City University of Hong Kong
Department of Computer Science

CS6537 Guide Study – Final Report

Obstacles Avoidance Based on Reinforcement Learning

Student: Li Xinyan (55670594)
Supervisor: Dr. SONG, Linqi

Table of Contents

1 Introduction

1.1 Background

Reinforcement learning algorithms have kept making progress in recent decades: AlphaGo, developed by DeepMind in 2016[1], used RL algorithms and defeated the world's Go masters. At this point, RL have renewed the attention of scholars and is more widely used in modern robot control. Major technology companies now are giving a higher priority to the development of RL than before. It can be said that RL is affecting and changing the world.

As an important machine learning method,reinforcement learninghas beenwidelyused in complex decision-making optimization and control problems. Unlike supervised learning and unsupervised learning, reinforcement learning can solve problemsusing the interaction between agents and environment for modeling. Environment in reinforcement learning represents the world where the agents are in, and agentsarereinforcement learning components that make decision of what action to take in environment. This procedure can be described as: agent’s goal is to maximize the total reward,every time when the agent is about to choose an action, it will perceive its state in the environment, usestrategiesto select an action,take this action, then it will get an instant reward (feedbacks) from environment, reach a new state. During this interaction, agent gradually learn to improve its strategies.

Obstacle avoidance of mobile robot [2] (like UAV, unmanned submarine vehicle, medical robot, humanoid robot, etc.)requiresmobile robottoperceive static or dynamic obstacles through sensors, effectively avoid them according to certain methods [3].

Considering the similarity between typical reinforcement learning problem and obstacle avoidance problem, there are promising prospects in using reinforcement learning algorithm to solve obstacle avoidance problem.

Reinforcement learning algorithms can be divided into tabular algorithms (Q-Learning algorithm, SARSA algorithm) and deepalgorithms (Deep Q Network algorithm). In both kind of algorithms, the agent relies on value function to evaluate the value of every states and actions, the statevalue deeply influences the agent to take particular action. The value

function in tabular algorithms is maintained by a table (also called Q-Table) and stored in memory, but in deepalgorithms, the value function is replaced by a neural network.By introducing neural network, there is no need to store the value table, which may be much more massive in complex problems.

Deep Q Network is uncontrollable for its varied hyper parameters (from both reinforcement learning and neural network aspects): learning rate, discount factor, batch size, update step, replay memory size, greedy factor, etc. The hyper parameters and the structure of neural network is two key factor affect the model's performance. So there is indeed a need for parameter adjustment in order to get a better training outcome. Due to the long training time, this study tries multi-threading implementation, parallel computing method.

1.2 Objectives

The goal of this study is to use reinforcement learning algorithm to solve obstacles avoidance problem. To realize the goal, this study will build up an obstacles avoidance emulation environment, redefine obstacles avoidance to a reinforcement learning problem, use different algorithms to solve the problem, observe the performance of algorithms, adjust parameters, then improve the algorithm.

In the beginning, to explore reinforcement learning principle, this study goes deep into fundamental reinforcement learning theory: Markov Property, Markov Process, Markov Decision Process, Bellman equation, value iteration and policy iteration [4]. Reinforcement learning algorithm like Q-learning algorithm and neural network based Deep Q Network algorithm are discussed. For a better performance, DQN algorithm is improved from many aspects.

2 Related Work

To fully understand reinforcement learning problems, this chapter will start from the underlying architecture: Markov Property, MarkovProcess, Markov Decision Process, to describe knowledge representation of reinforcement learning problem from the bottom, and will point out the feasibility of using reinforcement learning method to solve obstacle avoidance problem. In order to pave the way for deep reinforcement learning, some basic knowledge needed for the application of neural network in reinforcement learning algorithm is introduced. Because the neural networkshave complicated and various contents that involves many aspects, only the part related to reinforcement learning algorithms is described,the rest is not repeated.

2.1 Reinforcement Learning Overview

Reinforcement learning solves the problem of interaction between agent (s) and environment: agent perceives, observes, makes decision and acts in the environment; at each moment, after the agent acts, the state of the environment changes, meanwhile produces a Return signal (this signal gives agent rewardor punishment, also known as reinforcement signal, teacher signal, or simply called Return).In general, the Return isa scalar using positive number to represent reward and negative number for punishment. Agent studies from Return after taking numerous actions, gradually improving itspolicy. The agentlearns tochoose action that leads to reward and avoid action that leads to punishment. When the environment reaches the terminal state, this old state-action-reward-new state loop also terminates, this is the end of episode. The goal of reinforcement learning is the agent learn to find an optimal policy for

action selection, in order to maximizecumulative Reward. Figure 2-1 shows the interaction between agent and environment.

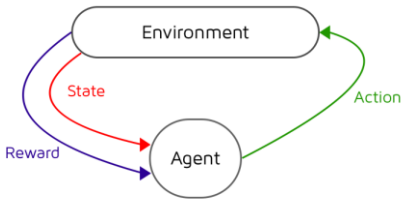


Figure 2-1 Interaction between Agent and Environment

Early studies in reinforcement learning proposed Random Policy Search and Genetic Algorithms to directly find the optimal policy, but these random algorithmsare blind. Complex problems with more states and actions will low down the solution efficiency and the optimal solution may not be guaranteed. There are two ways to solve a typical reinforcement learning problem: First, to search an optimal policy directly: find an optimal sequence of actions, agent can maximize cumulative reward by following this policy to choose action. Second, to searchan optimal value function: value function evaluates every states and actions. When the agent is going to choose an action, actions with higher potential will be recommended by value function.

Because it is difficult to find an optimal policydirectly (mentioned in last paragraph), scientists according to the second strategy, proposed a framework to solve most of the reinforcement learning problems: Markov Decision Process (MDP). Following the order of Markov Property, Markov Process (MP), Markov Decision Process (MDP), Bellman Equation, the paper introduces the solution of reinforcement learning problem step by step. Solve the theoretical basis.

2.2Reinforcement Learning Model

2.2.1 Markov Property and Markov Process

In a stochastic process, the conditional probability distribution of the future state is only related to the current state rather than any previous state, then we claim this stochastic process has Markov property.Let the state of stochastic process at time tas St, and the next state as St+1. If the stochastic process has Markov property, it should satisfy formula (2-1):

$$\boldsymbol{P}[\boldsymbol{S_{t+1}}|\boldsymbol{S_t}] = \mathbf{P}[\boldsymbol{S_{t+1}}|\boldsymbol{S_1},\ \boldsymbol{S_2}, \dots, \boldsymbol{S_t}] \text{(Formula2-1)}$$

Here is a given example to illustrate the Markov property [5]: In an opaque bag with two red balls and one green ball, all the properties of the three balls are the same except for the color. Suppose that you take a ball out of the bag every day and never put it back. If the balls are taken on day1, day2, day3in the orderred, red, green. On day2, when you pick up a red ball, you can calculate the probability of getting agreen ball on day3: it must be 100% since you already know the ball picked on day1 is red.So the probability distribution of the future state(the color of the ball pick up in day3) is related to both current state (pick up a red ball on day2) and previous state (pick up a red ball on day1),that is, this stochastic process does not have Markov property [6]. On the other hand, if we change the rule: we take out the ball, mark its color and put it back to the bags. Replay the game, the probability of getting agreen ball on day3 will be 2/3 regardless of the color of the ball picked up on day1 and day2. At this point, the probability distribution of future state has nothing to do with previous state and this random process has Markov Property.

Stochastic processes with Markov propertyis called Markov process. In Markov process, the state changes from current state $S_t$to the next state S_{t+1}based on the probability distribution in matrix P.

$$\boldsymbol{P} = \begin{bmatrix} \boldsymbol{P_{11}} & \boldsymbol{P_{12}} & \cdots & \boldsymbol{P_{1n}} \\ \boldsymbol{P_{21}} & \boldsymbol{P_{22}} & \cdots & \boldsymbol{P_{2n}} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{P_{n1}} & \boldsymbol{P_{n2}} & \cdots & \boldsymbol{P_{nn}} \end{bmatrix} \text{(Formula2-2)}$$

Thus, Markov Processisrepresented as$\boldsymbol{MP} = < S, P >$, where $\boldsymbol{S} = \{\boldsymbol{S_1}, \boldsymbol{S_2}, \dots, \boldsymbol{S_n}\}$ is the set listing all the possible state$\boldsymbol{S_i}$, and $\boldsymbol{P}$is the probability of state transferring from$\boldsymbol{S_i}$ to $\boldsymbol{S_j}$at any time. By using the probability distribution matrix $\boldsymbol{P}$, we can easily get the transfer probabilityof any state at any time.

2.2.2Markov Decision Process

Markov Decision Process can be used as a mathematical model for describing reinforcement learning problems by introducing the concept of Action $\boldsymbol{A}$ and Reward $\boldsymbol{R}$. We already got

$\boldsymbol{MP} = < S, P >$, by adding $\boldsymbol{A}$ and $\boldsymbol{R}$, we can now represent Markov Decision Process as $\boldsymbol{MDP} = < S, A, P, R >$, where:

$\boldsymbol{S} = \{\boldsymbol{S_1}, \boldsymbol{S_2}, \dots, \boldsymbol{S_n}\}$ is the set listing all the possible states of agent.

$\boldsymbol{A} = \{\boldsymbol{A_1}, \boldsymbol{A_2}, \dots, \boldsymbol{A_n}\}$ is the set listing all the possible actions of agent.

$\boldsymbol{P}$ is probability function, calculating the probability of transferring from state$\boldsymbol{s}$ to$\boldsymbol{s'}$ by taking action $\boldsymbol{a}$.

$$\boldsymbol{P} = \boldsymbol{P_{ss'}^{a}} = \mathbf{P}[\boldsymbol{S_{t+1}} = \boldsymbol{s'} | \boldsymbol{S_t} = \boldsymbol{s}, \boldsymbol{A_t} = \boldsymbol{a}] \text{(Formula2-3)}$$

$\boldsymbol{R}$ is reward function, providing the mathematical expectation oninstantreward of transferring from state $\boldsymbol{s}$ to$\boldsymbol{s'}$ by taking action $\boldsymbol{a}$at time$\boldsymbol{t}$.

$$\boldsymbol{R} = \boldsymbol{R_s^a} = \mathbf{E}[\boldsymbol{R_{t+1}} | \boldsymbol{S_t} = \boldsymbol{s}, \boldsymbol{A_t} = \boldsymbol{a}] \text{(Formula2-4)}$$

Other definitions that help to understand MDP model:

$\boldsymbol{\gamma}$is called discount factor with value ranges between 0 to 1. This parameter is added in front of every instant$\boldsymbol{R}$ when sum up all the reward to produce the total reward, the total reward is also called Return.By adding in $\boldsymbol{\gamma}$, MDP model will have a better presentation skill. When $\boldsymbol{\gamma} = \mathbf{0}$, the agent will ignore all the reward that might be earn in the future, only focus on instant reward, this is very short-view choice. When $\boldsymbol{\gamma} = \mathbf{1}$, all the future will be considered have equally importance as the instant reward. When $0 < \boldsymbol{\gamma} < 1$, we can assume that the future reward is discounted or its value depreciate by time.$\boldsymbol{\gamma}$increases the 'horizon' of agent to solve 'delay satisfaction' problem.

Assuming that the agent is exploring a maze, there are two paths to choose from: one of which is shorter with less reward, the other is longer but with more reward. When$\boldsymbol{\gamma}$is closer to 0, the robot tends to be 'greedy'with short-termreward; conversely, when$\boldsymbol{\gamma}$ is closer to 1, the robot needs to consider not only the instant reward, but also the potential long-term reward the following steps. This is howthe decision-making process changes by increasing$\boldsymbol{\gamma}$.

$\boldsymbol{G_t}$is the sum of all the instant reward obtained by every state transition. The goal of reinforcement learningproblem is to maximize this cumulative reward:

$$\boldsymbol{G_t} = \boldsymbol{R_{t+1}} + \boldsymbol{\gamma R_{t+2}} + \boldsymbol{\gamma^2 R_{t+3}} \dots = \sum_{\boldsymbol{k=0}}^{\infty} \boldsymbol{\gamma^k R_{t+k+1}} \text{(Formula2-5)}$$

$\boldsymbol{\pi}(\boldsymbol{a}|\boldsymbol{s})$is the policy mapping from state set$\boldsymbol{S}$to action set $\boldsymbol{A}$.$\boldsymbol{\pi}$representing the distribution function (probability of taking each action) of action $\boldsymbol{A}$whengiven state$\boldsymbol{S_t} = \boldsymbol{s}$.

$\boldsymbol{V_\pi(s)}$is state value function. State value is defined as themathematical expectation of return from state $\boldsymbol{S_t = s}$, following policy $\boldsymbol{\pi(a|s)}$ and sampling on all the actions $\boldsymbol{A}$.$\boldsymbol{V_\pi(s)}$can help us to measure the value of every state in an intuitively way. In practice, we usually depend on $\boldsymbol{Q_\pi(s,a)}$ to evaluate state-action value, $\boldsymbol{V_\pi(s)}$ is only used for theoretical derivation.

$$\boldsymbol{V_\pi(s) = E_\pi[G_t|S_t = s] = E_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}\,|S_t = s\right]}\text{(Formula2-6)}$$

$\boldsymbol{Q_\pi(s,a)}$is state-action value function. State-action value is defined as themathematical expectation of return from state $\boldsymbol{S_t = s}$, following policy $\boldsymbol{\pi(a|s)}$ and sampling on a particular action $\boldsymbol{A_t = a}$.$\boldsymbol{Q_\pi(s,a)}$ can help us to measure the value of every state-action pairs in an intuitively way. By comparing the state-action value of state-action pairs, the agent can choose the action with higher value to produce a higher expectation of return.

$$\boldsymbol{Q_\pi(s,a) = E_\pi[G_t|S_t = s, A_t = a] = E_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}\,|S_t = s, A_t = a\right]}\text{(Formula2-7)}$$

If we do the sampling on allactions in $\boldsymbol{A}$, then will find theconnections between $\boldsymbol{V_\pi(s)}$and $\boldsymbol{Q_\pi(s,a)}$:

$$\boldsymbol{V_\pi(s) = \sum_{a\in A}\pi(a|s) * Q_\pi(s,a)}\text{(Formula2-8)}$$

Solving the MDP is to find an optimal policyto maximize the cumulative reward represented by Formula (2-7). MDP problem now turns into an optimization problem.

