# Reinforcement Learning 2021
# Practical Assignment 2: Function Approximation

Xinrui Shan(s2395738) Pengyu Liu(s2604531) Zhe Deng(s2696266)

## 1 Introduction

Reinforcement learning is a machine learning field studying how the agent responds to an environment with the goal of maximizing the cumulative reward. A standard Reinforcement learning environment-agent loop is represented in Figure 1.

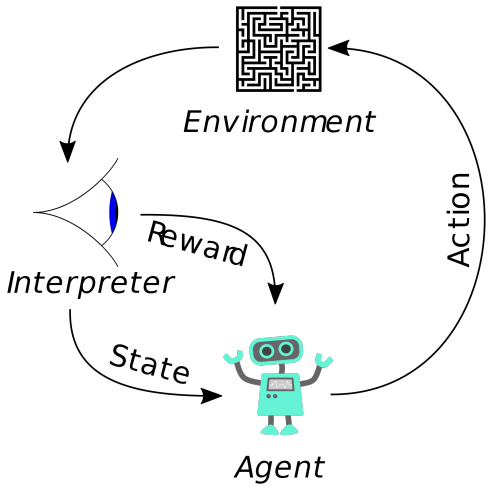



Figure 1: From Wikipedia "Reinforcement learning" page [1]. A typical Reinforcement Learning loop. The agent takes an action on the environment. The environment then returns the new state and reward to the agent. The agent keeps take action and the environment responds until the game is over. As goes with round and round of games, the intelligent agent will learn from previous knowledge to perform better, in our Cartpole is maintaining longer time and get higher cumulative rewards.

Q-learning is one of the model-free Reinforcement learning algorithms (it does not require the mathematical or physical model to solve the problem). Q value will be updated at each iteration based on the Bellman equation showing below, it is a weighted average of the old value and the new information:

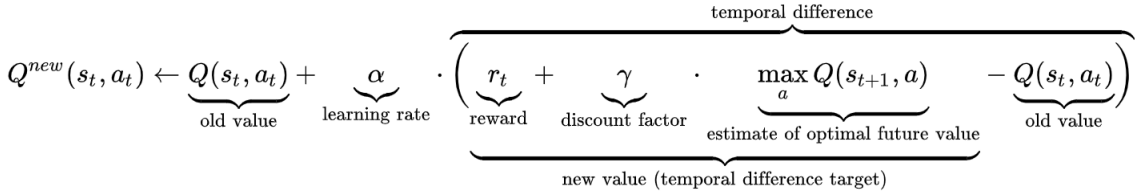



Figure 2: Bellman equation from Wikipedia "Q-learning" page.

There are three adjustable variables in the Bellman equation, learning rate $\alpha$, discount factor $\gamma$, initial conditions $Q_0$.

- learning rate $\alpha$ denotes at what extend we want to use the new information, for example, the agent learns nothing if $\alpha = 0$ and makes decisions only based on the latest information if $\alpha = 1$. Based on researchers' experience, around $\alpha = 0.1$ is always a good choice.
- discount factor $\gamma$ represents how much we want to look into the future, $Q(s_{t+1,a})$ is the estimate of optimal future value as shown in the equation. $\gamma = 0$ means the agent is short-sight only considering reward in the current iteration. Note the algorithm may diverge if setting $\gamma$ close to 1.
- initial state $Q_0$, we must carefully choose the initial state because the Q value is updated from it, a good choice of initial state could help learn faster.

The above paragraph about Q-learning introduction is referred from Wikipedia "Q-learning" page [2].

In this assignment, we will study the CartPole v1 problem. The goal of this control game is to let the pole not deviate more than 12 degrees from vertical, and keep the cart does not move more than 2.4 units from the center by applying a force on a cart left or right. We implemented three function approximation algorithms, Tabular Q-learning, Deep Q-learning, Monte-Carlo policy gradient.

# 2 Tabular Q-learning

Tabular Q-learning uses a table to record $Q(s, a)$ for every state and will be updated at each iteration via the Bellman equation. The procedure of this algorithm and parameter setup is following:

1. discretize the environment. The environment returns four continuous parameters at each step that are cart position, cart velocity, pole angle, pole velocity at the tip. Generally, the smaller intervals of parameters could make decisions more accurately, the limit case is the table could give one-to-one "personalized" advice which is the continuous case. However, it has two disadvantages: (1) it required large storage because there are four

dimensions (parameters), (2) there are so many elements in Q-table that require many episodes running even to update all elements in Q-table once. Moreover, through our pre-experiment, divide four or five groups for each parameter is sufficient to have a good preference (with cumulative rewards reach 500 after around 700 episodes in 1 minute), the help is little if implement with more groups. **Eventually we discretize the state as follows bins: cart position=4, cart velocity=2, pole angle=4, pole velocity=2.** You may notice the groups for velocity parameters are only two because we think with information whether it would move left or right (positive or negative velocity) is kind of enough. The comparison experiment is included in section 2.1.2.

2. initialize the Q table: **we choose to initialize Q table to zero everywhere to give equal weight at the beginning.** We tried random initial table as well, the convergence rate and eventually preference is similar as you can see in section 2.1.3.

3. When the game is not done, **choose the action with the highest Q value of the state under a probability of** $1 - \epsilon$ where $\epsilon$ decreases as episode increase meaning we give this algorithm a chance to test beyond from the Q table at the start of the experiment so that it won't be stuck in some local minimum, then strictly follow the Q table at episode larger than 700. If you examine most of the figures showing in section 2.1, you will find the episodes before 700 are always bad because of this $\epsilon$ randomness. We have tried not to introduce $\epsilon$ or only set these test experiments to 100 episodes but comes out with a low convergence rate and low convergence value. Therefore we finally decide to choose figure 3 curve.

