Reinforcement learning

WANG Zeyao,WANG Chengzhi,CHEN Chen

January 2019

Contents

1 Introduction

1.1 Reinforcement learning

Reinforcement learning (RL) is an area of machine learning concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward.
Reinforcement learning can be understood using the concepts of agents, environments, states, actions and rewards, all of which we'll explain below. Capital letters tend to denote sets of things, and lower-case letters denote a specific instance of that thing; e.g. A is all possible actions, while a is a specific action contained in the set.
·Agent: An agent takes actions; for example, a drone making a delivery, or Super Mario navigating a video game. The algorithm is the agent. In life, the agent is you.
·Action (A_t): A is the set of all possible moves the agent can make. An action is almost self-explanatory, but it should be noted that agents choose among a list of possible actions. In video games, the list might include running right or left, jumping high or low, crouching or standing still. In the stock markets, the list might include buying, selling or holding any one of an array of securities and their derivatives. When handling aerial drones, alternatives would include many different velocities and accelerations in 3D space.
·Discount factor: The discount factor is multiplied by future rewards as discovered by the agent in order to dampen thse rewards' effect on the agent's choice of action. Why? It is designed to make future rewards worth less than immediate rewards; i.e. it enforces a kind of short-term hedonism in the agent.
·Environment: The world through which the agent moves. The environment takes the agent's current state and action as input, and returns as output the agent's reward and its next state. If you are the agent, the environment could be the laws of physics and the rules of society that process your actions and determine the consequences of them.
·State (S_t): A state is a concrete and immediate situation in which the agent finds itself; i.e. a specific place and moment, an instantaneous configuration that puts the agent in relation to other significant things such as tools, obstacles, enemies or prizes. It can the current situation returned by the environment, or any future situation.
·Reward (R_t): A reward is the feedback by which we measure the success or failure of an agent's actions. From any given state, an agent sends output in the form of actions to the environment, and the environment returns the agent's new state (which resulted from acting on the previous state) as well as rewards, if there are any. Rewards can be immediate or delayed. They effectively evaluate the agent's action.
·Policy: The policy is the strategy that the agent employs to determine the next action based on the current state. It maps states to actions, the actions that promise the highest reward.
·Value (V_t): The expected long-term return with discount, as opposed to the

short-term reward R. We discount rewards, or lower their estimated value, the further into the future they occur. See discount factor.
·Q-value or action-value (Q): Q-value is similar to Value, except that it takes an extra parameter, the current action a. $Q(s_t, a_t)$ refers to the long-term return of the current state s_t, taking action a under policy. Q maps state-action pairs to rewards.
·Trajectory: A sequence of states and actions that influence those states.
So environments are functions that transform an action taken in the current state into the next state and a reward; agents are functions that transform the new state and reward into the next action. We can know the agent's function, but we cannot know the function of the environment. It is a black box where we only see the inputs and outputs. It's like most people's relationship with technology: we know what it does, but we don't know how it works. Reinforcement learning represents an agent's attempt to approximate the environment's function, such that we can send actions into the black-box environment that maximize the rewards it spits out.
The reinforcement learning follows the Markov chain below:

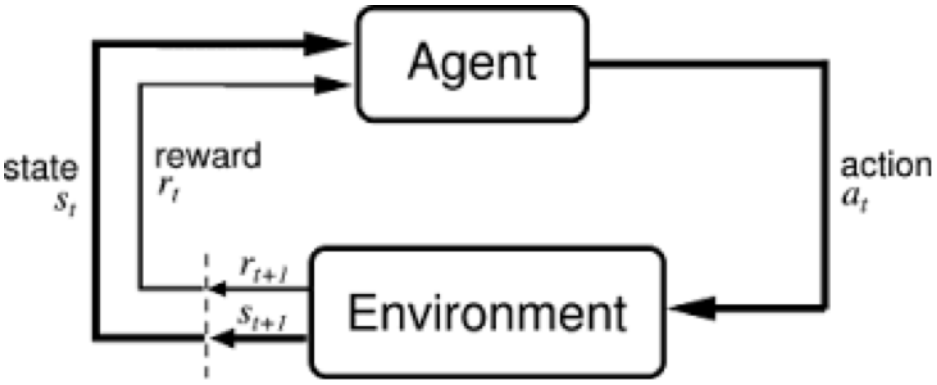


Figure 1: Markov chain

We start with an initial environment. It doesn't have any associated reward yet, but it has a state S_t
Then for each iteration, an agent takes current state (S_t), picks best (based on model prediction) action (A_t) and executes it on an environment. Subsequently, environment returns a reward (R_{t+1}) for a given action, a new state (S_{t+1}) and an information if the new state is terminal. The process repeats until termination.

2 Physical method of cartpole problem

2.1 Cartpole problem

Cartpole - known also as an Inverted Pendulum is a pendulum with a center of gravity above its pivot point. It's unstable, but can be controlled by moving the pivot point under the center of mass.
The following figure shows an inverted pendulum. The aim is to move the wagon along the x direction to a desired point without the pendulum falling. The wagon is driven by a DC motor, which is controlled by a controller (analog in our implementation). The wagons x position (not in our case) and the pendulum angle è are measured and supplied to the control system.

