Reinforcement Learning: Theory and Applications

by
DONG,Hanwen

A report
submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
The Chinese University of Hong Kong

November in 2022
Supervisor: Prof. John C.S.Lui

Abstract

Reinforcement learning (RL) refers to the process by which an agent learns behaviors through interactions with a given environment. This report discusses the basic framework of RL and the principles of core algorithms. The details are as follows.

First, we introduce the basic concepts about RL. We briefly introduce what RL is, how it differs from supervised learning and some industrial applications in RL. After that, We describe the framework of the RL, including value functions, expected returns, Markov decision processes and other key elements in RL. Additionally, we describe in detail the principles of the algorithms in RL that have learned in this term. Dynamic programming, Monte Carlo methods, temporal difference (TD) learning and deep Q-network will be covered in this section.

Several experiments are conducted based on the above algorithms and some experimental results are presented in experiment section. We apply dynamic programming and Monte Carlo methods to an environment called frozen lake, which is a 4×4 maze, where we must control the agent to pick up all the rewards and get to the termination. Dynamic programming and Monte Carlo methods can achieve great results in this environment. TD learning and deep Q-network are applied to another environment called cart pole, where we need to keep the pole's balance as long as possible. The results indicate that TD learning does not work well if the environment states are continuous, whereas deep Q-network far outperforms it.

The plan for next term is presented in the last section. I plan to do some research on AlphaGo Zero, a famous RL system. I am going to learn its structure, try to implement it and use it to play a game called Gobang or Five in a Row.

Contents

Abstract **i**

1 Introduction **1**

2 Background **1**

2.1 Basic Framework of Reinforcement Learning 1

2.1.1 Setting 2

2.1.2 Returns 2

2.1.3 Markov Property and Markov Decision Processes 3

2.1.4 Value Functions 3

2.1.5 Optimal Value Functions 4

2.2 Related Algorithms 5

2.2.1 Dynamic Programming 5

2.2.2 Monte Carlo Methods 5

2.3 Software Tools 6

3 What I have learned **6**

3.1 Dynamic Programming 6

3.1.1 Policy Iteration 6

3.1.2 Value Iteration 8

3.2 Monte Carlo Methods 9

3.3 Temporal Difference Learning 10

3.3.1 SARSA Algorithm 11

3.3.2 Q-learning 12

3.3.3 On-Policy Algorithm and Off-Policy Algorithm 12

3.4 Deep Q-Network 12

3.4.1 Experience Replay . . . 13
3.4.2 Target Network . . . 14

4 Experiments **14**

4.1 Environment Setting . . . 14
4.1.1 Frozen Lake . . . 14
4.1.2 Cart Pole . . . 15
4.2 Dynamic Programming . . . 16
4.2.1 Results for Policy Iteration . . . 17
4.2.2 Results for Value Iteration . . . 18
4.3 Monte Carlo Methods . . . 19
4.4 SARSA Algorithm . . . 20
4.5 Q-Learning . . . 21
4.6 Deep Q-Network . . . 22

5 Plan for Next Term **23**

References **24**

1 Introduction

The process of figuring out how to behave in a certain environment in order to maximize a numerical reward signal is known as reinforcement learning (RL). The learner is not given specific instructions; instead, it must try through experiments to determine which action yields the greatest rewards. In the most challenging circumstances, actions might affect not just the current reward but also all future benefits. Sequential decision-making, which involves selecting the order of actions to do in a particular environment in order to maximize rewards based on prior experience, is the central topic in RL [1].

Reinforcement learning differs from supervised learning, which has been the major focus of much recent research in machine learning, statistical pattern recognition, and artificial neural networks. Following along with examples supplied by an external supervisor is what it means to learn under supervision. Although this is an important sort of learning, it cannot be learned through interaction. People would expect an agent in an unknown area to be able to draw lessons from its own experience. This is what RL mainly focuses on.

The combination of RL with deep learning has become increasingly popular as well. This combination, known as deep RL, is especially helpful in high-dimensional state-space problems. Deep RL overcomes the limitation that previous RL could only be used in low dimensional problems. Thus, we are able to apply reinforcement learning to more complex environments.

RL or deep RL have a broad variety of applications. In fields of image recognition, Mnih et al. [2] proposed the recurrent attention model (RAM) for image classification and object identification, which focuses on a specified series of areas or places from an image or video. The use of policy search for visual object recognition was proposed by Mathe et al. [3]. By optimizing the long-term reward associated with localization accuracy across all objects, Jie et al.[4] proposed a tree-structure reinforcement learning strategy to search for items sequentially while taking into account both the current circumstance and prior search results. In fields of motion analysis, Supancic and Ramanan[5] proposed an online decision-making approach for tracking in scenarios when picture frames may be unclear and computing resources are restricted. They described it as a partially observable decision-making process (POMDP) and used deep RL algorithms to learn policies. This procedure defines when to reinitialize, where to seek for the object, and when to update the object's appearance model. In addition, deep RL also has real-world applications like as robots [6] [7] [8], self-driving automobiles [9] and finance [10].

2 Background

2.1 Basic Framework of Reinforcement Learning

2.1.1 Setting

The purpose of reinforcement learning is to solve the challenge of learning from the environment through interaction in order to attain a goal. The learner is referred to as the agent. The agent behaves in a particular environment. The environment reacts to the agent's actions by providing the agent with new states. Rewards, which are mostly numerical values that the agent tries to optimize, are also produced by the environment.

Formally, we consider a sequence of discrete time steps. At each time step t, the agent gets the environment's state, $S_t \in \mathcal{S}$, where $\mathcal{S}$, also known as state space, is the collection of possible states. The agent performs an action, $A_t \in \mathcal{A}$, based on that state S_t, where $\mathcal{A}$ is the set of possible actions, or action space. The consequence of the agent's behavior is a numerical reward, $R_t \in \mathcal{R}$, after a time step. After then, it will be in state S_{t+1}.

How an agent selects its actions is defined by a policy. In reality, policy is a mapping of states to probabilities of selecting an action, denoted as π, where $\pi(a|s)$ is the probability that $A_t = a$ when $S_t = s$. . It is defined as follows:

$$\pi(a|s) = Pr(A_t|S_t = s) \tag{1}$$

How the agent's policy changes as a result of its experience is described by reinforcement learning methodologies. In general, the agent seeks to maximize the total amount of reward it receives over the long run.

2.1.2 Returns

The agent seeks to maximize its reward throughout the course of a long period of time. Assume the agent receives a series of rewards indicated as $R_t, R_{t+1}, R_{t+2}, ...$ after each time step t. In general, our goal is to maximize the expected return, where the return, G_t, represents the sum of all rewards. The definition of the return is as follows:

$$G_t = R_t + R_{t+1} + R_{t+2} + R_{t+3} + ...R_T \tag{2}$$

where T is the last time step. This equation makes sense when there is a final time step in the environment, such as in a game or a maze. However, in many cases, there is no final time step, where the interaction between agent and environment continues without limits. The return will be infinite. Therefore, in reinforcement learning, we usually use discounted return, which is denoted as follows:

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \gamma^3 R_{t+3} = \sum_{k=0}^{\infty} \gamma^k R_{t+k} \tag{3}$$

where γ is the discount rate, $0 \leq \gamma \leq 1$. If $\gamma < 1$, the sequence of reward is bounded. Thus, the sum has a finite value.

