

# Abstract

ZHANG Tianlin

tianlin.zhang@auditeur.ec-lyon.fr

ZHANG Zhaofeng

zhaofeng.zhang@auditeur.ec-lyon.fr

research project

report

Reinforcement Learning

We are going to some basic research on Reinforcement Learning, a paradigm based on reward and punishment. More goals are achieved by a robot (an agent), more reward it will receive. Conversely, errors will be penalized. Specific algorithms and applications will be discussed in this report.

Key words: Reinforcement Learning(RL), agent, reward, algorithm, application.

Contents

[Abstract 1](#_Toc516839463)

[1 Introduction 3](#_Toc516839464)

[1.1 Research theme and background 3](#_Toc516839465)

[1.2 Research methods 4](#_Toc516839466)

[2 Research results 5](#_Toc516839467)

[2.1 Literature research 5](#_Toc516839468)

[2.1.1 Games 5](#_Toc516839469)

[2.1.2 Robotics 6](#_Toc516839470)

[2.1.3 Continuous Control 7](#_Toc516839471)

[2.2 Algorithms and Examples 9](#_Toc516839472)

[2.2.1 Q Learning 9](#_Toc516839473)

[2.2.2 Sarsa 12](#_Toc516839474)

[2.2.3 Deep Q Network 16](#_Toc516839475)

[2.2.4 Policy Gradient 16](#_Toc516839476)

[3 Conclusion 24](#_Toc516839477)

[4 Reference 24](#_Toc516839478)

# 1 Introduction

## 1.1 Research theme and background

Reinforcement learning is a big class in the machine learning family. Using reinforcement learning allows the machine to learn how to get high scores in the environment and show excellent results. More generally it can make machines learning from mistakes, finding the law and learning how to achieve the goal through continuous attempts. There are many examples of reinforcement learning in practice. The most famous one, AlphaGo, defeated the human master in the Go field for the first time.

The key elements of reinforcement learning are: environment, reward, action, and state. With these elements we can build a reinforcement learning model. The problem of reinforcement learning is to get an optimal policy for a specific problem with which the reward obtained under this strategy is maximized. The so-called policy is a series of actions. That is, sequential data. Reinforcement learning can be characterized by the following figure, which is to first extract an environment from the task to be completed, abstract the state, the action, and the instantaneous reward accepted by the execution of the action.[1]

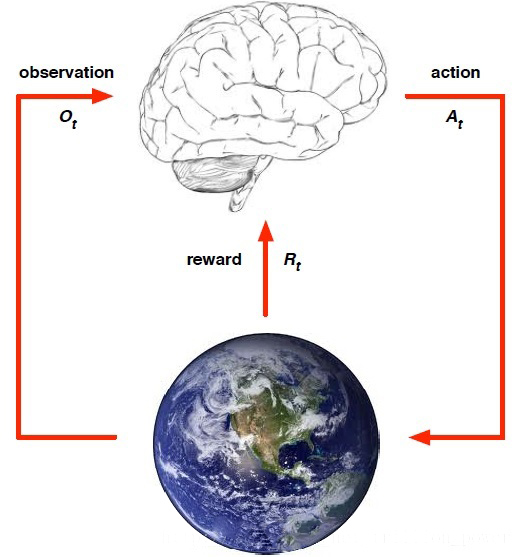


Figure 1 Fundamental concepts of RL

To make further discussion easier to understand, here we give some explications to these key concepts one by one. Reward is usually denoted as Rt, indicating the return value after the nth time step. As a matter of fact, all reinforcement learning is based on the reward assumption. Reward is a scalar. Action is from action space. The agent determines what action to perform by analyzing the current state and the reward of the previous state. The execution of action will lead to maximize the expected reward, until the final algorithm converges, and the resulting policy is a series of sequential data of those actions. State refers to the current situation of the agent. Let’s take the pong game (Atari game) for example, the state of the game is the position of the ball of the current time step while the state in Flappy bird is the position of the bird in the plane.

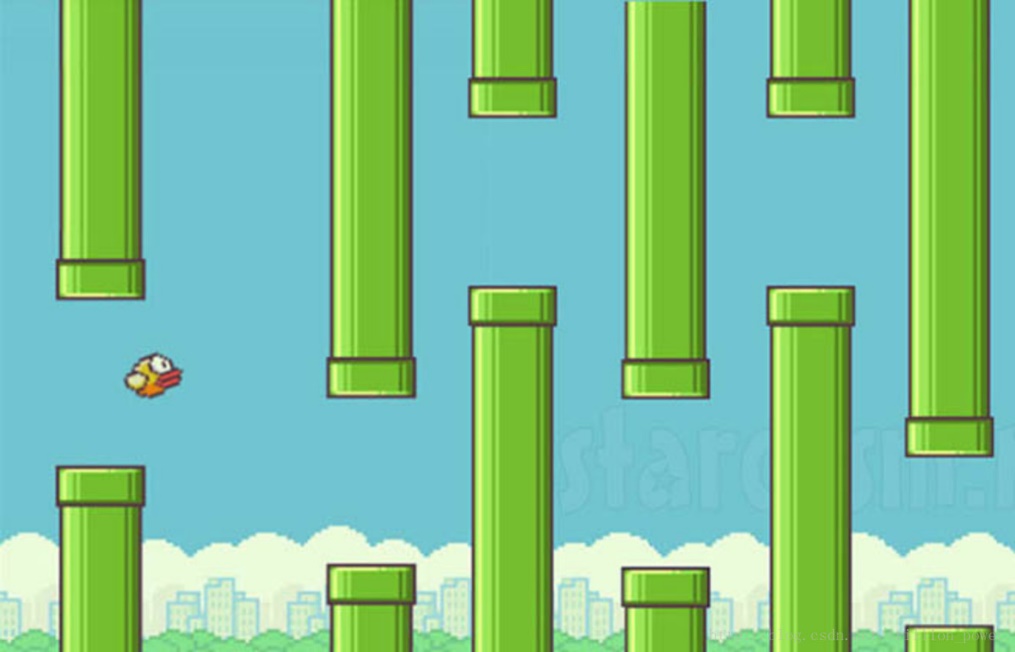


Figure 2 State is the current situation

Policy is kind of agent’s characteristic. It’s a mapping from a state to an action. It is divided into two parts: the determination strategy and the random strategy. The determination strategy is a certain action in a certain state. The random policy is described by a probability, that is, the probability of performing this action in a certain state: . Reinforcement learning can be summed up today in this way: to obtain an optimal strategy by maximizing reward. However, if one instantaneous reward is maximal, the algorithm will select the related action from the action space all the time. This becomes the simplest Greedy policy. It’s a good idea to constantly add the current reward into the total reward and make the last one largest. Therefore, a value function is constructed to describe this total reward. The expression is as follows:

Valued at [0,1], γ is called discount factor which is to reduce the impact of future rewards on the current action. Now the value function is maximized by selecting the appropriate policy. We should also be familiar with the Markov process. It’s actually the process of state transfer. The most important thing is the probability matrix of state transition by one step. As long as we have such a one-step transition probability matrix, the entire Markov process could be described.