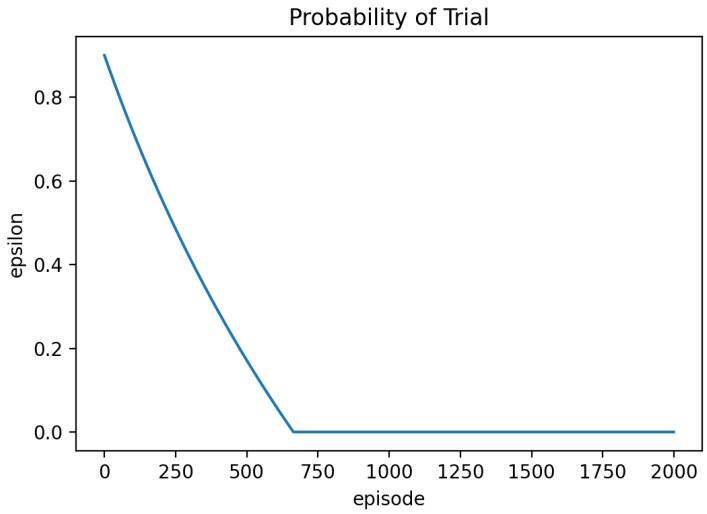



Figure 3: choose the action with the highest Q value under a probability of $1 - \epsilon$

4. update Q table via Bellman equation, there are two adjustable parameters, learning rate $\alpha$, discount factor $\gamma$. We finally choose $\alpha = 0.1, \gamma = 0.9$. We have tested the $\alpha$ from

a grid of [0.,0.2,0.3,0.4] and $\gamma$ from a grid of [0.8,0.9]. The experiment details are in section 2.1.4.

5. reward choice: the environment automatically returns a reward value 0 if the game fails at the current iteration otherwise returns 1. We found this bi-reward is not accurate enough, hence we define a new continuous reward that can reflect the goodness of the state (how close it is toward the center position and 0 angle deviation)

$$reward = (1 - \frac{|location|}{2.4}) \times (1 - \frac{|angle|}{12}). \tag{1}$$

## 2.1 experiments

Here are experiments about (1) baseline accuracy, (2) discretization bin choice, (3) initial Q-table, (4) learning rate and discount factor. All figures are the **average of the latest 20 episode cumulative rewards** because we want to see the trend clearly and every time plot has obvious ups and downs. **We keep this parameter set as default when running following scientific control experiments: bins=(4,2,4,2), initialization with a zeros array,** $\alpha = 0.1$,$\gamma = 0.9$**, and attached a curve for this default setting at section 2.1.5.**

**Run following experiments with an explicit random seed, actually, you cannot reproduce the exact curve as ours, but would get a really similar one**, we think this might be caused by the randomness in Gym environment.

### 2.1.1 baseline accuracy

Here we take action with random choice, no algorithm supported, can be called "baseline accuracy. The average cumulative reward of all episodes is around 22. If looking at the following sections experiments, the performances of them are better as goes with episode and eventually can reach an average of 300 cumulative rewards, even hit 500 for some individual episode. Therefore we could say that Q Tabular learning is kind of good learning algorithm solving Cartpole problem.

### 2.1.2 Tabular Q-learning discretization choice

Here is the result of the experiments of different discretized table choices. Figure 5 are three typical cases for two discretization choice, (4,2,4,2) and (4,4,4,4). Actually, we run many tests, found sometimes (4,2,4,2) is superior, but sometimes is not. The most important thing is though we set the random seed, the result changes every time, we think there may be some randomness inside the Gym environment, the second cause might be our Q-table update is dependent on the behavior of every episode (and the episode behavior is random, sad). Eventually, this leads to a non-stable result. As a conclusion of this experiment, since fewer bins have storage and running time advantages, we take two velocity bins setting.

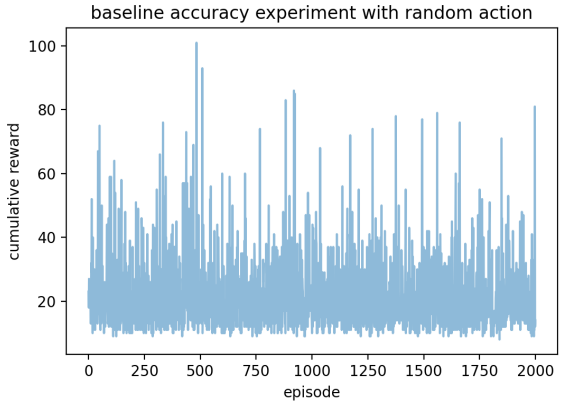
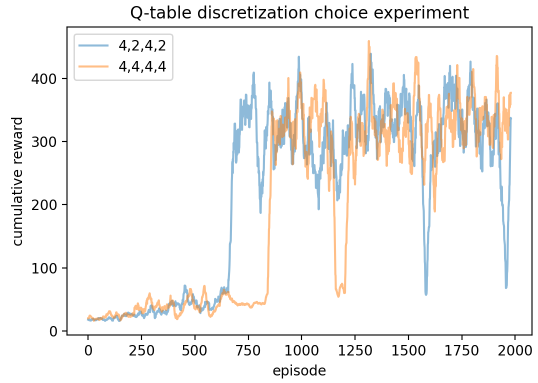
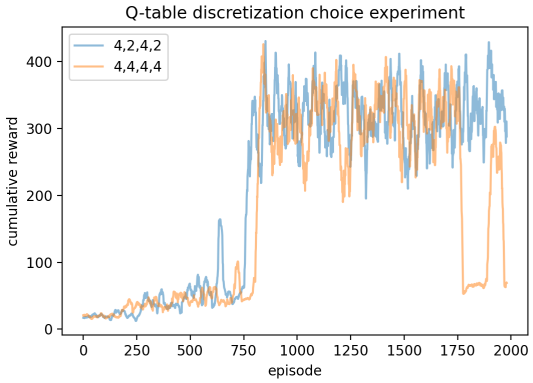


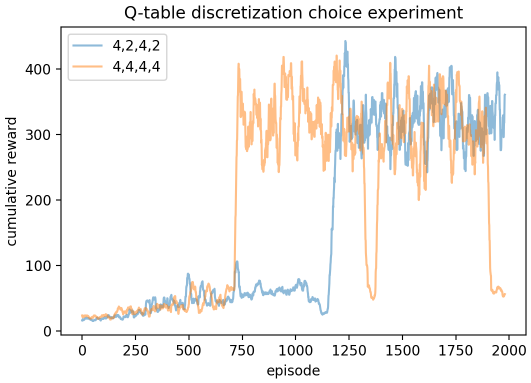

Figure 4: Make action with random choice, no algorithm supported, can be called "baseline accuracy.




