, as long as they can be differentiated. Each term can be simplify in following form, in which $g(x)$ denoted activation function, and W denoted weight matrix.

$$\frac{dC}{dx} = \frac{dC}{da^L} \cdot \left(g^L\right)' \cdot W^L \cdot \left(g^{L-1}\right)' \cdot W^{L-1} \cdot \ldots \circ \left(g^1\right)' \cdot W^1 \tag{15}$$

Reverses the order of the multiplication:

$$\nabla_x C = \left(W^1\right)^T \cdot \left(g^1\right)' \cdot \ldots \cdot \left(W^{L-1}\right)^T \cdot \left(g^{L-1}\right)' \cdot \left(W^L\right)^T \cdot \left(g^L\right)' \cdot \nabla_{a^L} C. \tag{16}$$

Let delta represent the error at layer l, that is, the gradient of the input value at layer l, then we can get:

$$\begin{aligned}
\delta^1 &= \left(g^1\right)' \cdot \left(W^2\right)^T \cdot \left(g^2\right)' \cdot \ldots \cdot \left(W^{L-1}\right)^T \cdot \left(g^{L-1}\right)' \cdot \left(W^L\right)^T \cdot \left(g^L\right)' \cdot \nabla_{a^L} C \\
\delta^2 &= \left(g^2\right)' \cdot \ldots \cdot \left(W^{L-1}\right)^T \cdot \left(g^{L-1}\right)' \cdot \left(W^L\right)^T \cdot \left(g^L\right)' \cdot \nabla_{a^L} C \\
\delta^{L-1} &= \left(g^{L-1}\right)' \cdot \left(W^L\right)^T \cdot \left(g^L\right)' \cdot \nabla_{a^L} C \\
\delta^L &= \left(g^L\right)' \cdot \nabla_{a^L} C
\end{aligned} \tag{17}$$

The delta can be calculated recursively from right to left shown below, this formula means we can get the delta from the last layer to the first layer, which means we can get the gradient of the cost function, then we can use it to update the parameters.

$$\delta^{l-1} := \left(g^{l-1}\right)' \cdot \left(W^l\right)^T \cdot \delta^l \tag{18}$$

From the derivative process, we can see that if we calculate each layer's error from the first layer, then we need to do many repeat calculations. But if we calculate from the final layer, then in each layer, we can just multiply the derivative of the last layer's activation value and weight matrix. This can reduce calculation and make the process faster.

Now we have the neural network model and we know how it works through forward process and backprogagation. So we can integrate them from a holistic perspective, the forward process output predicted value based on input value and current weight, then the backprogagation to calculate the error between true value and predicted value according to the generated prediction to update the weights. The updated weights are then used in the next round of the forward process, and the whole process is repeated until the model converges to a satisfactory solution.

If the process is a bit abstract for you, don't worry, as we will discuss a specific example and show the code implementation in the next section.

Example of a four layer network

The structure of the example neural network is similar to Figure 11, and the neurons in each layer can be different.

The forward propagation is:

$$\begin{aligned}
a^{(1)} &= x \\
z^{(2)} &= W^{(1)} a^{(1)} \\
a^{(2)} &= g\left(z^{(2)}\right) \quad \left(\text{ add } a_0^{(2)}\right) \\
z^{(3)} &= W^{(2)} a^{(2)} \\
a^{(3)} &= g\left(z^{(3)}\right) \quad \left(\text{ add } a_0^{(3)}\right) \\
z^{(4)} &= W^{(3)} a^{(3)} \\
a^{(4)} &= h_w(x) = g\left(z^{(4)}\right)
\end{aligned} \tag{19}$$

The backpropagation is:

$$\begin{aligned}
\delta^{(4)} &= a^{(4)} - y \\
\delta^{(3)} &= \left(W^{(3)}\right)^T \delta^{(4)} \cdot * g'\left(z^{(3)}\right) \\
\delta^{(2)} &= \left(W^{(2)}\right)^T \delta^{(3)} \cdot * g'\left(z^{(2)}\right)
\end{aligned} \tag{20}$$

According to:

$$\frac{\partial}{\partial w^{(l)}} J(W) = a^{(l)} \delta^{(l+1)} \tag{21}$$

To calculate the gradient using error of each layer and update parameters:

$$\Delta^{(l)} := \Delta^{(l)} + a^{(l)} \delta^{(l+1)} \tag{22}$$

$$w^{(l)} = w^{(l)} + \Delta w^{(l)} \tag{23}$$

2.2.4 Planar Data Classification

Decision boundary means the boundary line used to divide different categories in machine learning, and classification task is to find the decision boundary.

In this section, we will use neural network to classify planar data, that is, a public data set with 2-dimensional data, data distribution shown in Figure 13 with the shape of a flower. As we can see, the data categories are cross-distributed, so if we only use a line to classify these data, we can't get a satisfactory result.

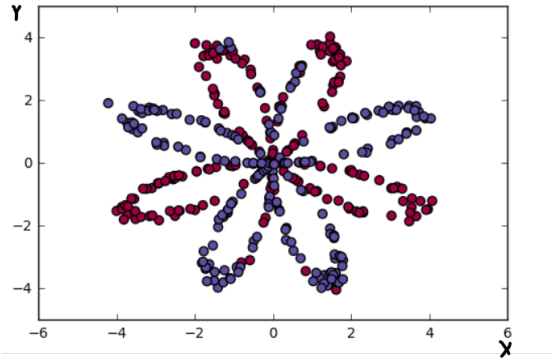

Figure 13: Planar Data

Logistic Regression

Logistic regression uses logistic function (also called sigmoid function) to map variables into number range in [0,1], although its' name is regression, but it's a classification algorithm that can divide data into two-categories. It's used in medicine area to predict whether a people has a disease, and other binary classification problem in economic and social fields.

Logistic function shown below, it's visually shown in Figure 7:

$$f(x) = \frac{1}{1 + e^{-(x)/s}} \tag{24}$$

The cost function is:

$$J = \sum_{(x,y)\in D} -y \log\left(f(x)\right) - (1-y) \log\left(1 - f(x)\right) \tag{25}$$

In above formula, data point denoted as (x, y), where x is a data point in the data set D and y is the true value in the data set. The true value in logistic regression is binary, taking the values 0 or 1. The function f(x) represents the predicted value produced by the logistic model, which falls in the range [0, 1].

When logistic regression is used to define a decision boundary, each data point (x, y) serves as an input. The output is a real number between 0 and 1

representing the probability of belonging to a particular category. By applying a threshold, hypothesis set at 0.5, if the output is greater than 0.5, the point is classified into one category; otherwise, it belongs to another category.

Figure 14 shows the classification results, where the model separates the red area into one category and the blue area into another. However, when compared to the data distribution, this division appears to be suboptimal.

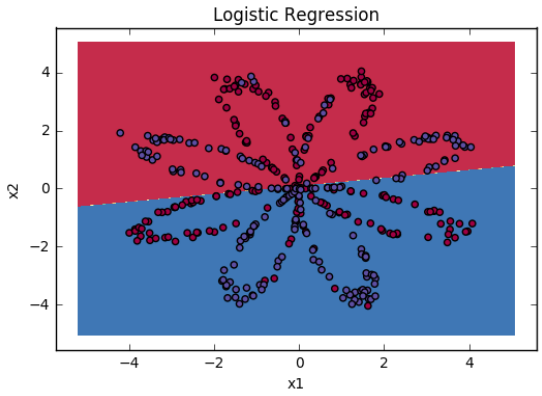

Figure 14: Logistic Regression on Planar Data

Neural Network

Now we try to use neural network to do the classification task, the neural network model shown in Figure . . . , using the cost function of logistic regression as the cost function represented in (), and using tanh function as activation function.

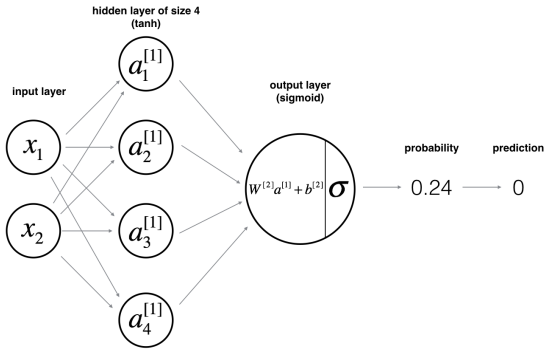

Figure 15: Neural Network Model

To elaborate, let's take a data point as an example. Initially, the weights in the neural network are randomly initialized. Then, with the data point's x as

input, the forward process is executed to obtain activation values and outputs. Subsequently, these values, along with the true value and weight matrix, are used to compute the gradient. The gradient is then employed to update the weights. This process is repeated until the error is sufficiently small or the maximum training rounds are reached.

Through the processes of forward propagation and backpropagation, we can achieve improved results, as depicted in Figure 16. As the figure shows, the model has learned the flower patterns hidden in the data set, compared with logistic regression, the descision boundaries are highly non-linear, so it's get more better result.