The problem of reinforcement learning usually has the following characteristics: (a)Different actions produce different rewards. (b)Reward is usually delayed. (c)The reward for an action is based on the current state.

## 1.2 Research methods

As beginners in this area, we decided to do our research by three phases. Phase one: language learning. As the programming work always play an important role in machine learning research and the language Python is officially required, we spent one month learning and practicing it to be well prepared for next phases. Phase two: literature reading. This part of the work is done regularly and continuously. Thanks for the training courses provided by Mr. Nicolas Jardin we got a wide range of paper collection sources: on-line library of the school, Google Scholar, Scopus and Sudoc. Reinforcement learning is a large family. It contains many algorithms such as methods for selecting specific behaviors through the value of all behaviors, including Q-learning, Sarsa, Deep Q Network with the neural network and Policy Gradients that directly output behaviors. The phase three is getting ourselves down to those specific algorithms, learning their principles, programming little examples. We carry out this phase concurrently with the literature study.

We contact our instructor Mr. Alexandre Saidi by email and he does give us many precious advices. Two mid-term face-to-face conferences respectively in April and May guaranteed that we are on the right way.

# 2 Research results

## 2.1 Literature research

First, we read a book named Reinforcement Learning: An Introduction, which is a great book for beginners. It helped us construct the basic structure about Reinforcement Learning and let us know plenty of professional theories and algorithms, such as Monte-Carlo Tree Search and Policy Gradient. We will introduce some of them in Chapter 3.

After knowing basic knowledges of Reinforcement Learning, we started focusing on its applications and innovations, especially in robotic and game domains. So, we searched several papers and knew the contemporary development of Reinforcement Learning. These papers were all published in important magazines from 2014 to 2016. Most of them are innovations and optimizations based on the algorithms we have learned.

### 2.1.1 Games

In the history of AI development, games play an indispensable role. In recent years, many people have applied algorithms in reinforcement learning to games. For example, there are lots of research about the resolution of Atari Games, such as Pacman.

Most people start to focus on the application of Reinforcement Learning on games by the paper which is written by the company named Deepmind[10].They have introduced a now algorithm: Deep Q-Network. With this algorithm, they can play all the Atari games with the same codes. That means it is a total self-learning system. No one told it the rules and it learned all by itself. It is a breakthrough in combining model-free reinforcement learning with deep learning, achieves the best real-time agents thus far. Planning-based approaches achieve far higher scores than the best model-free approaches.

Monte-Carlo Tree Search[1] could be used in Atari Games. Deep Q-Network exploit information that is not available to human players, and they are orders of magnitude slower than needed for real-time play. So, in other paper, they had built a better real-time Atari game playing agent than Deep Q-Network. The central idea is to use the slow planning-based agents to provide training data for a deep-learning architecture capable of real-time play. We proposed new agents based on this idea and show that they outperform Deep Q-Network. Real-Time Atari Game has been researched by using repeated incremental planning via Monte-Carlo Tree Search methods such as UCT (Upper Confidence Bound Apply to Tree) to avoid the perception problem entirely by eschewing the building of an explicit policy[11].

Talking about Atari Games, one paper is named *Playing Atari with Deep Reinforcement Learning* have contributed a deep learning model which could successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. They applied their method to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or learning algorithm. They found that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them[12].

Another big problem for Atari Games is that while not composed of natural scenes, frames in Atari games are high-dimensional in size. They can involve tens of objects with one or more objects being controlled by the actions directly and many other objects being influenced indirectly, can involve entry and departure of objects, and can involve deep partial observability. Two deep neural network architectures have been proposed that consist of encoding, action conditional transformation, and decoding layers based on convolutional neural networks and recurrent neural networks. Experimental results show that the proposed architectures can generate visually-realistic frames that are also useful for control over approximately 100-step action-conditional futures in some games[8].

After researches about Atari Games, everyone began to try to use deep reinforcement learning algorithm to play games. For example, MazeBase, which is an environment for simple 2D games, designed as a sandbox for machine learning approaches to reasoning and planning. A variety of neural models (fully connected, convolutional network, memory network) are deployed via reinforcement learning on these games, with and without a procedurally generated curriculum. Despite the tasks’ simplicity, the performance of the models is far from optimal, suggesting directions for future development[5].

Another example, the program AlphaGo achieved a 99.8% winning rate against other Go programs and defeated the human European Go champion by 5 games to 0. a new approach has been invented that uses ‘value networks’ to evaluate board positions and ‘policy networks’ to select moves. These deep neural networks are trained by a novel combination of supervised learning from human expert games, and reinforcement learning from games of self-play. Without any lookahead search, the neural networks play Go at the level of state-of-the-art Monte Carlo tree search programs (which has been also used in the research of Atari) that simulate thousands of random games of self-play. They also use a new search algorithm that combines Monte Carlo simulation with value and policy networks [4].

In my opinion, there are plenty of ripe algorithms about playing chess game and some ancient video games, such as Atari, super Mario and so on. After that, game research might be some more complex games such as Warcraft, hero League, King glory, Grand Theft Auto V and so on. The pixels of these games are even higher, which means that the amount of information input will be much larger. At present, no algorithm has been able to compete with human players in these games.

### 2.1.2 Robotics

Robots are widely used to complete various manipulation tasks in industrial manufacturing factories where environments are relatively static and simple. Problems in robotics are often best represented with high-dimensional, continuous states and actions. In robotics, it is often unrealistic to assume that the true state is completely observable and noise-free.

Reinforcement learning offers to robotics a framework and set of tools for the design of sophisticated and hard-to-engineer behaviors. Reinforcement learning (RL) enables a robot to autonomously discover an optimal behavior through trial-and-error interactions with its environment. Instead of explicitly detailing the solution to a problem, in reinforcement learning the designer of a control task provides feedback in terms of a scalar objective function that measures the one-step performance of the robot[2].