2.3 Reinforcement Learning Algorithm

2.3.1 Bellman Equation and Optimal Bellman Equation

Bellman equation, or called as dynamic programming equation, is used to represent the relationship between adjacent/sequential states. Through Bellman equation, the optimal decision-making problem of one state can be transformed into the optimal decision-making sub-problem of next state [7]. By doing the equation iteration, the optimal decision-making problem of the initial state can be iterated step by step, becoming the optimal decision-making sub-problem of the terminal state, which can be solved easily(because the sub-problem will have small state and action space). Most of the optimal decision-making problems can be solved by using state valuefunction to construct Bellman equation.

By using the definition of $\boldsymbol{G_t}$in MDP, the Bellman equation of state value function and state-action value function can be deduced:

$$\boldsymbol{V_\pi(s) = E_\pi[G_t|\, S_t = s] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right]}$$
$$\boldsymbol{= E_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \,|\, S_t = s]}$$
$$\boldsymbol{= E_\pi[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \cdots)|\, S_t = s]}$$
$$\boldsymbol{= E_\pi[R_{t+1} + \gamma\, G_{t+1}|\, S_t = s]}$$
$$\boldsymbol{= E_\pi[R_{t+1} + \gamma\, V_\pi(S_{t+1})|\, S_t = s]}\text{(Formula2-9)}$$

Do the same deformationon state-action value function:

$$\boldsymbol{Q_\pi(s,a) = E_\pi[R_{t+1} + \gamma\, Q_\pi(S_{t+1},\, A_{t+1})|\, S_t = s,\, A_t = a]}\text{(Formula2-10)}$$

Figure 2-2 shows the recursive process, white pointsstandfor state, black points stand for is action:

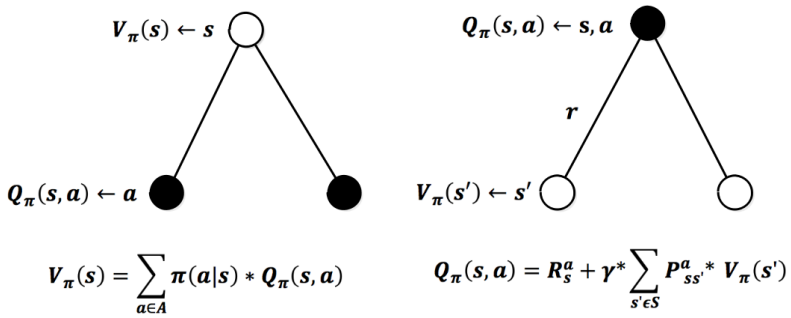


Figure 2-2: Recursive process ofstate-action valuefunction

Figure 2-3 shows the combination of the two processes above, this is how the Bellman Equation of state value function conducted.

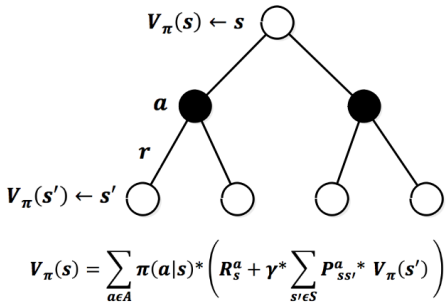


Figure 2-3: Recursive process ofstate valuefunction

Bellman Equation transforms the problem of maximizing$\boldsymbol{V(s)}$ is into a sub-problem of maximizing $\boldsymbol{V(s')}$, finally into $\boldsymbol{V(terminal\ state)}$, solves MDP problem recursively.

Given apolicy$\boldsymbol{\pi(a|s)}$, the state value $\boldsymbol{V_\pi(s)}$of any state can be calculated according to formula in Figure 2-3. Similarly if we solve the Bellman equation (i.e. find the optimal state value function), we find the optimal policy$\boldsymbol{\pi_*}$,solve corresponding this MDP, solve the corresponding reinforcement learning problem.

From all the value function generated by all the possible policy$\boldsymbol{\pi(a|s)}$, the optimal state value function is the state value function that maximizes the value of all the states $\boldsymbol{s}$, and the optimal state-action value functionis the state-action value function that maximizes the value of all the state-action pairs$\boldsymbol{(s,a)}$.

$$\boldsymbol{V_*(s) = \max_\pi V_*(s)}\text{(Formula2-11)}$$

$$\boldsymbol{Q_*(s,a) = \max_\pi Q_\pi(s,a)}\text{(Formula2-12)}$$

The optimal value function represents the best performance that MDP may achieve. With the optimal value function, the optimal value of any state-action pairs is known. The optimal policy$\boldsymbol{\pi_*}$ is to select the action$\boldsymbol{a}$that has the maximum state-actionvalue$\boldsymbol{Q(s,a)}$ at state $\boldsymbol{s}$, and the MDP solution is obtained.

For policies, we prescribe a ***policy comparison rule***to compare the pros and cons:

$$\boldsymbol{for\ all\ s \in S, if\ V_\pi(s) \geq V_{\pi'}(s), then\ \pi\ is\ better\ than\ \pi'}$$

To obtain the optimalpolicy, all polices $\boldsymbol{\pi}$ are sorted according to the above rule. The optimal policy has properties:

a. Any MDP problem must have more than one optimal policy, which is superior to (at least not inferior to) other polices;

b. All optimal polices have the same state value function and state-actionvalue function:

$$\boldsymbol{V_{\pi*}(s) = V_*(s) = \max_\pi V_\pi(s)}\text{(Formula2-13)}$$

$$\boldsymbol{Q_{\pi*}(s,a) = Q_*(s,a) = \max_\pi Q_\pi(s,a)}\text{(Formula2-14)}$$

To generate the optimal policy$\boldsymbol{\pi_*}$: always select the action $\boldsymbol{a}$with the largest $\boldsymbol{Q_*(s,a)}$value.

$$\boldsymbol{\pi_*(a|s) = \begin{cases} 1 & if\ a = \underset{a\in A}{argmax}\ Q_*(s,a) \\ 0 & else \end{cases}}\text{(Formula2-15)}$$

Substituting formula (2-13) (2-14) into the formula in Figure 2-2, get the optimal state value function and state-action value function:

$$V_*(s) = \max_a R_s^a + \gamma * \sum_{s' \epsilon S} P_{ss'}^a * V_*(s') \text{(Formula2-16)}$$

$$Q_*(s,a) = R_s^a + \gamma * \sum_{s' \epsilon S} P_{ss'}^a * \max_{a'} Q_*(s', a') \text{(Formula2-17)}$$

With the optimal state and state-action value function, the value of each state s and state-action pair(s, a) can be obtained. According to ***policy comparison rule***, the optimal policywill be found and the solution of reinforcement learning problem will be obtained.

2.3.2MC Method and TD Method

Agents need to learn from the environment by interactingto get some experience (sampling) for updating the current policy.Monte-Carlo method (MC) and Temporal-Difference method (TD) are two learning method for agent to make use of their experience.

If the probability function P of an MDP is unknown, we can not use formula 2-16 or 2-17 to obtain optimal value function. Instead, MC or TC method will be used to estimate the state value and state-action value.

MC method does sampling on MDP randomly and simulatesmany complete state transition sequences. Complete state transition sequence refers to a whole state transition sequence of an agent transfer from state $s$to the termination state. For any state s, the state value is calculated as the average return$G_t$of all complete state transition sequences that containing s.

By sampling, MC method does not depend on $P$to calculate value function but learns directly from the complete state transition sequences, then approximates state value withthe average returnG_t of these sequences. In theory, the more complete state transition sequences are obtained by sampling, the more abundant the samples are, and the more accurate the estimation results are.

MC method is used to compare return value$G_t$so as to evaluate which policy is better. Incremental mean method is used to calculate the average value of the state.

$$\mu_k = \frac{1}{k}\sum_{j=1}^{k} x_j = \frac{1}{k}\left(x_k + \sum_{j=1}^{k-1} x_j\right) = \frac{1}{k}(x_k + (k-1)\mu_{k-1})$$

$$= \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1}) \text{(Formula2-18)}$$

In incremental mean method, after a new data is sampled, the updating of average value does not need any historical sample. The new average value will be generated only from the old average value$\boldsymbol{\mu_{k-1}}$, the new sample$\boldsymbol{x_k}$ and totalsampling times$\boldsymbol{k}$.

When sample $\boldsymbol{x_k}$ is generated by the k-th times sampling, calculates the difference between$\boldsymbol{x_k}$ and the old average value$\boldsymbol{\mu_{k-1}}$, then multiplied the difference by the reciprocal of the sampling times $\boldsymbol{k}$. Replace $\boldsymbol{\mu_{k-1}}$, $\boldsymbol{x_k}$ and$\boldsymbol{k}$ inincremental mean method with return $\boldsymbol{G_t}$, value function$\boldsymbol{V(S_t)}$, the number of visits to state$\boldsymbol{S_t}$(visits state$\mathbf{S_t}$for the $\boldsymbol{K}$-th times).What's more, registersthe reciprocal of the sampling times $\boldsymbol{k}$as $\boldsymbol{\alpha}$with range from 0 to 1, $\boldsymbol{\alpha}$ is called learning rate, which decides how much to learn from the $\boldsymbol{K}$-th samples.As shown in Formula 2-19.

$$\boldsymbol{K \leftarrow \mathrm{K} + 1}$$

$$\boldsymbol{V(S_t) \leftarrow V(S_t) + \frac{1}{K}(G_t - V(S_t))}$$

$$\boldsymbol{V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t))} \text{(Formula2-19)}$$

MC method needs complete state transition sequence, it must wait until finishing the sampling episode to obtain $\boldsymbol{G_t}$ for updating the state value estimation (episode-dependence). Later, a new method was proposed to solve the problem of episode-dependence: Temporal-Difference method(TD).

TD method only samples on part of an episode, learning the incomplete state transition sequence, and updating the state estimation at the end of each time step instead of episode. Incomplete state transition sequence refers to an incomplete state transition sequence of an agent transfer from state ***s***to any state.

TD method is bootstrapping and the updating of state value and state-action value depends on other known statevalue and state-action value:

$$\boldsymbol{V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))} \text{(Formula2-20)}$$

$$\boldsymbol{Q(S_t, A_t) \leftarrow \mathrm{Q}(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))} \text{(Formula2-21)}$$

TD method in formula 2-20 calculates the value of state $\boldsymbol{S_t}$: add instant reward$\boldsymbol{R_{t+1}}$with$\boldsymbol{\gamma}$ multiply by the value of next state $\boldsymbol{V(S_{t+1})}$, multiplying the sum with learning rete $\boldsymbol{\alpha}$. The sum consists of return and the value of next state, is used to replace $\boldsymbol{G_t}$used in MC method above.

TD method can start learning before the end of episodes, or learning without return (if the episode lasts forever). However, MC methodonly start learning at the end of episodes. When updating the state value, the target value$\boldsymbol{R_{t+1} + \gamma V(S_{t+1})}$in TD is a biased value estimate of $\boldsymbol{S_t}$, while MC updates the value of $\boldsymbol{S_t}$by using the actual value of return$\boldsymbol{G_t}$, which is an unbiased estimate of the state value under a certain policy. Although the estimation of state value obtained by TD method is biased, the mean-square error of TD method is lower than MC method, and it is sensitive to initial value. At the same time, it benefits from the flexibility of updating the learning value of TD. As a result, TD method is usually more efficient than MC method.

2.3.3Greedy Policy

Greedy policy is needed for MC and TD in sampling: the action selection strategy is always to select the action with the maximum state-actionvalue, by using greedy policy, we can significantly accelerate the speed of finding the optimal policy in dynamic programming. Complete greed policy does not take enough into account about the state that actually exists but not experience when sampling, or does not accurately estimate the state value of states that have not experienced many times, which may lead to the existence of some more valuable but unexplored states, so complete greed policy is a one-sided policy.

ε-Greedy policy optimizes the complete greedy policy: in the exploration process, agent has the probability of $1-\varepsilon$to choose the action with the maximumstate- action value, and the probability of $\varepsilon$to choose randomly from other actions. ε-Greedy greed policy ensures the probability of exploring unknown states in sampling:

$$\boldsymbol{\pi_*(a|s)} = \begin{cases} \mathbf{1}-\varepsilon & \boldsymbol{if\ a = \overset{argmax}{a\in A}\ Q_*(s,a)} \\ \varepsilon & \mathbf{else} \end{cases} \text{(Formula2-22)}$$

On one hand, ε-Greedy policy does not want to lose state-action pairs with higher value, on the other hand, with the number of sampling increase, ε-Greedy will reducethe value of $\varepsilon$to make algorithm terminate atan optimal policy.

2.3.4 Q-Learning Algorithm

Q-Learning is a typical value iteration algorithm and a typical model-free learning algorithm. In practical reinforcement learning problems, agent often dives into the problem without any

priori knowledge about the environment (that is, unknown probability distribution function $\boldsymbol{P}$of model).

To obtain experience, the agent need to explore the environmentactively and collect state transition sequences from interacting with environment. Agent learns from the environment by trial and error, graduallyimproves thepolicy, the algorithm will finally end at an optimal policy.

Q-Learning inherited the ideas of TD method in Formula 2-21. Consider state transition sequences $(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{r}, \boldsymbol{s}', \boldsymbol{a}')$ in Figure 2-4, we can write an equationin based on TD method in Formula 2-23.

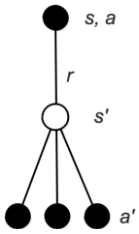


Figure 2-4: State transition sequences

$$\boldsymbol{V}(\boldsymbol{s}) \leftarrow \boldsymbol{V}(\boldsymbol{s}) + \boldsymbol{\alpha}(\boldsymbol{R} + \boldsymbol{\gamma}\boldsymbol{V}(\boldsymbol{s}') - \boldsymbol{V}(\boldsymbol{s}))\text{(Formula2-23)}$$

In order to show action selection intuitively, Q-Learning uses iteration on state-action value instead of state value:

$$\boldsymbol{Q}(\boldsymbol{s}, \boldsymbol{a}) \leftarrow \boldsymbol{Q}(\boldsymbol{s}, \boldsymbol{a}) + \boldsymbol{\alpha}(\boldsymbol{R_{t+1}} + \boldsymbol{\gamma}\, \underset{\boldsymbol{a}'}{\boldsymbol{max}}\, \boldsymbol{Q}(\boldsymbol{s}', \boldsymbol{a}') - \boldsymbol{Q}(\boldsymbol{s}, \boldsymbol{a}))\text{(Formula2-24)}$$

In state $\boldsymbol{s}$, the agent judges and executes action$\boldsymbol{a}$ according to policy (this is action selection policy), leads to a state-action pair $(\boldsymbol{s}, \boldsymbol{a})$ and gets instant reward $\boldsymbol{r}$ from environment, then enters the next state $\boldsymbol{s}'$. In state $\boldsymbol{s}'$, the agent finds all the possible action set$\boldsymbol{A}$, but does not execute any action immediately, instead,the agent calculates the value of all possible next state-action pairs'value $\boldsymbol{Q}(\boldsymbol{s}', \boldsymbol{a}')$, then, the agent chooses the action$\boldsymbol{a}'$corresponding to the maximum value $\boldsymbol{Q}(\boldsymbol{s}', \boldsymbol{a}')$ as the next step (this is value updating policy). After that, the sum of $\boldsymbol{Q}(\boldsymbol{s}', \boldsymbol{a}')$and $\boldsymbol{r}$is used to update the last state-action pair's value $\boldsymbol{Q}(\boldsymbol{s}, \boldsymbol{a})$.