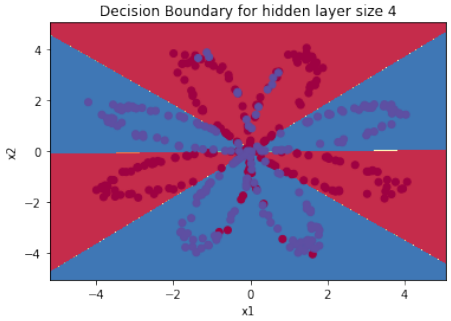

Figure 16: Neural Network Classification Result

Use Python to build and train a neural network to solve the planar data classification problem shown below:

```
import numpy as np
import matplotlib.pyplot as plt
from testCases_v2 import *
from public_tests import *
from planar_utils import plot_decision_boundary, sigmoid,
    load_planar_dataset, load_extra_datasets

# load the dataset
X, Y = load_planar_dataset()

# Visualize the data:
plt.scatter(X[0, :], X[1, :], c=Y, s=40, cmap=plt.cm.Spectral);

# get data shape
def layer_sizes(X, Y):
    n_x =  X.shape[0]
    n_h = 4
    n_y = Y.shape[0]
    return (n_x, n_h, n_y)


# initialize the parameters in neural network, including weigh and
    bias in each layer.
def initialize_parameters(n_x, n_h, n_y):
```

```
23
24      W1 = np.random.randn(n_h,n_x) * 0.01
25      b1 = np.zeros((n_h,1)) * 0.01
26      W2 = np.random.randn(n_y,n_h) * 0.01
27      b2 = np.zeros((n_y,1)) * 0.01
28
29      parameters = {"W1": W1,
30                    "b1": b1,
31                    "W2": W2,
32                    "b2": b2}
33
34      return parameters
35
36
37  # forward_propagation
38  def forward_propagation(X, parameters):
39
40      W1 = parameters['W1']
41      b1 = parameters['b1']
42      W2 = parameters['W2']
43      b2 = parameters['b2']
44
45      Z1 = np.dot(W1, X) + b1
46      A1 = np.tanh(Z1)
47      Z2 = np.dot(W2, A1) + b2
48      A2 = sigmoid(Z2)
49
50      assert(A2.shape == (1, X.shape[1]))
51
52      cache = {"Z1": Z1,
53               "A1": A1,
54               "Z2": Z2,
55               "A2": A2}
56
57      return A2, cache
58
59  # use logistic function as cost function, calculate loss base on
        prediction and true value
60  def compute_cost(A2, Y):
61
62      m = Y.shape[1] # number of examples
63      logprobs = np.multiply(np.log(A2), Y) + np.multiply((1 - Y), np
            .log(1 - A2))
64      cost = - np.sum(logprobs) / m
65      cost = float(np.squeeze(cost))
66
67      return cost
68
69  # calculate the gradient
70  def backward_propagation(parameters, cache, X, Y):
71
72      m = X.shape[1]
73
74      W1 = parameters['W1']
75      W2 = parameters['W2']
76
77      A1 = cache['A1']
```

```
78      A2 = cache['A2']
79
80      # calculate dW1, db1, dW2, db2.
81      dZ2= A2 - Y
82      dW2 = (1 / m) * np.dot(dZ2, A1.T)
83      db2 = (1 / m) * np.sum(dZ2, axis=1, keepdims=True)
84      dZ1 = np.multiply(np.dot(W2.T, dZ2), 1 - np.power(A1, 2))
85      dW1 = (1 / m) * np.dot(dZ1, X.T)
86      db1 = (1 / m) * np.sum(dZ1, axis=1, keepdims=True)
87
88      grads = {"dW1": dW1,
89               "db1": db1,
90               "dW2": dW2,
91               "db2": db2}
92
93      return grads
94
95
96  def update_parameters(parameters, grads, learning_rate=1.2):
97
98      W1 = parameters['W1']
99      b1 = parameters['b1']
100     W2 = parameters['W2']
101     b2 = parameters['b2']
102
103     dW1 = grads['dW1']
104     db1 = grads['db1']
105     dW2 = grads['dW2']
106     db2 = grads['db2']
107
108     W1 = W1 - learning_rate * dW1
109     b1 = b1 - learning_rate * db1
110     W2 = W2 - learning_rate * dW2
111     b2 = b2 - learning_rate * db2
112
113     parameters = {"W1": W1,
114                   "b1": b1,
115                   "W2": W2,
116                   "b2": b2}
117
118     return parameters
119
120
121 # train the model
122 def nn_model(X, Y, n_h, num_iterations=10000, print_cost=False):
123
124     np.random.seed(3)
125     n_x = layer_sizes(X, Y)[0]
126     n_y = layer_sizes(X, Y)[2]
127
128
129     parameters = initialize_parameters(n_x, n_h, n_y)
130     W1 = parameters['W1']
131     b1 = parameters['b1']
132     W2 = parameters['W2']
133     b2 = parameters['b2']
134
```

```
135
136     for i in range(0, num_iterations):
137
138         # values used in backpropagation are stored in a cache.
139         A2, cache = forward_propagation(X, parameters)
140
141         cost = compute_cost(A2, Y)
142
143         grads = backward_propagation(parameters, cache, X, Y)
144
145         parameters = update_parameters(parameters, grads)
146
147         # Print the cost every 1000 iterations
148         if print_cost and i % 1000 == 0:
149             print ("Cost after iteration %i: %f" % (i, cost))
150
151     return parameters
152
153 # using the model to predict
154 def predict(parameters, X):
155
156     A2, cache = forward_propagation(X, parameters)
157     predictions = np.round(A2)
158
159     return predictions
160
161
162 parameters = nn_model(X, Y, n_h = 4, num_iterations = 10000,
        print_cost=True)
163 # Plot the decision boundary
164 plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)
165 # plt.title("Decision Boundary for hidden layer size " + str(4))
```

2.3 Convolution Neural Network

In the field of deep reinforcement learning (DRL), convolutional neural networks (CNNs) have proven to be powerful tools for handling complex input data, especially when it comes to visual information. Unlike traditional fully connected neural networks, CNNs are designed to effectively process grid-like data, making them particularly well-suited for tasks involving images, which are prevalent in many real-world environments.

Convolutional Neural Networks (CNNs), widely used in computer vision, computer vision has a wide range of applications, such as image classification, object detection in self-driving car and surveillance system, and image optimization in mobile phone cameras. When dealing with images, the pixels are fed into the neural network as features. However, since images have three RGB channels and potentially large dimensions, the number of input features can reach several million. This results in significant computational and storage costs, which can be impractical or even infeasible. In addition, an excessive number of parameters can lead to overfitting. To address this challenge, researchers introduced the concept of convolutional operations and subsequently developed convolutional neural networks.

Convolutional Neural Networks are inspired by the biological basis of the brain's visual processing. if you want to know more, you can search about visual cortex and neocognitron.

In this section, we will comprehensively cover the key components of a Convolutional Neural Network. This includes basic operations such as convolution and pooling, and how they are used to construct CNN. And we will introduce a seminal CNN model known as LeNet-5, finally, how to integrate CNN in Reinforcement learning will be introduced.

2.3.1 Components of Convolution Neural Network

Convolutional Operation

The convolution operation is a specialized linear calculation. Consider a 3x3 dimensional matrix as the input and a 2x2 dimensional matrix as the filter or kernel. The convolution operation is illustrated in the Figure 17. The input matrix is also called input feature map, the second matrix is called kernel or filter.

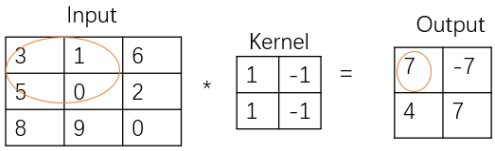

Figure 17: Convolution Example

To calculate each element in the output matrix, refer to the Figure 18. Take the element inside the circle, perform a wise product with the kernel matrix, resulting in a single element in the output matrix. Then move the circle to a new column or row element and repeat the process.

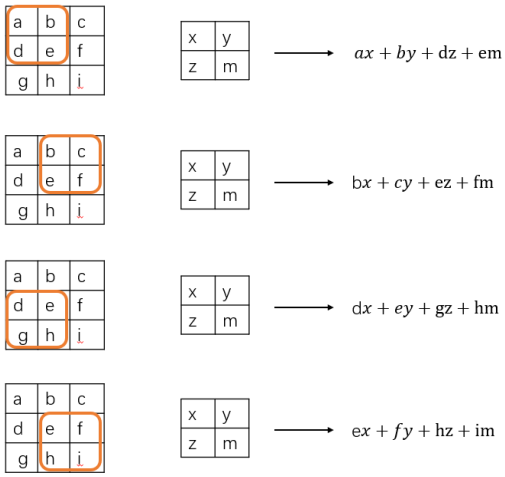

Figure 18: Convolution Operation

Choosing the kernel size and determining the number of kernels has sparked much debate and experimentation. In cases where the problem becomes complex and the input dimensions are large, manually choosing the number of kernels becomes impractical. Instead, we can treat it as parameters and use backprop-

agation to learn it. This process allows the kernel parameters to absorb the underlying patterns in the data, since it uses the data itself. As a result, the results tend to be superior to manual selection. The automatically learned parameters for the neural network to recognize low-level features of the data. More details will be discussed in later sections.

When it comes to implementing the convolution operation in a programming language, it has already been incorporated into numerous programming languages and deep learning frameworks. So it's convenient to use it in your neural network.

2.3.2 The Advantages of Convolutions

If we take a 24 x 24 x 3 dimensional image as input and use a 5 x 5 x 3 dimensional kernel with six kernels, the resulting output matrix will be 20 x 20 x 6 in dimensions. In this scenario, the number of parameters for the kernels is 156.

However, if we choose to use forward neural networks instead of convolutions, the number of parameters becomes $(24 * 24 * 3) * (20 * 20 * 6) = 1728 * 2400$, it's about 4 million. This represents a computational and storage cost that is a ten thousand times greater than using convolutions. Given the relatively small size of the input image in this case, it's not worth spending too many parameters in the small image.

But it's not the only reason to keep the size of parameters small, there are other reasons that convolutions need to run on small size of parameters, which are parameter sharing and sparsity of connections.

Parameter Sharing
In image processing, the same feature detector can be applied to different parts of an image. For example, when training a kernel to detect edges in images, from our intuition we can know that the edge is the pixel's RGB channels change abruptly, A well-trained edge detector can detect this feature in different parts of the image. Thus, a detector can be applied to the entire image as well as to other images, you don't need to train a different kernel for each partition or image, feature detectors can be shared across the entire image dataset.

Sparsity of Connections
As shown in Figure 17, the circled element (seven) in the output matrix is only connected to the circled 2 x 2 matrix in the input matrix. It is connected only to the four circled input features, while the rest of other elements have no influence on the output value. This demonstrates the sparsity of the connections.

In addition to these advantages, the convolutional structure maintains the property of translation invariance. This means that an image can be shifted by a number of pixels in different directions, but it remains the same image and can still be recognized. Because the kernel used to convolution different parts of image, it can recognize similar features even if the image is shifted by a few pixels.

2.3.3 Convolution Layer

The first step is to perform the convolution operation. After convolving the input with the two kernels, we obtain a 4 x 4 x 2 output. Next, we add a bias, which is a real number, to the output. Following this, we apply a nonlinear function. After these operations, we finally obtain a 4 x 4 x 2 dimensional output matrix. This structure and processing represent one layer of a convolutional neural network.

When comparing it to a standard neural network without a convolutional layer, the parameters of the kernel can be considered as the weight, denoted as $w^{[1]}$, and the input feature map can be thought of as $a^{[0]}$. Thus, the convolution operation essentially performs the same function as $w^{[1]} * a^{[0]}$. The entire process can still be viewed as an activation function unit. The key distinction between a standard neural network and a convolutional neural network lies in the $w * a$ operation, which is the convolution operation in a convolutional neural network.

In a convolutional network, regardless of the size of the input matrix, the size of the parameters remains invariant depend on the size of the kernels. While in some cases one kernel may detect one feature, in others, several kernels may work together to detect a single feature. Therefore, the total number of kernels determines the variety and size of features. Once you have trained several feature detectors, you can apply them to larger images than those in the training dataset, the property can avoid overfitting.

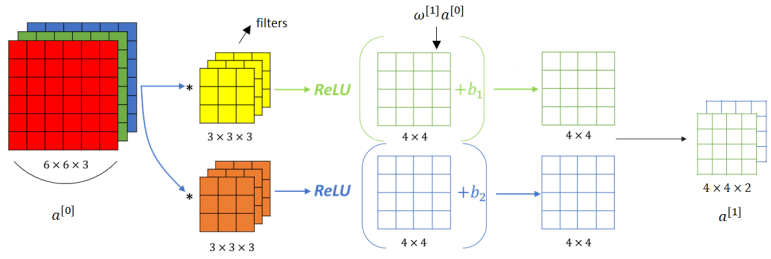

Figure 19: One Convolution Layer

2.3.4 Pooling

To reduce the size of a convolutional neural network and speed up computation, it is beneficial to add a pooling operation.

Max Pooling
As shown in Figure 20, the input matrix is divided into partitions determined by the Max Pooling parameter. Then, the maximum element in each partition corresponds to the number in the output matrix. The process is similar to using a 2 x 2 kernel and moving with a stride of two, where both the size of the kernel and the stride serve as parameters for max pooling.

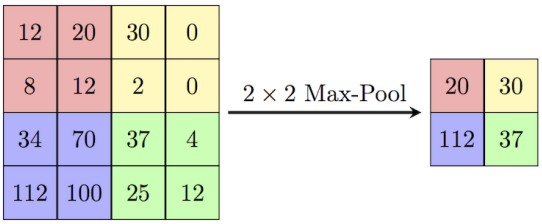

Figure 20: Max Pooling

The pooling parameters do not require training and should be set as hyper-parameters before starting training.

Basically, pooling works by examining each partition independently. If a partition has the feature the detector is looking for, the maximum value is selected. Even if the partition lacks the feature, it can still be represented by its maximum value.

Average Pooling
Average pooling calculates the average of each partition to derive the corresponding input value, as shown in Figure 21.

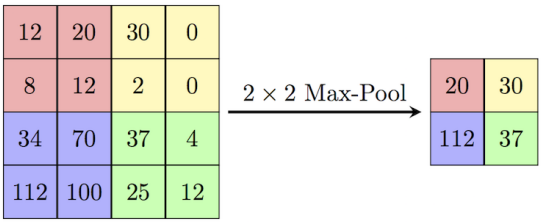

Figure 21: Max Pooling

Why Pooling ?
The first obvious advantage of pooling is that it reduces the number of parameters, thereby reducing computational and storage costs, as well as the required

processing time.

A second purpose of pooling is to preserve invariance to translations. This means that if we shift the input by a few pixels, most of the pooling output will remain unchanged. In certain scenarios, we are more interested in what the feature is than where it is located. Thus, this invariance to translation proves beneficial because it shifts our attention to the feature itself rather than its location. For example, in face recognition, we may be primarily interested in determining the presence of a face in an image, without requiring precise pixel-level details.