From the research called *Towards Vision-Based Deep Reinforcement Learning for Robotic Motion Control*[6]*,* they showed that the DQN-based system is feasible to learn performing target reaching from exploration in simulation, using only visual observation with no prior knowledge. However, the agent (Agent B) trained in simulation scenarios failed to perform target reaching in the real-world experiment using camera images as inputs. Instead, in the real-world experiment using synthetic images as inputs, the agent got a consistent success rate with that in simulation. These two different results show that the failure in the real-world experiment with camera images was caused by the input image differences between real and simulation scenarios. To determine the causes of these more work is required. So they made a DQN-based learning system for a target reaching task, trained agents in simulation and evaluate them in both simulation and real-world target reaching experiments. After that, they identified and discussed a number of issues and opportunities for future work towards enabling vision based deep reinforcement learning in real-world robotic manipulation.

### 2.1.3 Continuous Control

After reading papers about Deep Q-Network, we started concentrating on the amelioration of this algorithm. The paper *Continuous control with deep reinforcement learning* [7]. adapt the ideas underlying the success of Deep Q-Network to the continuous action domain.

To use Deep Q-Network, deep neural network function approximators were used to estimate the action-value function. However, while Deep Q-Network solves problems with high-dimensional observation spaces, it can only handle discrete and low-dimensional action spaces. Many tasks of interest, most notably physical control tasks, have continuous (real valued) and high dimensional action spaces. Deep Q-Network cannot be straightforwardly applied to continuous domains since it relies on a finding the action that maximizes the action-value function, which in the continuous valued case requires an iterative optimization process at every step.

Their model-free approach which we call Deep DPG (DDPG) can learn competitive policies for all our tasks using low-dimensional observations (e.g. cartesian coordinates or joint angles) using the same hyper-parameters and network structure. In many cases, they are also able to learn good policies directly from pixels, again keeping hyperparameters and network structure constant.

Here is the algorithm of DDPG. In this algorithm, they had just given the gradient of J. In our calculation,.

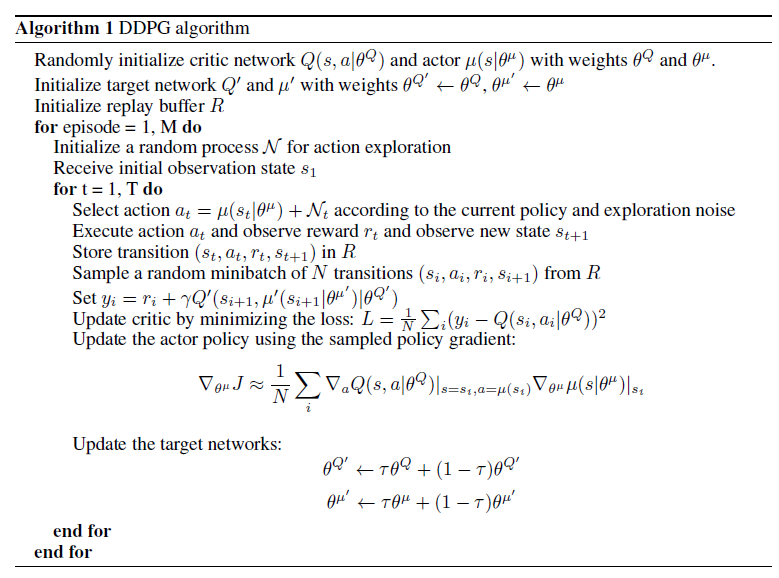


Figure 3 DDPG algorithm

Another paper about continuous control task is *Continuous Deep Q-Learning with Model-based Acceleration*[3] they explored algorithms and representations to reduce the sample complexity of deep reinforcement learning for continuous control tasks.

They proposed two complementary techniques for improving the efficiency of such algorithms. First, they derived a continuous variant of the Q-learning algorithm, which they called normalized advantage functions (NAF), as an alternative to the more commonly used policy gradient and actor-critic methods. NAF representation allows us to apply Q-learning with experience replay to continuous tasks, and substantially improves performance on a set of simulated robotic control tasks. To further improve the efficiency of our approach, they explored the use of learned models for accelerating model-free reinforcement learning. They showed that iteratively refitted local linear models are especially effective for this and demonstrate substantially faster learning on domains where such models are applicable.

To solve interesting continuous control problems, we also read a paper called *Learning Continuous Control Policies by Stochastic Value Gradients*[9]*.* They presented a unified framework for learning continuous control policies using backpropagation. It supports stochastic control by treating stochasticity in the Bellman equation as a deterministic function of exogenous noise. The product is a spectrum of general policy gradient algorithms that range from model-free methods with value functions to model-based methods without value functions.

They applied these algorithms first to a toy stochastic control problem and then to several physics-based control problems in simulation. One of these variants, SVG, shows the effectiveness of learning models, value functions, and policies simultaneously in continuous domains. They have shown that two potential problems with value gradient methods, their reliance on planning and restriction to deterministic models, can be exorcised, broadening their relevance to reinforcement learning. They have shown experimentally that the SVG framework can train neural network policies in a robust manner to solve interesting continuous control problems.

## 2.2 Algorithms and Examples

To have a better understanding of the papers, we have conducted theoretical study and practice for different reinforcement learning algorithms.

### 2.2.1 Q Learning

(1) What is Q Learning

Q Learning is a learning technique that does not require a model of the environment. It can handle problems with stochastic transitions and rewards, without requiring adaptations. For any finite Markov decision process (FMDP), Q-learning eventually finds an optimal policy, in the sense that the expected value of the total reward return over all successive steps, starting from the current state, is the maximum achievable. Q-learning can identify an optimal action-selection policy for any given FMDP.

(2) Updating and decision-making

Reinforcement learning usually includes two entities: agent and environment. The interaction between the two entities is as follows. In the state st of the environment, the agent action at to obtain reward rt and enter state st+1.

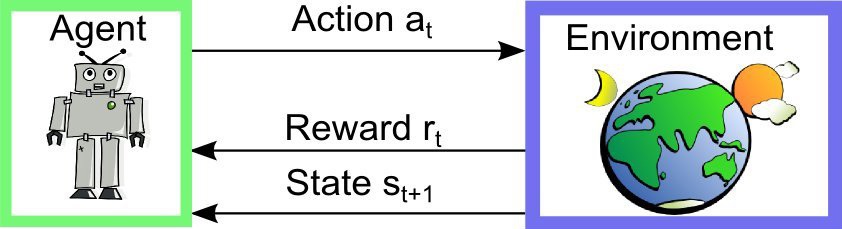


Figure 4 Interaction between agent and environment

The core of Q Learning is Q-table. The Q-table’s rows and columns represent respectively the values of state and action, and the Q-table’s value of Q(s, a) measures how well would it be if we take action a at the current states s. During the training, we used Bellman equation to update the Q-table. Here’s the Bellman equation:

The Q-value is equal to the sum of the reward after taking action a at the current state s and the best reward ever after a certain discount γ. According to Bellman equation, we can get the entire Q-table by the algorithm follows: (a)Count the simulation times in external loop. (b)In inner loop, Count number of steps per simulation. (c)Select the action based on the current state and q-table. (d)Get the next state and reward based on the current state and action. (e)Update Q-table by: and . Shortly, this principle can be illustrated like this:

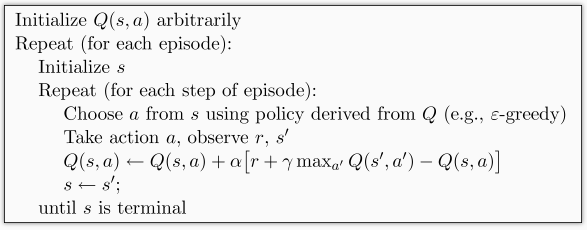


Figure 5 Main idea of Q Learning

Where ε-greedy is a strategy used in decision-making, for example ε=0.9, it means that in 90% of the cases, I will choose the behavior according to the optimal value of the Q-table and use randomly selected behavior for 10% of the time. α is called learning rate, used for determining how much an error can affect and γ is the discount factor as introduced.

(3) Example

Here, we create a 4\*4 maze in which a red explorer will learn how to find the yellow heaven while trying to avoid to get in the black hell.

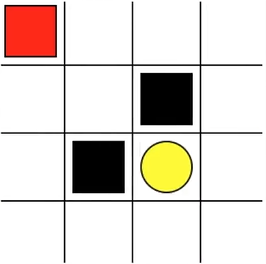


Figure 6 4\*4maze with explorer, heaven and hell

When executing runthis.py in the folder Q\_learning\_maze, we can see that at beginning, the red explorer falls into the black hell but after half time of simulation, he can always find the yellow heaven accurately and quickly.

Just like what have been illustrated before, the whole algorithm depends on constantly updating the value in the Q table and taking actions according to the new value. Let’s go straight to the code. First of all, we import two modules, maze\_env is our environment module, It’s a virtual environment written by a simple GUI module called ‘tkinter', we won’t go to details for all the environment design in this research. And the other part RL\_brain includes the main idea of reinforcement learning.

from maze\_env import Maze  
from RL\_brain import QLearningTable

And we can correspond the following code according to the algorithm in the image above. This is the most important part of the entire Q Learning iterative update part.

def update():  
 for episode in range(100):  
 # initial observation  
 observation = env.reset()  
  
 while True:  
 # fresh env  
 env.render()  
  
 # RL choose action based on observation  
 action = RL.choose\_action(str(observation))  
  
 # RL take action and get next observation and reward  
 observation\_, reward, done = env.step(action)  
  
 # RL learn from this transition  
 RL.learn(str(observation), action, reward, str(observation\_))  
  
 # swap observation  
 observation = observation\_  
  
 # break while loop when end of this episode  
 if done:  
 break  
  
 # end of game  
 print('game over')  
 env.destroy()  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 env = Maze()  
 RL = QLearningTable(actions=list(range(env.n\_actions)))  
  
 env.after(100, update)  
 env.mainloop()

Now Let’s see the other important part of code that is responsible for decision-making and thinking. We will define Q Learning as a class and call it QLearningTable.

class QLearningTable:  
 def \_\_init\_\_(self, actions, learning\_rate=0.01, reward\_decay=0.9, e\_greedy=0.9):  
   
 def choose\_action(self, observation):  
  
 def learn(self, s, a, r, s\_):

def check\_state\_exist(self, state):

At the beginning we initialize all parameters by:

def \_\_init\_\_(self, actions, learning\_rate=0.01, reward\_decay=0.9, e\_greedy=0.9):  
 self.actions = actions # a list  
 self.lr = learning\_rate  
 self.gamma = reward\_decay  
 self.epsilon = e\_greedy  
 self.q\_table = pd.DataFrame(columns=self.actions, dtype=np.float64)

Here is how to select the action based on the state in which it is located or the observation value on this state. Here we introduce the concept of epsilon greedy. Because in the initial stage, exploring the environment randomly is usually better than behavior-fixed patterns, and it’s also for accumulating experience. In fact, ε is between 0 and 1, as we hope that explorers will not be so greedy, so it’s good to choose ε=0.9, which means that 90% of the time we choose the optimal strategy and 10% of the time to explore. (randomly choose action)

def choose\_action(self, observation):  
 self.check\_state\_exist(observation)  
 # action selection  
 if np.random.uniform() < self.epsilon:  
 # choose best action  
 state\_action = self.q\_table.loc[observation, :]  
 state\_action = state\_action.reindex(np.random.permutation(state\_action.index)) # some actions have same value  
 action = state\_action.idxmax()  
 else:  
 # choose random action  
 action = np.random.choice(self.actions)  
 return action

We decide how to update q\_table based on whether it is at the terminal state.

def learn(self, s, a, r, s\_):  
 self.check\_state\_exist(s\_)  
 q\_predict = self.q\_table.loc[s, a]  
 if s\_ != 'terminal':  
 q\_target = r + self.gamma \* self.q\_table.loc[s\_, :].max() # next state is not terminal  
 else:  
 q\_target = r # next state is terminal  
 self.q\_table.loc[s, a] += self.lr \* (q\_target - q\_predict) # update

Another function is required to check whether we can find the current state information in the q\_table, and if the answer is no, we will insert asset of all zero data as the initial values of this state.

### 2.2.2 Sarsa

(1) What is Sarsa

Sarsa's decision-making part is the same as Q learning, we use Q table again to make decision. Larger action values in the Q table will be chosen to impose on the environment in exchange for rewards. The difference between Q Learning and Sarsa is reflected in the action selection. Q Learning always choose the action with the best value, and Sarsa is always following the control strategy to act. In practical projects, Sarsa will be more secure, and Q Learning will be full of adventurousness. For example, in the maze game, Q Learning tends to try boldly, no matter if the agent falls into a pit; When a Sarsa agent is near the pit, it will be hesitant and wandering, after all, his actions are too cautious.

Whether it is Q-Learning or Sarsa, they all belong to Temporal-Difference Learning algorithm. Timing difference algorithm combines dynamic programming and Monte Carlo algorithm to simulate a scenario. After each action step, according to the value of the new state, estimate the pre-execution state value.

(2) Updating and decision-making

Just like Q Learning, the entire algorithm keeps updating the values in the Q table, and then uses the new values to determine what action to take at a state. However, Sarsa does not only think of the state’s corresponding action at the current state, but also thinks about the next state s\_ and the next action a\_. And when updating the value of Q(s, a), Sarsa needs Q(s\_, a\_) while Q Learning needs maxQ(s\_).