(a)

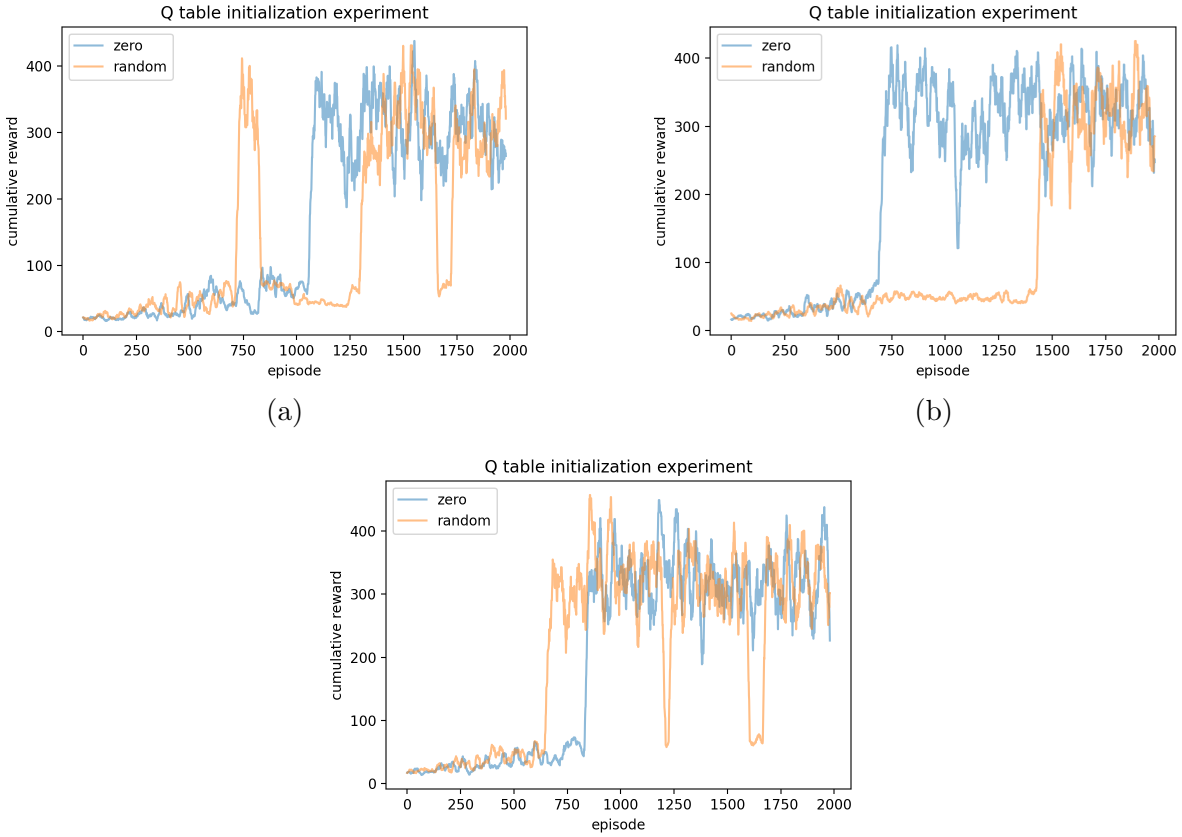



(b)




(c)

Figure 5: There are three typical cases for two discretization choices, (4,2,4,2) and (4,4,4,4). Actually, we run many tests, found sometimes (4,2,4,2) is superior, but sometimes is not.




(a)

(b)

(c)

Figure 6: The comparison between random initialization Q table (-0.05,0.05 uniform distribution) and filled with zeros initialization. This figure shows some typical curves. Sometimes the randomness initialization is better, sometimes it is not.

### 2.1.3 Q table initialization experiment

Figure 6 is the comparison between random initialization Q table (-0.05,0.05 uniform distribution) and filled with zeros initialization. We found the same issue here as in section 2.1.2. The performance of convergence rate is not stable and there is no absolute good choice for randomness or zeros initialization.

### 2.1.4 learning rate, discount factor experiments

There are two adjustable parameters, learning rate $\alpha$, discount factor $\gamma$. We have tested the $\alpha$ from a grid of [0.,0.2,0.3,0.4] and $\gamma$ from a grid of [0.8,0.9]. We found the convergence rate is similar for some parameter sets, for example $\alpha = 0.1$ $\gamma = 0.9$ and $\alpha = 0.2$ $\gamma = 0.9$. We believe the reason might be it is not quite sensitive in this selected parameter space.

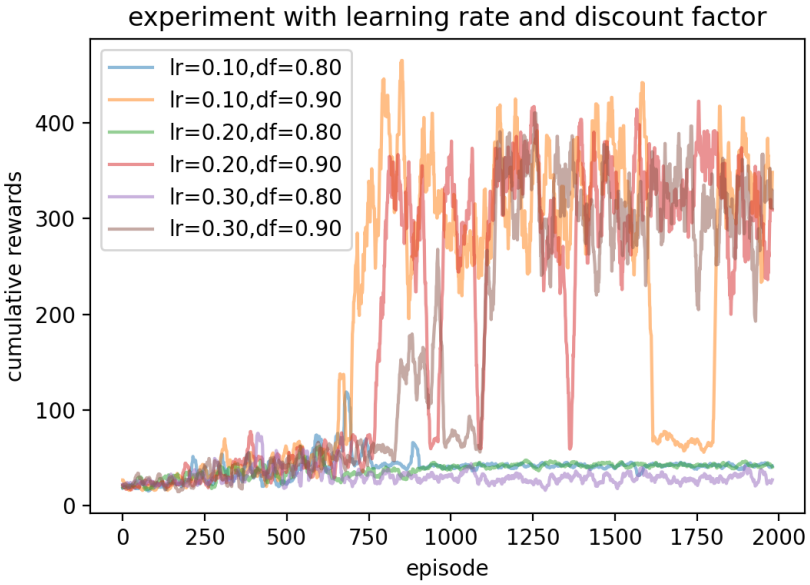



Figure 7: Six cumulative rewards curves in learning rate and discount factor space.

### 2.1.5 final chosen experiment parameter set

We made a plot in 10,000 episodes for our final parameter set: bins=(4,2,4,2), initialization with a zeros array, $\alpha = 0.1$,$\gamma = 0.9$. As you can see in figure 8, the first 700 episodes the cumulative rewards curve shows an upward trend, and when the random trial is not applied anymore (after 700 episodes), the total rewards hit 500 at some episode, overall keeps an average of 320.


Figure 8: Our final chosen parameter set experiment. The blue curve is the cumulative rewards of the individual episodes, the yellow curve is the moving average of the latest 20 episodes.

# 3 Deep Q-learning

## 3.1 Algorithm

Beyond Tabular Q-learning, we implement DQN. In reality, most problems have a huge state space or action space. If a Q table needs to be built, sometimes it is absolutely not allowed considering memory, and the amount of data and time overhead are also problems.