2.3.5 LeNet-5 Architecture

LeNet-5 is a classic and important neural network architecture in the development of deep learning. It's used to handwriting digit recognition. By studying LeNet-5, we gain insight into how to integrate the various components mentioned above to construct an effective convolutional neural network.

As shown in Figure 22, LeNet-5 consists of eight layers, including both input and output layers. It uses convolutional and pooling layers, as well as fully connected layers. The fully connected layer is the standard neural network layer as introduced in the last section.

The network takes an image with dimensions of 32 x 32 x 1 as input and produces a vector indicating the digit represented in the input image. Since the input images are grayscale, they have only one channel. The process begins by applying six 5 x 5 filters with a stried of one. This convolutional operation reduces the image dimensions to 28 x 28. A average pooling is then applied with a filter size of two and a stride of two, resulting in a volume of 14 x 14 x 6.

The next step is to pass through a convolutional layer with 16 filters, each 5 x 5, resulting in a 10 x 10 x 16 volume with 16 channels. The volume then passes through another pooling layer to further reduce its size by a factor of two, resulting in a 5 x 5 result that retains 16 channels. This result, equal to 400, send to a fully connected layer with 120 nodes. This is followed by another fully connected layer, this time with 84 nodes. Finally, we apply these 84 features with a softmax function to obtain a final output. This output includes 10 numbers, each representing the probability corresponding to a digit from 0 to 9.

The LeNet-5 architecture was introduced in 1994. Since then, neural networks have many significant developments, leading to possible modifications in their contemporary implementations. However, the basic principles remain the same.

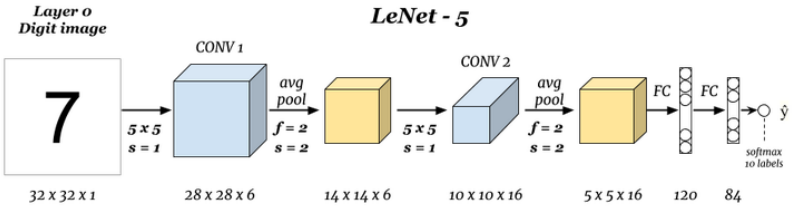

Figure 22: LeNet-5

2.3.6 CNN in Reinforcement Learning

In the context of reinforcement learning, CNNs are often used to process visual input from the environment. This is particularly common in applications where the agent interacts with the environment through visual observations, such as playing video games or navigating a robotic environment.

A notable example is the use of CNN in the famous Deep Q-Network (DQN) algorithm, which achieved groundbreaking results on a variety of Atari games. The CNN in DQN processes the game frames as input, allowing the algorithm to learn effective strategies directly from raw pixel data. Each frame of the game serves as input to the CNN, these frames capture the visual information about the game state, convolutional layers automatically extract hierarchical features from the frames, lower layers can capture basic patterns such as edges, while higher layers focus on complex structures such as game objects. CNNs inherently understand the spatial relationships within images, which is critical for tasks where the arrangement of objects is important. The learned features are then fed into fully connected layers to estimate Q-values for different actions; the agent uses these Q-values to make decisions and improve its policy over time. More details about DQN will be discussed in a later section.

In robotics and simulated environments, CNNs are used to extract meaningful features from camera images or depth maps. This visual information is then used by the reinforcement learning agent to make decisions and navigate the environment.

In summary, the integration of CNN with reinforcement learning has significantly improved the ability of agents to understand and interact with visual environments. The ability to extract meaningful features from raw pixels has opened new avenues for tackling complex real-world problems using DRL approaches.

2.4 Recurrent Neural Network

In the previous section, we learned CNN, a specialized model apply to process grid data like images. It excels at reducing large images to more manageable sizes without compromising detection accuracy. However, just as with images, there are sequential data types such as audio clips and text transcripts that require specific handling. We need a model capable of dealing with original sequential data, mapping it to another sequential format which we need, and efficiently processing large-scale data. Recurrent Neural Networks is proposed for the problem. RNN have found applications in a wide array of scenarios today, from speech recognition and language translation to even music generation and sentiment classification.

Recurrent Neural Networks (RNNs), used in RL, stand out as useful architectures in the field of Deep Reinforcement Learning (DRL) when dealing with tasks involving sequential information. This section will guide us through the construction and functioning of RNNs, exploring its forward propagation and backpropagation structures, end up with RNN used in Reinforcement Learning.

2.4.1 Forward Propagation

A basic RNN model, shown in Figure 23, differs from the standard neural network in its handling of sequential inputs. It takes the first variable x_1, feeds it into the network to produce the output y_1. Then it uses the information from the first step to process the second variable. This pattern continues, with each step building on the information gathered in the previous steps. Conceptually, you can imagine that each recurrent neural network has a core recurrent cell that takes in x and updates the RNN's internal hidden state. Subsequently, the updated hidden state interacts with the new input at each step to produce an output. This type of perspective is shown in Figure 24.

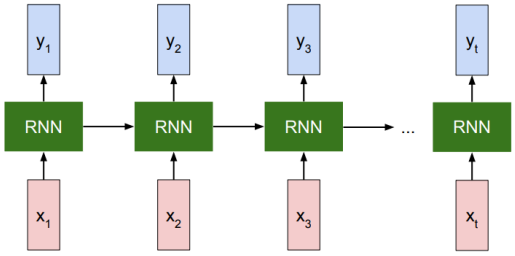

Figure 23: Recurrent Neural Network Model

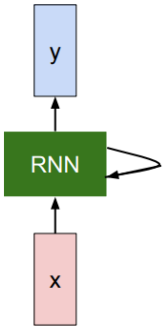

Figure 24: Rolled Recurrent Neural Network Model

Initially, we can make some fake data as the activation at step zero to feed into the first step. Typically, we set this to zero or some random data.

This process makes each step absorb information from previous steps, but the weakness is that it tends to underutilize information from steps further back. For example, when dealing with a sequence of text as input, it's often crucial to consider not only the preceding words, but also the following ones. Certain variants of RNNs are designed to overcome this limitation. If you are interested

in this topic, you can learn more by reading special books or papers on RNN.

We can represent the recurrent relation as a self-calling function using the mathematical formula below. It takes the current input $x^{\langle t \rangle}$ and the activation value from the previous step $a^{\langle t-1 \rangle}$ as inputs. This function is used to update the hidden layer parameters. When moving to the next step t+1, the activation value from the previous step $a^{\langle t \rangle}$ is fed back into the same function.

$$a^{\langle t \rangle} = f_w \left(a^{\langle t-1 \rangle}, x^{\langle t \rangle} \right) \tag{26}$$

The parameters from x_t to the hidden layer are denoted as W_{ax}, which is shared across all time steps. Similarly, the parameters between the hidden layers can be expressed as W_{aa}, which controls the output activation value, while W_{ya}, represents the parameters controlling the output values. As shown in Figure 25.

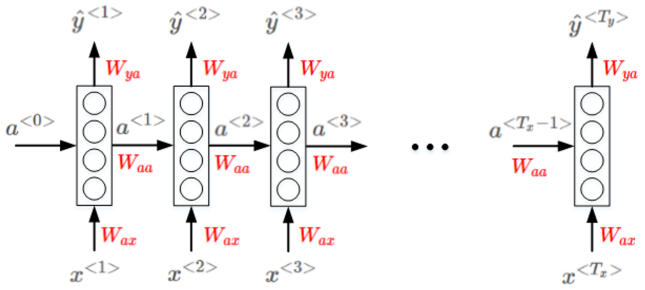

Figure 25: Recurrent Neural Network's Parameter

Taking $\hat{y}^{\langle 1 \rangle}$ as an example, the procedure for calculating $\hat{y}^{\langle 1 \rangle}$ is as follows:

$$a^{\langle 1 \rangle} = g_1 \left(\omega_{aa} a^{\langle 0 \rangle} + \omega_{ax} x^{\langle i \rangle} + b_a \right) \tag{27}$$

$$\hat{y}^{\langle 1 \rangle} = g_2 \left(\omega_{ya} a^{\langle 1 \rangle} + b_y \right) \tag{28}$$

The value of a_1 is determined by four components. The first is the initial activation value, which is usually set to zero and multiplied by the initial parameter matrix W_{aa}. The second part is the input value multiplied by the parameter matrix W_{ax}. The third part is the bias, and the fourth part is the result of applying the activation function to the sum of the first three parts. Next, a_1 is multiplied by the parameters W_{ya} and the bias is added. The result is passed through an activation function represented by $g(x)$ to give the output $\hat{y}^{\langle 1 \rangle}$. The choice of activation function depends on the desired output. For binary classification, the sigmoid function is appropriate, while softmax is appropriate for multi-classification. The process is then repeated for the next step and so on until all input values have been processed. It's important to note that several parameter matrices are shared across all steps.

The general mathematical expression is shown below. This formula outlines forward propagation in a recurrent neural network.

$$a^{\langle t \rangle} = g\left(\omega_{aa} a^{\langle t-1 \rangle} + \omega_{ax} x^{\langle t \rangle} + b_a\right) \tag{29}$$

$$\hat{y}^{\langle t \rangle} = g\left(\omega_{ya} a^{\langle t \rangle} + b_y\right) \tag{30}$$

The simplified formula below summarizes the above expressions. By using a parameter matrix W_a that encapsulates the information contained in W_{aa} and W_{ax}, this expression simplifies the notation for our convenience.

$$a^{\langle t \rangle} = g\left(\omega_a \left[a^{\langle t-1 \rangle}, x^{\langle t \rangle}\right] + b_a\right) \tag{31}$$

2.4.2 Backpropagation Through Time

During forward propagation in a recurrent neural network, we obtain output values. To update the parameters, we need to define a cost function that measure the distance between the predicted value and the true value. Let's consider an English sentence such as "recurrent neural network is widely used" that needs to be translated into Chinese. The whole process involves tasks such as data encoding, mapping to a dictionary, and more complex network structures. For this example, however, we'll focus only on backpropagation for clarity. The input is taken one word at a time, along with the corresponding output values. We can identify whether the input word and output word match in the dictionary. In more complex scenarios, you could define a measure of the distance between the output value and the true value, which would be the Chinese word corresponding to the input English word. Based on this, we use a cost function to compute the loss. We compute the loss for the whole sequence by summing the loss for each word. This process is illustrated in Figure 26, the mathematically represented as follows:

$$L(\hat{y}, y) = \sum_{t=1}^{T_y} L^{<t>}\left(\hat{y}^{<t>}, y^{<t>}\right) \tag{32}$$

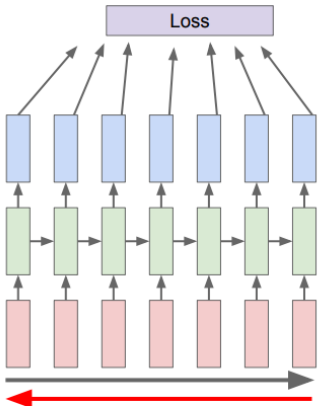

Figure 26: Loss calculation of Recurrent Neural Network

Once we have defined the cost function, we can use the gradient descent algorithm to find the parameter matrix that minimizes the cost function, and then update the parameter matrix. Backpropagation is from right to left, which is a reverse direction of increasing the time step, so the algorithm is aptly named "backpropagation through time".

2.4.3 RNN in Reinforcement Learning

In the context of reinforcement learning, RNNs are used when the state of the environment is not fully observable from a single observation, and information from past observations is crucial for decision making. This is particularly relevant in scenarios where the agent needs to remember past events or observations in order to make informed decisions.

In some environments, the agent's observation at a given time step may not provide complete information about the state. RNNs help overcome this challenge by maintaining an internal state that captures information from previous observations, effectively providing the agent with a form of memory. For example, consider an autonomous vehicle navigating a complex environment. At each time step, the vehicle receives an observation, but a single snapshot may not provide a complete view of the environment. RNNs fill this gap by maintaining a memory of past observations, allowing the vehicle to build context and make informed decisions based on the entire journey.