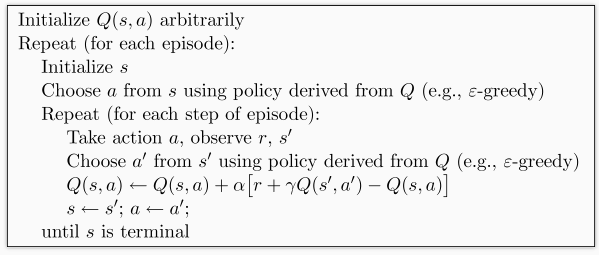


Figure 7 Main idea of Sarsa

(3) Example

The same maze example using Sarsa. When executing the file runthis.py in the folder Sarsa\_maze, we can see that the red explorer is doing the same thing to find the yellow heaven but this time, he prefers to take time to go back and forth beside the black hells, trying to avoid falling into them. As before, we import one environment module maze\_env and one decision-making module RL\_brain.

from maze\_env import Maze  
from RL\_brain import SarsaTable

The main updating code is given by translating the algorithm above into programming language.

def update():  
 for episode in range(100):  
 # initial observation  
 observation = env.reset()  
  
 # RL choose action based on observation  
 action = RL.choose\_action(str(observation))  
  
 while True:  
 # fresh env  
 env.render()  
  
 # RL take action and get next observation and reward  
 observation\_, reward, done = env.step(action)  
  
 # RL choose action based on next observation  
 action\_ = RL.choose\_action(str(observation\_))  
  
 # RL learn from this transition (s, a, r, s, a) ==> Sarsa  
 RL.learn(str(observation), action, reward, str(observation\_), action\_)  
  
 # swap observation and action  
 observation = observation\_  
 action = action\_  
  
 # break while loop when end of this episode  
 if done:  
 break  
  
 # end of game  
 print('game over')  
 env.destroy()  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 env = Maze()  
 RL = SarsaTable(actions=list(range(env.n\_actions)))  
  
 env.after(100, update)  
 env.mainloop()

In RL\_brain.py, we create SarsaTable as the same way of establishing QLearningTable. In fact, we can even define a main class RL and then use QLearningTable and SarsaTable as a derivative of the main class RL. We put the previous functions: \_\_init\_\_, check\_state\_exist, choose\_action and learn in this main structure and then change the corresponding contents according to different algorithms. With the parent class RL, we only need to write the function of learn respectively in QLearningTable and SarsaTable, because the other functions are the same as the parent class. Here are the one parent class and his two children:

class RL(object):  
 def \_\_init\_\_(self, action\_space, learning\_rate=0.01, reward\_decay=0.9, e\_greedy=0.9):  
 self.actions = action\_space # a list  
 self.lr = learning\_rate  
 self.gamma = reward\_decay  
 self.epsilon = e\_greedy  
  
 self.q\_table = pd.DataFrame(columns=self.actions, dtype=np.float64)  
  
 def check\_state\_exist(self, state):  
 if state not in self.q\_table.index:  
 # append new state to q table  
 self.q\_table = self.q\_table.append(  
 pd.Series(  
 [0]\*len(self.actions),  
 index=self.q\_table.columns,  
 name=state,  
 )  
 )  
  
 def choose\_action(self, observation):  
 self.check\_state\_exist(observation)  
 # action selection  
 if np.random.rand() < self.epsilon:  
 # choose best action  
 state\_action = self.q\_table.loc[observation, :]  
 state\_action = state\_action.reindex(np.random.permutation(state\_action.index)) # some actions have same value  
 action = state\_action.idxmax()  
 else:  
 # choose random action  
 action = np.random.choice(self.actions)  
 return action  
  
 def learn(self, \*args):  
 pass  
  
  
# off-policy  
class QLearningTable(RL):  
 def \_\_init\_\_(self, actions, learning\_rate=0.01, reward\_decay=0.9, e\_greedy=0.9):  
 super(QLearningTable, self).\_\_init\_\_(actions, learning\_rate, reward\_decay, e\_greedy)  
  
 def learn(self, s, a, r, s\_):  
 self.check\_state\_exist(s\_)  
 q\_predict = self.q\_table.loc[s, a]  
 if s\_ != 'terminal':  
 q\_target = r + self.gamma \* self.q\_table.loc[s\_, :].max() # next state is not terminal  
 else:  
 q\_target = r # next state is terminal  
 self.q\_table.loc[s, a] += self.lr \* (q\_target - q\_predict) # update  
  
  
# on-policy  
class SarsaTable(RL):  
  
 def \_\_init\_\_(self, actions, learning\_rate=0.01, reward\_decay=0.9, e\_greedy=0.9):  
 super(SarsaTable, self).\_\_init\_\_(actions, learning\_rate, reward\_decay, e\_greedy)  
  
 def learn(self, s, a, r, s\_, a\_):  
 self.check\_state\_exist(s\_)  
 q\_predict = self.q\_table.loc[s, a]  
 if s\_ != 'terminal':  
 q\_target = r + self.gamma \* self.q\_table.loc[s\_, a\_] # next state is not terminal  
 else:  
 q\_target = r # next state is terminal  
 self.q\_table.loc[s, a] += self.lr \* (q\_target - q\_predict) # update

(4) Sarsa(λ)

Actually, Sarsa is a one-step update method. We can put a bracket behind this Sarsa and say it is Sarsa (0) because it updates the code of conduct directly just after arriving at a new state. If this idea can be continued: After one step taken, if the agent takes one more step and then update, we can call him Sarsa (1). If n steps are taken, in a round, then we call the strategy Sarsa (n). To unify this process, we select a lambda value to replace the number of steps we want to choose. This is the origin of Sarsa(λ).

Single-step updates are easily trapped in local minimums, but round-off updates are more global, but they will wait until the end of training for this round to update parameters, so the efficiency of learning is not high enough for them. When λ is 0, it becomes a single-step update of Sarsa. When lambda takes 1, it becomes a round update, in such way the degrees of updating are the same for all steps. When lambda is between 0 and 1, the bigger λ is, the greater the degree of updating is for those steps near heaven is.

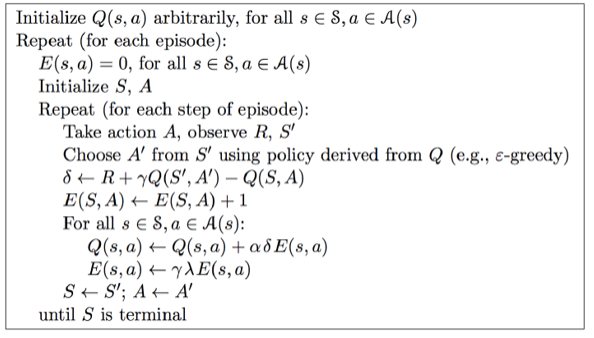


Figure 8 Sarsa lambda

### 2.2.3 Deep Q Network

The main method of the DQN algorithm is Experience Replay, which stores the data obtained from the exploration of environment and then randomly sampling the samples to update the parameters of the deep neural network. It also maximizes the reward on each action and environment state. What’s new to Deep Q Learning is that some improvements have been added like experience playback and dueling network architectures.