The Function Approximation method is to solve the problem of too large state space, also known as the "curse of dimensionality". By expressing $Q(s, a)$ as a function instead of a Q table, this function can be linear or non-linear. It can be described as the equation: $\hat{q}(s, a, \omega) \approx q_\pi(s, a)$. ($\omega$ is called "weight"). In order to find this weight (that is, to fit such a suitable function), we combine some supervised learning algorithms in the machine learning algorithm, extract the features of the input state as input, calculate the value function as the output, and then the function The parameter $\omega$ is trained until convergence. As for regression algorithms, there are linear regression, decision trees, neural networks, etc.

Thus, DQN (Deep Q-Network)[3] can be introduced. In fact, it is a combination of Q-Learning and neural network, which turns the Q table of Q-Learning into Q-Network.

## 3.2 Implementation of DQN

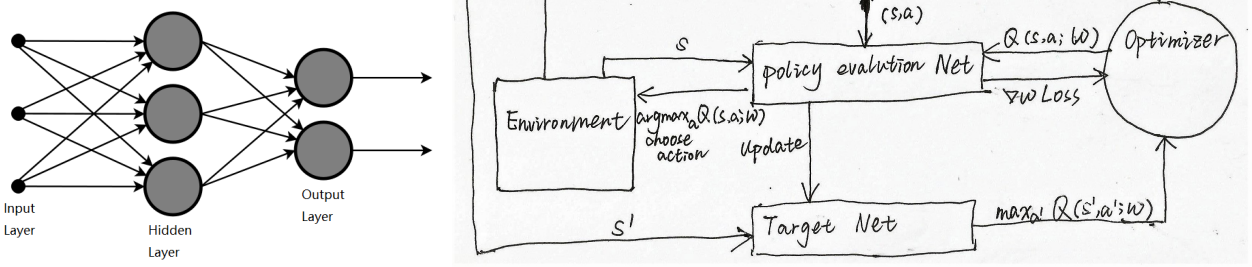
In order to train this network, in other words, to determine the network parameter $\omega$, we need a Loss Function and enough training samples. MSE loss function and Adam optimizer are adopted, as well as $\epsilon$-greedy strategy is used to generate training samples.

The update of the samples adopts the batch method, first collect a bunch of samples, and then sample a part of them to update the Q network, which is called "experience replay". In fact, the DQN proposed by DeepMind adopts the experience replay method. Why use the experience replay method? Because when training the neural network, it is assumed that the samples are independent and identically distributed. However, there is a correlation between the data collected through reinforcement learning. Using these data for sequential training, the neural network is of course unstable. Experience replay can break the association between data.

### 3.2.1 Neural Network Architecture

We finally decided to implement a fully connected neural network (FNN), since the CartPole environment's step() function returns four values: [observation, reward, done, info] which is exactly what we need.

Our choice on the architecture of the neutral network is simple, but it performs well on the CartPole problem. Besides the input layer. it has one hidden layer and one output layer. In this way, we build a fully connected neural network (FNN). The FNN we implement has one hidden layer, and there are 128 nodes in this layer. A ReLU() activation function is included between the input layer and the hidden layer. The batch size is 32. DQN with experience replay, the size of the replay memory is 1000, and the frequency of that update the memory is 100.




(a) Structure of the FNN (b) Draft of data flow

Figure 9: Fully connected neural network (FNN)

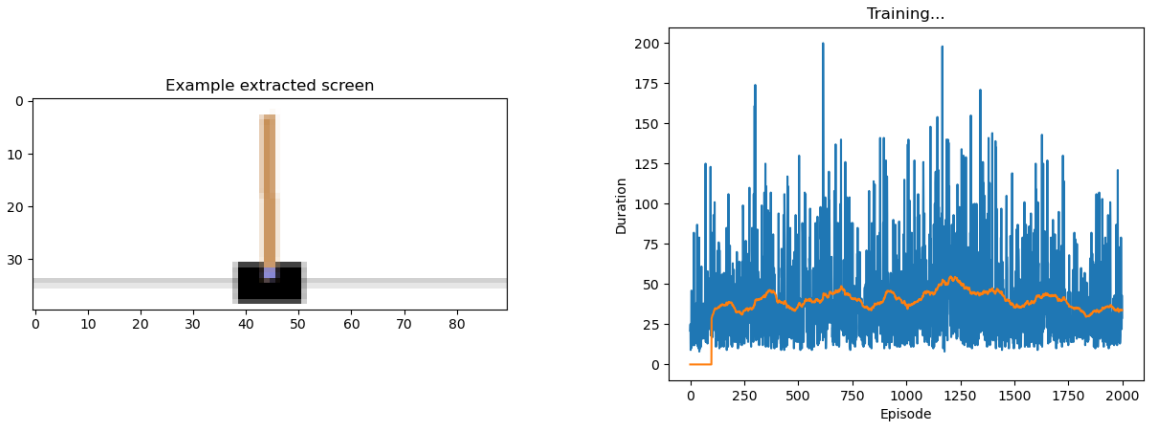
Since the problem is to optimize the value that the environment return, the dimension is low and there is no need to extract feature from image, we could only use FNN without convolutions. Convolutional layers are widely used to process image data as input.

However, we indeed test the idea of using convolutions:
In the beginning, our idea is to take a screenshot of each state frame of the dynamic CartPole

video, and then input it to the convolutional layers for feature extraction, then analyze the extracted features, and finally output the action that should be taken. According to this train of thought, we test and refer to the examples that have been implemented in the tutorial on the official website of Pytorch[4].

First of all, it extracts, crops and compresses the screenshot returned to 40*90, so that it can be better input into the convolutional neural network (CNN). In addition to the input layer and output layer, the CNN consists of 3 convolutional layers and a linear fully connected layer: Conv2d(3,16); Conv2d(16,32); Conv2d(32,32); Linear 512 nodes to 2 nodes. The action decision adopts the $\epsilon$-greedy policy, which means that there is a certain ratio and random behavior is selected (otherwise, it will act according to the best behavior predicted by the network). This ratio gradually decreases from 0.9 to 0.05, decreasing according to the exp curve.[4]

After running it for 2000 epochs, we can see that the average duration of the CartPole is about 50, and the max one is only 200. However, the CNN does not make full use of the information we have. If the problem is of high dimension or to deal with the image data, we could do like this. The OpenAI gym CartPole system could return the values that we want. we could make use of the vector they return to achieve a higher accumulated reward.