Tasks that involve sequential decision making, such as navigating a maze or playing a game with a changing environment, benefit from RNNs. The network's ability to retain information about past observations allows the agent to make decisions based on a history of experience. For example, imagine an agent playing a video game where the sequence of actions is important. In traditional feedforward networks, each action is treated in isolation. But in games with changing environments, past actions profoundly affect the outcome. RNNs allow the agent to consider the entire sequence of actions, ensuring a more nuanced and strategic decision-making process.

Learning to Play Pong with RNNs
Let's look at a concrete example to illustrate the power of RNN in reinforcement learning. Consider training an agent to play the game Pong. In each frame, the agent receives a picture of the current state. However, to understand the game, the agent must remember the trajectory of the ball and the movements of the opponent.

The agent equipped with an RNN processes a sequence of observations instead of individual frames. This sequence allows the RNN to capture the temporal dynamics of the game. The hidden state of the RNN acts as a memory, holding information about the ball's trajectory, the paddle's movements, and the agent's past actions.

When deciding the next move, the RNN considers not only the current frame, but also the historical context encoded in its hidden state. This allows the agent to learn strategies that depend on the entire sequence of observations.

In summary, recurrent Neural Networks play an important role in reinforcement learning by enabling agents to navigate through sequential and partially observable environments. Their ability to model temporal dependencies and retain memory provides agents with the tools necessary to address a wide range of challenges in dynamic and evolving scenarios.

2.5 Deep Q-Learning

In Chapter 1, we introduced the basic concepts of reinforcement learning and two classical algorithms: Q-learning and SARSA. However, it's clear that these methods rely on tabular representations of states. In real-world problems, the scale can become too large for tabular approaches to be practical. Even in relatively simple game settings, the number of states can reach 10^{20}. In more complex games, such as computer Go, the state space can balloon to 10^{170}. Moreover, in problems involving robots navigating continuous state spaces, the number of states becomes uncountably infinite. In all of these scenarios, the traditional approach of storing a table and assigning a separate value to each state becomes infeasible due to large storage and computation costs. This challenge is particularly pronounced in cases where the underlying Markov Decision Process (MDP) of the environment is unknown, making reliance on value functions alone for action selection impractical. This is where the action value function Q(s, a) comes in. By allowing us to evaluate each action under each state, we can select actions by maximizing Q. However, this requires storing a two-dimensional table of all states and actions, which further increases the storage and computational requirements, and the massive computation also take a lot of time.

Therefore, finding a solution that reduces the storage and computational burden is paramount. Ideally, we seek a method that can seamlessly handle continuous state spaces, even infinitely large ones, while reducing the overall computational time.

The answer lies in value function approximation. This involves using function approximation to estimate the state space. In this section, we'll delve into this technique, exploring how to find appropriate function approximations and how to update their parameters. And then we will learn about two key technologies that are experience replay and two network structure. These components are the core idea of the Deep Q-Network (DQN). Finally, we'll see how to combine this foundation pieces to construct Deep Q-learning and to handle large state spaces.

Deep Q-learning is a milestone structure in the development of reinforcement learning. Prior to the advent of DQN, numerous attempts were made to integrate deep neural networks into reinforcement learning, but the outcomes were less than optimal. Consequently, DQN stands out as one of the earliest algorithms to successfully incorporate deep neural networks into RL. Notably, its remarkable performance in game tests showed its effectiveness. The technology it introduced continues to be influential until today.

2.5.1 Value Function Approximation

When the state space is so large that it's infeasible to visit and estimate every single state, we need to estimate the unvisited states based on the states we already know. This challenge is especially prevalent in continuous state spaces. To address this, we assume the existence of a true value function $v(s)$ that map s to stat value. We then construct a set of functions to estimate this mapping relationship using all available data, and iteratively update the parameters of these functions until an optimal solution is reached. The input and output of the value function is presents in Figure 27.

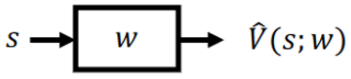

Figure 27: Input and Output Of value function approximator

In the previous section on loss functions, we examined an example of using a linear function to approximate the relationship between price and click rate. In this section, we do the same thing that to approximate the relationship between state s_t, and and state value $v(s_t)$ under policy π. In the first step to understand value function approximation, given a simple example that the relationship is linear:

$$\hat{v}(S, \mathbf{w}) = \mathbf{x}(S)^\top \mathbf{w} = \sum_{j=1}^{n} \mathbf{x}_j(S)\mathbf{w}_j \tag{33}$$

In this formula, the vector $\mathbf{w}$ represents the weight vector, $\mathbf{x}$ is the feature vector which can be different depend on specific problems, for example feature is price in the click rate predict problem, and $\hat{v}(S, \mathbf{w})$ signifies the value function used to approximate the relationship between state value and state.

By employing this formulation, we can store just two parameter vectors instead of the entire state space. This allows us to generalize to unvisited states by updating parameter vectors, which is a significant advantage compared to the tabular form that only allows updates to visited states. However, the drawback is that the state values are estimated, which may introduce some error compared to the true value.

The linear approximation example provided earlier is a simple case. In more complex scenarios, one can use techniques like neural networks, decision trees, high-order functions, and so forth. Neural networks are particularly prevalent in modern applications of function approximation, especially in cases involving non-linear relationships. We'll discuss deeper into this topic later. It's important to note that while more complex functions may achieve a more accurate fit to the relationship, they also tend to require more parameters.

Let's assume we have a true value function $v(s)$ and an estimated function $\hat{v}(S, \mathbf{w})$. In order to derive an estimated function that best approximates the

true value function for each state, we need to adjust the parameter vectors until the discrepancy between them is sufficiently small.

To quantify the gap between $v(s)$ and $\hat{v}(S, \mathbf{w})$ under all state, we define the objective function, denoted as:

$$J(\mathbf{w}) = \mathbb{E}_\pi \left[\left(v(S) - \hat{v}(S, \mathbf{w}) \right)^2 \right] \tag{34}$$

In the preceding section, we introduced the gradient descent algorithm to search for an optimal solution for the objective function, or loss function, as follows:

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \begin{pmatrix} \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_1} \\ \vdots \\ \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_n} \end{pmatrix} \tag{35}$$

Using the formulas above in conjunction with the chain rule applied to $J(\mathbf{w})$, and depend on gradient descent algorithm, we get the $\Delta \mathbf{w}$, the formulas represent parameter updates in the direction of the gradient, scaled by the learning rate.

$$\begin{aligned} \Delta \mathbf{w} &= -\frac{1}{2} \alpha \nabla_{\mathbf{w}} J(\mathbf{w}) \\ &= \alpha \mathbb{E}_\pi \left[\left(v(S) - \hat{v}(S, \mathbf{w}) \right) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w}) \right] \end{aligned} \tag{36}$$

Then we can update the weight vector $\mathbf{w}$:

$$\mathbf{w}_{t+1} = \mathbf{w_t} - \alpha \nabla_{\mathbf{w}} J\left(\mathbf{w}\right) \tag{37}$$

In the gradient formula, we need to calculate an expectation, which is equivalent to a full gradient update. To avoid this computational burden, we can employ stochastic gradient descent, which is sampling from the gradient, as denoted in the formula below. The stochastic approximation theory demonstrates that this approach can still minimize the mean square error between the true value and predicted value across the entire dataset.

$$\Delta \mathbf{w} = \alpha \left(v(S) - \hat{v}(S, \mathbf{w}) \right) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w}) \tag{38}$$

2.5.2 TD Learning with Value Function Approximation

In the above formula, the true state value function $v(s)$ remains unknown, and in practice, we won't have direct access to it. Hence, we need a way to estimate the true value. There are two methods to achieve this through experience: Monte Carlo and TD learning. Based on Monte Carlo learning, we use the discounted return obtained from sample episodes as an estimate of the true state value. The DQN algorithm employs TD learning to estimate $v(s)$, with the formula as follows:

$$\Delta \mathbf{w} = \alpha \left(R_{t+1} + \gamma \hat{v} \left(S_{t+1}, \mathbf{w} \right) - \hat{v} \left(S_t, \mathbf{w} \right) \right) \nabla_{\mathbf{w}} \hat{v} \left(S_t, \mathbf{w} \right) \tag{39}$$

As we learned earlier, TD learning is a biased estimator of the true value function. However, we can update it at each step using TD learning. By utilizing the TD target as an estimate of the true state value, along with the estimated value, we can calculate $\Delta \mathbf{w}$ to update the parameters at each step. This process is similar to making a prediction about the future, then taking a step forward, receiving actual feedback, and using that feedback to adjust parameters. This cycle is repeated. Compared with supervised learning, it's like $\hat{v}(s, w)$ is the function we aim to update, the data comes from the TD target, which is $R_{t+1} + \gamma \hat{v} \left(S_{t+1}, \mathbf{w} \right)$. The complete process described in following pseudocode.

Algorithm 1 TD Learning with Function Approximation

Initialization: A function $\hat{v}(s; w)$ that is differentiable in w.
Initial parameter w_0.
Aim: Approximate the true state values of a given policy π.
for each episode generated following the policy π **do**
  for each step (s_t, r_{t+1}, s_{t+1}) **do**
    $\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha_t \left[r_{t+1} + \gamma \hat{v}(s_{t+1}; \mathbf{w}_t) - \hat{v}(s_t; \mathbf{w}_t) \right] \nabla_{\mathbf{w}} \hat{v}(s_t; \mathbf{w}_t)$
  end for
end for

Use TD target as estimate of true state value with linear example

Estimate value Function:

$$\hat{v}(S, \mathbf{w}) = \mathbf{x}(S)^\top \mathbf{w} = \sum_{j=1}^{n} \mathbf{x}_j(S) \mathbf{w}_j \tag{40}$$

Update parameter vector:

$$\nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w}) = \mathbf{x}(S) \tag{41}$$

$$\Delta \mathbf{w} = \alpha \left(v_(S) - \hat{v}(S, \mathbf{w}) \right) \mathbf{x}(S) \tag{42}$$

Using TD target to replace true value function, which make parameters update in the direction of TD target with learning rate alpha.

$$\begin{aligned}\Delta \mathbf{w} &= \alpha \left(R + \gamma \hat{v}\left(S', \mathbf{w}\right) - \hat{v}(S, \mathbf{w})\right) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w}) \\ &= \alpha \delta \mathbf{x}(S) \\ \delta &= R + \gamma \hat{v}\left(S', \mathbf{w}\right) - \hat{v}(S, \mathbf{w})\end{aligned} \tag{43}$$

Action-value function approximation

In a model-free scenario, where we do not have precise knowledge of the Markov Decision Process (MDP), determining which action to take becomes a unsolved problem even with the best approximation of the value function. Therefore, we need to find an action-value function approximator that allows us to select actions based on the best action value at each state, that also constructing the optimal policy.

Similar to the concept of the value function, we apply the same principle to the action-value function, changing our focus from the value function to the action-value function. The input and output of the action-value function are shown in Figure 28,and the difference with the tabular method is shown in the figure 29.

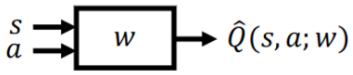

Figure 28: Input and Output Of action-value function approximator

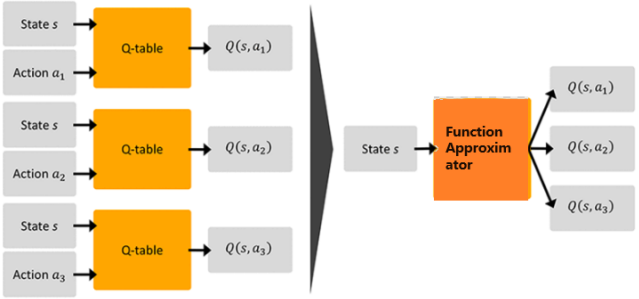

Figure 29: Compare with tabular method

Our goal is to construct a function with parameters w that approximates the action-value function and predicts the accumulated rewards for each state and action. We want to minimize the mean squared error between the approximate value $\hat{q}$ and the true value q under the policy π using stochastic gradient descent. This process can be described by the following formula.