[To be added.]

### 2.2.4 Policy Gradient

(1) What is Policy Gradient (PG)

Policy Gradient is another big family in RL. Unlike the value-based method Q Learning and Sarsa, it accepts environment information as well. What PG wants to deliver is a specific action instead of the value of it. Thus, PG skips the value observation and estimation phase. One of the biggest advantages of PG is that the output of this action can be a continues one. (The valued based methods we have talked about always output discontinuous values and then select the action with the highest value) Well, PG could select action from a continuous distribution.

(2) Updating and decision-making

The first algorithm we learn and use is the Reinforce method, which is based on the update of the data of a whole round. This method is very basic.

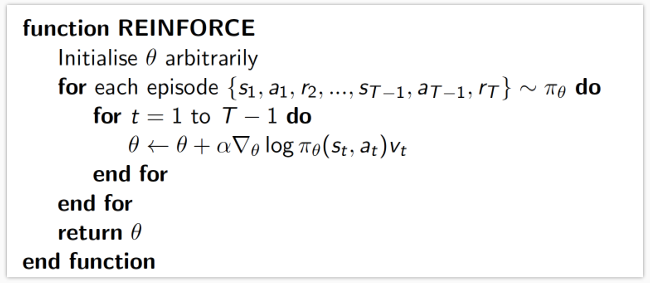


Figure 9 The reinforce method

The in indicates the degree of surprise of the selected action a in the state s. If the probability of is small, then the opposite value of will be large. If we got a large V (V represent reward) with a small , then the degree of surprise will be huge. (We will be surprised if we get a good reward while picking an action that we didn’t pick very often.) In this case, a modification of the neural network’s parameter ϴ needs to be done.[13]

(3) Example

We will resolve two problems: The Cart Pole problem and the Mountain Car problem. Here’s the description of the first one: A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 4 units from the center.

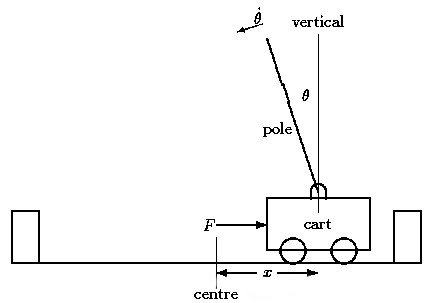


Figure 10 The Cart Pole problem

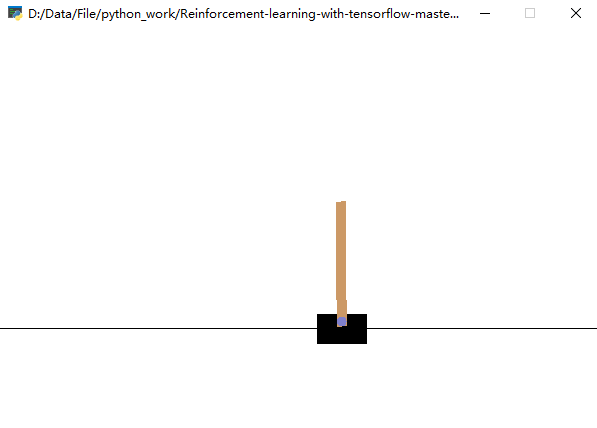


Figure 11 Simulation of Cart Pole problem

The module or the toolkit gym is for developing and comparing our reinforcement learning agents while matplotlib is a python plotting package that is widely used. We program first the main update loop like this:

import gym  
from RL\_brain import PolicyGradient  
import matplotlib.pyplot as plt  
  
DISPLAY\_REWARD\_THRESHOLD = 400 # renders environment if total episode reward is greater then this threshold  
RENDER = False # rendering wastes time  
  
env = gym.make('CartPole-v0')  
env.seed(1) # reproducible, general Policy gradient has high variance  
env = env.unwrapped  
  
print(env.action\_space)  
print(env.observation\_space)  
print(env.observation\_space.high)  
print(env.observation\_space.low)  
  
RL = PolicyGradient(  
 n\_actions=env.action\_space.n,  
 n\_features=env.observation\_space.shape[0],  
 learning\_rate=0.02,  
 reward\_decay=0.99,  
 # output\_graph=True,  
)

The content of this section below will be update only if one episode has completely finished.

for i\_episode in range(3000):  
  
 observation = env.reset()  
  
 while True:  
 if RENDER: env.render()  
  
 action = RL.choose\_action(observation)  
  
 observation\_, reward, done, info = env.step(action)  
  
 RL.store\_transition(observation, action, reward)  
  
 if done:  
 ep\_rs\_sum = sum(RL.ep\_rs)  
  
 if 'running\_reward' not in globals():  
 running\_reward = ep\_rs\_sum  
 else:  
 running\_reward = running\_reward \* 0.99 + ep\_rs\_sum \* 0.01  
 if running\_reward > DISPLAY\_REWARD\_THRESHOLD: RENDER = True # rendering  
 print("episode:", i\_episode, " reward:", int(running\_reward))  
  
 vt = RL.learn()  
  
 if i\_episode == 0:  
 plt.plot(vt) # plot the episode vt  
 plt.xlabel('episode steps')  
 plt.ylabel('normalized state-action value')  
 plt.show()  
 break  
  
 observation = observation\_

Mountain Car, a standard testing domain in Reinforcement Learning, is a problem in which an under-powered car must drive up a steep hill. Since gravity is stronger than the car's engine, even at full throttle, the car cannot simply accelerate up the steep slope. The car is situated in a valley and must learn to leverage potential energy by driving up the opposite hill before the car is able to make it to the goal at the top of the rightmost hill.