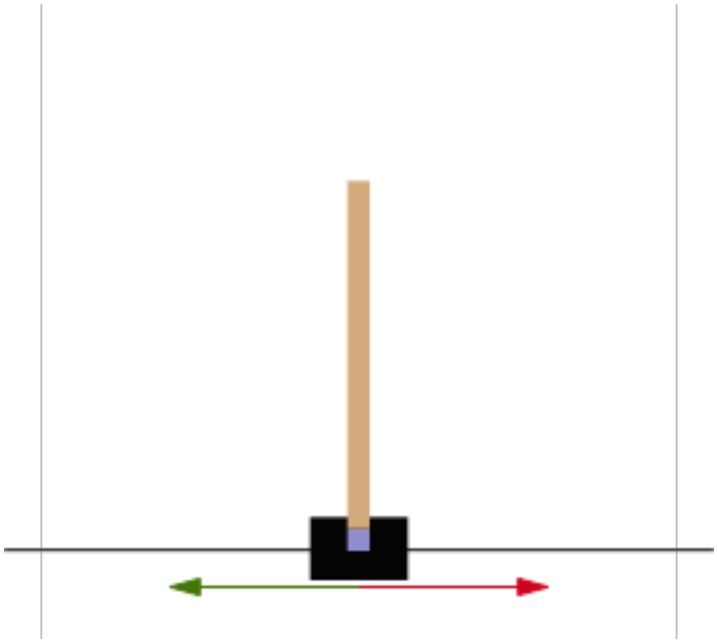


Figure 2: The cartpole

·Violet square indicates a pivot point
·Red and green arrows show possible horizontal forces that can be applied to a pivot point

2.2 Analyse the system

Here we consider a pendulum-car system
The mass of the pendulum : m
The length of the pendulum : l
The mass of the car : M

The angle of pendulum : θ
Moment of inertia : J
The control force u(t) acts along the x direction of the car

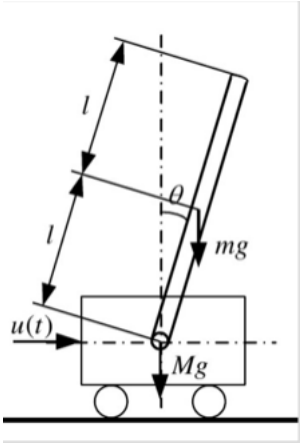


Figure 3:

According to the Figure 3. The force equation of car along the x direction is:

$$M\ddot{x} = u(t) - R_x \quad (1)$$

The force equation of the pendulum along the x direction is:

$$m\ddot{x} + ml\ddot{\theta}cos\theta = R_x \quad (2)$$

Torque equation of pendulum:

$$J\ddot{\theta} = mglsin\theta - ml\ddot{\theta}l - m\ddot{x}lcos\theta \quad (3)$$

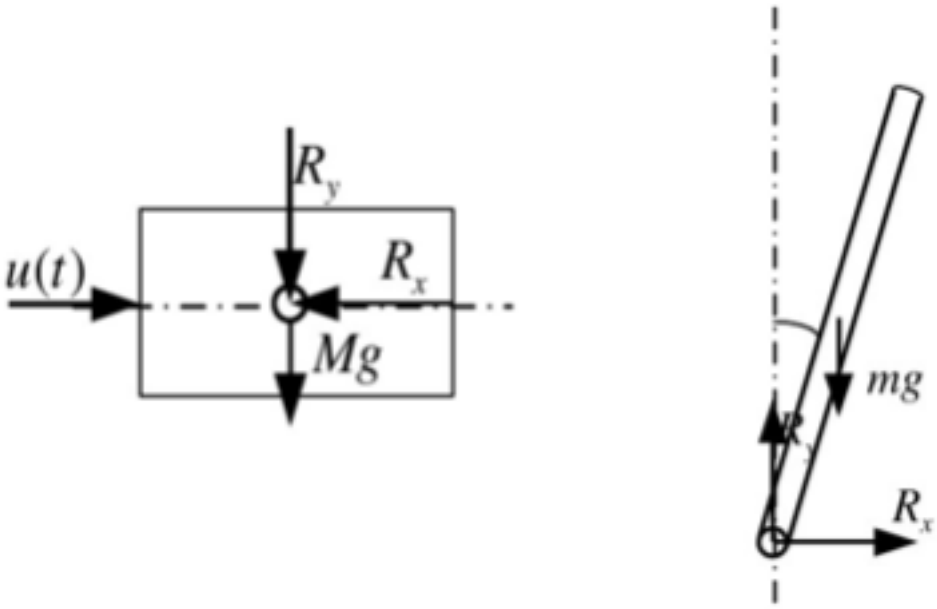


Figure 4:

The angle of the inverted pendulum angle is not very large, it can be approximated.

$$sin\theta = \theta, cos\theta = 1, \theta^2 = 0$$

Finally, the linearization model of the inverted pendulum system is:

$$(M+m)\ddot{x} + ml\ddot{\theta} = u(t) \quad (4)$$

$$(J+ml^2)\ddot{\theta} + ml\ddot{x} = mgl\theta \quad (5)$$

To obtain the transfer function of the linearized system equations analytically, we must first take the Laplace transform of the system equations. The Laplace transforms are:

$$(M+m)s^2X(s) + bsX(s) - mls^2\phi(s) = U(s)$$

$$(J+ml^2)s^2\phi(s) - mgl\phi(s) = mlX(s)s^2$$

When finding the transfer function, initial conditions are assumed to be zero. The transfer function relates the variation from desired position [Output] to the force on the cart [Input]

Re-arranging, the transfer function is:

$$\frac{\Phi(s)}{U(s)} = \frac{\frac{ml}{q} \cdot s^2}{s^4 + \frac{b(I+ml^2)}{q} \cdot s^3 - \frac{mgl(M+m)}{q} \cdot s^2 - \frac{bmgl}{q} \cdot s}$$

where,

$$q = (M+m)(I+ml^2) - (ml)^2.$$

The physical parameters of the system prototype are tabulated as follows:
The mass of the pendulum : m=0.1kg
The length of the pendulum : l=0.25m
The mass of the car : M = 1kg
So we have the transfer function with the physical parameters

$$\frac{\theta(s)}{U(s)} = \frac{3}{s^2 - 29.4}$$

2.3 Method of PID

2.3.1 Introduction

Our implementation contains only feedback from the pendulum angle (that is, only one out of the four states is used for feedback, the other states being carriage position, carriage velocity and pendulum angular velocity). The implementation may be enhanced by incorporation of cart-position control loop. In this problem, the pendulum is first positioned upright manually, that is, in a position of unstable equilibrium, or it is given some initial displacement (position). The controller is then switched in to balance the pendulum and to maintain this balance in the presence of disturbances. A simple disturbance may be a light tap on the balanced pendulum. A complex disturbance may be gusts of wind. This setup can be used to study the control of open loop unstable system. It is a demonstration of the stabilizing benefits of feedback control. A range of control techniques ranging from the simple phase advance compensator to neural net controllers can be applied.
In this experiment, the proportion (P) and integral (I) and differential (D) of deviation are linearly combined to form a control quantity, and the controlled object is controlled by this control quantity. Such a controller is called a PID controller.