(a) Extraction of the screen

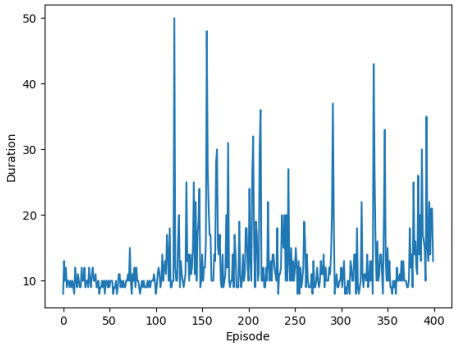
(b) Results of the example on Pytorch tutorial

Figure 10: Convolutional neural network (CNN)

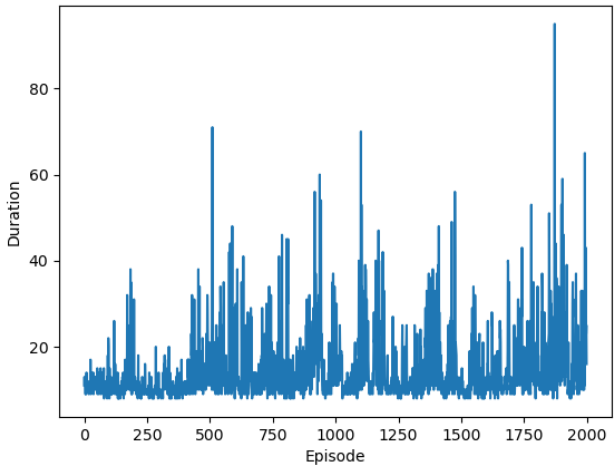
After testing the CNN example, we decided not to use convolution. Not only because we need to simplify the problem itself by using FNN, but also because of the lack of relevant knowledge, it is difficult to come up with a way to improve the performance of CNN.
By using FNN, in order to optimize the performance. we only need to consider the things related to algorithm and math. A series of models embedded in torch could be used, as well as the modified function we think that fits the problem.

## 3.3 Experiments and Results

Several experiments are carried out. The first 100 epochs are used to collect the memory for the experience replay, so the process of learning begins at $100^{th}$ epoch. 400 epochs' training seems not enough, and at the second time of training, 2000 epochs are chosen.




(a) 400 epochs, $\alpha$=0.01, $\gamma$=0.7




(b) 2000 epochs, $\alpha$=0.01, $\gamma$=0.7

Figure 11: Results of training with the original discrete "0 or 1" reward

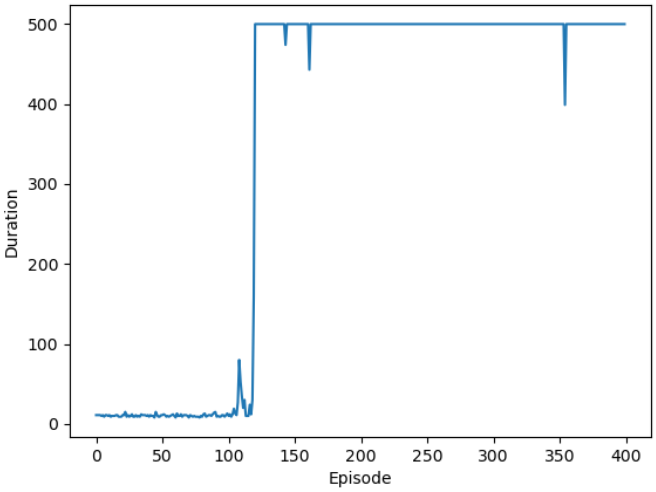
### 3.3.1 Parameter tuning

From figure 11 below, the accumulated reward of the DQN varies from 20 to 50, which performs not as well as we expect. Compared to the test of the CNN example before, both types of neutral network have accumulated reward around 50.

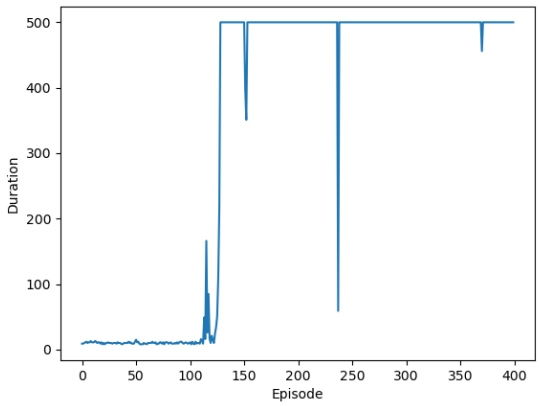
This is due to the discrete reward itself. The networks cannot learn the crucial key to make the CartPole hold for a longer time, because the environment automatically returns a reward value 0 if game fails at current iteration otherwise is 1. The reward which we use to learn is either 0 or 1.

To improve the performance, we modify the reward and use this modified one to train the networks. At the same time, we do the same accumulation of the original reward. The modified reward, which is named "score" in the DQN code, includes 2 main aspects of the evaluation of the current state: position of the cart, and angle of the pole. The way to calculated the score is illustrated in previous sections as Equation 1. In this way, the networks will learn that if the closer the cart is to the center and the closer the angle is to 0, the larger possibility of the CartPole can be achieved to make the system hold on for a long time.

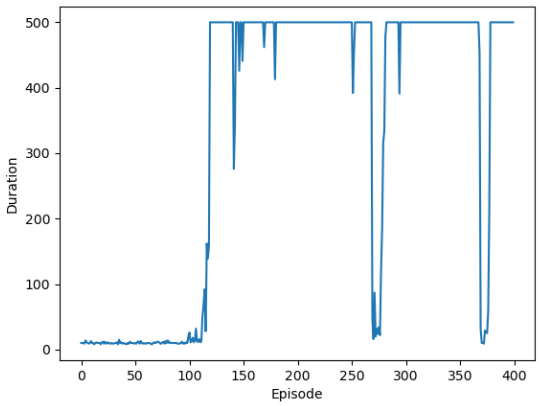
The results after the improvement are shown in the following figure 12. After training with the continuous value representation of the state, the performances increase sharply and greatly. Because the first 100 epochs is the process of collecting for the replay memory, the actual training process starts at the $100^{th}$ epoch. The accumulated reward could reach 500(full mark) at around $120^{th}$ epoch and keep 500 for following episodes. With the learning rate 0.01 and reward discount 0.7, the performance could stay at a high level, which is shown in the figure 12(a).