In the right side of Formula 2-24, $\boldsymbol{R_{t+1}} + \boldsymbol{\gamma}\, \underset{\boldsymbol{a}'}{\boldsymbol{max}}\, \boldsymbol{Q}(\boldsymbol{s}', \boldsymbol{a}')$ is the target value and $\boldsymbol{Q}(\boldsymbol{s}, \boldsymbol{a})$ is the current value, the difference between them is called error. With the system closer to convergence, error will gradually decrease to zero. The same, error can be used as an indicator of convergence (end of the algorithm and find an optimal policy).

In Q-Learning algorithm, the action selection policy is ε-Greedy, but the value updating policy is complete greedy: always use the maximum of $\boldsymbol{Q}(\boldsymbol{s}', \boldsymbol{a}')$ to update $\boldsymbol{Q}(\boldsymbol{s}, \boldsymbol{a})$.Q-Learning algorithm can not only make agent try to explore unknown part of environment (action selection policy is ε -Greedy), but also increase the speed of convergence to an optimal policy (value update policy is complete greedy). If both the action selection policy and value updating policy are complete greedy, that will be SARSA algorithm.

Q-Learning algorithm flow is as follows [8]:

a. Random initialization $\boldsymbol{Q}(\boldsymbol{s}, \boldsymbol{a})$ with a value as small as possible

b. For current state $\boldsymbol{s}$, action $\boldsymbol{a}$ is selected using ε-Greedy policy: $\boldsymbol{a} = \boldsymbol{random}(\boldsymbol{A})$with the probability of ε, and $\boldsymbol{a} = \boldsymbol{arg}\,\boldsymbol{\max_{a}}\, \boldsymbol{Q}(\boldsymbol{s}, \boldsymbol{a})$with the probability of $\boldsymbol{1} - \boldsymbol{\varepsilon}$

c. execute action $\boldsymbol{a}$, obtain reward $\boldsymbol{r}$ and reach state $\boldsymbol{s}'$, get state transition sequence$(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{r}, \boldsymbol{s}')$

d. update state-action pair's value: $\boldsymbol{Q}(\boldsymbol{s}, \boldsymbol{a}) \leftarrow \boldsymbol{Q}(\boldsymbol{s}, \boldsymbol{a}) + \boldsymbol{\alpha}(\boldsymbol{R_{t+1}} + \boldsymbol{\gamma}\,\boldsymbol{\max_{a'}}\, \boldsymbol{Q}(\boldsymbol{s}', \boldsymbol{a}') - \boldsymbol{Q}(\boldsymbol{s}, \boldsymbol{a}))$

e. $\boldsymbol{s} \leftarrow \boldsymbol{s}'$**,** return to b.

2.3.5 Neural Network Based Algorithm

The states and actions in MDP are discrete and the scale of reinforcement learning problems is small, while the scale of states and actions of actual problems is usually large andthe states are usually continuous. If use table to represent the state-action pairs, the problem representation space will become huge and reduce computational efficiency, even the solution will not be obtained.

After laying groundwork knowledge in 2.2, it has been determined that the solution of reinforcement learning problem needs to find the optimal state value function and state-action value function. The value function in MDP is discrete, because it uses a tableor dictionary to store state and action, which can not be directly used to find state value in continuous state space. Because the idea of finding the optimal value function is consistent, the MDP model can be generalized: discrete value function can be replaced by a continuous approximate value function to evaluate the value of continuous state.

By introducing appropriate parameters and choosing appropriate state features, neural network can approximate the value function and calculate statesand state-action pairs' value. For the neural networkwith parameters, as long as the parameters are determined, the value of states, state-action pairs can be approximated. This design does not need to store the value of each state or state-action pair, but only the neural network's structure and parameters. In this way, the problem of finding the optimal value function in reinforcement learning is transformed into designing a neural network(an optimal approximate function) and findingthe appropriate parameters.

Approximate functionscan be into linear approximation or non-linear approximation. The expressive power of non-linear approximation is stronger than that of linear approximation, so most of complex problem adopt non-linear approximation. Non-linear approximation can use gradient descent algorithm on parameters to train neural network. Neural network is introduced to solve the reinforcement learning problemswith large scale of states and actions. Practice has proved that it has achieved good results.

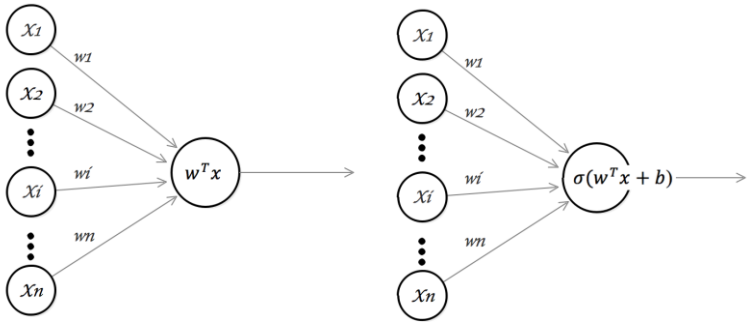


Figure 2-5: linear approximation and non-linear approximation

A feasible idea is to use neural network to approximately estimate the value function instead of doing value iteration to find state and state-action value (like Q-Learning algorithm does). Establish a neural network with a state value approximation function $\boldsymbol{V}^{\wedge}$and parametersset$\boldsymbol{\theta}$. The network's input is a continuous variable$\boldsymbol{s}$, which represents the state, and the output is an approximate value$\boldsymbol{V}^{\wedge}(\boldsymbol{s},\boldsymbol{\theta})$generated by the network. By adjusting the parameters, the approximation function $\boldsymbol{V}^{\wedge}$will gradually approach thefinalstate valueand convergenceto an optimalpolicy $\boldsymbol{\pi}$:

$$\boldsymbol{V}^{\wedge}(\boldsymbol{s},\boldsymbol{\theta}) \approx \boldsymbol{V}_{\boldsymbol{\pi}}(\boldsymbol{s})\text{(Formula2-25)}$$

Similarly, establish a neural network with a state-action value approximation function $\boldsymbol{Q}^{\wedge}$ and parametersset$\boldsymbol{\theta}$.The network's input are two continuous variable $\boldsymbol{s}$ and $\boldsymbol{a}$, representing states and actions. The output is the value of all state-action pairs' value $\boldsymbol{Q}^{\wedge}(\boldsymbol{s},\boldsymbol{a})$ corresponding to the particular state$\boldsymbol{s}$. By adjusting the parameters, the approximation function $\boldsymbol{Q}^{\wedge}$ will gradually approach the final state-action value and convergence to an optimal policy $\boldsymbol{\pi}$. As a result, the network is the approximate expression of the state-action value function$\boldsymbol{Q}^{\wedge}(\boldsymbol{s},\boldsymbol{a})$:

$$\boldsymbol{Q}^{\wedge}(\boldsymbol{s},\boldsymbol{a}) \approx \boldsymbol{Q}_{\boldsymbol{\pi}}(\boldsymbol{s},\boldsymbol{a})\text{(Formula2-26)}$$

In the value approximation function, the variable $\boldsymbol{s}$ representing the state is eigenvector composed of a set of data, each element of the eigenvector is a feature of $\boldsymbol{s}$, the value of element is eigenvalue. The parameter set$\boldsymbol{\theta}$ is also represented as a vector, the value of every parameters is solved by establishing an objective function, training the neural network and using the gradient descent method.

Universal Approximation Theorem points out that feedforward neural networks can fit functions of arbitrary complexity with arbitrary accuracy only if they have a single hidden layer anda finite number of neurons. A multi-layer neural network can have strong non-linear approximation ability by using non-linear functions such as Rectified Linear Unit (ReLU), Sigmoid and hyperbolic tangent function (tanh) as activation function [9].

In theory, any function can be used as an approximation function. Neural network is a commonly used non-linear approximation function. The basic unit of the neural network is the non-linear neurons. The neurons are arranged in the same layer and interconnected in different layers to realize the complex non-linear approximation structure. Non-linear approximation introduced bias $\boldsymbol{b}$ and non-linear activation function $\boldsymbol{\sigma}$to the neurons based on the linear approximation (shown in Figure 2-5). Therefore, the final output of a single neuron is a non-linear structure with abundant fitting contents:

$$\boldsymbol{y} = \boldsymbol{\sigma}(\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x} + \boldsymbol{b})\text{(Formula2-27)}$$

The value function updating equation using TD method in Formula 2-21:

$$\boldsymbol{Q}(\boldsymbol{S_t},\boldsymbol{A_t}) \leftarrow \mathbf{Q}(\boldsymbol{S_t},\boldsymbol{A_t}) + \boldsymbol{\alpha}(\boldsymbol{R_{t+1}} + \boldsymbol{\gamma Q}(\boldsymbol{S_{t+1}},\boldsymbol{A_{t+1}}) - \boldsymbol{Q}(\boldsymbol{S_t},\boldsymbol{A_t}))$$

In the learning process, there are deviations in the estimationof state-action value:

$$\boldsymbol{R_{t+1}} + \boldsymbol{\gamma Q}(\boldsymbol{S_{t+1}},\boldsymbol{A_{t+1}}) - \boldsymbol{Q}(\boldsymbol{S_t},\boldsymbol{A_t}) \neq \mathbf{0}$$

When each time step is finished, it is necessary to update the value functionalong the direction of the target value with learning rate $\boldsymbol{\alpha}$. After several times of updating, the value function tends to be balanced, which means that the estimated target value $\boldsymbol{R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})}$ of the approximate function is the same as the actual value $\boldsymbol{Q(S_t, A_t)}$. By replacing $\boldsymbol{Q(S_t, A_t)}$ in Formula 2-21 with $\boldsymbol{\hat{Q}(S_t, A_t, \theta)}$, the new formula of approximate value function with $\boldsymbol{\theta}$is obtained:

$$\boldsymbol{\hat{Q}(S_t, A_t, \theta) \leftarrow \hat{Q}(S_t, A_t, \theta) + \alpha(R_{t+1} + \gamma \hat{Q}(S_{t+1}, A_{t+1}, \theta) - \hat{Q}(S_t, A_t, \theta))}$$

(Formula2-28)

Assuming that finding appropriate parameters$\boldsymbol{\theta}$ make the value function converge and no longer update, it means finding the final value function or the optimal value function based on the optimal policy. However, in reality, it is difficult to find the perfect $\boldsymbol{\theta}$ to make the Formula 2-29 fully valid. At the same time, due to the rely on the sampled data when training approximation function, even if all the state transmission sequences obtained by the former sampling fulfill the Formula 2-29, it still can not guarantee that all the state transmission sequences obtained by later sampling fulfill the same formula.

$$\boldsymbol{R_{t+1} + \gamma \hat{Q}(S_{t+1}, A_{t+1}, \theta) = \hat{Q}(S_t, A_t, \theta)}\text{(Formula2-29)}$$

Using the $\boldsymbol{M}$state transmission sequencesobtained by sampling to calculate themean-square errorof the actual value$\boldsymbol{\hat{Q}(S_t, A_t, \theta)}$ and the target value $\boldsymbol{R_{t+1} + \gamma \hat{Q}(S_{t+1}, A_{t+1}, \theta)}$, we can construct the objective function$\boldsymbol{J(\theta)}$, and use the objective functionto judge the convergence of approximate value function. The convergence of approximate value function is reflected in the decreasing value of$\boldsymbol{J(\theta)}$. We know from Formula 2-30 that $\boldsymbol{J(\theta)}$is always positive,at the same time, and the objective function exists a minimum $\boldsymbol{J(\theta) = 0}$.

$$\boldsymbol{J(\theta) = \frac{1}{2M}\sum_{k=1}^{M}[R_k + \gamma \hat{Q}(S_{k+1}, A_{k+1}, \theta) - \hat{Q}(S_k, A_k, \theta)]^2}\text{(Formula2-30)}$$

For the partial derivatives of multivariate functions with respect to all the parameters, the partial derivatives of all the parameter in the form of vector is gradient.,The points on the function graph increasetheirs function values fastest along the direction of gradient vector; also decrease theirs function values fastest along the opposite direction of gradient vector [10]. When the gradient value of a point is equal tozero, the function obtains a local maximum or minimum value at that point. That is to say, along the gradient vector's direction, it is easier to find the extreme value of the function.

The objective function $\boldsymbol{J(\theta)}$is the mean-square errorof the actual value$\boldsymbol{Q^{\wedge}(S_t, A_t, \theta)}$ and the target value $\boldsymbol{R_{t+1} + \gamma Q^{\wedge}(S_{t+1}, A_{t+1}, \theta)}$. The purpose of gradient descent is to get the minimum value of $\boldsymbol{J(\theta)}$ as far as possible so that to accelerate the convergence of approximate value function.

There two possible neural network structure, shown in Figure 2-6.

Given a state$\boldsymbol{s}$, Q-Learning algorithm requires to calculatethe $\boldsymbol{Q(s, a)}$for all possible actions, then choose the action $\boldsymbol{a}$ corresponding to the largest $\boldsymbol{Q(s, a)}$ value to execute, so the forward propagation calculation is needed for all possible actions every time when calculating $\boldsymbol{Q(s, a)}$. If the structure of the neural network is very complex, the forward propagation calculation will surely take a long time, cause the low efficiency obviously. If the action is discrete and thescale of actions is acceptable, the network can be designed as structure inthe right side of Figure 2-6. This structure needs only one forward propagation calculation to obtain all $\boldsymbol{Q(s, a)}$. This structure is adopted by the DQN algorithm in the following paper.