The objective function or loss function is defined as:

$$J(\mathbf{w}) = \mathbb{E}_\pi \left[\left(q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}) \right)^2 \right] \tag{44}$$

Calculate its gradient using stochastic gradient descent:

$$\begin{aligned} -\frac{1}{2} \nabla_{\mathbf{w}} J(\mathbf{w}) &= \left(q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}) \right) \nabla_{\mathbf{w}} \hat{q}(S, A, \mathbf{w}) \\ \Delta \mathbf{w} &= \alpha \left(q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}) \right) \nabla_{\mathbf{w}} \hat{q}(S, A, \mathbf{w}) \end{aligned} \tag{45}$$

And the same problem arises that we do not know the true action value function. To address this, we use Monte Carlo or TD learning to estimate the true function. The underlying principle is similar to value function approximation. The following formula represents TD(0) as the target action value function estimate:

$$\Delta \mathbf{w} = \alpha \left(R_{t+1} + \gamma \hat{q} \left(S_{t+1}, A_{t+1}, \mathbf{w} \right) - \hat{q} \left(S_t, A_t, \mathbf{w} \right) \right) \nabla_{\mathbf{w}} \hat{q} \left(S_t, A_t, \mathbf{w} \right) \tag{46}$$

2.5.3 Deep Q-Learning

Q-Learning with Function Approximation
In the previous chapter, we covered Q-learning in detail. In this section, we use function approximation as the TD target. Instead of using a table to store action state values, we use a function approximator to approximate the relationship between action state and action state value. This approach has several advantages, as discussed in the previous section.

The process is outlined in the following pseudocode. The main difference is that we no longer update the $Q(s, a)$ values directly in the table. Instead, we adjust the parameters w to obtain an increasingly refined function approximator, resulting in more accurate action state values, so that we can make informed decisions based on specific policy to select the action.

Experience Replay in Deep Q-learning
In the previous sections, we observed that updating parameters at each step relies only on a single data case. However, this approach lacks data efficiency. Ideally, we would like each update to be based on a batch of data, allowing us to find the best possible fit for our function approximator across the entire batch of experiences.

To address this, we introduce the concept of a batch in reinforcement learning, which represents the collective experience of the agent. This includes states, actions, rewards, and next states. All of this information is stored in a large replay buffer called the $D = (s, a, r, s')$.

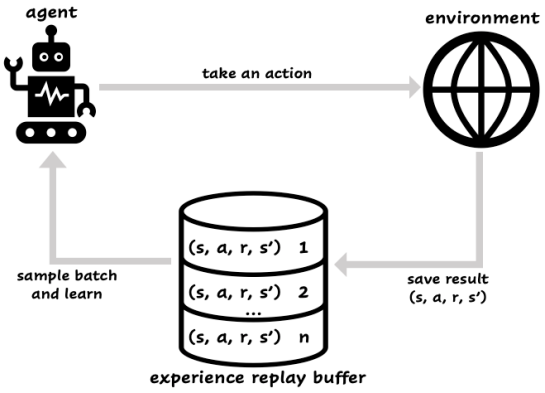

Figure 30: Experience Replay

The process shown in Figure 30, the agent interact with the environment, then store states, actions, rewards and next states in the replay buffer, in the learning stage, sample batch from the buffer and train the agent.

When we sample from this replay buffer, we do so randomly to achieve a uniform distribution. This ensures that each sample has an equal chance of being selected. The reason for this is that the agent has no prior knowledge about

the relative importance of different states or actions. Also, in reinforcement learning, the experience sequence is high correlated, and our policy is updated at every step, leading to changes in the target value function, so the value function can by non-stationary.

To deal with this non-stationarity, we need to break the correlation between the training data sequences. The use of non-i.i.d. (independent and identically distributed) data helps to achieve this. By randomly sampling, we can decorrelate the trajectories of each episode, thereby reducing the inter-step relationships. This leads to more stable updates, a significant difference and advantage over naive Q-learning. Another advantage is that we can reuse the samples. Since we're sampling randomly, there's a chance of getting repeated samples. This not only improves the efficiency of data use, but also reduces the pressure to collect new data.

Target Network in Deep Q-Learning

In the previous section, we discussed Q-learning with function approximation. Now, the question arises: which approximator should we choose to use? There are various methods to consider, including neural networks, decision trees, high-order functions, and more. Many studies have explored this topic, and through experiments, it has been found that using a deep neural network has the best performance in terms of results and data size. DQN uses a neural network as the function approximator.

The loss function of DQN is shown below:

$$J = \mathbb{E}\left[\left(R + \gamma \max_{a \in \mathcal{A}(S')} \hat{q}\left(S', a, w_T\right) - \hat{q}(S, A, w)\right)^2\right] \tag{47}$$

As we can see, there are two parameters w and w_T in this formula. In tabular Q-learning, it can be computed using a straightforward method. However, when using a neural network, we need to calculate the gradient for updates. The $\hat{q}(s, a, w)$ can be calculated straightforwardly,but the gradient of $\hat{q}(s', a, w)$ presents a challenge.

To handle the problem, In Deep Q-learning, we set

$$y \doteq R + \gamma \max_{a \in \mathcal{A}(S')} \hat{q}\left(S', a, w\right) \tag{48}$$

and assume w in the formula to be fixed.

In the formation (47), $J(w)$ has only one parameter variable w, making it easier to obtain the gradient related to w. And the fixed network called the target network, the continuously updated network is the main network. The gradient is denoted as follows, the formula is the same as in previous section.

$$\nabla_w J = \mathbb{E}\left[\left(R + \gamma \max_{a \in \mathcal{A}(S')} \hat{q}\left(S', a, w_T\right) - \hat{q}(S, A, w)\right) \nabla_w \hat{q}(S, A, w)\right] \tag{49}$$

So in DQN, we maintain two Q networks with different parameter vectors. For some iterations, we only update the main network $q(s, a, w)$, while the

target network, updates its parameters after a certain number of interations by copying the main network's parameters. This process shown in Figure 31, which is updating towards the target network's value for stable learning. After the Q network undergoes updates, acquiring new information about the environment and policy, we use this information to update the Q target network, guiding the next phase of learning.

If we don't employ the Q target network and use the same network for both the target and Q values, the process becomes unstable because the target and Q values exhibit a strong correlation. It's akin to chasing a moving target, after updating once, the target shifts slightly, especially with neural networks, leading to unstable behavior. This is why we utilize the target network: to prevent unstable updates and maintain a stable target over a defined period. However, the target network also needs to be updated to incorporate new information learned by the main network. Therefore, after the main network undergoes a specified number of updates, we copy the main network's parameters to the target network.

The objective is to train the Q main network towards the Q target network. We sample experiences from the replay buffer and use them to calculate target value y. This enables us to obtain the gradient of the loss function and subsequently update the main network's parameters. This process is repeated until we reach the interval at which the target network needs to be updated. These steps continue until a certain number of iterations are completed or the loss falls below a predefined threshold, the complete process will be discussed in the following section.

Deep Q-learning

In the previous sections, we discussed about Q-learning, neural networks, value function approximation, and experience replay. Now that we have all of these components in place, we're ready to put them together to form Deep Q-Learning. If you've followed this chapter carefully, you should now have a solid foundation for understanding the structure of Deep Q-Learning.

In this section, we'll learn how to build Deep Q-Learning by integrating all of these components. The first step is use a neural network as a value function approximator to estimate the Q-value, compare with Q-learning, DQN use deep network to replace the Q-table, the process shown in Figure 31.

Then add the target network structure for stable learning, the basic idea is to create a separate network, called the target network, which is a delayed copy of the original Q-network. This target network is not updated as frequently as the Q-network, thus introducing a degree of temporal stability into the learning process. The update frequency of the target network is a critical parameter. Instead of updating the target network at every step, it is updated periodically. This delayed update ensures that the target Q-values used in the learning process have some temporal consistency. Typically, the update involves copying the weights of the main network into the target network at fixed intervals.

To stabilize and improve learning efficiency, DQN employs experience replay.

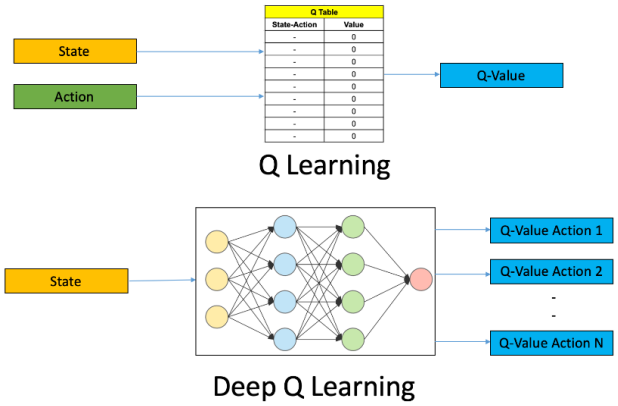

Figure 31: Compare with Q-learning and DQN

This involves storing and randomly sampling experiences from the agent's past interactions, breaking the temporal correlations in the data and making the learning process more robust. And the reuse of past experience increases data efficiency, allowing the model to learn from a wider range of scenarios.

Here's the overall process, as shown in Figure 32 and algorithm 2, we initialize the parameters of the neural network and the environment, and then start the agent. At each step, the agent chooses an action based on an ϵ-greedy policy represented by a neural network that estimates Q values. We store the action taken, the reward received, the current state, and the next state in the replay buffer. Then, at each step, we randomly sample batches from the replay buffer and use the gradients of the batches to optimize the mean squared error between the Q-values predicted by the main network and the Q-values predicted by the target network. This serves as the Q-learning objective. After a certain number of iterations, we copy the parameters of the main network to the target net. Repeat the process until certain iterations are complete or the loss is below to a certain threshold.

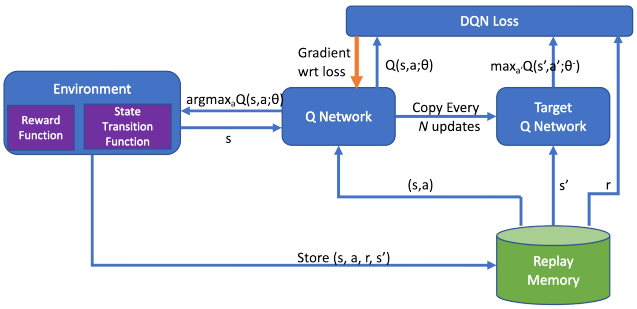

Figure 32: Diagram of Deep Q-learning

Algorithm 2 Deep Q-learning with Experience Replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights
Initialize target network weights $w^- \leftarrow w$
Initialize episode counter $n \leftarrow 1$
for episode $= 1, M$ **do**
  Initialize sequence $s_1 = x_1$ and preprocess sequence $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    Otherwise, select $a_t = \text{argmax}_a Q(\phi(s_t), a; w)$
    Execute action a_t and observe reward r_{t+1} and next state x_{t+1}
    Set $s_{t+1} = x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_{t+1}, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set target $y_j = r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; w^-)$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; w))^2$
  end for
  Update the target network weights every C steps: $w^- \leftarrow \tau w + (1 - \tau) w^-$
  Increment episode counter $n \leftarrow n + 1$
end for