2.1.3 Markov Property and Markov Decision Processes

Given the current state and every previous state. The conditional probability of a future state solely depends on the current state if a process possesses the Markov property. A RL task that satisfies the Markov property is known as a Markov decision processes(MDP). Assume there are a finite number of states and rewards. The probability of each potential future state s', given any state s and action a, is:

$$p(s'|s,a) = Pr(S_{t+1} = s'|S_t = s, A_t = a) \tag{4}$$

These probabilities are called transitions probabilities.

In addition, given any present state s, action a and next state s', the expected value of next reward is:

$$r(s,a,s') = \mathbf{E}[R_{t+1}|S_t = s, A_t = a, S_{t+1} = s'] \tag{5}$$

2.1.4 Value Functions

Value functions measure how advantageous it is for an agent to be in a particular state or to behave in a particular state under a specific policy. In reinforcement learning, the evaluation is defined based on expected return.

Consider a policy: $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0,1]$ and $\pi(a|s)$, the probability of performing action a while in state s. The expected return when an agent begins from state s and follows policy π is denoted as $V_\pi(s)$, which is the value of state s under policy π. For MDPs, $V_\pi(s)$ is defined as follows:

$$V_\pi(s) = \mathbf{E}[G_t|S_t = s] = \mathbf{E}[\sum_{k=0}^{\infty} \gamma^k R_{t+k}|S_t = s] \tag{6}$$

V_π is referred to as the state-value function for policy π in reinforcement learning. It shows the expected return beginning in state s and continuing after policy π.

In reinforcement learning, there is another function called action-value function for policy π. Its definition is given by the expression $Q_\pi(s,a)$, which stands for the expected value of return beginning from state s, choosing action a, and then applying policy π:

$$Q_\pi(s,a) = \mathbf{E}[G_t|S_t = s, A_t = a] = \mathbf{E}[\sum_{k=0}^{\infty} \gamma^k R_{t+k}|S_t = s, A_t = a] \tag{7}$$

Q_π is the action-value function for policy π. The relationship between state-value function and action-value function is as follows:

$$V_\pi(s) = \sum_{a \in A} \pi(a|s) Q_\pi(s,a) \tag{8}$$

Value functions used in reinforcement learning have the key characteristic of satisfying specific recursive

relationships. The expansion of V_π for any policy π and any state s is as follows:

$$\begin{aligned} V_\pi(s) &= \mathbf{E}[G_t|S_t = s] \\ &= \mathbf{E}[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + ...|S_t = s] \\ &= \mathbf{E}[R_t + \gamma(R_{t+1} + \gamma R_{t+2} + ...)|S_t = s] \\ &= \mathbf{E}[R_t + \gamma G_t|S_t = s] \\ &= \mathbf{E}[R_t + \gamma V_\pi(S_{t+1})|S_t = s] \\ &= r(s) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s) V_\pi(s') \end{aligned} \tag{9}$$

This is called the Bellman Expectation Equation for V_π. It illustrates the connection between a state's value and the values of its succeeding states. This equation holds for every state.

Similarly, Q_π can also be expanded as:

$$\begin{aligned} Q_\pi(s,a) &= \mathbf{E}[R_t + \gamma Q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a] \\ &= r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V_\pi(s') \end{aligned} \tag{10}$$

2.1.5 Optimal Value Functions

A RL task's objective is to identify a policy that will yield the maximum reward within a set amount of time, as was described in section 2.1.2. This is known as the optimal policy, which is indicated as π^*. There can be several optimal policies, and they all have the same optimal state-value function $V^*(s)$ and optimal action-value function $Q^*(s,a)$. $V^*(s)$ can be defined as:

$$V^*(s) = \max_\pi V_\pi(s) \tag{11}$$

for all $s \in \mathcal{S}$.

Similarly, $Q^*(s,a)$ can be described as:

$$Q^*(s,a) = \max_\pi Q_\pi(s,a) \tag{12}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$.

To maximize $Q_\pi(s,a)$, we need to follow the optimal policy after current state-action pair (s, a). Then Q^* can be written in terms of V^* as follows:

$$\begin{aligned} Q^*(s,a) &= \mathbf{E}[R_t + \gamma V^*(S_{t+1})|S_t = s, A_t = a] \\ &= r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V^*(s') \end{aligned} \tag{13}$$

This is the same as the link between $V_\pi(s)$ and $Q_\pi(s,a)$ under common policies. On the other hand, the value

while selecting the action that maximizes the best action value at the moment is known as the optimal state value:

$$V^*(s) = \max_{a \in \mathcal{A}} Q^*(s, a) \tag{14}$$

Based on the relationship between $V^*(s)$ and $Q^*(s, a)$, we have the following equations:

$$V^*(s) = \max_{a \in \mathcal{A}} \{r(s, a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, a) V^*(s')\} \tag{15}$$

$$Q^*(s, a) = r(s, a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, a) \max_{a' \in \mathcal{A}} Q^*(s', a') \tag{16}$$

which are the Bellman Optimality Equations for $V^*(S)$ and $Q^*(s, a)$ respectively.

The optimal policy π^* can be obtained directly from $Q^*(s, a)$:

$$\pi^* = \arg\max_{\pi \in \mathcal{A}} Q_\pi(s, a) \tag{17}$$

2.2 Related Algorithms

2.2.1 Dynamic Programming

Given a thorough description of the environment, the dynamic programming approach is a collection of algorithms that may be used to choose the optimum course of action [11]. The basic goal of these algorithms is to use incremental update of value functions to identify an ideal course of action that maximizes the long-term gain. Finding a good policy is accomplished by acting in accordance with it, updating the value of the state before the action using the values of all potential states that may be reached from the successor state.

Dynamic programming-based reinforcement learning algorithms often fall into one of two categories: value iteration or policy iteration. Policy evaluation and policy improvement are the two stages of policy iteration. In particular, the value iteration directly uses the Bellman optimal equation for dynamic programming to obtain the final optimal state value, whereas the policy evaluation in the policy iteration uses the Bellman expectation equation to obtain a value function under a particular policy, which is a dynamic programming process.

2.2.2 Monte Carlo Methods

Monte Carlo methods are numerical computation approaches that are based on probability and statistics. We normally employ repeated random sampling when utilizing the Monte Carlo methods, and then we use probability and statistics methods to infer the numerical estimation of the target we want to seek from the sampled data. Monte Carlo techniques are used in reinforcement learning to average complete returns from sampled action sequences rather than the total returns from all potential action sequences as in dynamic programming [11]. Only sample sequences of the environment are required, not the entire model. However, Monte Carlo methods require the tasks

that have episodic structure. That is, a task can be divided into episodes and each episode has a termination [12]. Only when an episode ends do the values and the policies change.

2.3 Software Tools

The experiments are conducted using Python 3.7. Particularly, when implementing deep Q-network, we use Pytorch 1.13.0. In order to create a better reinforcement learning environment, we use OpenAI Gym library. Gym is a Python framework that is available as open source and is used to create and evaluate reinforcement learning algorithms. It offers both a standard collection of environments that accept that API and a common API for interacting between learning algorithms and environments. The environments: frozen lake and cart pole are the main focus here, which will be described in section 4.1.

3 What I have learned

3.1 Dynamic Programming

2.2.1 has briefly introduced the basic idea about dynamic programming. More details about dynamic programming will be described in this section.

3.1.1 Policy Iteration

Policy iteration is the process of continuously evaluating and improving policies until the best policy is reached. This section will introduce these two processes in detail.