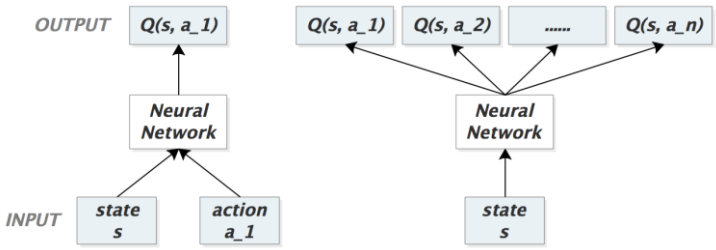


Figure 2-6: two kinds od neural network structure

There are two main advantages comparing the traditional Q-Learning algorithm to the Q-Learning algorithm combined with neural network. First, when the scale of states and actions is larger (for example, the input state is a two-dimensional image, or in a higher dimensional space), the traditional Q-Learning algorithm has the problem of curse of Dimensionality. The storage needed to maintain the $\boldsymbol{Q(s, a)}$value table is large, even can not be realized in program; while the neural network, regardless of the large scale of states and actions,they just store the parameters. Second, the traditional Q-Learning algorithm requires discrete and finite states and actions. When the state or action space is continuous, the problem needs to be discretized. For example, in pendulum car problem, the balance angle of the rod range from -24 degrees to 24 degrees. If we separated the state every 0.1 degrees, that is -24 degrees, -23.9 degrees, -23.8 degrees... There are more than 100 discrete states. So the

traditional Q-Learning algorithm also has the problem of storage space. In contrast, the neural network can directly accept continuous values as input withoutany processing of input.

3 Design and Algorithm of Obstacle Avoidance System

The design of the system consults the problem representation of OpenAI gym (shown in Figure 3-1). OpenAI gym is a python open source library for standardizing reinforcement learning problem. Gym's framework divides the reinforcement learning problem into two parts: environment and agent. An environment corresponds to a reinforcement learning problem, and the agent is controlled by the algorithm of solving the problem. Environment provides a unified interface env, through which agent can interact with the environment:execute actions, get observation value.

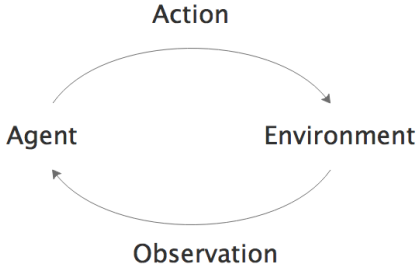


Figure 3-1: OpenAI gym's framework

This chapter will talk about the environment design and algorithm implement of the obstacle avoidance problem. First of all, we need to design the obstacle avoidanceemulating environment, and try to describe this problem in the structure of reinforcement learning problem. Secondly, we need to implement reinforcement learning algorithm for our agent.

3.1 Environment Design

3.1.1 Objects in Environment

The environment has two key objects: obstacle andagent.Objectisrepresented as a single circle in two-dimensional space, we can usethecircle's center coordinate $(\boldsymbol{x}, \boldsymbol{y})$and radius$\boldsymbol{r}$ to determine position and size of any object. Obstacles can be stationary or moving, so it is necessary to record the velocity information of each obstacle, which is expressed as $(\boldsymbol{V_x}, \boldsymbol{V_y})$ **to** represent the velocity components on the X-axis and Y-axis. As shown in Figure 3-2, the origin of the environment is located in the upper left corner of the two-dimensional space, the X-axis is the horizontal axis (X-value increase from left to right along the axis), and the Y-axis is the horizontal vertical axis (Y-value increase from top to down along the axis).In addition, we add a time attributes to each object in order to record their existing time in the environment. The unit of time is frame.

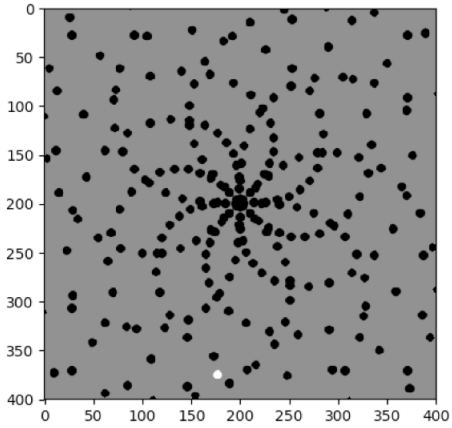


Figure 3-2 environment with objects: the white circlefor agent and the black for obstacles

Define the common base class ***EnvObject***, which contains the common attributes of obstacles and agent: existing time, position (circle center'scoordinates and radius). The attributes

existing time will be implemented by the default update method. Any time when the object needs to be updated, update method will be automatically called and theexisting timeattributes of the objectwill be increased by one frame.

```
class EnvObject:
    def __init__(self):
        self.x = 0
        self.y = 0
        self.r = R
        self.time = 0

    def update(self):
        self.time += 1
```

Figure 3-3: Common base class ***EnvObject***

Define the class ***Agent***, which is inherited from ***EnvObject***, besides the basic method and attributes mentioned above, it also has private attributes init_x, init_yto record initial position, which are used to initialize the position for the agent when resetting the environment for next episode.

```
class Agent(EnvObject):
    def __init__(self, init_x, init_y):
        super(Agent, self).__init__()
        self.x = init_x
        self.y = init_y
        self.init_x = init_x
        self.init_y = init_y

    def reset(self):
        self.x = self.init_x
        self.y = self.init_y
        self.time = 0
```

Figure 3-4: Subclass ***Agent***

Definition class ***Obstacle***, which are also inherited from ***EnvObject***. The classhasprivate attributes vx, vy to record the velocity information of obstacles.Subclass ***Obstacle*** overrides the update method and the method updates both position and existing time of the obstacle. The displacement of obstacle is obtained by multiplying the velocity and the time step. Since the environment is updated 60 times per second, the time step $\boldsymbol{dt} = \frac{\mathbf{1}}{\mathbf{60}}$.

```
class Obstacle(EnvObject):
    def __init__(self):
        super(Obstacle, self).__init__()
        self.vx = 0
        self.vy = 0

    def update(self):
        self.x += self.vx * dt
        self.y += self.vy * dt
        self.time += 1
```

Figure 3-5: Subclass ***Obstacle***

3.1.2 Interaction Logicin Environment

Define class ***Environment*** to manage obstacles and agent and synchronize information. An instantiated environment object needs to maintain a list of obstacles. When the environment needs to be updated, the update method of class ***Environment*** is called. Some important method in the class:

a. ***_generate_obstacles(self***)

This method is called to determine whether a new obstacle needs to be generated. The method defines the generation rules of location and velocity components of an obstacle. If a new obstacleis generated, it will be added to the obstacle list.

```
def _generate_obstacles(self):
    if((not self.done) and (self.time % 12 == 0)):
        offset = math.sin(self.time / 1440.0 * math.pi * 2) * math.pi * 2
        for i in range(8):
            angle = math.pi * 2 * i / 8 + offset
            o1 = Obstacle()
            o1.x = 200
            o1.y = 200
            o1.vx = math.cos(angle) * 90
            o1.vy = math.sin(angle) * 90
            self.obstacles.append(o1)

            angle = math.pi * 2 * i / 8 - offset
            o2 = Obstacle()
            o2.x = 200
            o2.y = 200
            o2.vx = math.cos(angle) * 60
            o2.vy = math.sin(angle) * 60
            self.obstacles.append(o2)
```

Figure 3-6: Obstacle generation function of class ***Environment***

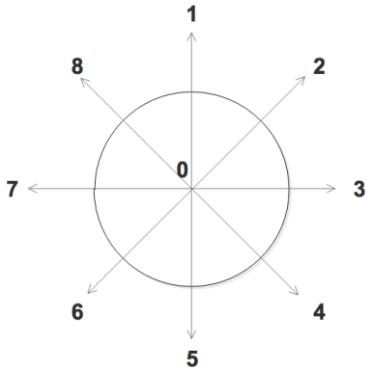
b. ***_update(self, action***)

Update function of class ***Environment***has four parts: obstacles update, agent update, collision detection and time update.

In obstacles update, each obstacle in the list will call their own update method to update location when ***Environment***calls the environment's update method. The coordinate range from***(0, 0)*** to***(400, 400)***, so the area of the environmentsizes***(400, 400)***. If an obstacle is located outside the environment area (i.e. totally invisible), this obstacle object willbe deleted from the obstacle list and will not be updated any more. The criteria for judging whether an

obstacle is outsidethe area: the abscissa is less than ***0***orthe abscissa is larger than ***400***or the ordinate is less than***0*** orthe ordinate larger than ***400***.

In agent update, the location of agent is updated according tothe argument ***action***. Agents can move to eight directions or make no motion, so there are nine alternative actions. Figure 3-7 shows the nine actions' code representation: (0) make no motion; (1) up; (2) up right; (3) right; (4) down right; (5) down; (6) down left; (7) left; (8) up left. It should be noted that the agent's speed is fixed, so the oblique motion (2), (4),(6), (8) needs to multiply the coefficients $\frac{\sqrt{2}}{2}$to both horizontal and verticalvelocity components. If the agent has been close enough to the environment area's boundary and still chooses to move outside, this action will be considered as invalid, and set the agent's all velocity components to 0.




```
'''
Actions:
8 1 2
7 0 3
6 5 4
'''
action_map = [
    (0, 0),(0, -1),(1, -1),
    (1, 0),(1, 1),(0, 1),
    (-1, 1),(-1, 0),(-1, 1)]
```

Figure 3-7: 9 actions of agent and the code representation

In collision detection, the judgement of whether two circlesintersect is used to detect whether an agent collides with an obstacle. The codes traverse all the objects in obstacle list, use Euclidean distance formula to calculate the distance between obstacle's circle center to the agent's circle center. If the distance is less than the sum of two circles' radius, we will conclude the collision between the agent and the obstacle happened. Episodes will end if agent collides with an obstacle or no collision occurs after 3600 frames.

```
# Detect collision
for o in self.obstacles:
    d = (self.agent.x - o.x) ** 2 + (self.agent.y - o.y) ** 2
    if(d <= (self.agent.r + o.r) ** 2):
        # End of episode
        self.done = True
        break
```

Figure 3-8: Code for collision detection

c. ***reset(self***)

After an episode ends, there is a need for a method to clean things up and to start a new episode: initialize the time, delete all the objects in the obstacle list, reset the continuous flag, agent's position and return the new initial observation to the new agent.

```
def reset(self):
    self.time = 0
    self.obstacles = []
    self.done = False
    self.agent.reset()

    observation, _, _ = self.step(0)
    return observation
```

Figure 3-9: ***reset***function of class ***Environment***

d. ***render(self***)

OpenCV is an excellent open source image processing library. Codes here use OpenCV correlation function to render the environment frame by frame, making images to a live scene and showing them in a video. The reason for using images instead of video is that the input of neural network usually shaped in vector, or matrix, which is the same form as pixel data.

Pixel data of image is represented and stored by NumPy array and sizes***400×400***(this is also the size of our environment). In order to facilitate the image processing by neural network based reinforcement learning algorithm, we render the environment on the grayscale image.Grayscale image has only one color channel and the pixel values range from 0 to 255. In the environment, background is grey (the pixel value of grey is 128), obstaclesare black (the pixel value of black is 0), and agent is white (the pixel value of white is255).

Render function is used to build up a visual environment. First, ***np.full()*** is called to create an array with the shape of ***(400, 400)*** and fill it with a value of 128, which draws the background. Then, for every object in the obstacle list, after obtaining the information of their circle center and radius, we draw a solid black circle with ***cv2.circle()***, which draws all the obstacles in the environment. Similarly, a solid white circle is drawn to representthe agent. Up till now, the image of one frame has been rendered. And we need to render a new frame every time the environment is updated.

```
def render(self):
    img = np.full((400, 400), 128, dtype = np.uint8)
    for o in self.obstacles:
        cv2.circle(img, (int(o.x), int(o.y)), int(o.r),\
                   0, -1, lineType = cv2.LINE_AA)
    cv2.circle(img, (int(self.agent.x), int(self.agent.y)),\
               int(self.agent.r), 255, -1, lineType = cv2.LINE_AA)

    return img
```

Figure 3-10:***render***function of class ***Environment***

e. ***step(self, action***)

When the agent chooses an action, the environment will call ***step*** function and pass the action through argument ***action***. In this function, the action will be executed and environment will return the reward for this action and give the agent a new observation of new environment.

Here, the agent and environment interaction is performed every 3 frames, so the observation returned by the functionis the collection of three consecutive frames. Similarly, the action passed through the argument is repeatedly executedon three consecutive frames. In order to cut down the size of problem, the image is reducedfrom***400×400***to ***160×160***, which also reduces the space requirement. Reward setting is very important to the success of algorithm and this part of the research will be discussed in Chapter 4.

```
def step(self, action):
    frames = []
    reward = 0.0

    for i in range(3):
        self._update(action)
        img = self.render()
        cv2.imwrite("img/{}.bmp".format(self.count), img)
        self.count += 1

        img = cv2.resize(img, (160, 160))
        frames.append(img)
        if(not self.done):
            reward += 0.1

    if(self.done and (self.time < 3600)):
        reward -= 1.0

    observation = np.expand_dims(np.stack(frames, axis = -1), axis = 0)
    return observation, reward, self.done
```

Figure 3-11:***step*** function of class ***Environment***

To sum up, the class diagram of class ***Environment*** is shown in 3-12.