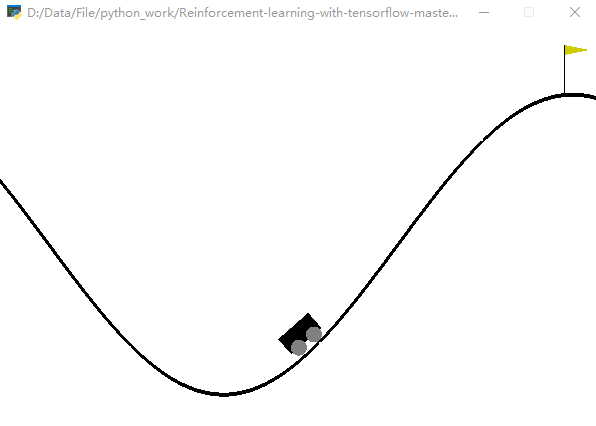
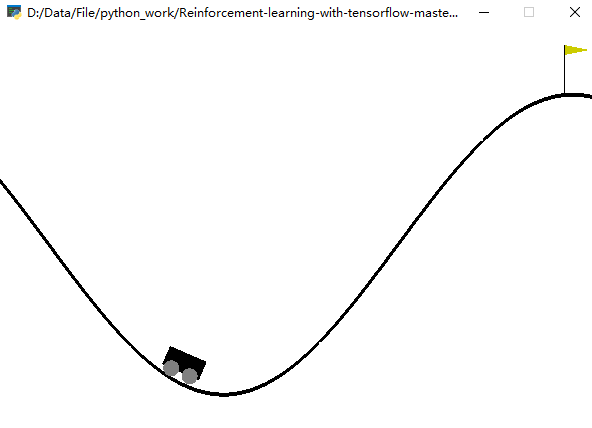


Figure 12 Simulation of the Mountain Car problem

Appropriate modification must be done in this problem. Here’s its parameter initialization and main loop programming:

import gym  
from RL\_brain import PolicyGradient  
import matplotlib.pyplot as plt  
  
DISPLAY\_REWARD\_THRESHOLD = -2000 # renders environment if total episode reward is greater than this threshold  
# episode: 154 reward: -10667  
# episode: 387 reward: -2009  
# episode: 489 reward: -1006  
# episode: 628 reward: -502  
  
RENDER = False # rendering wastes time  
  
env = gym.make('MountainCar-v0')  
env.seed(1) # reproducible, general Policy gradient has high variance  
env = env.unwrapped  
  
print(env.action\_space)  
print(env.observation\_space)  
print(env.observation\_space.high)  
print(env.observation\_space.low)  
  
RL = PolicyGradient(  
 n\_actions=env.action\_space.n,  
 n\_features=env.observation\_space.shape[0],  
 learning\_rate=0.02,  
 reward\_decay=0.995,  
 # output\_graph=True,  
)  
  
for i\_episode in range(1000):  
  
 observation = env.reset()  
  
 while True:  
 if RENDER: env.render()  
  
 action = RL.choose\_action(observation)  
  
 observation\_, reward, done, info = env.step(action) # reward = -1 in all cases  
  
 RL.store\_transition(observation, action, reward)  
  
 if done:  
 # calculate running reward  
 ep\_rs\_sum = sum(RL.ep\_rs)  
 if 'running\_reward' not in globals():  
 running\_reward = ep\_rs\_sum  
 else:  
 running\_reward = running\_reward \* 0.99 + ep\_rs\_sum \* 0.01  
 if running\_reward > DISPLAY\_REWARD\_THRESHOLD: RENDER = True # rendering  
  
 print("episode:", i\_episode, " reward:", int(running\_reward))  
  
 vt = RL.learn() # train  
  
 if i\_episode == 30:  
 plt.plot(vt) # plot the episode vt  
 plt.xlabel('episode steps')  
 plt.ylabel('normalized state-action value')  
 plt.show()  
  
 break  
  
 observation = observation\_

We program a class named PolicyGradient to include all related functions in this algorithm. The structure looks like those valued-based methods, for example, Q Learning.

class PolicyGradient:  
 def \_\_init\_\_(  
 self,  
 n\_actions,  
 n\_features,  
 learning\_rate=0.01,  
 reward\_decay=0.95,  
 output\_graph=False,  
 ):  
  
 def \_build\_net(self):  
  
 def choose\_action(self, observation):  
  
 def store\_transition(self, s, a, r):

def learn(self):  
  
 def \_discount\_and\_norm\_rewards(self):

At initialization, we need to give all static parameters and create a neural network.

def \_\_init\_\_(  
 self,  
 n\_actions,  
 n\_features,  
 learning\_rate=0.01,  
 reward\_decay=0.95,  
 output\_graph=False,  
):  
 self.n\_actions = n\_actions  
 self.n\_features = n\_features  
 self.lr = learning\_rate  
 self.gamma = reward\_decay  
  
 self.ep\_obs, self.ep\_as, self.ep\_rs = [], [], []  
  
 self.\_build\_net()  
  
 self.sess = tf.Session()  
  
 if output\_graph:  
 # $ tensorboard --logdir=logs  
 # http://0.0.0.0:6006/  
 # tf.train.SummaryWriter soon be deprecated, use following  
 tf.summary.FileWriter("logs/", self.sess.graph)  
  
 self.sess.run(tf.global\_variables\_initializer())

The neural network we are going to build this time is like:

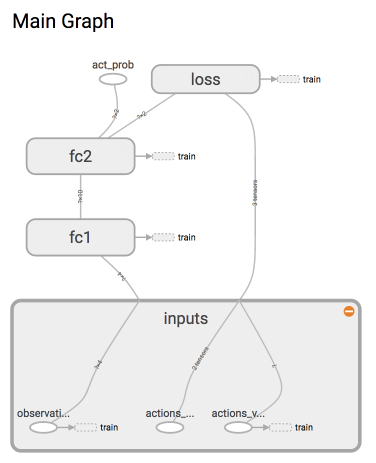


Figure 13 Sturcture of neural network

Because this is reinforcement learning, there is no y label in the neural network like in supervised learning. Instead, we choose the action. The loss function illustrated at the right-top of this figure often appears in a supervised learning problem measures the compatibility between a prediction (e.g. the class scores in classification) and the ground truth label. The data loss takes the form of an average over the data losses for every individual example. That is, where N is the number of training data.

def \_build\_net(self):  
 with tf.name\_scope('inputs'):  
 self.tf\_obs = tf.placeholder(tf.float32, [None, self.n\_features], name="observations")  
 self.tf\_acts = tf.placeholder(tf.int32, [None, ], name="actions\_num")  
 self.tf\_vt = tf.placeholder(tf.float32, [None, ], name="actions\_value")  
 # fc1  
 layer = tf.layers.dense(  
 inputs=self.tf\_obs,  
 units=10,  
 activation=tf.nn.tanh, # tanh activation  
 kernel\_initializer=tf.random\_normal\_initializer(mean=0, stddev=0.3),  
 bias\_initializer=tf.constant\_initializer(0.1),  
 name='fc1'  
 )  
 # fc2  
 all\_act = tf.layers.dense(  
 inputs=layer,  
 units=self.n\_actions,  
 activation=None,  
 kernel\_initializer=tf.random\_normal\_initializer(mean=0, stddev=0.3),  
 bias\_initializer=tf.constant\_initializer(0.1),  
 name='fc2'  
 )  
  
 self.all\_act\_prob = tf.nn.softmax(all\_act, name='act\_prob') # use softmax to convert to probability  
  
 with tf.name\_scope('loss'):  
 # to maximize total reward (log\_p \* R) is to minimize -(log\_p \* R), and the tf only have minimize(loss)  
 neg\_log\_prob = tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(logits=all\_act, labels=self.tf\_acts) # this is negative log of chosen action  
 # or in this way:  
 # neg\_log\_prob = tf.reduce\_sum(-tf.log(self.all\_act\_prob)\*tf.one\_hot(self.tf\_acts, self.n\_actions), axis=1)  
 loss = tf.reduce\_mean(neg\_log\_prob \* self.tf\_vt) # reward guided loss  
  
 with tf.name\_scope('train'):  
 self.train\_op = tf.train.AdamOptimizer(self.lr).minimize(loss)