We provide a simplified implementation of DQN algorithm using pytorch, and that is a basic implementation based on the pseudocode above, and if you want to use it, you need to modify it depend on your own environment and requirements.

```
import torch
import torch.nn as nn
import torch.optim as optim
import random
import numpy as np
import gym

# Define the Q-network model
class QNetwork(nn.Module):
    def __init__(self, input_size, output_size):
        super(QNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, output_size)

    def forward(self, state):
        x = torch.relu(self.fc1(state))
        x = torch.relu(self.fc2(x))
        return self.fc3(x)

# Initialize replay memory D to capacity N
replay_memory = []
replay_memory_capacity = 10000
```

```
24
25  # Initialize action-value function Q with random weights
26  num_actions = 4  # Replace with the actual number of actions in
        your environment
27  input_size =  state_size  # Replace with the actual size of your
        state
28  q_network = QNetwork(input_size, num_actions)
29  target_network = QNetwork(input_size, num_actions)
30  target_network.load_state_dict(q_network.state_dict())
31
32  # Other hyperparameters
33  epsilon = 0.1
34  gamma = 0.99
35  batch_size = 32
36  learning_rate = 0.001
37  num_episodes = 1000
38  max_steps_per_episode = 10000
39  update_target_network_freq = 100  # Update target network every C
        steps
40
41  optimizer = optim.Adam(q_network.parameters(), lr=learning_rate)
42
43
44  # Create an environment
45  env = gym.make ('CartPole -v1')
46
47  # Training loop
48  for episode in range(num_episodes):
49      state = env.reset()
50
51      for t in range(max_steps_per_episode):
52          # Epsilon-greedy action selection
53          if random.random() < epsilon:
54              action = random.randint(0, num_actions - 1)
55          else:
56              q_values = q_network(torch.FloatTensor([state]))
57              action = torch.argmax(q_values).item()
58
59          # Take action and observe next state and reward
60          next_state, reward, done, _ = env.step(action)
61
62          # Store transition in replay memory
63          replay_memory.append((state, action, reward, next_state,
                done))
64
65          # Sample random minibatch from replay memory
66          minibatch = random.sample(replay_memory, batch_size)
67          states, actions, rewards, next_states, dones = zip(*
                minibatch)
68
69          # Convert to PyTorch tensors
70          states = torch.FloatTensor(states)
71          actions = torch.LongTensor(actions)
72          rewards = torch.FloatTensor(rewards)
73          next_states = torch.FloatTensor(next_states)
74          dones = torch.FloatTensor(dones)
75
```

```
        # Update Q-network parameters using gradient descent
        q_values = q_network(states)
        target_q_values = target_network(next_states)

        targets = rewards + gamma * torch.max(target_q_values, dim
            =1).values * (1 - dones)
        selected_q_values = q_values.gather(1, actions.unsqueeze(1)
            )

        loss = nn.MSELoss()(selected_q_values, targets.unsqueeze(1)
            )

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Update target network parameters every C steps
        if t % update_target_network_freq == 0:
            target_network.load_state_dict(q_network.state_dict())

        if done:
            break

        state = next_state
```

2.6 DQN Variants and Rainbow

In this section, we will begin by addressing the overestimation problem identified in DQN. We'll analyze its manifestation through an example and explore methods to rectify it, leading us to the concept of double DQN. We will delve into the core idea behind double DQN and understand how it solves the overestimation problem.

Following that, we'll shift our focus to Dueling DQN, which introduces a novel network structure using the advantage function. This modification enhances the learning process, making it more efficient and robust.

Subsequently, we will introduce Rainbow, a comprehensive framework that combines six variants of DQN. using advanced techniques including prioritized experience replay, multi-step learning, distributional reinforcement learning, and noisy net, Rainbow constructs a new structure that can absorb all of their advantages. We will discuss each of these techniques individually to provide a clearer understanding of Rainbow.

2.6.1 Double DQN

Double Q-learning

As we learned in the previous section, Q-learning has many advantages, but people have found that it tends to overestimate the true Q value when applied to practical problems. In other words, there is a systematic error in estimating the Q value. With an increase in training iterations, the value estimation continues to rise, causing the estimated action-state value to be consistently higher than the true value. This problem exists in all naive Q-learning algorithms, and Double Q-learning is proposed to address this issue.

Before delving into the topic of Double Q-learning, let's consider how systematic overestimation arises through a simple and specific example. I'll illustrate this issue with a static example to help you understand how the max operator influences the final result, which is a crucial problem in Q-learning and serves as the basis for Double Q-learning.

Overestimation phenomenon in the maximum height estimation problem

Imagine we want to measure the maximum height of 100 people, assuming that all 100 people have the same heights of 160cm. We use a height measuring gauge with a standard deviation of 1cm, meaning it can be off by plus and minus 1cm (symmetrical noise on the measurement). We measure each person's height and record the maximum height. Regardless of how many times we measure, the result is consistently larger than 160cm due to the noise in the measuring process. Essentially, we sample from the noise distribution and select the highest value, which is always greater than zero. Thus, under these conditions, the maximum is biased toward overestimating the true value.

To address the overestimation problem, we introduce a symbolic process: let X^i denote the height of the i'th person, and set $Y = max_i X^i$. The idea is to measure the same person twice, denoted as X_1^i, X_2^i. Since the height measuring gauge produces independent noise, the two results are independent, each with its own independent noise. In the first measurement, we identify the person with the highest height, denoted as $n = argmax_i(X_1^i)$. Then, we use the second measurement of that person as the estimate of the maximum, denoted as $Y = X_2^n$. For example, if in the first measurement, the 30th person is found to have the highest height, we then take the second measurement of that specific person (the 30th person) as the final result. This approach ensures that the estimate of the maximum is not systematically higher than the true value, as the second measurement of a specific person is independent and may be higher or lower than the true value.

In summary, this process involves using two estimators: the first to determine which person has the highest height, and the second to select that person's second measurement of height.

In both Q-learning and DQN, the same problem arises as illustrated in the

example. In the Q-learning sections, the target for the network to update is selected as the maximum value of $Q(s,a)$. And the maximum value of $Q(s,a)$ is not only used as the estimated value of the state but also used to select an action, as shown in the formula:

$$Q_{t+1}\left(s_t,a_t\right)=Q_t\left(s_t,a_t\right)+\alpha_t\left(s_t,a_t\right)\left(r_t+\gamma\max_a Q_t\left(s_{t+1},a\right)-Q_t\left(s_t,a_t\right)\right) \tag{50}$$

Rewrite the target value as:

$$Y_t^{\mathrm{Q}}\equiv R_{t+1}+\gamma\max_a Q\left(S_{t+1},a\right) \tag{51}$$

This max operation leads to an overestimation of action-state values, and the overestimate phenomenon is common in practice due to the inherent structure of the algorithm. The idea to fix it remains the same that is to use two estimators, one for action selection (picking the action with the maximum action-state value) and the other for action-state value estimation. Formally, two independent Q-networks, Q^A with parameters w_a and Q^B with parameters w_b, are trained with disjoint datasets sampled from experience. When training Q^A, instead of just using Q^A to update itself, Q^B is used to choose the maximum action value, Similarly, Q^B is updated in the same way. The process is denoted by the following formulas:

Double Q-learning update:

$$a^*=\arg\max_a Q^A\left(s',a\right) \tag{52}$$

$$Q^A(s,a)=Q^A(s,a)+\alpha(s,a)\left(r+\gamma Q^B\left(s',a^*\right)-Q^A(s,a)\right) \tag{53}$$

$$b^*=\arg\max_a Q^B\left(s',a\right) \tag{54}$$

$$Q^B(s,a)=Q^B(s,a)+\alpha(s,a)\left(r+\gamma Q^A\left(s',b^*\right)-Q^B(s,a)\right) \tag{55}$$

In the maximum height estimation example, overestimation is caused by noise, but in DQN or Q-learning, other factors such as environment non-stationarity and function approximation may contribute to it. And the new update mechanism in double Q-learning can generally address all these factors.

Double DQN

In traditional DQN, the Q-value is estimated using the maximum Q-value over all possible actions for the next state. This can lead to overestimation,

especially in situations where the maximization operation is applied to noisy or exaggerated Q-values.

The idea of Double DQN is similar to Double Q-learning—disjointing the max operation in the Q-value update into value evaluation and action selection. Instead of introducing additional networks, as in the case of Double Q-learning, Double DQN makes optimal use of the existing main network and target network. The main network is used to select actions according to a greedy policy under a state, and the target network is used to evaluate the value of the picked action. Double DQN Maintain two separate neural networks, often referred to as the main network (refer to Q^B in Double Q-learning, which is used to select action) and the target network (refer to Q^A in Double Q-learning which is used for action evaluation). Other parts of Double DQN is the same with DQN, including algorithm process and parameters update that the target network still requiring a periodic copy from the main network. The only difference is the target as shown below:

$$Y_t^{\text{DQN}} \equiv R_{t+1} + \gamma \max_a Q\left(S_{t+1}, a; \boldsymbol{w}_t^{'}\right) \tag{56}$$

$$Y_t^{\text{DoubleDQN}} \equiv R_{t+1} + \gamma Q\left(S_{t+1}, \operatorname{argmax} Q\left(S_{t+1}, a; \boldsymbol{w}_t\right), \boldsymbol{w}_t^{'}\right) \tag{57}$$

In this formula, the network with parameters $\boldsymbol{w}_t$ is used to select actions, refer to Q^B in Double Q-learning, and the main network in DQN, and the network with parameters $\boldsymbol{w}_t^{'}$ is used to evaluate actions, refer to Q^A in Double Q-learning, and the target network in DQN.

In the paper proposing Double DQN, several experiments compared Double DQN with DQN have demonstrated that Double DQN exhibits better performance in the value estimation accuracy and policy efficiency. As expected, it avoids the systematic overestimation problem, making the learning process more stable and reliable. And the experiments' result in the "Deep Reinforcement Learning with Double Q-learning" paper compare DQN and Double DQN show in Figure 33, further details of the experiments can be found in that paper.

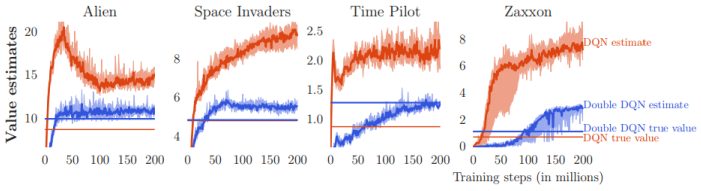

Figure 33: Compare DQN and Double DQN on Atari Games

2.6.2 Dueling DQN

Advantage Function

To introduce the idea behind Dueling DQN,let's review the basic problem that reinforcement learning need to solve: the sequential decision problem. The goal is to determine the optimal policy that maximizes cumulative rewards within a given environment, while adhering to predefined rules. To this end, we define V(s) as the state value, estimating the value of each state by accumulating rewards from that state to the endpoint, denoted as $V(s)$. However, in an environment where the Markov Decision Process (MDP) model is unknown and the actions leading to particular states are uncertain, we introduce the action-value function $Q(s, a)$. This function estimates the value of each action under a given state. Based on $Q(s, a)$, we can choose the action that maximizes the Q-value. $Q(s, a)$ encapsulates both the state value and the utility of an action, mixing the two values.

If we use only the Q-value to select actions, a problem arises. It is conceivable that the state value comprises a significant portion of the Q-value. In such cases, the specific action taken may not significantly affect the outcome, resulting in a waste of time and computational resources for each state-action pair. For example, consider the scenario shown in Figure 34 for the Enduro game, it doesn't matter to move left or right when there are no other cars nearby. The action becomes critical only when a collision is going to happen. In many situations, however, the action can have a significant impact on the return and subsequent state. In some cases, a single action can determine the final outcome. Therefore, we need to find a method that preserves the benefits of the Q-value while minimizing redundant computations. Moreover, the relationship between state and action cannot be adequately expressed by the Q-value alone, so it is important to address this aspect in the network.

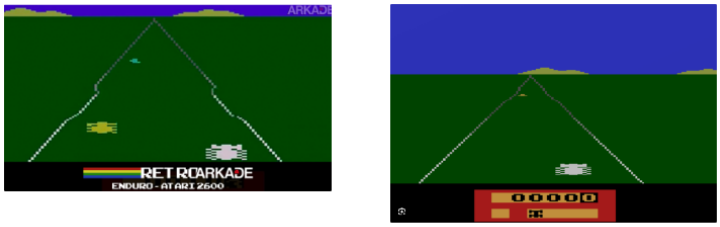

Figure 34: Scenario in the Enduro Game

To address these issues, we introduce the advantage function, defined as the subtraction of the state value from the Q-value, $A(s, a) = Q(s, a) - V(s)$. This function represents the gain of performing a certain action under a given state. After the action, the larger the advantage value, the more advantageous the action.

Let $Q^\star(s, a) = \max_\pi Q_\pi(s, a)$, and $V^\star(s) = \max_\pi V_\pi(s)$, then $A^\star(s, a) =$

$Q^\star(s,a) - V^\star(s)$ Then we can get: $Q^\star(s,a) = V^\star(s) + A^\star(s,a)$.