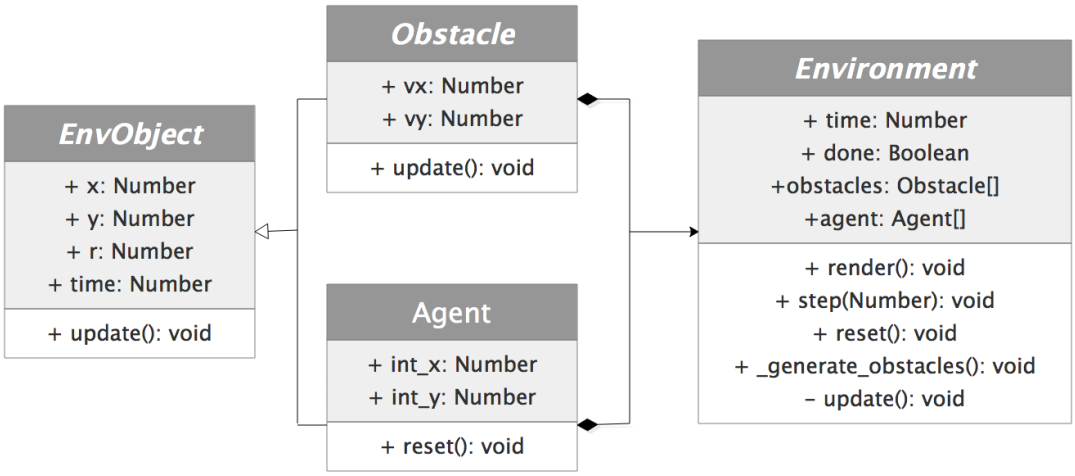


Figure 3-12:UML class diagram of ***Environment***

3.2 Algorithm Implementation

3.2.1 DQN Algorithm

Deep QNetwork algorithm is a deep reinforcement learning algorithm based on deep neural network proposed by DeepMind, it was developed on the basis of Q-Learning algorithm. DeepMind trained neural network modelsuccessfully using DQN algorithm on Atari games[11].Q-Learning algorithm update the state-action value $\boldsymbol{Q}(\boldsymbol{s},\boldsymbol{a})$ iteratively, so the input data of the algorithm are the old value and the output is the new value; but DQN algorithm approximates the value function by training neural network, the input of network is state and the output is state-action value.DQN algorithm makes two special improvementson the basic conventional neural network and Q-Learning algorithm: Experience Replay and Fixed Q Target Network [12].

The experience pool stores the sampled state transition sequences$(\boldsymbol{s},\boldsymbol{a},\boldsymbol{r},\boldsymbol{s}')$. When training the neural network, a fixed size (a.k.a. batch size) state transition sequences is randomly extracted from the experience pool as the training dataset. The purpose of Experience Replay is to reduce the correlation between data over a period of time and simulate a condition of independent and identical distribution for data in training set, which is the prerequisite for applying gradient descent method. Moreover, we can adjust the priority of experience in the experience pool, selectively givingmore opportunities to the low-frequency but more important experiences to be learned in the neural network.

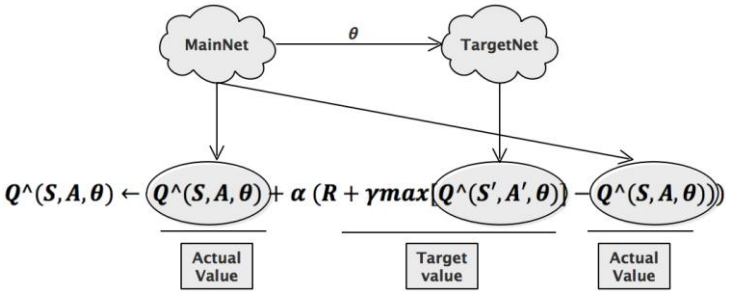
DQN has two sets of neural networks: TargetNet and MainNet. The twoneural networks with the same structure and the same parameters. The parameters of MainNet are updated with training process, while the parameters of TargetNet remain unchanged. After a period of time, the parameters of MainNet tend to be stable (the neural network obtains a convergence), then wecopy the parameters of MainNet to TargetNet (now TargetNet has the same parameters as MainNet), this process are recycled until the algorithm stops. This strategy makes the output of TargetNet to be stable for a period of time and can provide stable monitoring data for training MainNet.Since the calculation of value $\boldsymbol{Q}$ depends on the value function approximated by the neural network, updating the network's parameters can change thestate-action value of current state, but it will certainly affect the state-action value of the next state, as a result, there may be non-convergence or chasing phenomenon when the difference between the target value and actual valuenot decrease.

In Q-Learning algorithm (Figures 3-11),$\boldsymbol{Q}$is updated uses target valueand actual value, target value is the sum of reward $\boldsymbol{R_{t+1}}$and$\boldsymbol{\gamma \underset{a'}{max} Q(s', a')}$, actual valuecomes from the current state-action value$\boldsymbol{Q(s, a)}$. In DQN (Figures 3-12), Target value come from the sum of $\boldsymbol{R_{t+1}}$ and $\boldsymbol{\gamma \underset{a'}{max} Q\hat{}(s', a', \theta)}$ ($\boldsymbol{Q\hat{}(s', a', \theta)}$ is calculated by TargetNet), and actual value$\boldsymbol{Q\hat{}(s, a, \theta)}$ is calculated by MainNet. The TargetNetupdated its parameters only after a period of timethat the target value will not change in a period of time, so as to solve the correlation problem and stabilize the training process.

$$\boldsymbol{Q(S,A) \leftarrow \underbrace{Q(S,A)}_{\text{Actual Value}} + \alpha \underbrace{(R + \gamma max [Q(S',A')]}_{\text{Target value}} \underbrace{-Q(S,A)}_{\text{Actual value}})}$$

Updated value = Actual Value + Learning Rate × difference of Target and Actual

Figures 3-13: Actual value and target value of Q-Learning



Figures 3-14: Actual value and target value of DQN

DQN algorithm flow is as follows:

a. Define the capacity of experience pool:$\boldsymbol{N}$ and update frequency of TargetNet: $\boldsymbol{freq}$;

b. Initialize the parameters of MainNet randomly to $\boldsymbol{\theta}$ and the parameters of TargetNetto $\boldsymbol{\theta}' = \boldsymbol{\theta}$;

c. Initialize the environment and get the first state $\boldsymbol{s}$ and observation;

d. Using ε-Greedy policy to choose an action $\boldsymbol{a}$;

e. Execute $\boldsymbol{a}$, generate sequence sample$(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{r}, \boldsymbol{s}')$andstore it into experience pool. If the experience pool is full, replace the oldest sample.

f. From experience pool, randomly selected aMini-batch of samples as training data;

g. For all the sequence $(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{r}, \boldsymbol{s}')$in Mini-batch, calculate the difference of target value and actual value. If $\boldsymbol{s}'$ is terminal state, set the target value to$\boldsymbol{r}$;

h. Calculate the objective function $\boldsymbol{J}(\boldsymbol{\theta})$, use gradient descent method to update $\boldsymbol{\theta}$;

i. Reset the parameters of TargetNet $\boldsymbol{\theta}' \leftarrow \boldsymbol{\theta}$every$\boldsymbol{freq}$ steps;

j.If $\boldsymbol{s}'$ is terminal state, go to step c, else, go to step d.

3.2.2 Improvements on DQN

3.2.2.1 Double DQN

DQN algorithm has problem of over estimate: the algorithm generally overestimates the value of$\boldsymbol{Q}$, and the estimation error will increase as the number of actionsincrease. DQN algorithm uses the sum of reward $\boldsymbol{R_{t+1}}$and $\boldsymbol{\gamma} \boldsymbol{\max_{a}} \boldsymbol{Q}(\boldsymbol{s}', \boldsymbol{a}', \boldsymbol{\theta})$to estimate the target value of next state $\boldsymbol{s}'$, so we are using the TargetNet to select action $\boldsymbol{a}'$ for next state $\boldsymbol{s}'$, but we can not guarantee that $\boldsymbol{a}'$is optimal ($\boldsymbol{a}'$ is only guaranteed to be the optimal action chosen by TargetNet). Double DQN decouples the selection of$\boldsymbol{a}'$ and the calculation of target value, this dissociation can alleviate the over estimation problem. Double DQN select action$\boldsymbol{a}'$for next state $\boldsymbol{s}'$ using MainNetinstead of TargetNet (but still calculating the target value using TargetNet):

$$\boldsymbol{Target\ value} = \boldsymbol{R_{t+1}} + \boldsymbol{\gamma Q}(\boldsymbol{s}', \boldsymbol{\underset{a'}{argmax}} \boldsymbol{Q}(\boldsymbol{s}', \boldsymbol{a}', \boldsymbol{\theta}), \boldsymbol{\theta}) \text{(Formula3-1)}$$

Compared to the original target value:

$$\boldsymbol{Target\ value = R_{t+1} + \gamma \max_{a'} Q(s', a', \theta)} \text{(Formula3-2)}$$

Obviously,

$$\boldsymbol{Q(s', \operatorname{argmax}_{a'} Q(s', a', \theta), \theta) \leq \max_{a'} Q(s', a', \theta)}$$

This change, to some extent,alleviatesthe over estimateproblem and improves the stability of DQN [13].

3.2.2.2 Prioritized Experience Replay

Suppose that we have a situation: a robot can easily walk out of 50 steps, but after 50 steps, the robot suddenly encounters a complex terrain that is hard to cross and the episode ends here. According to howDQNcollecting data, the data samples in the first 50 steps in experience pool will occupy a high proportion, but the data samples after the 50 steps will be relatively rare. Sincethe training processselectssamples from experience pool uniformly and randomly, the probability of sampling data of the first 50 steps is greater than that of the later steps. Thus, the algorithm trains the samples of "simple" situation repeatedly and paying less attention to the samples of "complex" situation, so that the robot has mastered the first 50 steps of walking, but still can't do anything when encountering the complex terrain.

For an agent who is learning from complex environment, uniform random sampling is not an efficient way to use data, samples have different significance. In order to achieve a balanced training, use prioritizedexperience replay instead of uniform random sampling. Prioritized experience replaydefines the priority of each training sample, the programmer can givea higher priority to the most significant sample by improving the sampling weight.

A reasonable way to quantify priority is TD error. TD error indicates the difference between the target value and actual value, this is also the difference betweenideal and the reality. The larger the TD error of sample $\boldsymbol{i}$is, the greater potential for improving model's accuracy from $\boldsymbol{i}$, the more needs for $\boldsymbol{i}$ to be learned, that is, the higher priority value$\boldsymbol{P(i)}$ it should be given [14].

In prioritized experience replay, the sampling sample $\boldsymbol{i}$ has TD error $\boldsymbol{\delta_i}$ and the samplingprobability is $\boldsymbol{P(i)}$.

$$\boldsymbol{\delta_i = TD\ error = R + \gamma \max_{a'} Q(s', a') - Q(s, a)} \text{ (Formula3-3)}$$

$$\boldsymbol{p_i = |\delta_i| + \epsilon \quad (\epsilon > 0)} \text{(Formula3-4)}$$

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \quad (\alpha > 0) \text{(Formula3-5)}$$

In Formula 3-4, $\epsilon$is a small number, by adding ϵ, some sample with zero TD errorcan also have a probability to be extracted.In Formula 3-5, p_i^α is the priority of sample i, $\alpha$is a hyper parameters with value usually be 0.5.

The management of prioritized experience replay needs an efficient data structure to support the calculation and update weights. When the number of sampled date in experience pool reaches the capacity, sampled data with a lower weights need to be replaced or deleted. An intuitive sampling algorithm likes:

a. Store all p_i^α in array X;

b. Randomly generate value$s$,$s \in [0, \sum_k p_k^\alpha]$;

c. $i \leftarrow 0$;

d. Check the$i$-th element$X[i]$: if $X[i] \leq s$, choose sample i, end algorithm, else continue;

e. $s \leftarrow s - X[i]$, $i \leftarrow i + 1$, go back to d.;

As a result, every time when we select a sample, we need to traverse array X, with time complexity $O(n)$. Eachtraining round needs to obtain a minibatch of samples (usually 32) from the experience pool. When the capacity$n$ of the experience poolincreases, the sampling time $O(n)$ is not really profitable.

Segment tree can reduce the time complexity to $O(log\ n)$. Segment tree is a binary tree withevery node represent an interval, the left subtree for the left half interval and the right subtree for the right half interval, leaf node represents for unit interval. Segment tree with root node [0, n] will have$n$ leaf nodes, and the depth is at most$\lfloor logn \rfloor$+1. Figure 3-15 is a segment tree with 6 leaf nodes.

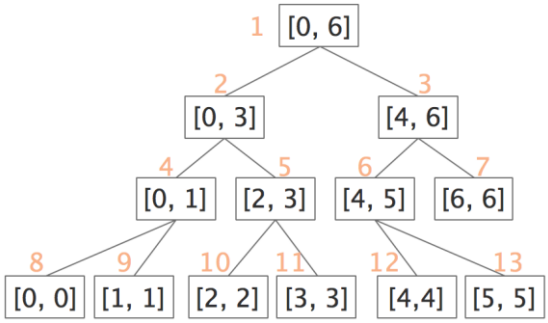


Figure 3-15: Segment tree [0, 6]

The segment tree can recursively query the interval maximum value or calculate the sum of an interval of complexity $\boldsymbol{O(log\ n)}$. Leaf nodes in segment tree records the value of priority$\boldsymbol{p_i^{\alpha}}$, to sum up the segment tree is to sum up the priority$\boldsymbol{p_i^{\alpha}}$. When sampling, a random interval is generated, and the selected samples can bequickly located in the segment tree using prefix sum.

3.2.2.3Dueling Network Architectures

In some situation, no matter what action at is selected at state $\boldsymbol{s}$, it will not have a great impact on the next state $\boldsymbol{s'}$, so the value of$\boldsymbol{Q(s,\ \cdot)}$ and $\boldsymbol{Q(s',\ \cdot)}$willhave not much difference. Assume that the agent is a car, state is the distance between car and destination, action is go forward or backward one meter, $\boldsymbol{s}$=100000m, if **a** =go forward, $\boldsymbol{s'}$=100001m; if **a** =go backward, $\boldsymbol{s'}$=99999m. At this point, we consider no matter which action the agent takes, it has no much effect on the next state $\boldsymbol{Q(s',\ \cdot)}$. In these cases, the significance of calculating state-action value$\boldsymbol{Q(s,\ \cdot)}$ is not as great as that of state value$\boldsymbol{V(s)}$.
As a result, we can dividestate-action value$\boldsymbol{Q}$ into two parts: state value $\boldsymbol{V}$ and advantage $\boldsymbol{A}$[15]:

$$\boldsymbol{Q(s,a;\ \theta,\alpha,\beta) = V(s;\ \theta,\beta) + A(s,a;\ \theta,\alpha)} \quad \text{(Formula3-6)}$$

Dueling Network Architectures only makes a few adjustments in DQN, in Figure 3-16, the below isDueling Network Architectures DQN, this structurelearns features extracted from the previous network layerrespectively in two branches, state value function $\boldsymbol{V}$ and advantage function$\boldsymbol{A}$ arethen combined into$\boldsymbol{Q}$as the output. $\boldsymbol{\theta}$corresponds to the parameters of the common part, $\boldsymbol{\alpha}$ corresponds to the parameters of the fully connected layer of the lower branch, and $\boldsymbol{\beta}$corresponds to the parameters of the fully connected layer of the upper branch.