Policy evaluation Policy evaluation is the process of calculating a policy's state-value function. According to equation 9 in section 2.1.4, Bellman expectation equation for $V_\pi(s)$ is given by:

$$\begin{aligned} V_\pi(s) &= r(s) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s) V(s') \\ &= \sum_{a \in \mathcal{A}} \pi(s,a)[r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V_\pi(s')] \\ &= \sum_{a \in \mathcal{A}} \pi(s,a) \sum_{s' \in \mathcal{S}} p(s'|s,a)[r(s,a,s') + \gamma V_\pi(s')] \end{aligned} \tag{18}$$

Knowing the reward function and the state transition function allows us to calculate the value of the current state based on the value of the following state. Calculating the value of the next potential state is a sub-problem, whereas determining the value of the current state is the current problem, according to the dynamic programming paradigm. Once the sub-problems are resolved, the current problem may be resolved. The state value function of the previous round are generally used to calculate the state value function of the next round when all the states are

taken into consideration, that is:

$$V_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(s,a) \sum_{s' \in \mathcal{S}} p(s'|s,a)[r(s,a,s') + \gamma V_k(s')] \tag{19}$$

Initial value V_0 is chosen arbitrarily. According to Bellman expectation equation, it can be proved that $V_k = V_\pi$ is fixed point for this formula. The sequence $\{V_k\}$ will converge to V_π as $k \to \infty$. Actually, when implementing algorithms, if the value $\max_{s \in \mathcal{S}} |V_{k+1} - V_k|$ is smaller than a threshold, we can terminate the policy evaluation in advance.

Policy improvement We may then improve the policy after calculating the state value function of the current one using the policy evaluation. Assume that we already know the value function for the policy V_π, V_π.

Let's say the agent chooses action a in state s and then executes policy π. The expected return at this point is therefore $Q_\pi(s,a)$. If $Q_\pi(s,a) > V_\pi(s)$, the expected return from taking action in state s will be greater than the expected return from the initial policy π. The above assumption only applies to one state; assuming a deterministic policy π' that, for each state s, meets the following conditions:

$$Q_\pi(s, \pi'(s)) > V_\pi(s) \tag{20}$$

Then the policy π' must therefore be better than, or as good as policy π. As a result, each state must provide it with a higher or equivalent expected return, which is

$$V_{\pi'}(s) \geq V_\pi(s) \tag{21}$$

This is the policy improvement theorem. We can obtain the proof of this theorem by expanding Q_π until we reach the terminal state, which is:

$$\begin{aligned}
V_\pi(s) &\leq Q_\pi(s, \pi'(s)) \\
&= \mathbf{E}_{\pi'}[R_t + \gamma V_\pi(S_{t+1}) | S_t = s] \\
&\leq \mathbf{E}_{\pi'}[R_t + \gamma Q_\pi(S_{t+1}, \pi'(S_{t+1})) | S_t = s] \\
&= \mathbf{E}_{\pi'}[R_t + \gamma R_{t+1} + \gamma^2 V_\pi(S_{t+2}) | S_t = s] \\
&\leq \mathbf{E}_{\pi'}[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \gamma^3 V_\pi(S_{t+3}) | S_t = s] \\
&\ldots \\
&\leq \mathbf{E}_{\pi'}[R_t + \gamma R_{t+1} + \gamma^2 V_\pi(S_{t+2}) + \gamma^3 R_{t+3} + \ldots | S_t = s] \\
&= V_{\pi'}(s)
\end{aligned} \tag{22}$$

To obtain the new policy π', we can greedily choose the maximal action value function in each state, that is:

$$\begin{aligned}\pi'(s) &= \arg\max_a Q_\pi(s,a) \\ &= \arg\max_a \{r(s,a) + \gamma \sum_{s'} p(s'|s,a) V_\pi(s')\} \\ &= \arg\max_a \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma V_\pi(s')]\end{aligned} \tag{23}$$

Policy iteration algorithm The policy iteration method generally operates as follows: we assess the current policy, get its state value function, improve the policy based on the state value function to get a better new policy, and then keep evaluating the new policy, improving policy, until we get the best policy:

$$\pi^0 \xrightarrow{evaluate} V^{\pi^0} \xrightarrow{improve} \pi^1 \xrightarrow{evaluate} V^{\pi^1} \xrightarrow{improve} \dots \xrightarrow{improve} \pi^* \xrightarrow{evaluate} V^*$$

Algorithm 1 shows how the policy iteration works.

Algorithm 1 Policy Iteration Algorithm

Initialize $\pi(s)$ and $V(s)$ arbitrarily for all $s \in \mathcal{S}$
while $\Delta > \theta$ **do**
  $\Delta = 0$
  for each state $s \in \mathcal{S}$ **do**
    $v = V(s)$
    $V(s) = \sum_{s'} p(s'|s, \pi(s)\{r(s, \pi(s), s') + \gamma V(s')\}$
    $\Delta = \max(\Delta, |v - V(s)|)$
  end for
end while
$\pi_{old} = \pi$
for each state $s \in \mathcal{S}$ **do**
  $\pi = \arg\max_a \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma V_\pi(s')]$
end for
If $\pi_{old} = \pi$ then return $V(s)$ and π, otherwise go to policy evaluation

3.1.2 Value Iteration

Value iteration updates the value function using the Bellman optimality equation:

$$\begin{aligned}V^*(s) &= \max_{a \in \mathcal{A}} \{r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V^*(s')\} \\ &= \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p(s'|s,a)[r(s,a,s') + \gamma V^*(s')]\end{aligned} \tag{24}$$

It is better to write it as an iterative update:

$$V_{k+1}(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p(s'|s,a)[r(s,a,s') + \gamma V_k(s')] \tag{25}$$

The fixed point for the Bellman optimality equation, which corresponds to the optimal value function V^*, is

reached when $V_{k+1} = V_k$. To determine the optimal policy, we may utilize the equation shown below:

$$\pi(s) = \arg\max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p(s'|s,a)[r(s,a,s') + \gamma V_{k+1}(s')] \tag{26}$$

Algorithm 2 illustrates how the value iteration works.

Algorithm 2 Value Iteration Algorithm

Initialize $V(s)$ arbitrarily for all $s \in \mathcal{S}$
while $\Delta > \theta$ **do**
  $\Delta = 0$
  for each state $s \in \mathcal{S}$ **do**
    $v = V(s)$
    $V(s) = \max_a \sum_{s'} p(s'|s,\pi(s))[r(s,\pi(s),s') + \gamma V(s')]$
    $\Delta = \max(\Delta, |v - V(s)|)$
  end for
end while
return a deterministic policy π, such that
$\pi(s) = \arg\max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p(s'|s,a)[r(s,a,s') + \gamma V_{k+1}(s')]$

3.2 Monte Carlo Methods

The expected returns from sampled action sequences are estimated using Monte Carlo methods, as was described in section 2.2.2. Recall that a state's value is equal to its expected return. We use a policy to sample many sequences, calculate the return from a particular state, and then determine its expectation when determining a state's value. According to the law of large numbers, the average should converge to the expected value as more returns are sampled. The updating formula for value function is given by:

$$V_\pi(s) = \mathbf{E}[G_t|S_t = s] \approx \frac{1}{N}\sum_{i=1}^{N} G_t^i \tag{27}$$