2.3.2 PID in the cartpole problem

The transfer function of the input of acceleration and the output of angular velocity: $\frac{\theta(s)}{R(s)} = \frac{3}{s^2-29.4}$
If the output is the angle of the pendulum relative to the vertical axis (in upright

position), we realize that the system is unstable, since the pendulum will fall down if we release it with a small angle. To stabilize the system, i.e., to keep the pendulum in upright position, a feedback control system must be used.
Here we use simulink to simulate this system, obviously it's unstable. The impulse response of open-loop uncompensated system is shown below:

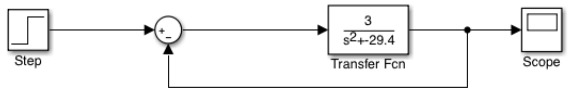
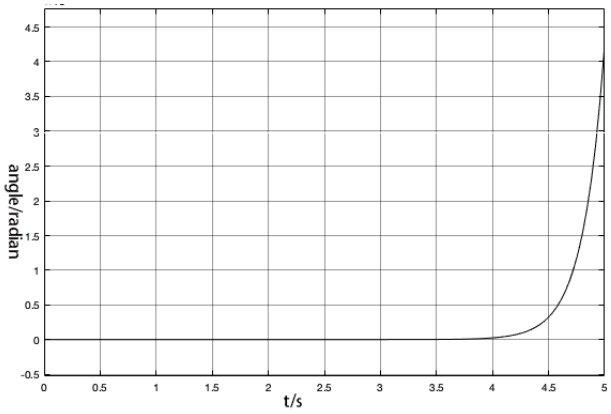


Figure 5: No PID



And then we used the method of PID. The transfer function can be written as $G(s) = \frac{R(s)}{E(s)} = K_P + \frac{K_I}{s} + K_D s = \frac{K_D s^2 + K_P s + K_I}{s}$
K_P: Proportion coefficient
K_I: Integral coefficient
K_D: Differential coefficient
We change different values of K_P,K_I,K_D and we find that:
(1)Increasing the proportion coefficient K_P can speed up the response of the system. But if it is too large, it can cause system instability
(2)Increasing the integral coefficient K_I can increase the stability of system but the speed of reducing the error will slow down.
(3)Increasing the differential coefficient K_D can speed up the response of the system. But the system's ability to suppress disturbances is weakened.
Of all these three factors, the proportion coefficient has the greatest influence.

We use the value: K_D=30 K_P=600 K_I=5000

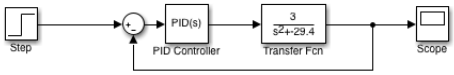
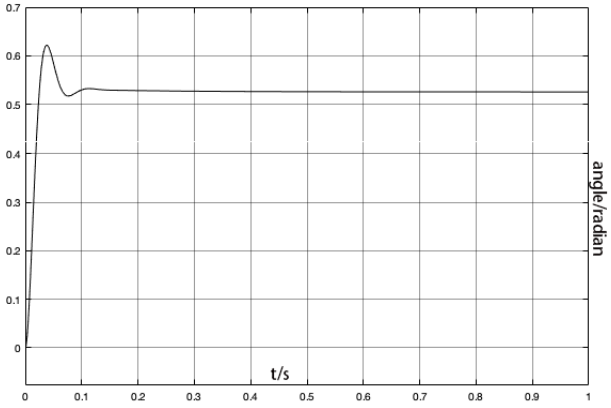


Figure 6: The impulse response of system with PID controller is shown below(the angle converge to 0.52 radian)



3 Data processing of cartpole problem

3.1 Environement

In order to simulate the cartpole problem, we have used OpenAI gym.
OpenAI is created for removing this problem of lack of standardization in papers along with an aim to create better benchmarks by giving versatile numbers of environment with great ease of setting up. Aim of this tool is to increase reproducibility in the field of AI and provide tools with which everyone can learn about basics of AI.
OpenAI gym includes these following elements:

3.1.1 Observation

Type: Box(4)

Num	Observation	Min	Max
0	Cart Position	-2.4	2.4
1	Cart Velocity	-Inf	Inf
2	Pole Angle	~ -41.8°	~ 41.8°
3	Pole Velocity At Tip	-Inf	Inf

3.1.2 Actions

Type: Discrete(2)

Num	Action
0	Push cart to the left
1	Push cart to the right

3.1.3 Reward

Reward is 1 for every step taken, including the termination step

3.1.4 Starting state

All observations are assigned a uniform random value between ± 0.05

3.1.5 Episode termination

(1)Pole Angle is more than ± 12
(2)Cart Position is more than ± 2.4 (center of the cart reaches the edge of the display)

3.1.6 Solved requirements

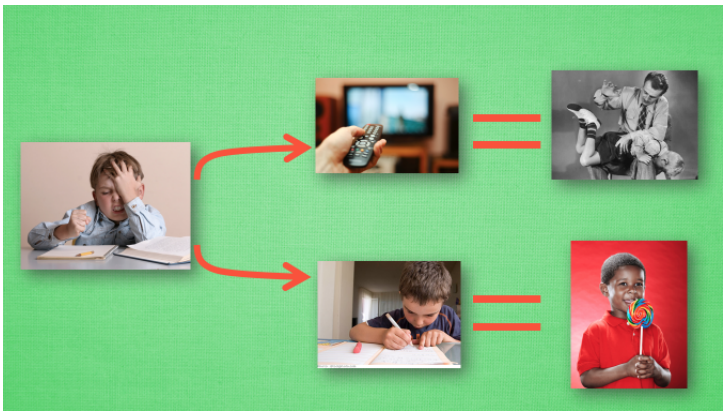
Considered solved when the average reward is greater than or equal to 195.0 over 100 consecutive trials.