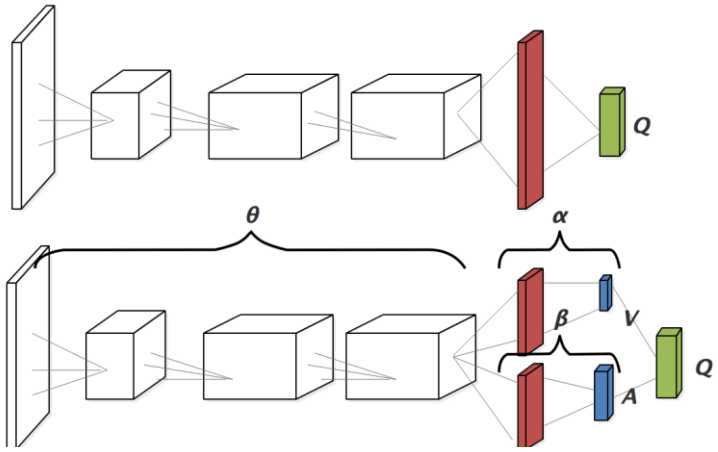


Figure 3-16: Basic DQN and Dueling Network Architectures DQN

Reference [15] also pointed out that if only using Formula 3-6, in some cases, it is difficult for neural network to distinguish state value function$\boldsymbol{V}$ and advantage function$\boldsymbol{A}$, resulting in the failure to identify the respective roles of$\boldsymbol{V}$ and$\boldsymbol{A}$. For example, in a given state has$\boldsymbol{V}(\mathbf{s})$= 0, the value of $\boldsymbol{Q}$ is mostly depends on $\boldsymbol{A}$, which is contrary to the original intention of dueling structure and introducing $\boldsymbol{A}$($\boldsymbol{Q}$mostly depends on $\boldsymbol{V}$). In order to introduce a better identifiability, the output of advantage function needs to be centralized, that is, adding an average constraint to make the value of advantage functionto 0. The practical combination formula is as follows:

$$\boldsymbol{Q}(\boldsymbol{s},\boldsymbol{a};\ \boldsymbol{\theta},\boldsymbol{\alpha},\boldsymbol{\beta}) = \boldsymbol{V}(\boldsymbol{s};\ \boldsymbol{\theta},\boldsymbol{\beta}) + \boldsymbol{A}(\boldsymbol{s},\boldsymbol{a};\ \boldsymbol{\theta},\boldsymbol{\alpha}) - \frac{\mathbf{1}}{|\boldsymbol{A}|}\sum_{\boldsymbol{a}'}\boldsymbol{A}(\boldsymbol{s},\boldsymbol{a}';\ \boldsymbol{\theta},\boldsymbol{\alpha})\text{(Formula3-7)}$$

3.2.2.4N-step Return

Chapter 2mentionedtarget value$\boldsymbol{R_{t+1}} + \boldsymbol{\gamma V}(\boldsymbol{S_{t+1}})$ is used as the approximate estimation of$\boldsymbol{V}(\boldsymbol{S_t})$in TD method. If we continue to use the Bellman equation to estimate $\boldsymbol{V}(\boldsymbol{S_t})$ in the following multiple steps, we can make the approximate estimation$\boldsymbol{V}(\boldsymbol{S_t})$from state $\boldsymbol{s}$as:

$$\boldsymbol{V}(\boldsymbol{S_t}) \leftarrow \boldsymbol{R_{t+1}} + \boldsymbol{\gamma R_{t+2}} + \boldsymbol{\gamma^2 R_{t+2}} + \cdots + \boldsymbol{\gamma^{n-1} R_{t+n}} + \boldsymbol{\gamma^n V}(\boldsymbol{S_{t+n}})\text{(Formula3-8)}$$

To some extent, this approximate estimation method can accelerate the convergence speed of value function by considering the future N-step return in an iteration [16]. However,N requires to be reasonablyselected, since the estimated value $\boldsymbol{V}(\boldsymbol{S_{t+n}})$of the future state $\boldsymbol{S_{t+n}}$ changes continuously with iterations, which means the target value also changes constantly, resulting in unstable training. Therefore, N=1 or a large N value is not as good as a middle

one,the value of N is selected through experiments according to the specific reinforcement learning problems.

3.2.2.5Soft TargetNet Updates

DQN algorithm has two set of networks, MainNetandTargetNet, TargetNet is introduced to provide a stable supervises training data. The parameters of TargetNet is fixed and only updates after ina fixed number of iterations, we copy the parameters $\boldsymbol{\theta}$ of MainNet to TargetNet. We describe this regularly update as hard update because TargetNet's parameters are suddenly changed, which may cause the unstable output of TargetNet. This unstable change innetwork's output against our initial goal, to stabilize supervised training data. So Soft TargetNet Updates is a way of doing soft updates on parameters of TargetNet:

$$\boldsymbol{\theta}' \leftarrow (\mathbf{1}-\boldsymbol{\tau})\boldsymbol{\theta}' + \boldsymbol{\tau}\boldsymbol{\theta} \text{(Formula3-9)}$$

Here $\boldsymbol{\tau}$ is a small positive number usually with value 0.01 or 0.001. Researchers[17] point out that soft updates of the parameters can improve the stability of training.

3.2.2.6 Parallel computing

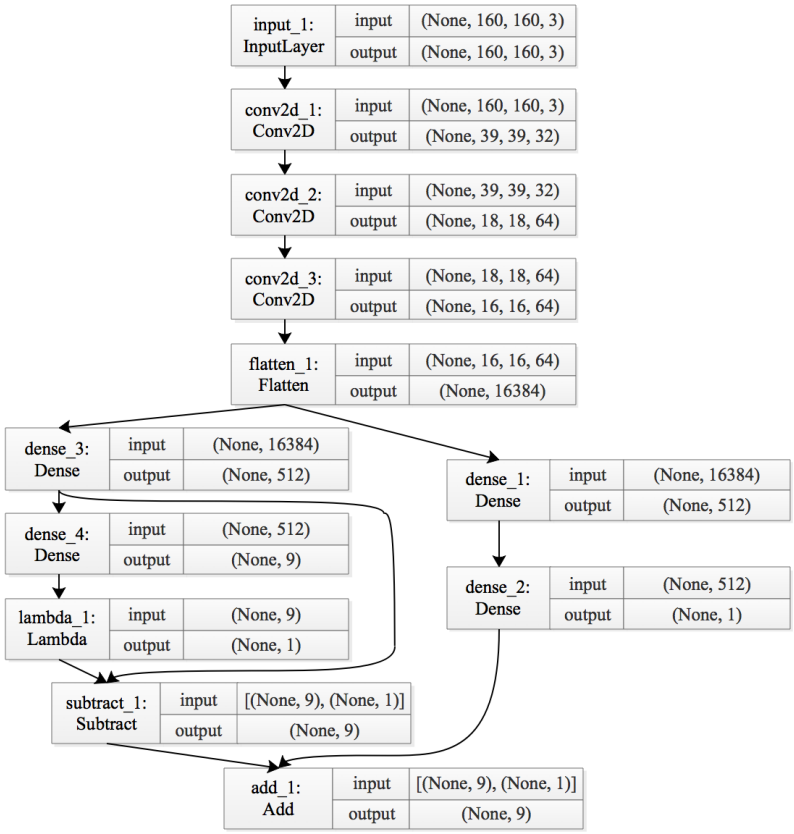
In DQN algorithm, agent uses TargetNet to select action when interacts with the environment, when some complete state transition sequences $< s, a, r, s' >$ are acquired, algorithm storesthem in experience pool; then take a certain number of data samples from experience pool for training, update the parameters of networks. Since the training samples are taken from the big experience pool, the update of networkparameters will not immediately affect all the samples in memories (the big pool). Therefore,multithreading or multiprocessing techniques can be introduced to split the data collectionprocessand training process, handle them separately.Furthermore, multiple agents can be used to interact with the environment at the same time, improvingstate transition sequencescollectingspeed and the overall efficiency of the program.

When the structure of network is more complex, algorithm like back-propagation needs longer time for forward or backward derivation and parameter updating, parallel computing can improve the utilization of CPU or GPU,speed up the training.

But Parallel computing is not always so efficient as we think it can be, there is a trade off between data collectiontimeand$\boldsymbol{Q}$value computing time, since all the $\boldsymbol{Q}$ value are computed depends on the two networks, even though there is a higher speed of collecting state transition sequences$< s, a, r, s' >$, it still need to wait for the $\boldsymbol{Q}$ value to decide action $\boldsymbol{a}'$. So parallel computing can only improve program in limited extent.

3.2.3 Final Algorithm

The above DQN algorithm and improvements are realized in pythonusing keras and tensorflow. The neural network design is in Figure 3-17.



```
def build_model():
    input = keras.layers.Input(shape = INPUT_DIMS)

    x = keras.layers.Conv2D(
        filters = 32,
        kernel_size = 8,
        strides = 4,
        padding = "valid",
        activation = "relu",
        kernel_initializer = keras.initializers.he_normal()
    )(input)
    x = keras.layers.Conv2D(
        filters = 64,
        kernel_size = 4,
        strides = 2,
        padding = "valid",
        activation = "relu",
        kernel_initializer = keras.initializers.he_normal()
    )(x)
    x = keras.layers.Conv2D(
        filters = 64,
        kernel_size = 3,
        strides = 1,
        padding = "valid",
        activation = "relu",
        kernel_initializer = keras.initializers.he_normal()
    )(x)
    x = keras.layers.Flatten()(x)

    q = keras.layers.Dense(
        units = 512,
        activation = "relu",
        kernel_initializer = keras.initializers.he_normal()
    )(x)
    q = keras.layers.Dense(units = 1)(q)

    a = keras.layers.Dense(
        units = 512,
        activation = "relu",
        kernel_initializer = keras.initializers.he_normal()
    )(x)
    a = keras.layers.Dense(units = NUM_ACTIONS)(a)

    mean = keras.layers.Lambda(lambda x: K.mean(x, axis = -1, keepdims = True))(a)
    a = keras.layers.Subtract()([a, mean])
    q = keras.layers.Add()([q, a])

    model = keras.models.Model(inputs = input, outputs = q)
    model._make_predict_function()
    return model
```

Figure 3-17: DQN design and code implements

The network's input are three images with the size of 160×160 travel throughthree convolution layers, thethree convolution layers extract features of input images, after that is a flatten layer. The sameoutput of flatten layerflows into two branches, two branches compute the state value $\boldsymbol{V}$and advantage $\boldsymbol{A}$respectively, then addthe result to estimated$\boldsymbol{Q}$, this is implementation of dueling network architectures in3.2.2.3, finally the output is $\boldsymbol{Q}$ of nine different actions.

In 2.1.2.b, the environment designed with the sizes of ***400×400***, before input images to network, we need preprocessing to reduce image to ***160×160***. OpenCV library provide method ***cv2.resize()*** for scaling images.Preprocessingoneach pixel canfacilitate the processing of convolution neural network, so we mappixel value from [0, 255] to [-1, 1] using Formula 3-10, then input image becomes like Figure 3-16.

$$\boldsymbol{x} \leftarrow \frac{\boldsymbol{x}}{\mathbf{128}} - \mathbf{1} \text{(Formula3-10)}$$

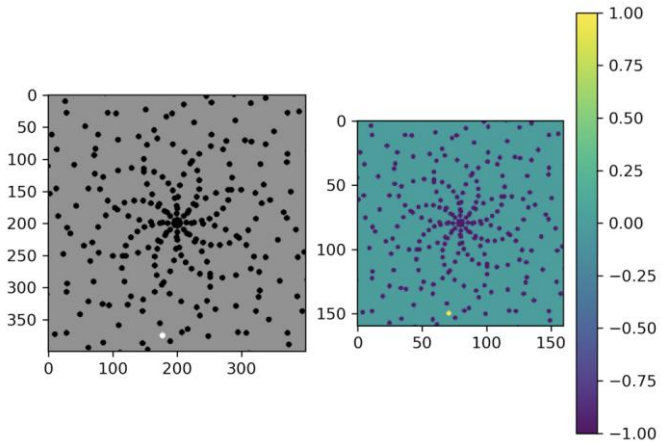


Figure 3-18: Preprocessing on pixel

But preprocessing on pixel also has its disadvantage, the 8-bit (one byte) per pixel (8bpp) format stores one pixel per byte. After preprocessing, pixel value is transferred to a double, which needs 32-bit (8 byte) space. Obviously, preprocessingincreases the space required for storage.

The loss function (or call objective function $\boldsymbol{J}(\boldsymbol{\theta})$ used here is mean square errorfunction and optimizer used is Adam, they are popular common combination in deep learning.
Now, consider the parallel computing part, theoretically, we can use multithreading or multiprocessing. Python's interpreter is Cpython, and Cpython has ahistorical issueof global interpreter lock (GIL), every thread needs to obtainGIL before executing and thread can only execute code when obtains GIL (one thread only has one GIL), which means multithreading applications have serious operational efficiency problems in intensive computing tasks, so multiprocessingis often adopted rather thanmultithreading. Codes build network model with keras and tensorflow is thread safe, but the underlying session object cannot be shared among multiple processes at the same time (so the computing of $\boldsymbol{Q}$must be in one thread).

The final parallel computingplan is tocollect interaction data using multiprocessing agent in subprocess.Agents in subprocess interact with environment, send observation data into a shared queue, when there is a need to evaluate$\boldsymbol{Q}$, subprocess pipe the observation to the main process, wait for the main process to return the evaluation result of $\boldsymbol{Q}$. The main process has two threads, one is responsible for receiving observationfrom pipe, and returning the evaluation results; the otheris responsible for fetching data from the shared queue and adding

it to the experience pool. At the same time, experience pool's access must be ensured safe withthread's mutex mechanism.