(a) $\alpha$=0.01, $\gamma$=0.7




(b) $\alpha$=0.005, $\gamma$=0.7




(c) $\alpha$=0.01, $\gamma$=0.9

Figure 12: Results of training with the modified continuous "0 to 1" score

# 4 Monte-Carlo policy gradient (MCPG)

## 4.1 Algorithm

The third method we used is the Monte Carlo policy gradient, a policy search algorithm. It is an off policy algorithm. Different from Q-learning, we parameterize the policy network and update it during the training instead of focusing on Q value. Also because we focus on the policy network and rewards, we don't need to discretize the continuous environment. Monte Carlo means we sample traces and policy gradient means we calculate gradient to optimize the policy network. The algorithm we implemented can be described as the following five steps:

1. policy network: we built a four-layer neural network using TensorFlow. As shown in Table 1, we used two dense layers with 128 units and put one dropout layer between them to prevent overfitting. The input is the observation of current state. In this case, it's an array of four elements: position, velocity, angle, and angular velocity. The output of the network is the probability of taking action 0 or 1, which controls the cartpole to move left or right. We also used the tensorflow_probability package to sample an action given the probability and calculate the log probability. For the initialization of network parameters, we used the default initializer of each layer.

Table 1: Policy network built by tensorflow

| layer(type) | Output(shape) | Param number | activation |
|---|---|---|---|
| dense | (none, 128) | 640 | relu |
| dropout | (none, 128) | 0 | |
| dense | (none, 128) | 16512 | relu |
| dense | (none, 2) | 258 | soflmax |
| Total params: 17,410 | Trainable params: 17,410 | | |

2. Monte Carlo trace: for each episode/game, we reset the environment and input the observation of current state to the policy network; sample an action using the output of the network; take this action and obtain a reward and observation; if the game is not over, put the new observation into the policy network and take an action. We keep records of observations(s), actions(a) and rewards(r) during the game. That is the trace $h_t = s_t, a_t, r_t, s_{t+1}, ..., a_{t+n}, r_{t+n}$. In the code, we can choose to sample a single trace or several traces before we update the policy network.

3. Discount reward and policy gradient: after we sample an trace, we can calculate the total reward using the discount factor $\gamma$: $R(h_t) = \sum_{i=0}^{n} \gamma^i r_{t+i}$. The policy network is paramterized by $\theta$ and the policy is $p_\theta(a/s)$. We want to maximize the expected return from the start state: $J(\theta) = \mathbb{E}_{h_0 \sim p_\theta(h_0)}[R(h_0)]$. In the code, we have two ways to calculate the reward after each action. The first one is directly using the reward returned from the game and it is 1 as

long as the game is not over. Because during the training, we observed that the cart moved slowly to the edge and still gave a very good reward, which is not the best state we want. So we also use another way to take the position and angle into account. Because the game is done if the position or angle exceeds a max deviation, the best state is the cart is in the center and the pole is right upward. We calculate a continuous reward from 0 to 1 using equation 1.

4. Policy gradient and loss function: We used gradient method to maximize $J(\theta)$. The gradient is given by [5]:

$$\nabla_\theta J(\theta) = \mathbb{E}_{h_0 \sim p_\theta(h_0)} \left[ \sum_{t=0}^{n} \nabla_\theta \log \pi_\theta \left( a_t \mid s_t \right) \cdot R \left( h_t \right) \right] \tag{2}$$

But in practice, we can use the automatic optimizer from tensorflow, so we need a loss function. The loss function is given by [5]:

$$L(\theta) = -\frac{1}{M} \sum_{i=1}^{M} \left[ \sum_{t=0}^{n} R \left( h_t^i \right) \log \pi_\theta \left( a_t \mid s_t \right) \right] \tag{3}$$

The minus sign transferred the maximizing problem into minimizing problem. M is the number of traces we sampled before updating the network (We usually only run a single trace and update the network, so M is usually 1). After defining the loss function, we used the Adam optimizer to differentiate it and update the parameters of the policy network.

5. Repeat step 2,3 and 4 until it converges or after a very large number of iterations.

## 4.2 Experiments and Results

For MCPG, we did experiments to explore the influence of learning rate of Adam (lr), discount factor (gamma), the number of traces M per episode, and normalization of discount rewards, which we will explain in each subsection. Since there are already many parameters we could tune, we fixed the architecture of the policy network. For the following content, one episode means updating the policy network once. The cumulative reward in the following figures means the number of steps it can achieve in each game (the maximum is 500 for cartpole v1).

### 4.2.1 learning rate and discount factor

For this experiment, we fixed M=1, so we updated the policy network after each game/episode. We used the second method to calculate reward from 0 to 1 and didn't use any discount rewards normalization. We chose two learning rate:0.0003 and 0.001; three discount factor: 0.9, 0.95, 0.99. Figure 13 shows the training curves of the six experiments. For all the six experiments, our MCPG showed its learning ability: it can learn from the environment. After about 1000 episodes, it can reach the max cumulative rewards sometimes. But MCPG is

unstable and has lots of fluctuations during training : it can stay in 500 for a while, but jumps down suddenly. Indeed, we also tried to run the experiment for a longer time (10000 episodes), it still could not reach a perfect convergence, and jumped up and down. Actually, instability is very common in MCPG. Sometimes, we could not obtain the same training curve by just redoing the same experiment. But the total trend is similar. As for the learning rate, we cannot tell which one is better, since all the curves have big fluctuations. When lr=0.001, the training curve can stay in 500 for while; when lr=0.0003, the training curve jumps more frequently, but the average seems better. We cannot tell which discount factor (gamma) is better either. However, we still can learn something from these experiments: our MCPG is not very sensitive to learning rate from 0.0003 to 0.001 and discount factor above 0.9. It is safe to choose values between these ranges and it can give us a good result.(We also did experiments when lr=0.01, the result is much worse.)