However, this equation is not identifiable. Adding a constant to $Q^\star(s,a)$ and subtracting the same constant from $A^\star(s,a)$ (eg. $Q^\star(s,a) = V^\star(s) + c + A^\star(s,a) - c$). This can lead to instability in the training process, as both the value function and the advantage function change to the same extent, and in opposite directions, then you can get the same output . Learning accurate value function and advantage function under such conditions becomes challenging. To address this, we introduce a third term into the equation, which is max, then the new equation is below.

Theorem 1:

$$Q^\star(s,a) = V^\star(s) + A^\star(s,a) - \max_a A^\star(s,a) \tag{58}$$

The term $\max_a A^\star(s,a)Z$ represents the maximum advantage value across all actions for a given state s. Adding this term to the value function helps capture the maximum advantage that can be obtained from the state. This is essential for accurately estimating the intrinsic value of being in a given state, regardless of the action chosen. Besides, adding this term is beneficial to the stability in the training process, since it can avoid value function and advagentage function to change in opposite direction with the same constant to get the same result.

why can we add the maxA(s,a) to the equation?

Theorem 2:

$$V^\star(s) = \max_a Q^\star(s,a) \tag{59}$$

The definition of the optimal advantage function:

$$A^\star(s,a) = Q^\star(s,a) - V^\star(s) \tag{60}$$

Maximize both sides of this equation with respect to a, then we get:

$$\max_a A^\star(s,a) = \max_a Q^\star(s,a) - V^\star(s) \tag{61}$$

With theorem 2:

$$\max_a A^\star(s,a) = 0 \tag{62}$$

Dueling DQN

In DQN, after obtaining input vectors, they are first sent to the convolutional layers to extract features. Then, through fully connected layers and a softmax operation, output vectors are generated that represent the Q-values for each action under a given state.

In Dueling DQN, the network structure incorporates the advantage function. It uses the network $A(s,a;w_a)$ to approximate $A^\star(s,a)$, and use network $V(s;w_v)$ to approximate $V^\star(s)$. The goal is to capture the relationship between the state value and the gain produced by an action, thus providing better es-

timates for both state and action. To achieve this, the two networks must be combined while maintaining the shape of the final output vector during the training process.

Considering Theorem 1, where the Q-value is the sum of the state value and the advantage value, the output of the Q network becomes the sum of the value network and the gain network, formulated as $Q(s,a;w) = V(s;w_v) + A(s,a;w_a) - \max_a A(s,a;w_a)$.

The network structure is shown in Figure 35. After the input vector passes through convolutional layers and acquires features, these features are separately fed into two fully connected layers to obtain advantage vectors and state values. These are then combined by adding them together, subtracting the maximum of the advantage vectors, resulting in the final vector where each element represents the value of an action. The action with the highest value is selected to execute in the environment.

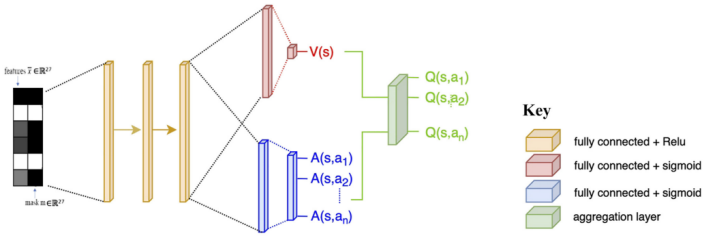

Figure 35: Structure of Dueling DQN

In practice, the max operator is often replaced by an average, which improves stability because the target moving toward the average is more stable than the maximum. Subtracting the mean value preserves the relative rank of the advantage values, thus preserving the final decision about which action to take. Formulated as $Q(s,a;w) = V(s;w_v) + A(s,a;w_a) - mean_a A(s,a;w_a)$.

The input and output of DQN and Dueling DQN are the same, and their training processes and functions are also the same, with the goal of approximating $Q^\star(s,a)$. The main difference lies in the network structure, with Dueling DQN introducing the advantage network to identify the gains brought by actions and the existing state value for all actions. This change leads to faster convergence and better policy in Dueling DQN. A comparison between DQN and Dueling DQN is shown in Figure 36.

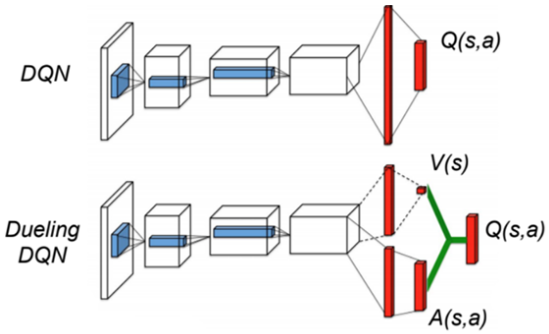

Figure 36: Compare Dueling DQN with DQN

2.6.3 Distributional DQN

Distributional Reinforcement Learning (Distributional RL) is a paradigm in reinforcement learning that focuses on estimating the distribution of accumulated rewards, rather than just their expected values. Traditional RL aims to estimate the expectation of the action state value (Q-value), which is essentially the expectation of the sum of the rewards. However, in certain scenarios, understanding the distribution of action-state values is critical, especially in risk-sensitive tasks such as those found in finance.

In distributional RL, the goal is to capture the full range of outcomes and their associated probabilities, providing more detailed information about the uncertainty in the probability distribution. So it provides a more nuanced understanding of the returns, allowing the agent to make decisions considering both the central tendency and the variability of outcomes.

Let Z be a random variable representing the return of an action under a given state, with its expectation being the Q-value. $Z(s, a)$ is the distribution of the action value for a action under state s, and is referred to as the value distribution. The distributional Bellman equation is given by

$$Z(s, a) \stackrel{D}{=} R(s, a) + \gamma Z\left(S', A'\right) \tag{63}$$

Distributional DQN (C51) is an extension of the traditional Deep Q-Network (DQN) that incorporates distributional reinforcement learning principles. C51 demonstrates how incorporating distributional principles into deep reinforcement learning algorithms can lead to improved performance and a more comprehensive understanding of uncertainty in the learning process. C51 models the entire distribution of returns as a categorical distribution. We'll take an an overview of the C51 algorithm to understand how distributional RL works. An overview about difference between DQN and C51 is shown in Figure 37 and the algorithm description is shown in algorithm 4.

Instead of estimating the expected value (as in traditional RL), C51 aims to learn the probabilities associated with different discrete outcomes. C51 dis-

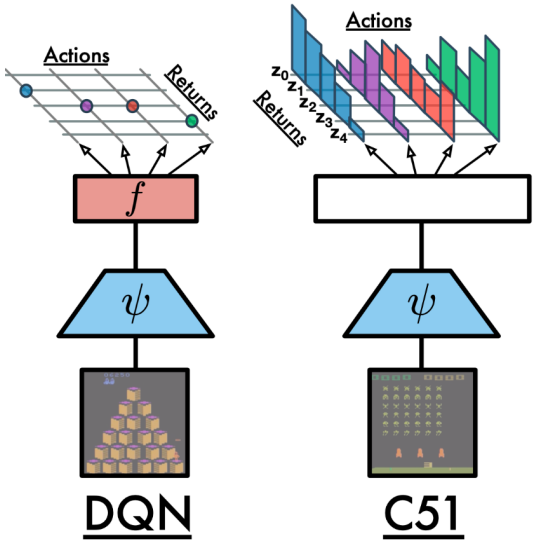

Figure 37: Compare With DQN And C51

Algorithm 3 C51 Algorithm

Ensure: Optimal Q-value distribution function $Z(s, a, \theta)$

Initialize neural network with parameters θ
Initialize target network with parameters θ^-
Initialize distribution support z_i for $i = 1, ..., N$
Initialize replay buffer D
for episode $= 1, M$ **do**
    Initialize episode s_1
    for timestep $= 1, T$ **do**
        With probability ϵ select a random action a_t
        Otherwise, select $a_t = \text{argmax}_{a'} Z(s_t, a', \theta)$
        Execute action a_t, observe reward r_t and next state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in D
        Sample a minibatch of transitions (s_i, a_i, r_i, s_{i+1}) from D
        Compute target distribution $T_i = R_i + \gamma Z(s_{i+1}, a', \theta^-)$
        Compute temporal difference error $\delta_i = T_i - Z(s_i, a_i, \theta)$
        Update priorities in replay buffer D based on δ_i
        Update neural network parameters θ using gradient descent on δ_i
        Update target network parameters θ^- periodically
    end for
end for

cretizes the returns into a set of bins and learns the probability mass function for those bins. But for continuous range, it is difficult to learn the probabilities, so we define a set of discrete values mapping to these bins called "atoms", denoted as discrete support z with N_{atoms}, define as:

$$z^i = v_{\min} + (i-1)\frac{v_{\max} - v_{\min}}{N_{\text{atoms}} - 1} \tag{64}$$

Dividing the return range by these bins, then we just need to learn the probability of each atom.

The probability mass of each atom, denoted by $p_\theta^i(S_t, A_t)$, the distribution d_t at time t, defined as $d_t = (\boldsymbol{z}, \boldsymbol{p}_\theta(S_t, A_t))$.

The value distribution function represents the distribution of action values for a given state-action pair. This distribution is modeled as a categorical distribution with probabilities assigned to each atom.

For updating, C51 focuses on updating the distribution of each atom based on the distributional Bellman equation. The goal is to update the distribution to approximate the actual distribution of returns. The loss function measures the gap between the predicted categorical distribution and the target distribution derived from the distributional Bellman equation denoted as:

$$d_t' \equiv \left(R_{t+1} + \gamma_{t+1}\boldsymbol{z}, \quad \boldsymbol{p}_{\bar{\theta}}\left(S_{t+1}, \bar{a}_{t+1}^*\right)\right) \tag{65}$$

$$\bar{a}_{t+1}^* = \text{argmax}_a\, q_{\bar{\theta}}(S_{t+1}, a) \tag{66}$$

And using the Kullbeck-Leibler divergence to measure the gap between d_t and d_t', The loss function is represented as $D_{\text{KL}}(\Phi_{\boldsymbol{z}} d_t' \| d_t)$, where Φ is the L2 projection of the target distribution onto the substrate z, and KL is the Kullback-Leibler divergence.

Distributional RL allows for a more complete representation of the return space, avoiding the cancellation of important information by expected values. It provides a more accurate and robust learning process, as demonstrated in experiments such as Atari games. In summary, the advantages of Distribution RL and algorithms like C51 lie in their ability to provide a more nuanced understanding of uncertainty, handle risk-sensitive scenarios, and improve the stability and robustness of learning processes in comparison to traditional RL methods.

2.6.4 Rainbow

In addition to the two DQN variants mentioned above, there are other variants, each of which improves the performance of DQN by addressing specific drawbacks. Considering these variants together, it becomes clear that they share a common foundation - the Q-learning framework and deep neural networks. It is possible to integrate all these improvements into a unified structure, and this is what Rainbow does. Rainbow combines six important and complementary variants, including the previously discussed Double DQN and Dueling DQN, along with Prioritized Experience Replay, Multi-Step Learning, Distributional DQN, and Noisy DQN. In this section, we will introduce the remaining four DQN variants, explore their benefits, and then look at how they can be seamlessly integrated, and each component's contribution to the overall results.

Prioritized Experience Replay

While Experience Replay effectively addresses the issue of relevant transition sequences and reduces the need for the agent to constantly interact with the environment for training data, it introduces another drawback - uniform sampling, which can slow down updates in certain scenarios. For example, in a robot arm grasping scenario, during the initial random movements, the probability of successfully grasping the target is minimal. The replay buffer contains many error samples, and uniform sampling from this buffer slows down the update process. Intuitively, some samples are more critical for the agent to learn useful information. Giving more weight to these samples would improve the training process, motivating the concept of Prioritized Experience Replay.

The question then arises: how should we assign different weights to these transitions? Frequent transitions already influence the network updates, resulting in a small gap between the current estimate and the estimate of the next step. On the other hand, infrequent transitions require substantial adjustments to the network parameters to adapt to new events, making them more informative. The TD error can express this information - it indicates the difference between the current estimate and the transitions. These transitions with high TD errors should be given more weight to reduce the error.