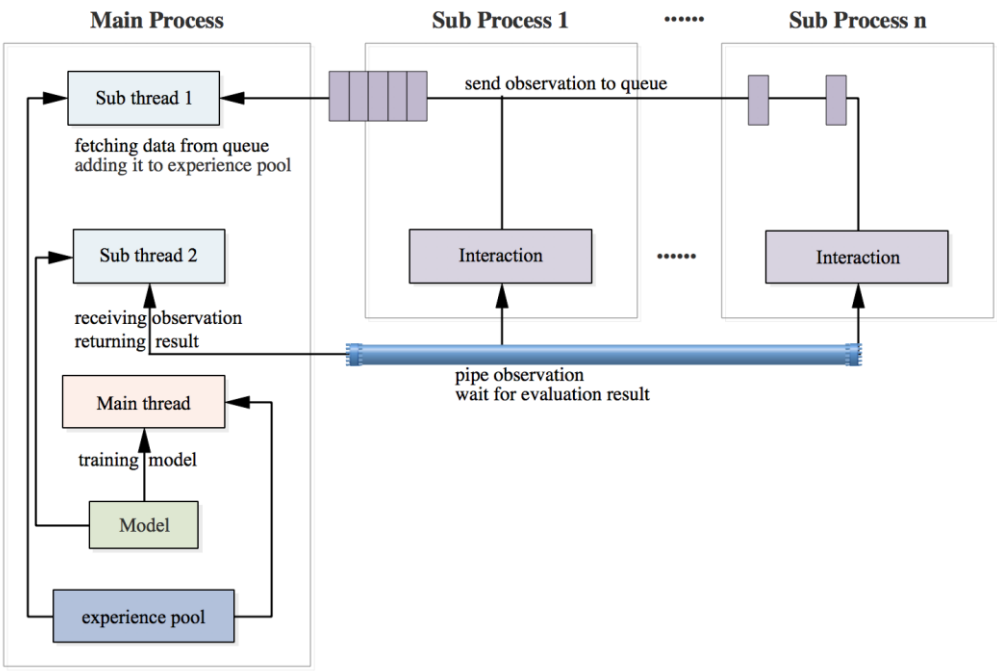


Figure 3-19: Communication between main process and subprocess

Python multiprocessing library encapsulatesQueue, Lock, Value, Pipe and other objects to facilitate the communication between processes.In the following code, ***prediction_pipe*** is a two-way pipeline for inter process communication, ***replay queue*** is a shared queue for data fetching, and ***replay_lock*** is the thread mutex for accessing experience pool. In subprocess, the collected data isfirst stored in a listand sum up the N-step return before it is added to the shared queue.

```
class Worker(mp.Process):
    def __init__(self, replay_queue, prediction_pipe, epsilon, global_steps):
        mp.Process.__init__(self, daemon = True)
        self.replay_queue = replay_queue
        self.prediction_pipe = prediction_pipe

        self.epsilon = epsilon
        self.global_steps = global_steps

        self.R = 0
        self.buffer = []

    def predict(self, s):
        self.prediction_pipe.send(s)
        q = self.prediction_pipe.recv()
        return q

    def run_episode(self):
        env = Environment1()
        s = env.reset()

        local_steps = 0
        total_reward = 0

        while(True):
            # Choose action
            eps = self.epsilon.value(self.global_steps.value)
            if(np.random.RandomState().uniform() < eps):
                a = np.random.RandomState().randint(9)
            else:
                a = np.argmax(self.predict(s)[0])

            # Execute action
            next_s, r, done = env.step(a)
            self.add_replay(s, a, r, next_s, done)

            s = next_s
            total_reward += r
            self.global_steps.value += 1
            local_steps += 1

            if(done):
                # print(self.global_steps.value, local_steps, total_reward)
                break
```

Figure3-20: Codes for subprocess interaction

```
def add_replay(self, s, a, r, next_s, done):
    if(len(self.buffer) < N_STEPS):
        self.R += r * (GAMMA ** len(self.buffer))
        self.buffer.append((s, a, r, next_s, done))
    else:
        self.buffer.append((s, a, r, next_s, done))
        s_0, a_0, _, _, _ = self.buffer[0]
        _, _, _, s_n, d = self.buffer[N_STEPS - 1]
        self.replay_queue.put((s_0, a_0, self.R, s_n, d))
        self.R = (self.R - self.buffer[0][2] + r * GAMMA_N) / GAMMA
        self.buffer.pop(0)

    if(done):
        while(len(self.buffer) > 0):
            n = len(self.buffer)
            s_0, a_0, _, _, _ = self.buffer[0]
            _, _, _, s_n, d = self.buffer[n - 1]
            self.replay_queue.put((s_0, a_0, self.R, s_n, d))
            self.R = (self.R - self.buffer[0][2]) / GAMMA
            self.buffer.pop(0)
        self.R = 0

def run(self):
    while(True):
        self.run_episode()
```

Figure 3-21: Codes for calculating N-step reward

```
class PredictionServer(threading.Thread):
    def __init__(self, pipes):
        threading.Thread.__init__(self)
        self.pipes = pipes
        self.should_stop = False

    def run(self):
        while(not self.should_stop):
            time.sleep(0.001)
            for p in pipes:
                if(p[0].poll()):
                    s = p[0].recv()
                    p[0].send(predictQ(s))

    def stop(self):
        self.should_stop = True

class PullServer(threading.Thread):
    def __init__(self, replay_queue):
        threading.Thread.__init__(self)
        self.replay_queue = replay_queue
        self.should_stop = False

    def run(self):
        while(True):
            time.sleep(0.001)
            while((not self.should_stop) and (not replay_queue.empty())):
                s, a, r, next_s, done = replay_queue.get()
                with replay_lock:
                    replay.add(s, a, r, next_s, done)

    def stop(self):
        self.should_stop = True
```

Figure 3-22: Codes for two kinds of subthread in main process

4 Experiment Result and Analysis

4.1 Overall Experiment Result

First, the experiment evaluates the overall performance of our model, variable-controlling approach is considered to be used, some variables setting are shown in Table 4-1.

Variable	Value	Description
INPUT_DIMS	(160, 160, 3)	3 frames of ***160×160*** size
NUM_ACTIONS	9	Number of actions
GAMMA	0.99	Discount factor
N_STEPS	4	Used 4-setp reward
REPLAY_SIZE	200000	Experience pool capacity
BATCH_SIZE	32	The mini_batch size
LEARNING_RATE	0.0001	Learning rate
MAX_STEPS	150000	Overall maximum iteration episode
MAX_FRAMES	3600	Frame capacity of one episode
NUM_WORKERS	16	Number of parallel process
EPSILON_INITIAL	1	Initial value of ε inε-Greedy policy
EPSILON_FINAL	0.1	Final value of ε inε-Greedy policy
EPSILON_DECAY_STEPS	6000000	Steps of for ε to change linearly from initial value to final value
α(ALPHA)	0.6	α in prioritized experience replay

Table 4-1: Fixed parameters in experiment

During this experiment, some tracks of obstacles (black dots) and agent (white dots) are visualized in a series of quick shots. These quick shots can be converted to video using OpenCV's methods, which is a better way to obverse agent's movement.

Observe these tracks, first, we notice thatunder the control of RL algorithm, the agent unanimouslychooses to go to the lower left corner, which is the area with sparse obstacles.Second, in some cases, agent do not passively keep distance from all the obstacles,

but actively pass through them (For example, in the first six images Figure 4-2(2/3), the agent goes through between a lineof blackobstacles).

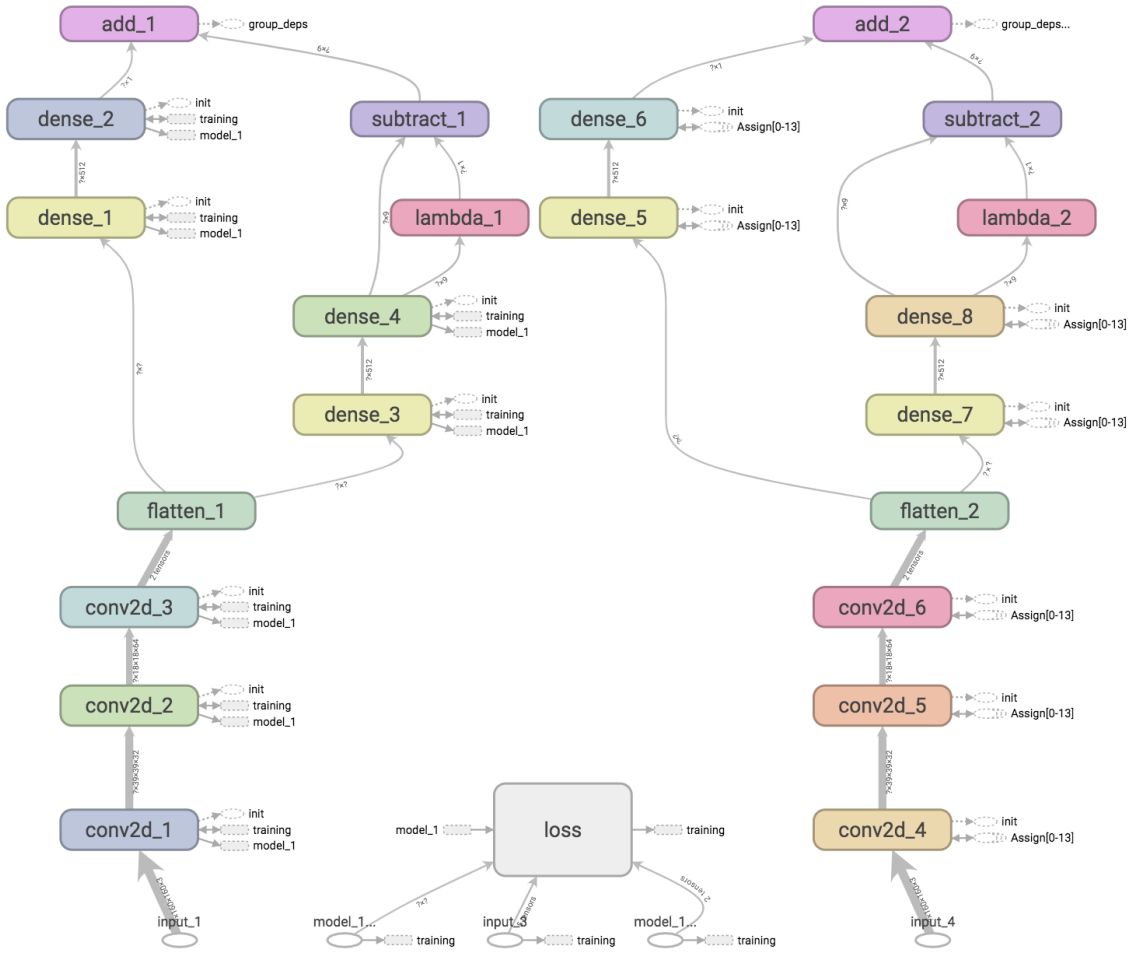


Figure 4-1: Model structure supported by TensorBoard

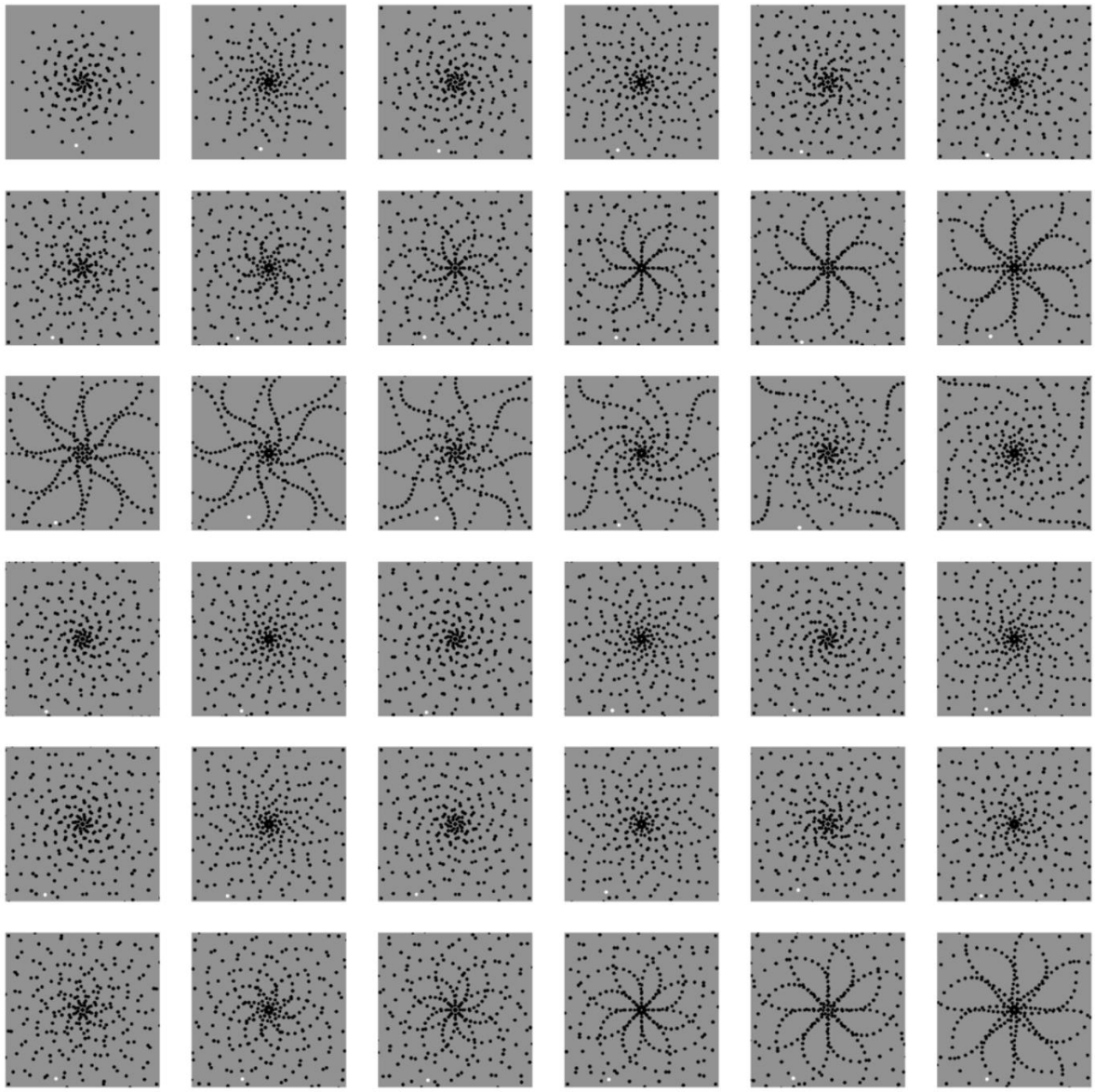


Figure 4-2: Tracksof obstacles and agent (1/3)