3.2 Q learning

In order to use reinforment learning to let the cartpole keep balanced, we decide to use Q learning at first. To explain it well, we have posed the following exemple:

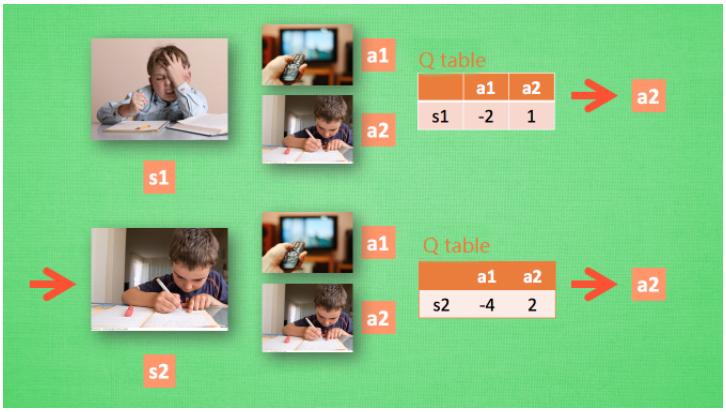
3.2.1 Exemple of Q learning

We suppose now that we are in the state of writing assignments and we have not tried to watch TV while writing homework, so now we have two choices here:
(1) continue to write homework
(2) go to watch TV.
Because I have not been punished before, so I chose to watch TV. And now the state changed to watching TV. at the second choice I chose to continue watching TV. Then I watched TV. Finally, my parents went home and found that I didn't finish my homework and went to watch TV. I was punished once,



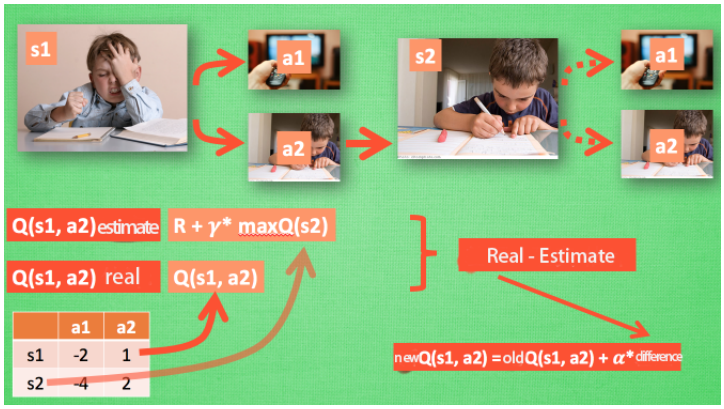
and then I profoundly recorded this experience, and changed the behavior of "watching TV without writing homework" into negative behavior in my mind.

3.2.2 Decision making of Q learning



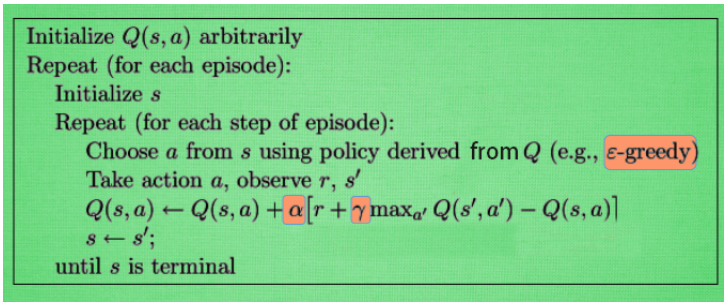
We suppose now that the policy has already been learned. Now we are in state s1 which is writing homework, and meanwhile I have two choices of actions a1, a2, which are watching TV and continuing to write homework. According to my experience in this state s1, the potential rewards brought by a2(writing homework) are better than a1(watching tv). The potential reward here can be replaced by a Q table with s and a. In my memory Q table, Q(s1, a1) = - 2 is smaller than Q(s1, a2) = 1. So we decided to choose a2 as the next action. Now our state is updated to s2, we still have two identical choices. Repeat the above process to find the values of Q(s2, a1)Q(s2, a2) in Q table and compare their sizes. Pick the larger one. Then according to a2 we reach s3 and repeat the above decision process.

3.2.3 Update of Q learning



Let's go back to the previous process. According to the estimation of the Q table, we chose a2 in s1 and reached s2 through the previous decision method because the value of a2 is relatively large in s1. At this moment, we started to update the Q table for decision, and then we did not take any behavior in practice, but imagine myself taking each action in s2, and see which of the two actions has a larger Q value. For example, the value of Q(s2, a2) is greater than Q(s2, a1), so we multiply the Q(s2, a2) by an attenuation value γ and add the reward R obtained when we reach s2 (the lollipop has not been obtained yet, so The reward is 0) and we use this as the value of my real Q(s1, a2). But we used to estimate the value of Q(s1, a2) according to the Q table. So with the reality and the estimated value, we can update Q(s1, a2). Based on the difference between the estimate and the reality, multiply this gap by a learning efficiency alpha and add the value of the previous Q(s1, a2) to a new value. Although we have estimated the state s2 with maxQ(s2), we have not yet done it in s2. The action of s2, the action decision of s2 will wait until the update is completed and then do it again.