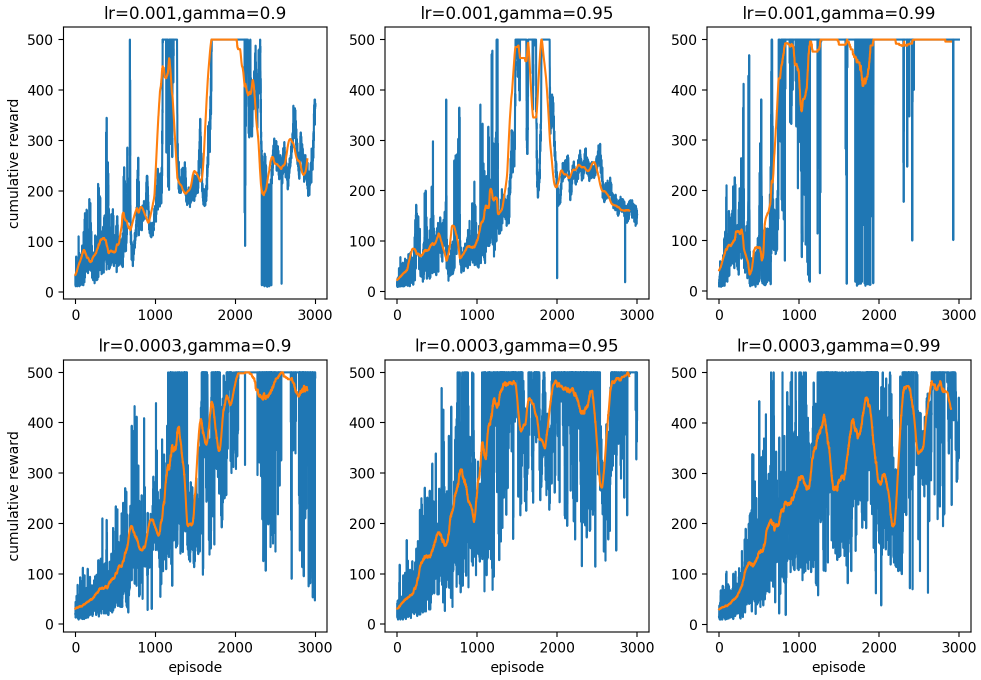



Figure 13: Training curves of the six MCPG experiments of learning rate(lr) and discount factor (gamma). The y axis is represents the cumulative reward and the maximum is 500. Blue lines are the original training curves and orange lines are moving average of the original training curves over 100 episodes. They show the learning ability and instability of MCPG. MCPG is not very sensitive to the range of lr and gamma we chose.

### 4.2.2 The behaviour of loss

One interesting thing we found that is the behaviour of loss during training. Figure 14 shows the training curve of cumulative reward and loss for the experiment of lr=0.001 and

gamma=0.9. In general, loss keeps a similar trend as the cumulative reward: increases with cumulative reward and decreases when cumulative reward jumps down. This is against our instinct, since we used Adam to minimize the loss. From our understanding, we should look at the equation 3 of how we calculated the loss. There are two parts: discount reward R and minus log probability of the action. When R is large and the probability of taking that action is also large, while the minus log probability is small, it is the state we want. But the two parts have opposite effects to loss. When R increases/decreases (it's our goal to maximize reward), it makes loss also increase/decrease, that's why we see the similar trend between cumulative reward and loss. After about 1000 episode, the cumulative reward jumps around 500 and the loss begin to decrease. We think in that part, MCPG begins to know the maximum of reward is 500 and learn to increase its probability, which corresponds to the decrease of minus log probability. So loss began to decrease. But because MCPG is unstable and rewards jump up and down, which have a relative big influence on the loss calculation, the loss still has a big relation to the reward. We have to say loss is not a performance indicator of our MCPG.

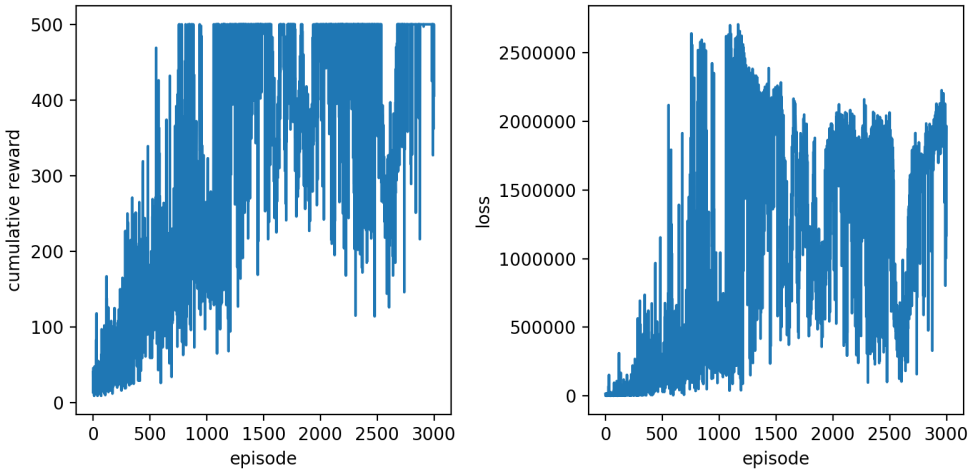



Figure 14: Training curve of cumulative reward and loss for the experiment of lr=0.001 and gamma=0.9. The Loss curve has a similar trend as the cumulative curve, but has a decreasing trend when the cumulative reward keeps at 500.

### 4.2.3 The number of traces M

During each episode, we can choose the number of traces (M) to sample and calculate the average loss of them. Figure 13 shows the training curve of M=5, lr=0.001, gamma=0.9, which sampled five traces and calculate the average loss of them. Compared with the previous results of M=1, the results of M=5 is much worse. The average cumulative reward is about 150.

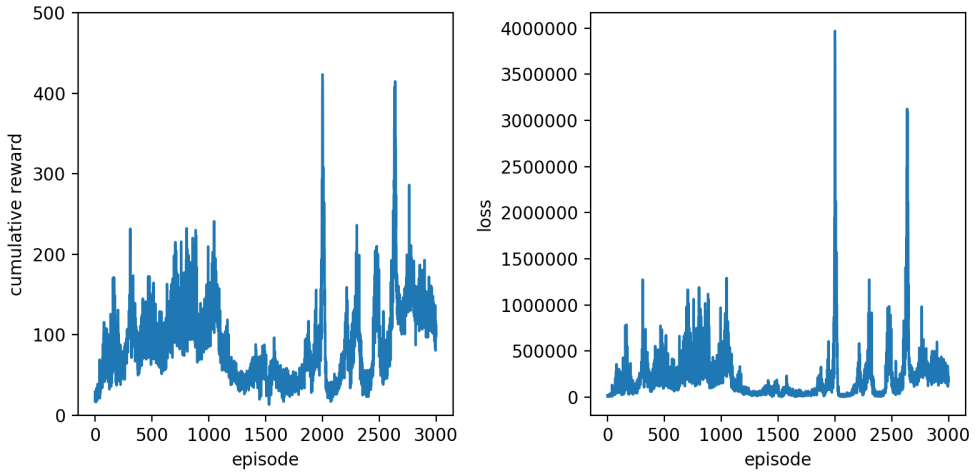



Figure 15: Training curve of M=5, lr=0.001, gamma=0.9. Sampling five traces per episode is worse than sampling one only trace.