There are two ways of estimating expected returns. One technique is known as the every-visit Monte Carlo method, which determines $V_\pi(s)$ as the average of the returns following each visit to s in a collection of episodes. The other is the first-visit Monte Carlo method, which calculates the reward only once, meaning that the following cumulative reward is computed only when this state comes for the first time in this sequence, and when this state appears again later, it will be ignored. Here we mainly focus on first-visit Monte Carlo method , which is shown in algorithm 3

The value function of each state is evaluated using Algorithm 3, which is very similar to how policies are evaluated in dynamic programming. Policy evaluation is followed by a policy improvement, as demonstrated by dynamic programming. We can also apply the similar procedure to Monte Carlo to improve the policy. Instead of calculating the value function, the new Monte Carlo calculates $Q(s,a)$ for each state-action pair (s,a). When

Algorithm 3 First-visit Monte Carlo Method

Initialize $\pi(s)$, $V(s)$ and a list of $Returns(s)$ arbitrarily for all $s \in \mathcal{S}$
repeat
  Generate a episode using policy π
  for each state s in episode **do**
    $G \leftarrow$ the return following the first occurrence of s
    Add G to $Returns(s)$
    $V(s) \leftarrow$ average of $Returns(s)$
  end for
until end

improving the policy, it uses an $\epsilon - soft$ method to update the policy [12], which is shown as follows:

$$\pi(a|s) = \begin{cases} \frac{\epsilon}{|\mathcal{A}|} + 1 - \epsilon, & if\ a = \arg\max_{a'} Q(s, a') \\ \frac{\epsilon}{|\mathcal{A}|}, & otherwise \end{cases} \tag{28}$$

In this case, the new Monte Carlo method called $\epsilon - soft$ Monte Carlo method [12] is shown in algorithm 4

Algorithm 4 $\epsilon - soft$ Monte Carlo Method [12]

Initialize π, $Q(s, a)$ and a list of $Returns(s, a)$ arbitrarily for all $s \in \mathcal{S}$, $a \in \mathcal{A}$
repeat
  Generate a episode using policy π
  for each state-action pair (s, a) in episode **do**
    $G \leftarrow$ the return following the first occurrence of (s, a)
    Add G to $Returns(s, a)$
    $Q(s, a) \leftarrow$ average of $Returns(s, a)$
  end for
  for each s in the episode **do**
    for all $a \in \mathcal{A}$ **do**
      $\pi(a|s) = \begin{cases} \frac{\epsilon}{|\mathcal{A}|} + 1 - \epsilon, & if\ a = \arg\max_{a'} Q(s, a') \\ \frac{\epsilon}{|\mathcal{A}|}, & otherwise \end{cases}$
    end for
  end for
until end

3.3 Temporal Difference Learning

To estimate the value function of a policy, the temporal difference(TD) learning combines the concepts of Monte Carlo and dynamic programming. The TD learning and Monte Carlo are similar in that they can learn from sample data without knowing the environment in advance; The TD learning and dynamic programming are similar in that, according to the Bellman equation, the value of the subsequent state is used to update the current state value. For Monte Carlo methods, we rewrite the updating form of value function as an incremental way:

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)] \tag{29}$$

where α is a constant step-size parameter and G_t stands for the real return at time t. While the TD learning simply needs to wait until the next step, the Monte Carlo approach must wait until the termination of the entire episode to

calculate the $V(S_t)$. To be more specific, the TD learning uses the present reward plus the value of the future state as the reward earned in the present state, namely:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_t + \gamma V(S_{t+1}) - V(S_t)] \tag{30}$$

where $r_t + \gamma V(S_{t+1}) - V(S_t)$ is called temporal difference error. The reason why G_t is replaced by $r_t + \gamma V(S_{t+1})$ is based on equation 9:

$$\begin{aligned} V_\pi(s) &= \mathbf{E}[G_t|S_t = s] \\ &= \mathbf{E}[R_t + \gamma V_\pi(S_{t+1})|S_t = s] \end{aligned} \tag{31}$$

The Monte Carlo method takes the first line of equation 31 as the update target, and the TD learning takes the last line of the above formula as the update target. Therefore, when using the policy to interact with the environment, every sampling step, we can use the temporal difference algorithm to update the state value. Equation 30 is the simplest TD method, known as $TD(0)$. Based on $TD(0)$, there are two commonly used TD methods, which are SARSA and Q-learning.

3.3.1 SARSA Algorithm

Now that we can use the $TD(0)$ to estimate the value function, a natural question is whether we can use a method similar to policy iteration for reinforcement learning. Policy evaluation can be achieved by the $TD(0)$, next is to improve the policy. The action value function can be directly estimated using TD method, which is:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_t + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{32}$$

Here comes another question that how we decide the action. If we always select the action that maximizes the $Q(s, a)$ according to greedy algorithm, some state-action pairs may be overlooked and their action value can not be estimated. Therefore, it cannot be guaranteed that after policy improvement, the policy is better than the original one. One of simple and often used solutions is $\epsilon - greedy$ algorithm: there is a probability of $1 - \epsilon$ to perform the action with the highest action value and a probability of ϵ to randomly choose an action from the action space. Algprithm 5 demonstrates the process of SARSA.

Algorithm 5 SARSA Algorithm

Initialize $Q(s, a)$ arbitrarily for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$
for each episode e **do**
  Obtain initial state s
  Select action a in state s from Q using $\epsilon - greedy$
  for each time step in episode **do**
    Take action a and obtain r and s'
    Select action a' in state s' from Q using $\epsilon - greedy$
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$, $a \leftarrow a'$
  end for
end for

3.3.2 Q-learning

The main distinction between Q-learning and Sarsa is that Q-learning uses the following updating mechanism:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_t + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{33}$$

Algorithm 6 shows the process of Q-learning. In this case, Q-learning actually estimates Q^*, the optimal action value function, regardless of the policy being followed, since given any (s, a, r, s') we can always update the Q function. However, SARSA algorithm needs the data sampled by the current policy to update Q, since a' in $Q(s', a')$ is the action in state s' under current policy. We refer to Q-learning as an off-policy algorithm and SARSA as an on-policy algorithm.

Algorithm 6 Q-learning Algorithm

Initialize $Q(s, a)$ arbitrarily for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$
for each episode e **do**
  Obtain initial state s
  for each time step in episode **do**
    Select action a in state s from Q using $\epsilon - greedy$
    Take action a and obtain r and s'
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$
  end for
end for

3.3.3 On-Policy Algorithm and Off-Policy Algorithm

The behavior policy is the one that samples data from environment, and the target policy is the one that uses these data to update function. The on-policy algorithm indicates that the behavior policy and the target policy are the same policy, whereas the off-policy algorithm indicates that these two policies are different. Sarsa is a typical example of on-policy algorithm, because its updating formula uses the tuple (s, a, r, s', a') where all the elements come from the current policy. Q-learning is an example of off-policy algorithm. Q-learning chooses a' that maximizes the Q function, which is not generated by current policy. An important way to distinguish whether it is an on-policy algorithm or off-policy algorithm is to see whether the data used to update temporal difference come from the current policy.