3.2.4 Algorithm of Q learning

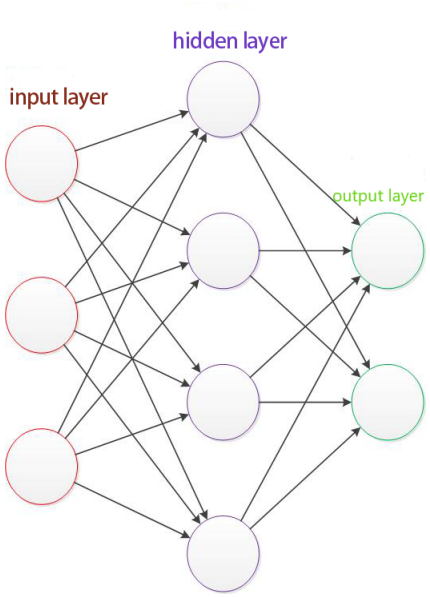


We use Q reality and Q estimation for each update. In the reality of Q(s1, a2), it also contains a maximum estimate of Q(s2), which will be the reality of this step by estimating the maximum attenuation of the next step and the current reward.
ϵ greedy is a strategy used in decision making. For example, when $\epsilon = 0.9$, it means that in 90% of cases, I will choose the action according to the optimal value of the Q table, and 10% of the time will use the random selection of action.
α is the learning rate to determine how much of this error is to be learned, α is a number less than 1.
γ is the attenuation value for future reward.

3.3 Deep Q network

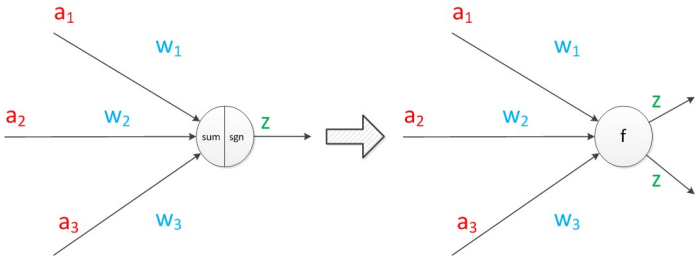
In the normal Q learning, we use Q-table to find the Q value corresponding to the state and action. But if the problem is too complicated and you use the table to store them, our computer won't have enough memory, and it is time consuming to search for the corresponding state in such a large table. So we tried to use another method.

3.3.1 Neural network



This is a neural network with three levels. Red is the input layer, green is the output layer, and purple is the middle layer (also called the hidden layer).
Input Nodes – The Input nodes provide information from the outside world to the network and are together referred to as the "Input Layer". No computation is performed in any of the Input nodes – they just pass on the information to the hidden nodes.
Hidden Nodes – The Hidden nodes have no direct connection with the outside world (hence the name "hidden"). They perform computations and transfer information from the input nodes to the output nodes. A collection of hidden nodes forms a "Hidden Layer". While a feedforward network will only have a single input layer and a single output layer, it can have zero or multiple Hidden Layers.
Output Nodes – The Output nodes are collectively referred to as the "Output Layer" and are responsible for computations and transferring information from the network to the outside world.

Connection is the most important thing in a neuron. There is a weight on



each connection.
A neural network training algorithm is to adjust the weight of the weight to the best, so that the prediction of the entire network is the best.
We use a to represent the input and w to represent the weight. A directed arrow indicating a connection can be understood as follows: at the beginning, the transmitted signal is still a, and there is a weighting parameter w in the middle. After this weighted signal, it becomes a*w, so at the end of the connection, the signal The size becomes a*w.
A neuron can lead to multiple directed arrows representing the output, but the values are the same

$$z = sgn(a1w1 + a2w2 + a3w3)$$

and here we combine function sum with sgn and become function f.

3.3.2 Learning process of neural network

Learning progress of neural network is to find the parameters(weight) corresponding to the loss function when it is minimized. The loss function measures the error between the predicted value of the model versus the sample and the real one. Minimized solutions generally use Gradient Descent. The steps include:
(1)Initialize the parameter
(2)Calculate the gradiant of function loss corresponding to the parameters
(3)Update the value of the parameters accoring to the gradiant
(4)Get the optimal parameter after iteration and finish the learning In our code we have used the following to finish the learning of the neural network:

```
history = self.model.fit(np.array(states), np.array(targets_f),
    epochs=1, verbose=0)
#Np.array(state) is an array of states, np.array(targets_f) is
    the corresponding updated q value, epoch = 1 means training
    once, verbose=0 means that the log is not displayed in the
    output
```

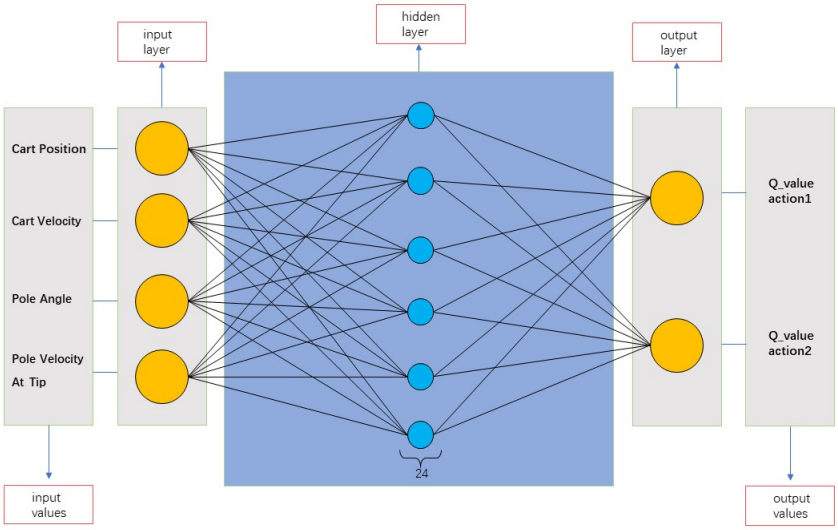


Figure 7: Our structure of neural network

3.3.3 Neural network in cartpole problem

There are 24 neurons. All these inputs are four-dimensional because state space is four-dimensional. All these outputs are two-dimensional because action space is two-dimensional.
Input layer has 4 inputs and 24 outputs.
Hidden layer has 24 inputs and 24 outputs.
Output layer has 24 inputs and 4 outputs.
To set up the neural net with Python:

3.4 Advantages and disadvantages between Q learning and deep Q learning

So, in conclusion, as we mentioned above, we have two inputs, 'State', 'Action' and one output, 'Q value' for the traditional Q-learning method. We use a Q-table to look for the Q-value for the specific state and action. The advantage of the traditional method is the simplicity, all we need is a table to store the q-value, and we can get it by giving it state and action. But, as a matter of fact, the simplicity is also the drawback of it. If the environment is too complicated, it's hard to create a table to store all the q-value and even if we did create it, it will cost us too much memory and cause a problem of efficiency.
For the Deep Q learning, we use a neural network to do the job of q-table, but the differences are the neural network only have one input, the 'State' and more than one outputs. The good news is that we don't need to store all the q-value

or find them out like looking up a dictionary, the neural network can calculate the q-value itself, and we can handle a situation much more complicated than before, no problem of it. But the bad news is we must train the neural network in advance, it will also take us some time. Here, for the 'cartpole' environment, the states are a little complicated and have a large amount of them. It's impossible for us to calculate all the q-value and put them into a table. So, we choose to use the Deep Q-learning to solve this problem. It save us memory and it works more efficiently.

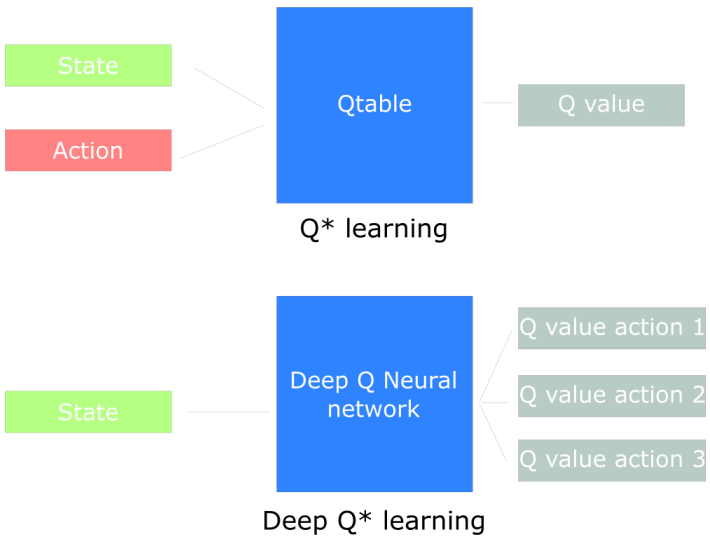


Figure 8: Our structure of neural network

4 Numerical experiments(common conditions of two methods)

4.1 Physical conditions of the car and the pole, and the physical environment

4.1.1 Physical environment

A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum starts upright, and the goal is to prevent it from falling over by increasing and reducing the cart's velocity. No any friction.

4.1.2 Physical condition

Gravity = 9,8 m/s2
Mass cart = 1.0kg
Mass pole = 0.1kg
Length pole = 0.25m

4.2 Initial conditions of cart and pole

For the cartpoleV0, the initial conditions are a random float between -0.05 0.05 for each variable in the observation space.
For the PID control, the four variables in the observation space is 0, but give it an instance acceleration of 0.05m/s2.

4.3 Control conditions for cart and pole

For the pole, we have no control on it. In the gym environment cartpoleV0, the control for the cart is simply give the cart a force with 10N (a force to the right) or a force with -10N (a force to the left).
For the PID control, the force is given by a function

$$\frac{\theta(s)}{U(s)} = \frac{3}{s^2 - 29.4}$$

4.4 Failure condition

The original failure conditions for cartpoleV0 are
(1)Pole Angle is more than ± 12
(2)Cart Position is more than ± 2.4 (center of the cart reaches the edge of the display)
In order to compare PID control and Reinforcement Learning, we change the failure condition for the cartpole V0 to:
(1)Pole Angle is more than ± 0.1 radian (As close as possible to 0)
(2)Cart Position has no limits

So that the two methods can have the same failure condition. It will be more accurate for us to compare two methods in the same failure conditions.

4.5 Stability conditions for cart and pole

For the PID control, the definition of stability is the radian of the pole converge to a number which close to 0.
For the cartpole V0, the definition of stability is the average score is greater than or equal to 195.0 which is the m-score over 100 consecutive trials.
Score- The number of actions before the experiment fails

5 Numerical experiments(Comparison between two methods)

5.1 The Platform and Software Employed by the Two Experiments

Physic: Simulink in Matlab
Reinforcement learning: python
Environment: OPEN AI–cartpole V0
Neural Network: tensorflow keras

5.2 Comparisons between the preparations for the two methods)

Physical Method: Calculation of the PID Parameter, The Development of the MATLAB Model
Reinforcement Learning Method: Construction of the Algorithm for Reinforcement Learning, Building of Neural Network

5.3 Stable time and the result

We use the value: K_D=30 K_P=600 K_I=5000 It takes around 0.2s for PID control to converge its radian to 0.51 and it stays very stable, hardly has perturbation.

The first graph contains the score for each run. We can learn from it that at first 80 runs the scores are very low, it's because of the Q-value hasn't been updated enough to show the real (or right, proper) Q-value for the pair of state and the action. Also, the epsilon in the epsilon-greedy strategy is very high at first, so the agent often gives us a random action in order to explore. After 100 runs, we start to detect the average score to see if it achieves the stability condition, so we have the red line start from 100th run, it continually increases and it reach the aim score at 159th run, cost us 402.24s. After reach the aim score, the average score stays beyond the aim score with some perturbations.
The second graph is about the value of the loss function. The loss function