### 4.2.4 Two ways of rewards calculating

As we mentioned in the step 3 in section 4.1 algorithm description, we have two choices to calculate reward of each step: one is to use 1 returned by the gym; another one is to calculate reward from 0 to 1 using cart location and pole angle. In previous experiments, we used the second way to calculate reward. Here we did an experiment to see the result of directly using the reward of 1 regardless of location and angle. Figure 16 shows the result of this experiment when lr=0.0003 and gamma=0.95. There is no much differencce between the two methods of calculating rewards in MCPG.

### 4.2.5 Normalization of discount rewards

Because we found that our MCPG is unstable and has big fluctuations, we tried to look for some normalization method to make it stable. In the blog [6], they mentioned policy gradient method has high variance in the gradients, and thus is unstable during learning. We adopted the simplest way they discussed to normalize the discount rewards by subtracting the mean value and dividing the standard deviation of them. Figure 17 shows the normalization result, where lr=0.0003, gamma=0.95 and M=1. We can see that this method does not work for our algorithm. When we added the normalization procedure, our MCPG could not learn.

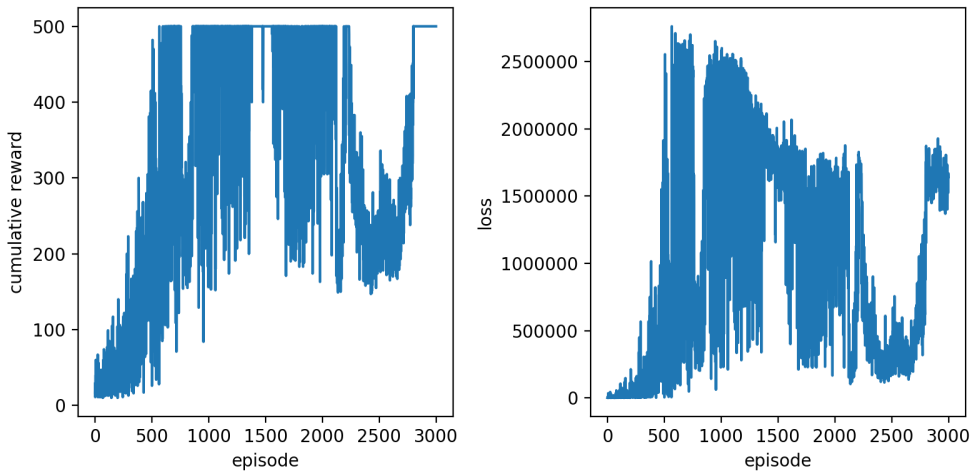



Figure 16: Training curve of using reward returned directly from the gym(1). For this experiment, learning rate is 0.0003 and discount factor is 0.95, M=1. It has no much difference between the two methods of calculating rewards in MCPG.

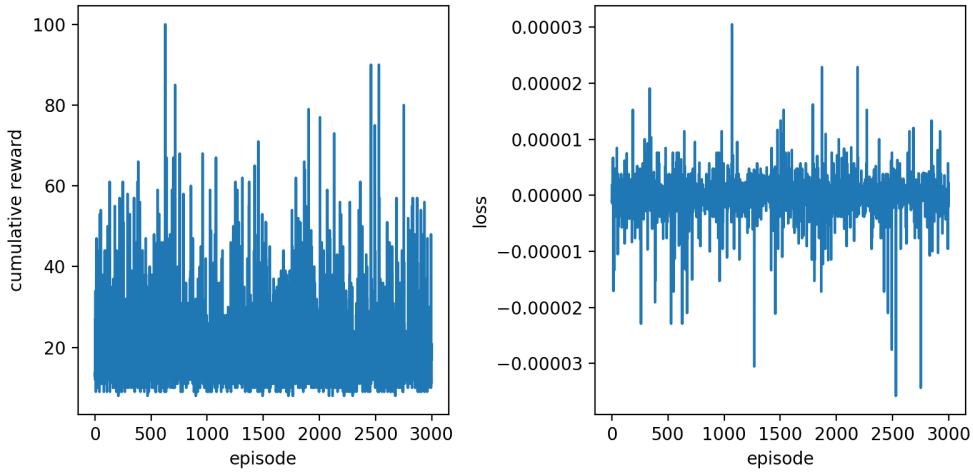



Figure 17: Training curve of normalizing discount rewards. Learning rate is 0.0003, discount factor is 0.95 and the number of trace is 1 per episode. Be aware that the cumulative reward is still the duration of each game, and thus maximum is 500. The normalization method does not work.

# 5 Conclusion and Discussion

1. DQN can be thought of as a combination of Q-Learning and some DNN. These DNNs are used to fit the samples found by Q-Learning. When the task becomes complex, the search space is extremely large, and the time and storage overhead associated with Q-Learning become unacceptable.

   At this time, the neural network is introduced, with the help of its generalization ability, to see the small knowledge, reduce the search space, and improve the space efficiency of storage.

2. Compared with tabular and DQN, MCPG's training stability is worse. The training curves have larger fluctuations. Especially compared with DQN, MCPG cannot stay at 500 for a long time, while DQN's training curve is very flat. Also, the learning speed of MCPG is slower than DQN. MCPG needs about 1000 episodes to reach the maximum cumulative reward and stay on it for a short period of time, while DQN can almost converge after about 150 episodes, dozens of episodes after the collection of the training set. Though both of them use neural networks, DQN parameterizes the Q value, while MCPG parameterizes the policy network.

3. As is mentioned in the previous sections, a much better performance can be achieved when we use the modified continuous reward function. If we only use the default discrete reward function of the environment, how to achieve higher performance is a problem worthy of time to explore.

# References


[1] Wikipedia. $https://en.wikipedia.org/wiki/reinforcement\_learning$, April 2, 2021.

[2] Wikipedia. $https://en.wikipedia.org/wiki/q-learning$, March 17, 2021.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[4] Adam Paszke. Reinforcement learning (dqn) tutorial, 2017.

[5] Thomas Moerland. Lecture notes: Continuous markov decision process and policy search, 2020.

[6] Fork Tree. https://medium.com/@fork.tree.ai/understanding-baseline-techniques-for-reinforce-53a1e2279b57, Oct 17, 2019.