3.4 Deep Q-Network

In the Q-learning algorithm described in section 3.3.2, the function $Q(s, a)$ represents expected return selecting aciton a in state s and following a certain policy afterwards. It works quite well when the states and actions are discrete data and when there is a small number of states and actions. However, when the numbers of states and actions are very large, this approach is not applicable. Furthermore, when the states and actions are continuous

data, there are infinite state-action pairs, and we cannot store the values for every (s, a) pair using this function. In this case, we need to estimate and fit the Q value using the function approximation approach. Since deep neural network performs quite well in function approximation, we can use neural network to represent the Q function. If the action is continuous (infinite), the neural network takes the state and the action as input and then outputs a scalar representing the value that can be attained by performing this action in the state. If the action is discrete (finite), we can input the state into the neural network and it can output the value for each action. The neural network used to fit the Q function is referred to as the Q-network.

One of the crucial parts in neural network is the loss function. As with any other neural network training, we need to define the loss function for Q-network so that the Q-network can update the weights by utilizing the loss function. Recall that the updating rule for Q-learning as shown in section 3.3.2:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)] \tag{34}$$

This formula uses temporal difference to incrementally update $Q(s, a)$ by learning the target $r + \gamma \max_{a'} Q(s', a')$. That is, we need to update $Q(s, a)$ to make it as close to the target as possible. In this case, we can create the loss function for Q-network in the form of mean square root(mse):

$$\omega^* = \arg\min_{\omega} \frac{1}{2N} \sum_{i=1}^{N} [Q_\omega(s_i, a_i) - (r_i + \gamma \max_{a'} Q_\omega(s_i', a')]^2 \tag{35}$$

At this stage, we may expand Q-learning into the neural network - deep Q network (DQN) algorithm. DQN also includes two critical modules: experience replay and target network, both of which can improve the robustness and performance of DQN.

3.4.1 Experience Replay

In general supervised learning, assuming that the training data is independent and identically distributed(IID), in each epoch, we arbitrarily sample one or more data from the training data to perform gradient descent. As the training continues, some data may be used more several times. However, in the Q-learning, each data point is only utilized once to update the Q value. To better combine Q-learning and neural networks, DQN uses experience replay method [13]. The particular way is to create a container called replay buffer and store the quadruple data (state, action, reward, next state) in it. When training the Q network, some of samples are randomly taken from the buffer. There are two advantages using this method:

1) Make sure the examples adhere to the independence assumption. The state at this time is tied to the state at the previous moment, hence the data acquired by interactive sampling in MDP does not meet the independence assumption. The training of neural networks is significantly impacted by non-IID input, which causes the neural network to fit the most recent training data. Experience replay can make samples meet the independent assumption by removing the correlation between them.

2) Improve the effectiveness of sample utilization. Each sample may be utilized several times, making it ideal for gradient descent in a deep neural network.

3.4.2 Target Network

DQN's purpose is to obtain $Q_\omega(s,a)$ as close to the target $r + \gamma \max_{a'} Q_\omega(s',a')$ as possible. However, because the target includes the neural network's output, the target will constantly change as the parameters are changed, which results in an unstable training process and makes the neural network difficult to converge. DQN employs the concept of target network to overcome this problem [13]: Use two networks instead of one. On one hand, the policy network is used to adjust the parameters, as is done in most neural networks. The target network, on the other hand, is utilized to compute the target. The network structures of the target network and the policy network are identical. During the training phase, the target network's parameters remain unchanged. Following a brief batch of training, the latest parameters of policy network are replicated to the target network.

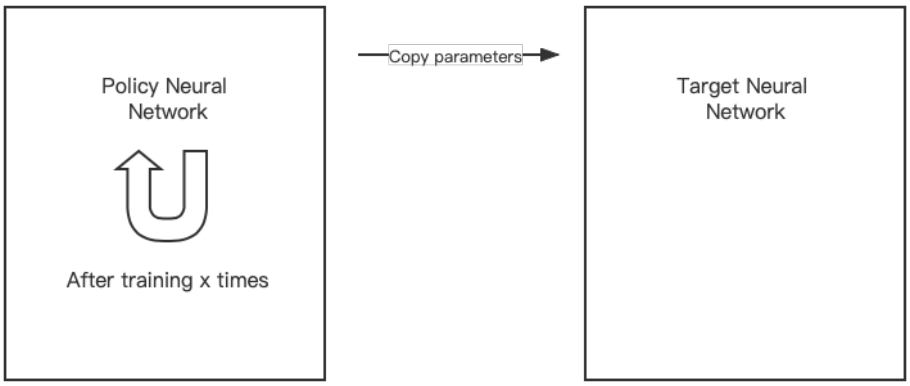


Figure 1: Q-network

By combining experience replay and target network, the procedure of DQN is shown in Algorithm 7

4 Experiments

4.1 Environment Setting

4.1.1 Frozen Lake

Frozen lake is a classical and simple maze in OpenAI gym, as shown in Figure 2(a). It is a 4×4 maze in which you must travel across a frozen lake (F) from the Start(S) in the top left corner to the Goal(G) in the bottom right corner while avoiding falling into any Holes (H). There are 4 actions for agent represented by 4 numbers, which are 0: LEFT, 1: DOWN, 2: RIGHT, 3: UP. The state, calculated by (current row)* nrows + (current col), is a value that represents the agent's current position. The row and col both start at 0. For example, the goal position can be

Algorithm 7 Deep Q-network

Initialize network $Q_\omega(s,a)$ arbitrarily
Copy parameters to the target network $Q_{\omega'}(s,a)$
Initialize experience replay buffer
for each episode e **do**
  Obtain initial state s_1
  for each time step in episode **do**
    Select action a_t from $Q_\omega(s,a)$ using $\epsilon - greedy$
    take action a_t and obtain reward r_t and s_{t+1}
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer
    if the amount of data in buffer reaches a threshold **then**
      Select N samples $\{(s_i, a_i, r_i, s_{i+1})\}_{i=1,...N}$
      Calculate target $y_i = r_i + \gamma \max_a Q_{\omega'}(s_{i+a}, a)$ for each sample
      Minimize loss function $L = \frac{1}{2N}\sum_i (y_i - Q_\omega(s_i, a_i))^2$
      Update target network
    end if
  end for
end for

calculated as: 3 * 4 + 3 = 15. The reward for goal or termination is 1 while rewards for other positions are 0. Once the agent reaches the termination or drops into holes, current episode will terminate.

In order to make this maze more complex, we added two small rewards called gifts to frozen lake, as shown in figure 2(b). The state that contains gift is called gift state. In the figure, the gift states are 2 and 9. Each gift can be picked up only once. Therefore, after the agent steps into a gift state, the reward should be zero. So the transition probability matrix must be modified once the agent get to the gift state. Essentially, we need to update rewards for its neighbor states, which are left, down, right and up. This is because the reward from neighbour state to gift state should be zero once the agent picks up the gift.

(a) Frozen lake (b) New frozen lake

Figure 2: Frozen lake environment

This environment will be used in dynamic programming and monte carlo method.

4.1.2 Cart Pole

Apart from the maze, we also use another environment called cart pole, as depicted in Figure 3. A joint links a pole to a cart that travels on a frictionless track. On the cart, the pole is erect. The purpose is to balance the pole

by moving the cart to the left or right. There are two discrete actions in action space represented by value 0 and 1, which are 0: move to the left and 1: move to the right. The observation denotes the current state for the cart pole. It is an array that has a length of 4 containing 4 elements. The information is shown in table 1. Because the aim is to maintain the pole upright for as long as possible, a reward +1 for each step is given. The reward threshold is 475. Note that the episode terminates when any one of following situations occur:

1) The cart pole angle leaves the range (-0.2095 rad (-12°), 0.2095 rad (12°)).
2) The cart position leaves the range (-2.4, 2.4).
3) The reward is greater than 475.