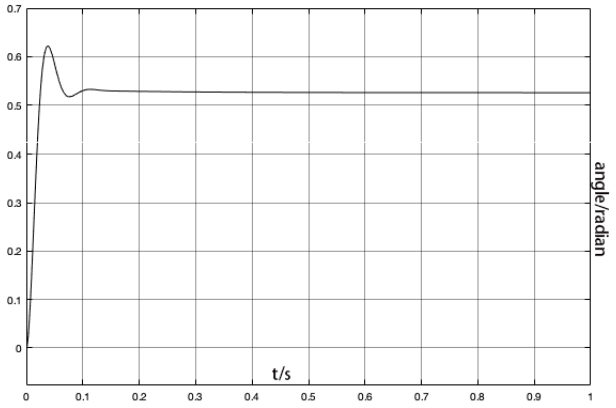
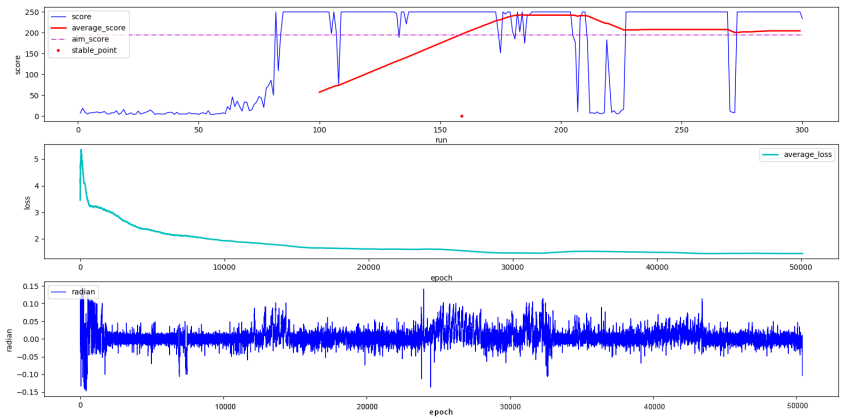


Figure 9: Result of PID method




```
Stable time:  402.24s
Stable run: 159
```

Figure 10: Result of reinforcement learning

represents how well the model fits the data. The better the model fits the data, the less the value of loss function would be. As we can see from the graph, the value of loss function continually decreases, the value is almost 0 after 4000 epochs.

The third graph shows the angle (as radian) for every epoch. At the end the

radians of the pole are mainly between -0.05 0.05, it's rather close to 0. But it still has some perturbations, we think it's because we set the limit of the angle to -0.1 radian to 0.1 radian, so the agent of reinforcement learning learned how to prevent the pole from fall over -0.1 radian or 0.1 radian, it's normal to have some perturbations, after all the perturbations didn't break the rules(the agent didn't been trained that strictly). So, if we change the units for y axis from 0.05 radian to 0.1 radian, we may have a better converge in this graph.

5.4 Advantages and Disadvantages

Advantages of Physical Method:
(1) Stability
(2) Less time spent to achieve stability
Disadvantages of Physical Method:
(1) Lots of preparations for the experiment
(2) Large amounts of manual calculation
(3) Parameters needed to be adjusted by practical situation and human experiment.
Advantages of Reinforcement Learning Method:
(1) Little preparation for experiment
(2) Little calculation.
Disadvantages of Reinforcement Learning Method:
(1) Lack of Stability

6 Conclusion

In the inverted pendulum experiment, the PID seems to perform better (faster convergence and better stability), but the PID requires a lot of manual calculations in the early stage and relies on human experience to adjust the parameters, while the reinforcement learning only needs more time to learn itself and it can also get enough accurate results. So if there is a more complicated system, reinforcement learning is a better choice.