To implement this idea, we store the TD-error along with each transition in the replay buffer and assign a priority to these transitions. Based on the priority, we assign different probabilities to these transitions.

There are two ways to define the priority of transition i, denoted as p_i, the first way is $p_i = |\delta_i| + \epsilon$, the second way is $p_i = 1/rank(i)$, $rank(i)$ represents the rank of transition i according to $|\delta_i|$ in the replay buffer. Both methods are monotonic to $|\delta_i|$. Based on the priority of the transition, we can define the probability of sampling transition i:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{67}$$

This probability calculation ensures that high TD error transitions are sampled more frequently than low TD error transitions. Transitions with low TD

error are also sampled to avoid missing information and overfitting.

Similar to uniform sampling from the replay buffer, which changes the distribution of the replay buffer and introduces bias, we must multiply by the importance sampling weight before updating the network parameters. The importance sampling weight is defined as:

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^{\beta} \tag{68}$$

β is an adjustable hyperparameter, and the probability of the sampling transition and the weight have a negative correlation, correcting the bias.

The algorithm process of DQN with Prioritized Experience Replay is shown in algorithm 3, the algorithm adapts traditional DQN by introducing prioritization of experiences based on their importance in the learning process. This adaptation aims to focus learning on more informative experiences, potentially leading to faster and more effective training.

Algorithm 4 DQN with Prioritized Experience Replay

Initialize replay memory $\mathcal{D}$ with capacity N
Initialize Q-network with weights θ
Initialize target Q-network with weights $\theta^- = \theta$
Initialize prioritized replay buffer hyperparameters
for episode = 1 to M **do**
    Initialize state s
    for timestep = 1 to T **do**
        With probability ϵ select a random action a, otherwise $a = \arg\max_{a'} Q(s, a'; \theta)$
        Execute action a, observe reward r and next state s'
        Calculate TD-error: $\delta = r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)$
        Update priorities: $p = |\delta| + \epsilon$
        Store transition (s, a, r, s') and priority p in $\mathcal{D}$
        Sample a minibatch of transitions $\mathcal{B}$ from $\mathcal{D}$ based on priorities
        for each transition (s_i, a_i, r_i, s_i') in $\mathcal{B}$ **do**
            Calculate target: $y_i = r_i + \gamma \max_{a'} Q(s_i', a'; \theta^-)$
            Update Q-network weights using TD-error: $\theta = \theta - \alpha \delta_i \nabla_\theta Q(s_i, a_i; \theta)$
        end for
        Update target Q-network: $\theta^- = \tau\theta + (1 - \tau)\theta^-$
    end for
end for

Multi-Step Learning

DQN use one-step reward to update the estimation of the current state, multi-step variant of DQN use multi-step reward to calculate the loss, and in this way, the learning process can be faster and more stable. The accumulated reward can be represented by the following formulas:

Accumulated Reward

$$G_t = R_t + \gamma \cdot G_{t+1} \tag{69}$$

$$G_t = R_t + \gamma \cdot R_{t+1} + \gamma^2 \cdot G_{t+2} \tag{70}$$

$$G_t = R_t + \gamma \cdot R_{t+1} + \gamma^2 \cdot R_{t+2} + \gamma^3 \cdot G_{t+3} \tag{71}$$

$$G_t = \sum_{i=0}^{m-1} \gamma^i \cdot R_{t+i} + \gamma^m \cdot G_{t+m} \tag{72}$$

The m-step TD target for DQN is :

$$y_t = \sum_{i=0}^{m-1} \gamma^i \cdot R_{t+i} + \gamma^m \cdot \max_{a'} Q'\left(S_{t+m}, a'\right) \tag{73}$$

The loss function is:

$$J = \left(\sum_{i=0}^{m-1} \gamma^i \cdot R_{t+i} + \gamma^m \max_{a'} Q'\left(S_{t+m}, a'\right) - Q\left(S_t, A_t\right) \right)^2 \tag{74}$$

DQN with Multi-Step Learning extends the traditional Deep Q-Network (DQN) algorithm by incorporating the concept of multi-step returns. Multi-step learning considers not only the immediate reward and the next state, as in standard DQN, but also includes additional steps into the future.

The multi-step returns is calculated by summing the returns over the next n steps, discounted by the appropriate factor. It provides a more comprehensive estimate of the cumulative return over a longer period of time.

Including multiple steps in the return can help reduce the variance in Q-value estimates, leading to more stable and accurate learning. And it also allow the algorithm to bootstrap information over multiple time steps, allowing for more efficient exploration of the state-action space.

Noisy Net

The core concept behind Noisy DQN is the introduction of parametric noise into the network weights to enhance exploration. While the epsilon greedy policy provides exploration capabilities, it may not be sufficient for extensive exploration, especially in scenarios that require numerous steps to achieve positive

rewards. Previous methods have often been used for small state-action spaces or simple approximators. However, in the context of neural networks or more complex scenarios, a novel approach is required to address the exploration challenge and accelerate the learning process.

Considering a linear layer of a neural network, expressed as $y = wx + b$, parametric noise is added to it, in which ϵ is a non-zero vector with a stable distribution, denoted below:

$$\boldsymbol{y} = (\boldsymbol{b} + \mathbf{W}\boldsymbol{x}) + \left(\boldsymbol{b}_{\text{noisy}} \odot \epsilon^b + (\mathbf{W}_{\text{noisy}} \odot \epsilon^w)\,\boldsymbol{x}\right) \tag{75}$$

As we know, trade-off between exploration and exploitation is critical problem in RL, while traditional exploration strategies such as ϵ-greedy can introduce limited randomness, noise injection provides a more continuous and adaptive form of exploration, this allows the agent to explore a wider range of actions and states, potentially discovering more optimal strategies. And traditional exploration methods, like ϵ-greedy, are inherently discrete, they randomly choose an action with some probability. In contrast, noise injection introduces parameterized noise into the entire weight space of a neural network, allowing continuous and more extensive exploration within the action space, this is especially valuable in complex environments.

Besides that, Noise helps break the dependence on local optima. By introducing noise into the weights, the neural network is encouraged to explore a broader region of the parameter space, preventing it from getting stuck in suboptimal solutions and facilitating the discovery of better global solutions. In environments with uncertainty and variability, noise in the weights allows the agent to adapt more effectively. It encourages the agent to try new, potentially beneficial actions and helps it cope better with changing conditions.

In summary, noise introduces flexibility and proactive exploration, allowing the agent to learn more efficiently. It can accelerate the learning process by providing a more dynamic and adaptive exploration strategy.

So how to design the noise is important, it can influence the learning effectiveness, parameterized noise models are commonly used, where the noise parameters are treated as learnable parameters in the network, this allows the network to adaptively adjust the intensity and characteristics of the noise as it learns, we'll introduce several noise design strategies.

Factorized Gaussian Noise introduces Gaussian noise with learnable parameters for each weight, this allows for independent noise for each weight, and the network learns how much noise to apply to different parts of the model.

Ornstein-Uhlenbeck Process uses a stochastic process such as Ornstein-Uhlenbeck to generate correlated noise, this type of noise can capture dependencies between different weights and allow for more structured exploration.

Annealed noise reduces the intensity of the noise over time, this can be useful to balance exploration and exploitation, starting with more exploration in the early stages of learning.

Adaptive noise lets the noise intensity adapt to the level of uncertainty in the environment or the agent's experience, that allows the agent to explore more

when uncertainty is high.

The key is to design noise that introduces enough randomness for exploration while ensuring that the learning process remains stable and converges to meaningful solutions. The specific design choices often depend on the characteristics of the learning task and the environment.

Rainbow

Rainbow combines the advantages of different DQN variants into a unified architecture. It uses DQN as the basic model and incorporates modifications such as TD target adjustment through multi-step learning, prioritized sampling from replay memory based on TD error, using the C51 architecture to estimate the distribution of the entire return space, adding parameterized noise to the network weights for more extensive and faster exploration, and incorporating an advantage function into the network structure. It also adopts the modification of double DQN to reduce overestimation.

While Rainbow does not introduce a novel concept, it systematically integrates the strengths of advanced techniques in deep reinforcement learning. It is well structured and implemented in an engineering sense.

The experiments on Atari games compare Rainbow with several DQN variants published in "Rainbow: Combining Improvements in Deep Reinforcement Learning" is shown in Figure 38, from the result, we can see that the performance of Rainbow is much better than any DQN variants, not only in learning speed, but also in the final score. And from the training process shown in the curve ascending process, we can say Rainbow exhibits more stable learning performance.

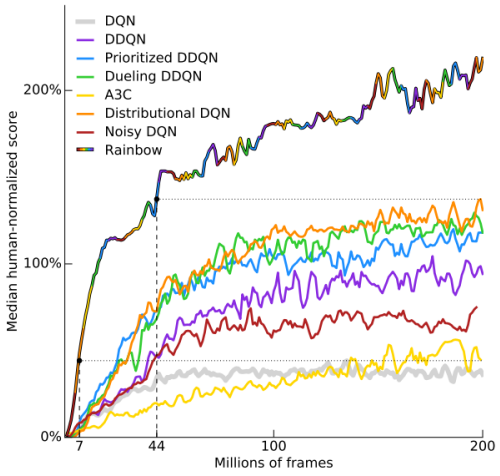

Figure 38: DQN Variants On Atari Games

To understand the contribution of each component in the rainbow, the researchers ran experiments where they removed each component from the rain-

bow and compared their performance. The result of the experiment is shown in Figure 39, prioritized replay and multi-step learning are the two most important components, as removing either of them results in a significant drop in median performance.

Distributed Q-learning ranks closely behind the techniques previously associated with agent performance. It is worth noting that there is no obvious difference in early learning, with distribution-ablation performing as well as the rainbow. However, without distribution, the agent's performance begins to fall behind.

Noisy network also has an important influence in the final result, but we don't observe an obvious decline in the final result when remove dueling network and double Q-learning, and for finding out their subtle difference in different environment, researchers did more experiments, you can see more details in the original paper of Rainbow.

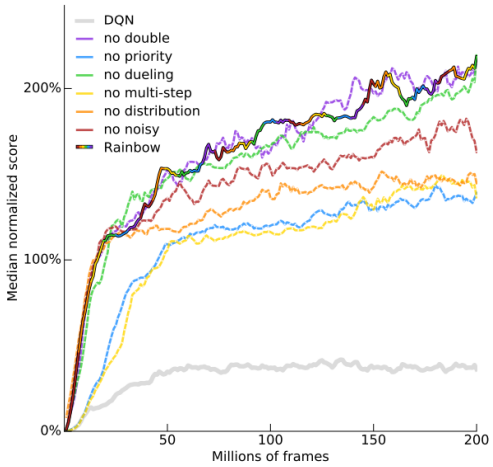

Figure 39: Remove Component From The Rainbow

Experimental results have shown that Rainbow DQN often outperforms baseline RL algorithms in a variety of tasks, demonstrating its effectiveness in practical applications.

In summary, all the advantages of individual DQN variants are combined in Rainbow, so it can solve all the problems mentioned in individual DQN variants. The methods proposed in the DQN variants used in Rainbow enhance the learning capabilities of the agent and improve its generalization to new and unseen environments. Moreover, Rainbow DQN enables agents to tackle complex tasks with multiple obstacles, sparse rewards, and long-term dependencies. It allows agents to learn effective strategies for dealing with challenging and uncertain environments. And Rainbow improves sample efficiency, allowing agents to learn from experience more effectively and accelerate the learning process. Rainbow DQN addresses issues such as overestimation bias and exploration-exploitation

tradeoffs, resulting in more stable and robust training of agents.

Overall, all applications suitable for DQN and it's variants are also suitable for Rainbow, including games, robotics and control, and finance and trading and recommendation system and so on. The application of Rainbow DQN contributes to advancing the capabilities of agents in handling complex decision-making tasks and navigating uncertain environments, making it a valuable tool in the field of reinforcement learning.