This environment will be used in SARSA, Q-learning and DQN.

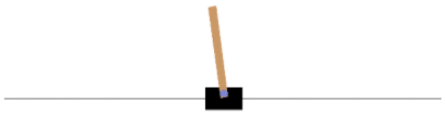


Figure 3: Cart pole

Table 1: Information about observation

Num	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	-0.418 rad (-24°)	0.418 rad (24°)
3	Pole Angular Velocity	-Inf	Inf

4.2 Dynamic Programming

Since the transition probability matrix must be modified once the agent steps into ‘gift state’, the policy should also be modified. So before playing the game, we have to evaluate 2^n policies if there are n gifts. The reason that the number is 2^n is that each gift state will either be visited or not be visited. There are two situations for each gift state. Here we only have two gifts, meaning that we need to evaluate 4 policies, which corresponds to 4 situations:

1) Both gifts have not yet been picked up
2) Gift 1 has been picked up while gift 2 remains still
3) Gift 2 has been picked up while gift 1 remains still
4) Both gifts have been picked up

If the agent picks up either gift for the first time, it will then follow another policy.

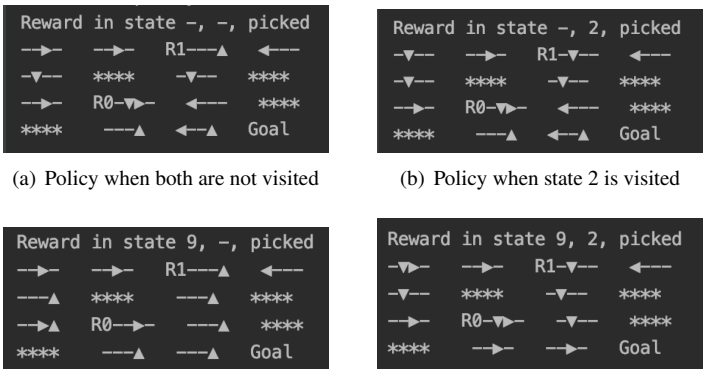
The main problem is finding which policy the agent should follow. I first evaluate 4 policies corresponding

to the above 4 situations. After that, the policies are stored in a dictionary. We generated 2^n 0-1 vectors that have length of n. Here $n = 2$, so we have 4 vectors $(0, 0)$, $(0, 1)$, $(1, 0)$, $(1, 1)$ and 1 represents that the gift state has been visited. The gift states are stored in a list. Here the gift states are $9, 2$, so the list is $(2, 9)$. Then we did the dot product between vector and list. For example, $(0, 1) \cdot (9, 2) = (0, 2)$, which means state 2 has been visited. The dot product is the key in the dictionary, the value is the policy. Take $(0, 2)$ as an example. Its corresponding policy is the one evaluated under the situation where state 2 has been visited and state 9 has not been visited. When playing game, we will check whether each gift state has been visited and then find the corresponding policy for agent in the dictionary.

We implemented dynamic programming algorithm using policy iteration and value iteration based the frozen lake environment. The gift reward in gift state 2 is set as 35; the gift reward in gift state 9 is 16. The termination reward is 50. So the maximum reward that agent can get is 101.

4.2.1 Results for Policy Iteration

We first used the policy iteration to implement dynamic programming. After evaluation, we generated the 4 policies under 4 situations, which are shown in Figure 4.



(a) Policy when both are not visited

(b) Policy when state 2 is visited

(c) Policy when state 9 is visited

(d) Policy when both are visited

Figure 4: Policies under 4 situations using policy iteration

In each figure, $R1$ represents the gift in state 2 and $R0$ represents the gift in state 9. '****' denotes the hole and 'Goal' means the termination. There are 4 actions represented by 4 kinds of black arrows. Each figure has 4×4 items, which correspond to the states in the maze one by one.

Starting from state 0 at the top left corner, when both gifts have not been picked up, the agent follows the policy shown in Figure 4(a). Under the guidance of this policy, it moves to right until it reaches state 2. Once it gets to the state 2 which is $R1$ in the figure, it follows the policy shown in Figure 4(b). At this time, following the policy, instead of moving to up, the agent moves to down and reaches state 9, which is $R0$ in the figure. At this moment, the agent has picked up all the gifts, so it will follow the policy shown in the Figure 4(d) and eventually get the termination. The agent finds two paths to get to the termination while picking up all the gifts, as illustrated

in Figure 5.

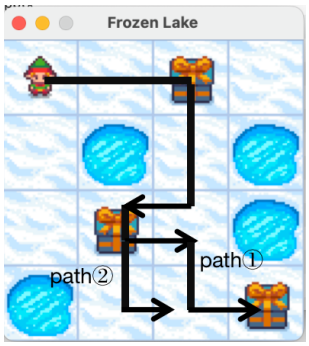


Figure 5: Two paths generated by policy iteration

We ran a 100-episode test to observe the effect of the policy. The results are illustrated in Figure 6. In 100 episodes, the agent can always pick up all the gifts in the maze and finally get to the termination.

```
Times of getting maximal reward:  100
Times of reaching the termination:  100
Average reward:  101.0
```

Figure 6: Test results by policy iteration

4.2.2 Results for Value Iteration

We also used the value iteration to implement dynamic programming. We generated 4 policies under different situations, as shown in Figure 7.

```
Reward in state -, -, picked
-->-  -->-  R1---^  <---
-v--  ****   -v--  ****
-->-  R0-v>-  <---  ****
****   ---^  <--^  Goal
```

(a) Policy when both are not visited

```
Reward in state -, 2, picked
-v--  -->-  R1-v--  <---
-v--  ****   -v--  ****
-->-  R0-v>-  <---  ****
****   ---^  <--^  Goal
```

(b) Policy when state 2 is visited

```
Reward in state 9, -, picked
-->-  -->-  R1---^  <---
---^  ****   ---^  ****
-->^  R0-->-  ---^  ****
****   ---^  ---^  Goal
```

(c) Policy when state 9 is visited

```
Reward in state 9, 2, picked
-v>-  -->-  R1-v--  <---
-v--  ****   -v--  ****
-->-  R0-v>-  -v--  ****
****   -->-  -->-  Goal
```

(d) Policy when both are visited

Figure 7: Policies under 4 situations using value iteration

The policies generated by value iteration are exactly the same as the policies generated by policy iteration. We also ran a 100-episode test for value iteration. The results can be seen in Figure 8. Similarly, the agent can also collect all of the gifts and eventually reach the end.

```
Times of getting maximal reward:  100
Times of reaching the termination:  100
Average reward:  101.0
```

Figure 8: Test results by value iteration

4.3 Monte Carlo Methods

Here we implement $\epsilon - soft$ Monte Carlo method described in algorithm 4. We trained about 300 episodes and recorded the average return that the agent get in each episode. The result for training is shown in Figure 9. The vertical line represents average returns and horizontal line represents the episodes. Before about 250th episode, despite some significant fluctuations, the average returns continue to rise. After 250th episode, the average returns reach the peak, meaning that the agent have already found a path that can get the maximal reward. We output the policy to see how agent acts in each state, as depicted in Figure 10. This path is the same as one of the path generated by dynamic programming.