7 Appendix

```
import random
import gym
import numpy as np
import matplotlib.pyplot as plt
from collections import deque
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam

### parameters
EPISODES = 1000
MAX_EPISODES_STEP = 250
ENV_NAME = 'CartPole-v0'
MEMORY_SIZE = 2000
GAMMA = 0.95
ALPHA = 0.8
EPSILON_MIN = 0.01
EPSILON_DECAY = 0.995
LEARNING_RATE = 0.001
STABLE_RUNTIME = 100
STABLE_THRESHOLD = 195

### when finish, build the graph of score and loss
class Score_Logger:
    def __init__(self):
        self.score = []
        self.run = []
        self.mean_score = []
        self.loss = []
        self.epoch = []
        self.mean_loss = []
        self.stable_run = []
        self.angle = []
        self.mean_angle = []
        self.step = []
        self.mean_run = []

    def add_score(self,score,run):
        self.score.append(score)
        self.run.append(run)

    def add_loss(self,loss,epoch,mean_loss):
        self.loss = loss
        self.epoch = epoch
        self.mean_loss = mean_loss

    def add_angle(self,angle,step):
```

```
    self.angle.append(angle)
    self.step.append(step)
    self.average_angle()

def average_angle(self):
    nsum = 0
    for i in range(len(self.angle)):
        nsum += self.angle[i]
    self.mean_angle.append(nsum / len(self.angle))

def getAngel(self):
    return self.angle

def getStep(self):
    return self.step

def getAverage(self):
    return self.mean_score[len(self.mean_score)-1]

def getLoss(self):
    return self.loss

def getStable(self):
    return self.stable_run

def average_score(self):
    score = []
    for i in range(1,STABLE_RUNTIME+1):
        score.append(self.score[len(self.score)-i])
    nsum = 0
    for i in range(len(score)):
        nsum += score[i]
    mean = (nsum/len(score))
    self.mean_score.append(mean)
    self.mean_run.append(len(self.score))

def isStable(self):
    if len(self.stable_run) > 0:
        return
    score = []
    for i in range(1,STABLE_RUNTIME+1):
        score.append(self.score[len(self.score)-i])

    nsum = 0
    for i in range(len(score)):
        nsum += score[i]
    mean = (nsum/len(score))
    if mean >= STABLE_THRESHOLD:
        self.stable_run.append(len(self.score))
```

```
    def calTime(self):
        time = 0.0
        if len(self.stable_run) == 1:
            index = 0
            index = self.stable_run[0]
            for i in range(0,index):
                time += self.score[i]
            time = time * 0.02
            return str(time)+'s'+'\nStable run: '+str(index)
        else:
            return 'Not stable'

    def plot(self):
        aim = np.full((EPISODES,),STABLE_THRESHOLD)
        stable = np.full((len(self.stable_run),),0)
        pic_score = plt.subplot(3,1,1)
        pic_loss = plt.subplot(3,1,2)
        pic_angle = plt.subplot(3,1,3)

        plt.sca(pic_score)
        plt.plot(self.run,self.score,'b-',linewidth=1,label='score')
        plt.plot(self.mean_run,self.mean_score,'r-',linewidth=2,label='average_score')
        plt.plot(self.run,aim,'m-.',linewidth=1,label='aim_score')
        plt.plot(self.stable_run,stable,'r.',linewidth=3,label='stable_point')
        plt.legend(loc='upper left')
        plt.xlabel('run')
        plt.ylabel('score')

        plt.sca(pic_loss)
        plt.plot(self.epoch,self.mean_loss,'c-',linewidth=2,label='average_loss')
        plt.legend(loc='upper right')
        plt.xlabel('epoch')
        plt.ylabel('loss')
        plt.show()

        plt.sca(pic_angle)
        plt.plot(self.step,self.angle,'b-',linewidth=1,label='radian')
        plt.legend(loc='upper left')
        plt.xlabel('epoch')
        plt.ylabel('radian')


### principal class for DQN algorithm
class DQNAgent:
    def __init__(self, state_size, action_size):
        self.state_size = state_size # the size for input is the size of
            state space which is 4
        self.action_size = action_size # the size for output is the size
            of action space which is 2
```

```
    self.memory = deque(maxlen=MEMORY_SIZE)# create the container of
        the memory
    self.epsilon = 1.0 # exploration rate
    self.model = self._build_model()
    self.epoch = 0
    self.epoch_list = []# record the number of epoch
    self.loss_list = []# record the loss for each epoch
    self.mean_loss = []


def _build_model(self):# build of neural network for Deep-Q learning
    Model
    model = Sequential()# using a sequntial model
    model.add(Dense(24, input_dim=self.state_size,
        activation='relu'))# input layer
    model.add(Dense(24, activation='relu'))# hidden layer
    model.add(Dense(self.action_size, activation='linear'))# output
        layer
    model.compile(loss='mse',optimizer=Adam(lr=LEARNING_RATE))#
        after build,compile the DNN
    return model

def remember(self, state, action, reward, next_state, done):#store
    the data for memory
    self.memory.append((state, action, reward, next_state, done))

def act(self, state):#using the epsilon-greedy strategy to choose
    the action
    if np.random.rand() <= self.epsilon:
        return random.randrange(self.action_size)
    act_values = self.model.predict(state)
    return np.argmax(act_values[0]) # returns action

def replay(self, batch_size):
    minibatch = random.sample(self.memory, batch_size)
    states, targets_f = [], []
    for state, action, reward, next_state, done in minibatch:
        target = reward
        if not done:
            ### principal code for DQN,we update the Q-value by using
                the q-value formula
            target =
                (1-ALPHA)*(self.model.predict(state)[0][action])+ALPHA*(reward
                + GAMMA *
                    np.amax(self.model.predict(next_state)[0]))
        target_f = self.model.predict(state)
        target_f[0][action] = target
        # Filtering out states and targets for training
        states.append(state[0])
        targets_f.append(target_f[0])
```

```
        history = self.model.fit(np.array(states), np.array(targets_f),
            epochs=1, verbose=0)# using the state and the new q-value to
            train the DNN for once
        # Keeping track of loss
        loss = history.history['loss'][0]
        self.loss_list.append(loss)
        self.average_loss()
        self.epoch_list.append(self.epoch)
        self.epoch += 1
        if self.epsilon > EPSILON_MIN:
            self.epsilon *= EPSILON_DECAY

    def load(self, name):
        self.model.load_weights(name)

    def save(self, name):
        self.model.save_weights(name)

    def average_loss(self):
        nsum = 0
        for i in range(len(self.loss_list)):
            nsum += self.loss_list[i]
        self.mean_loss.append(nsum / len(self.loss_list))

    def getLoss(self):
        return self.loss_list

    def getEpoch(self):
        return self.epoch_list

    def getMeanloss(self):
        return self.mean_loss


### main loop
if __name__ == "__main__":
    ### gym environment set up start
    env = gym.make(ENV_NAME)
    env._max_episode_steps = MAX_EPISODES_STEP
    state_size = env.observation_space.shape[0]
    action_size = env.action_space.n
    agent = DQNAgent(state_size, action_size)
    score_logger = Score_Logger()
    done = False
    batch_size = 32
    # agent.load("./save/cartpole-dqn.h5")
    ### gym environment set up end
    run = 0
    stateStep = 0
    for e in range(EPISODES):
```

```
        if run >= STABLE_RUNTIME:
            score_logger.isStable()
            score_logger.average_score()
        run += 1
        state = env.reset()# when the done is 'true' or in this episode
             we score 499,reset the environment and start again
        state = np.reshape(state, [1, state_size])
        score = 0
        for time in range(500):
            score += 1
            stateStep += 1
            env.render()# to show the graphic interface
            action = agent.act(state)# get action from the agent
            next_state, reward, done, _ = env.step(action)
            score_logger.add_angle(next_state[2],stateStep)
            reward = reward if not done else -10# if the in this time the
                pole didn't fall we set reward as 1 else we set it as -10
            next_state = np.reshape(next_state, [1, state_size])
            agent.remember(state, action, reward, next_state, done)#
                store the data in the memory
            state = next_state
            if done:
                print("episode: {}/{}, score: {}, e: {:.2}"
                      .format(e, EPISODES, time, agent.epsilon))
                score_logger.add_score(score,run)
                break
            if len(agent.memory) > batch_size:
                agent.replay(batch_size)# do the update of the q-value
                    and train the DNN
    agent.save("./save/cartpole-dqn.h5")
    stable = score_logger.getStable()
    step = score_logger.getStep()
    angle = score_logger.getAngel()
    score_logger.add_loss(agent.getLoss(),agent.getEpoch(),agent.getMeanloss())
    score_logger.plot()# plot the graph
    print('Stable time: ',score_logger.calTime())
```