We use as the loss function because this probability logarithmic is in fact the classification error of cross-entropy. The label in a classification problem is the correspondence between x and y while in PG, x is the state and y is a mark of action taken by referring state x. In supervised learning, this action could be always right and y can be a standard towards which we correct the parameters’ value. However, in RL, to make y again a standard tag, we must multiply our loss by vt to tell if the gradient calculated by this cross-entropy is a trustworthy gradient. If vt is small or negative, we can tell that the gradient descent is in a wrong direction. Thus, we are supposed to update the parameters in the other direction. Oppositely, if this vt is positive or large, the gradient out of the cross-entropy will be praised.[14]

When it comes to choose an action, we abandon the value-based method and use probability to choose them. Thus, even if there is not such a ε-greedy, we can still have some certain randomness, which represents the exploration part in a RL algorithm. It’s simple to store the episode by adding observation, action and rewards of this episode to a list. Noticing that the list has to be cleared before next episode, we will realize this in the function learn(). Here’s the code of these two parts:

def choose\_action(self, observation):  
 prob\_weights = self.sess.run(self.all\_act\_prob, feed\_dict={self.tf\_obs: observation[np.newaxis, :]})  
 action = np.random.choice(range(prob\_weights.shape[1]), p=prob\_weights.ravel()) # select action w.r.t the actions prob  
 return action

def store\_transition(self, s, a, r):  
 self.ep\_obs.append(s)  
 self.ep\_as.append(a)  
 self.ep\_rs.append(r)

In function learn(), we need to make all rewards more suitable to be learned by introduce a discount factor γ to decay future rewards. In order to reduce the variance of PG among episodes, we standardize the state-action value of each episode.

def learn(self):  
 # discount and normalize episode reward  
 discounted\_ep\_rs\_norm = self.\_discount\_and\_norm\_rewards()  
  
 # train on episode  
 self.sess.run(self.train\_op, feed\_dict={  
 self.tf\_obs: np.vstack(self.ep\_obs), # shape=[None, n\_obs]  
 self.tf\_acts: np.array(self.ep\_as), # shape=[None, ]  
 self.tf\_vt: discounted\_ep\_rs\_norm, # shape=[None, ]  
 })  
  
 self.ep\_obs, self.ep\_as, self.ep\_rs = [], [], [] # empty episode data  
 return discounted\_ep\_rs\_norm

Finally, how we realize the attenuation of future rewards.

def \_discount\_and\_norm\_rewards(self):  
 # discount episode rewards  
 discounted\_ep\_rs = np.zeros\_like(self.ep\_rs)  
 running\_add = 0  
 for t in reversed(range(0, len(self.ep\_rs))):  
 running\_add = running\_add \* self.gamma + self.ep\_rs[t]  
 discounted\_ep\_rs[t] = running\_add  
  
 # normalize episode rewards  
 discounted\_ep\_rs -= np.mean(discounted\_ep\_rs)  
 discounted\_ep\_rs /= np.std(discounted\_ep\_rs)  
 return discounted\_ep\_rs

Here are test results of simulation respectively of and Cart Pole and Mountain Car. We can see about when the agent renders environment.

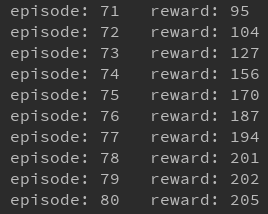
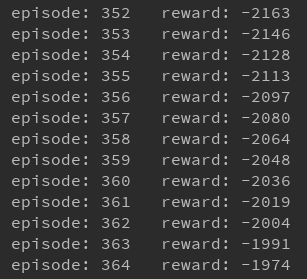


Figure 14 Convergence check

# 3 Conclusion

Through this research study in the field of reinforcement learning, we learned a lot of interesting knowledge and skills, and completed some exciting small examples. However, these are just the tip of the iceberg. There are still too much to be explored, both from the code level and theoretical knowledge level. If there is time, we plan to implement a small game on the Steam platform: GridWord. It is like a giant maze and the creatures inside will learn to avoid disasters and multiply.

# 4 Reference

1. Reinforcement Learning: An Introduction, Richard S. Sutton and Andrew G. Barto, 2015.
2. [J. Kober, J. Andrew (Drew) Bagnell, and J. Peters, “Reinforcement Learning in Robotics: A Survey,” International Journal of Robotics Research, July, 2013.](http://www.ias.tu-darmstadt.de/uploads/Publications/Kober_IJRR_2013.pdf)
3. Continuous Deep Q-Learning with Model-based Acceleration, Shixiang Gu et al., arXiv, 2016.
4. Mastering the game of Go with deep neural networks and tree search, D. Silver et al., Nature, 2016.
5. MazeBase: A Sandbox for Learning from Games, S. Sukhbaatar et al., arXiv, 2016.
6. Towards Vision-Based Deep Reinforcement Learning for Robotic Motion Control, F. Zhang et al., arXiv, 2015.
7. Continuous control with deep reinforcement learning, T. P. Lillicrap et al., ICLR, 2016.
8. Action-Conditional Video Prediction using Deep Networks in Atari Games, J. Oh et al., NIPS, 2015.
9. Learning Continuous Control Policies by Stochastic Value Gradients, N. Heess et al., NIPS, 2015.
10. Human-level control through deep reinforcement learning, V. Mnih et al., Nature, 2015.
11. Deep Learning for Real-Time Atari Game Play Using Offline Monte-Carlo Tree Search Planning, X. Guo et al., NIPS, 2014.
12. Playing Atari with Deep Reinforcement Learning, V. Mnih et al., NIPS Workshop, 2013.
13. Policy Gradient Methods for Reinforcement Learning with Function Approximation, Richard S. Sutton et al, Advances in Neural Information Processing Systems 12, MIT Press, 2000.
14. Deep Reinforcement Learning: Pong from Pixels, Andrej Karpathy, Hacker’s guide to Networks, 2016.