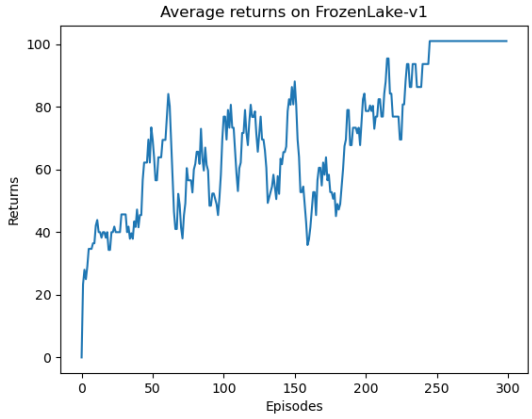
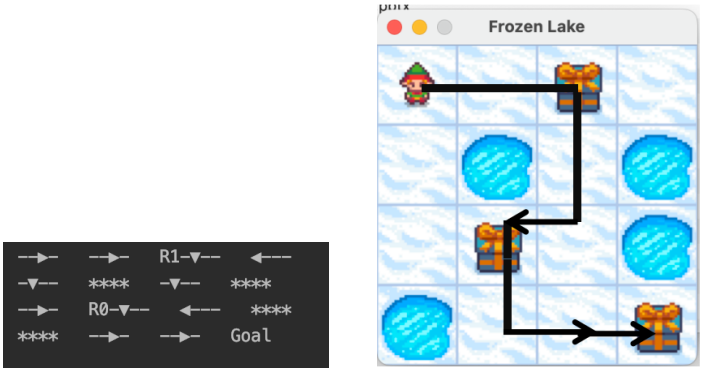


Figure 9: Training curve for Monte Carlo method



(a) Policy generated by Monte Carlo method (b) Path generated by Monte Carlo method

Figure 10: Policy and path generated by Monte Carlo method

We also ran a 100-episode test. The results are shown in Figure 11. In 100 episodes, we can see that the agent can always pick up all the gifts in the maze and finally get to the termination.

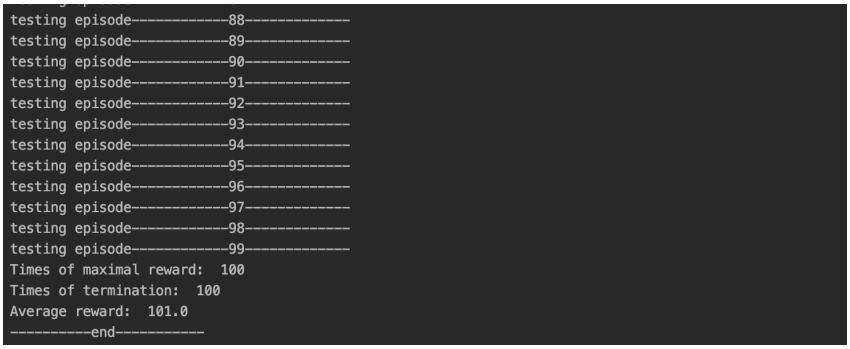


Figure 11: Test results for Monte Carlo method

4.4 SARSA Algorithm

We plan to implement SARSA based on cart pole environment. Since each observation has 4 elements and the value of each element is a continuous number, we cannot implement SARSA using the functionQs, a. Therefore, we need to discretize the observation into a tuple containing 4 integers and then convert this tuple into an integer representing the state. By doing this, we finally convert an observation into a state. Then we can use algorithm5 to implement SARSA.

Next we will introduce how to convert an observation into a state. First, we need a function from Numpy in Python called digitize. This function returns the bin indices for each value in the provided array. Given an input array x and an array of bins, it will return an array of index where each index i satisfies $bin[i-1] \leq x < bin[i]$. For example, given an input $x = [0.2, 2.5]$ and a bin $[0, 2, 4]$, this function will return $[1, 2]$ since $bin[0] \leq 0.2 < bin[1]$ and $bin[1] \leq 2.5 < bin[2]$. Then each continuous data in x is converted into discrete data. When we implement the algorithm, x is one of the elements in observation and bin is its range. Take one of the elements $cartposition$ in observation as an example. The range of cart position is actually $(-2.4, 2.4)$. We divided the range into 8 smaller ranges, which is $(-2.4, -1.8, -1.2, -0.6, 0, 0.6, 1.2, 1.8, 2.4)$. Given a continuous data cart position such as -2.3456, we can map this value to an integer 1. We repeated the operation on other observation elements. By doing this, a 4-element observation is converted into a 4-element integer tuple.

Next we will convert the integer tuple into an integer representing the state. Recall that we divided the range into 8 small ranges, which means that each element in the integer tuple actually has 8 possible values. We can perceive the tuple as an octal number. What we need to do is convert this octal number into a decimal number. By doing this, the integer tuple is finally converted into an integer. Thus, we can convert the observation into an integer that represents the state. As a result, we can use the function $Q(s, a)$ and implement SARSA based on algorithm 5.

We trained about 1000 episodes and the image of training curve is shown in Figure 13. The vertical line represents average returns and horizontal line represents the episodes. As shown in the figure, before the 200th episode, the returns fluctuate significantly, reaching more than 70. However, the returns drop to just around 20 and fluctuate between 20 and 40, which are far less than maximal return. We also recorded number of completing

episodes and calculate the overall average return, as depicted in Figure 13 Clearly, there is a lot to improve for SARSA algorithm.

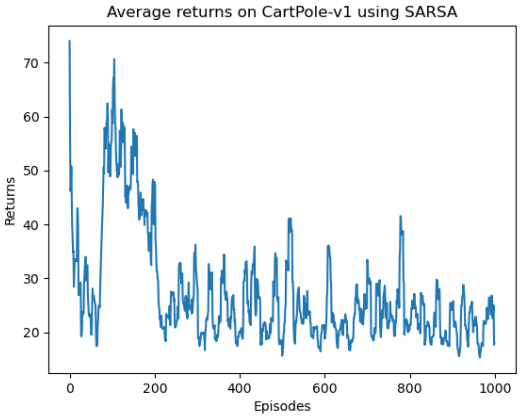


Figure 12: Training curve for SARSA

```
988 Episode: Finished after 18 time steps
989 Episode: Finished after 29 time steps
990 Episode: Finished after 44 time steps
991 Episode: Finished after 20 time steps
992 Episode: Finished after 16 time steps
993 Episode: Finished after 22 time steps
994 Episode: Finished after 20 time steps
995 Episode: Finished after 22 time steps
996 Episode: Finished after 50 time steps
997 Episode: Finished after 18 time steps
998 Episode: Finished after 17 time steps
999 Episode: Finished after 18 time steps
Complete episode: 0
Average reward is: 27.381
```

Figure 13: Training result for SARSA

4.5 Q-Learning

Similar to SARSA, we also need to dicretize the observation and convert it into an integer. The process is exactly the same as the process described section 4.4. What we need to modify here is the updating rule for $Q(s,a)$. The rest parts of the algorithm are the same as SARSA.

We also trained 1000 episodes and the image of training curve for Q-learning is shown in Figure 14. Before about the 100th episode, the highest return reaches almost 80. However, the returns drop to just around 20 and fluctuate between 20 and 40 afterwards, which is similar to the training curve for SARSA. Figure 15 shows the number of completing episodes and the overall average return. The average return is just 25.661, which is similar to the result of SARSA.

From the results of SARSA and Q-learning, we can conclude that when dealing with the environment whose states are continuous, both algorithms cannot achieve good results. One way to improve is to increase the number of discrete states, meaning that we need to divide the range of each element in the observation into smaller ranges.