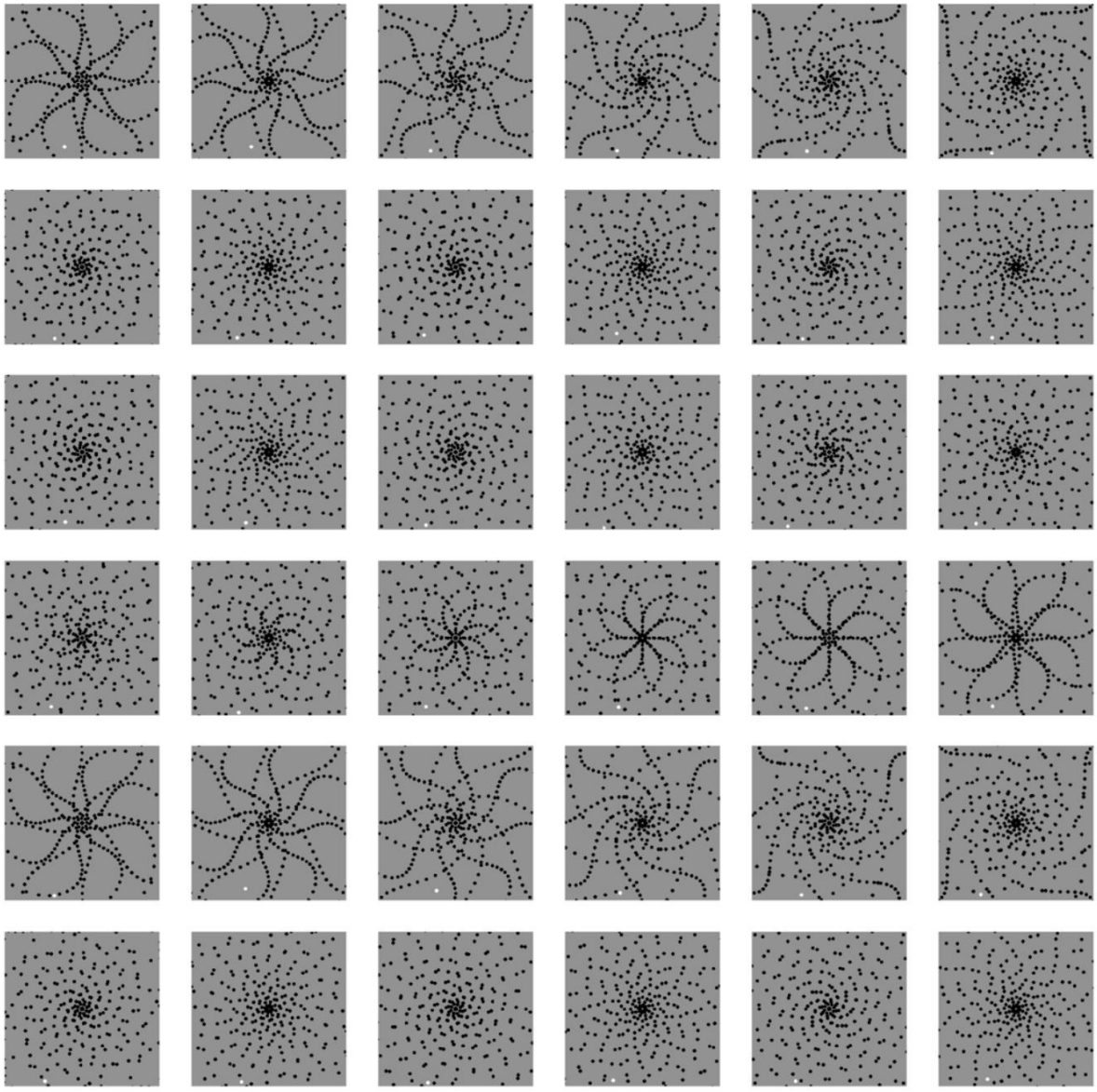


Figure 4-2: Tracks of obstacles and agent (2/3)

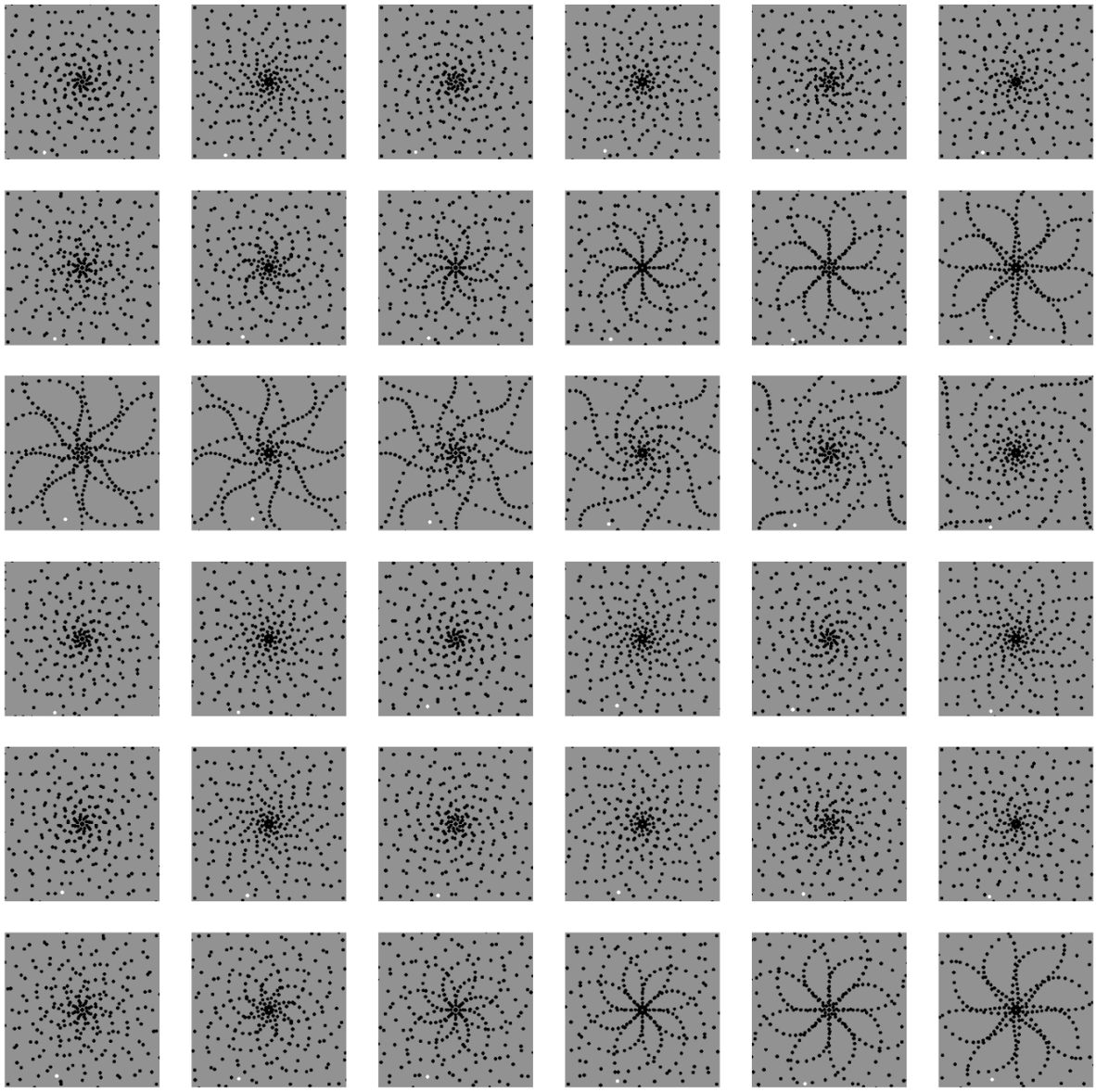


Figure 4-2: Tracks of obstacles and agent (3/3)

The overall evaluation is based on the parameters inTable 4-1, using the Episodes-Steps curve to measure result. An episode starts from the environment being reset to the agent andan obstacle collides or the reaches frame capacity (or the agent has already moved 1200 steps, since frame capacity is 3600 and three frames equal to one step).

Figure 4-3 shows the Episodes-Steps curve, the blue scatter represents the maximum moving steps of every agent in each episode; thered curve is a smoothing operation, representing theaverage maximum moving stepsof all agents in each episode. When episodes reach about 110000, step reach 1179, this is the maximum value of our model. Notice that, the first 60

frames of every episode are skipped (1200-60/3+1=1180), because at that time the distance between the agent toany obstacles is far and there is no need to record this period.

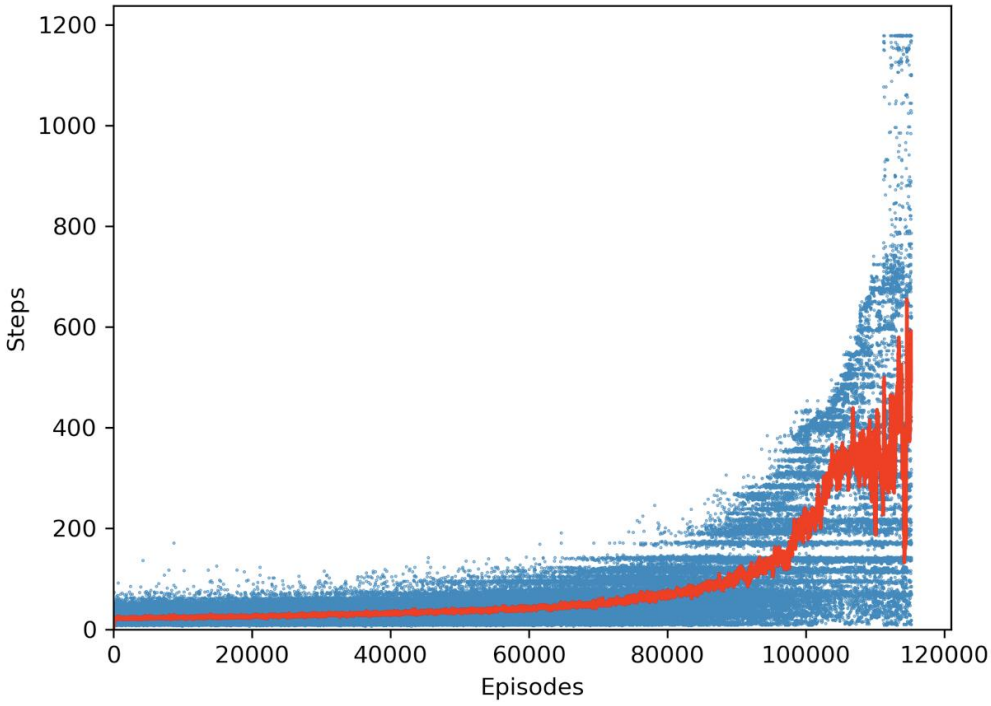


Figure 4-3: Episodes-Steps curve

Now, compare the trained model with an untrained model based on random policy: load the trained modelafter the aboveexperiment, and run 100 episodes, record the maximum steps, the minimum steps andcalculate the average steps respectively.

After the training process, model can reach a satisfied performance, improving the obstacle avoidance ability, moving more steps without collision than a random model.

Model	Maximum steps	Minimum steps	Average steps
Trained model	82	7	21.64
Untrained model	1179	35	704.51

Table 4-2: Fixed parameters in experiment

4.2 Analysis and Future Work

This guide study applied deep reinforcement learning to an agent in a 2D obstacle avoidance environment. The guide study report provides clear theoretical knowledgebackground and related concepts of RL, introduces some mathematical tools for describing RL problems and RL algorithms. The most important algorithm is Deep Q Networkalgorithm, to which I put a lot of efforts to study its improvements (DoubleDQN, prioritized experience replay, dueling network architectures, N-steps return, parallel computing).

Based on the improvedDQN algorithm, aneural network structure is proposed, which takes images' pixel data as input and outputs the state-action value for agent to decide its action in a dynamic obstacle environment. Even though the performance is improved after the training process, but the program's running time and memory cost in barely satisfactory. At the same time, how to extend this problem to a 3D environment still needs more focus.
In the future, some works can be done to improve the model's running cost, study how the reward function or initial function value impacts the convergence speed, test the agent's region of interest.

5 Reference

[1] Silver D, Schrittwieser J, Simonyan K, et al. Mastering the game of go without human knowledge[J]. Nature, 2017, 550(7676): 354.

[2] Bojarski M, Del Testa D, Dworakowski D, et al. End to end learning for self-driving cars[J]. arXiv preprint arXiv:1604.07316, 2016.

[3] Stanley K O, Miikkulainen R. Evolving neural networks through augmenting topologies[J]. Evolutionary computation, 2002, 10(2): 99-127.

[4] Sutton R S, Barto A G. Reinforcement learning: An introduction[M]. MIT press, 2018.

[5] Example of a stochastic process which does not have the Markov property[EB/OL]. https://math.stackexchange.com/q/89414, 2017-04-13.

[6] Dynkin E B. Markov processes[M]//Markov Processes. Springer, Berlin, Heidelberg, 1965: 77-104.

[7] Bertsekas D P. Dynamic programming and stochastic control[J]. Mathematics in science and engineering, 1976, 125: 222-293.

[8] Watkins C J C H, Dayan P. Q-learning[J]. Machine learning, 1992, 8(3-4): 279-292.

[9] Csáji B C. Approximation with artificial neural networks[J]. Faculty of Sciences, Etvs Lornd University, Hungary, 2001, 24: 48.

[10] Sutton R S, McAllester D A, Singh S P, et al. Policy gradient methods for reinforcement learning with function approximation[C]//Advances in neural information processing systems. 2000: 1057-1063.

[11] Mnih V, Kavukcuoglu K, Silver D, et al. Playing atari with deep reinforcement learning[J]. arXiv preprint arXiv:1312.5602, 2013.

[12] Schaul T, Quan J, Antonoglou I, et al. Prioritized experience replay[J]. arXiv preprint, 2015: 365-367.

[13] Van Hasselt H, Guez A, Silver D. Deep reinforcement learning with double q-learning[C]//Thirtieth AAAI Conference on Artificial Intelligence. 2016.

[14] Schaul T, Quan J, Antonoglou I, et al. Prioritized experience replay[J]. arXiv preprint arXiv:1511.05952, 2015.

[15] Wang Z, Schaul T, Hessel M, et al. Dueling network architectures for deep reinforcement learning[J]. arXiv preprint arXiv:1511.06581, 2015.

[16] Peng J, Williams R J. Incremental multi-step Q-learning[M]//Machine Learning Proceedings 1994. Morgan Kaufmann, 1994: 226-232.

[17] Lillicrap T P, Hunt J J, Pritzel A, et al. Continuous control with deep reinforcement learning[J]. arXiv preprint arXiv:1509.02971, 2015.