If we divide the range into infinitely small ranges, then each small range actually corresponds to a real number. Every continuous state will correspond to a discrete state. This is obviously impossible because the function $Q(s, a)$ cannot store an infinite amount of data. In next section, we can see that deep Q-network works quite well for continuous state.

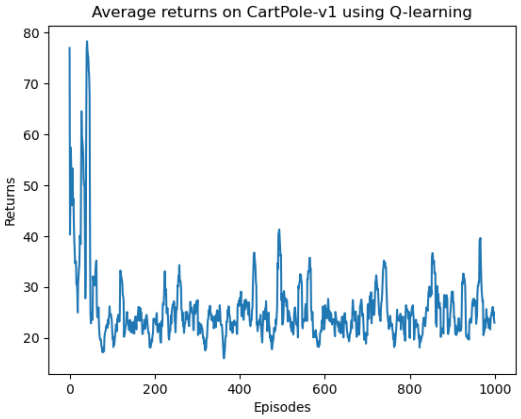


Figure 14: Training curve for Q-learning

```
991 Episode: Finished after 40 time steps
992 Episode: Finished after 24 time steps
993 Episode: Finished after 18 time steps
994 Episode: Finished after 27 time steps
995 Episode: Finished after 35 time steps
996 Episode: Finished after 20 time steps
997 Episode: Finished after 22 time steps
998 Episode: Finished after 26 time steps
999 Episode: Finished after 23 time steps
Complete episode: 0
Average reward is: 25.661
```

Figure 15: Training result for Q-learning

4.6 Deep Q-Network

We used Pytorch to implement DQN based on the procedure described in algorithm 7. Our DQN has one hidden layer containing 128 neurons. The input layer has 4 neurons, which is the same as the dimension of observation. The output layer has 2 neurons because there are only two actions in action space. Hidden layer uses ReLU as the activation function. We used Adam as the optimizer. We took each observation as an input and output the action value in this observation. After training 1000 episodes, the figures of training curves for returns and loss function are shown in Figure16 and Figure 17.

In the early stage, the returns grow dramatically, reaching the peak before the 350th episode, while the loss function decreases considerably. After that, although in some episodes, the returns drop to low level, most returns can be maintained at high levels. There are some sharp fluctuations in the loss function as well, but overall it remains at a relatively low value. The results of completing episodes and overall average returns are shown in

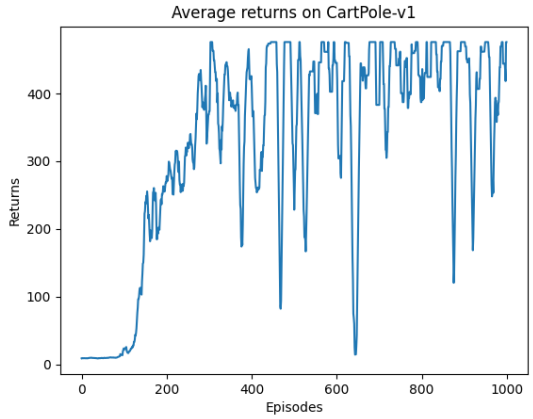


Figure 16: Training curve for returns using DQN

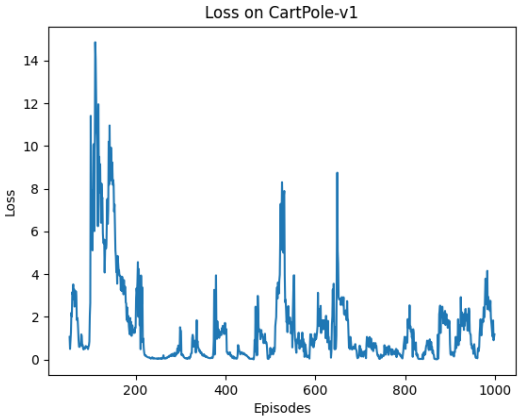


Figure 17: Training curve for loss function using DQN

Figure 18. The results are clearly superior to those of SARSA and Q-learning, indicating that DQN outperforms SARSA and Q-learning when dealing with continuous states.

```
100%|██████████| 1000/1000 [07:17<00:00,  2.28it/s, episode=1000, return=325.819]
Complete episode: 459
Average reward is: 325.819
```

Figure 18: Training results for DNQ

5 Plan for Next Term

Reinforcement learning has achieved great success in the field of games. One of the most successful algorithms is AlphaGo Zero [14]. AlphaGo Zero was born in October 2017. It started completely from scratch and became invincible only through self-play. It combines deep RL, convolutional neural networks (CNN) and Monte Carlo tree search (MCTS) and has achieved great success in many games such as Go, Chess and Shogi, which attracts me a lot. Next term, I plan to do some research on AlphaGo Zero. First, I will read some literature about AlphaGo Zero. I will try to figure out its principles, basic structure and related algorithms such as MCTS and CNN. Next, I will try to implement it using Python, Pytorch or Tensorflow. Finally, I am going to use the implemented algorithm to play a simple game, Gobang. I will train the model and make it able to play with humans. The final deliverable

will be a system based on the framework of AlphaGo Zero that enables humans to play Gobang with the machine.

References

[1] V. François-lavet, P. Henderson, R. Islam, M. G. Bellemare, V. François-lavet, J. Pineau, and M. G. Bellemare, "An Introduction to Deep Reinforcement Learning. (arXiv:1811.12560v1 [cs.LG]) http://arxiv.org/abs/1811.12560," *Foundations and trends in machine learning*, vol. II, no. 3 - 4, pp. 1–140, 2018.

[2] V. Mnih, N. Heess, A. Graves *et al.*, "Recurrent models of visual attention," *Advances in neural information processing systems*, vol. 27, 2014.

[3] S. Mathe, A. Pirinen, and C. Sminchisescu, "Reinforcement learning for visual object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2894–2902.

[4] Z. Jie, X. Liang, J. Feng, X. Jin, W. Lu, and S. Yan, "Tree-structured reinforcement learning for sequential object localization," *Advances in Neural Information Processing Systems*, vol. 29, 2016.

[5] J. Supancic III and D. Ramanan, "Tracking as online decision-making: Learning a policy from streaming videos with reinforcement learning," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 322–331.

[6] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1334–1373, 2016.

[7] D. Gandhi, L. Pinto, and A. Gupta, "Learning to fly by crashing," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017, pp. 3948–3955.

[8] L. Pinto, M. Andrychowicz, P. Welinder, W. Zaremba, and P. Abbeel, "Asymmetric actor critic for image-based robot learning," *arXiv preprint arXiv:1710.06542*, 2017.

[9] S. Yun, J. Choi, Y. Yoo, K. Yun, and J. Young Choi, "Action-decision networks for visual tracking with deep reinforcement learning," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 2711–2720.

[10] Y. Deng, F. Bao, Y. Kong, Z. Ren, and Q. Dai, "Deep direct reinforcement learning for financial signal representation and trading," *IEEE transactions on neural networks and learning systems*, vol. 28, no. 3, pp. 653–664, 2016.

[11] G. Coqueret and T. Guida, "Reinforcement learning," *Machine Learning for Factor Investing*, pp. 247–260, 2020.

[12] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[14] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. V. D. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, and K. Kavukcuoglu, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7585, pp. 484–489, 2016. [Online]. Available: http://dx.doi.org/10.1038/nature16961