Assignment Four Report

Hang Zheng 520021911347

April 19, 2023

Contents

1 Introduction

In this assignment, I implemented a program that realizes DQN and a variant of DQN called Double DQN, in order to solve the MountainCar problem, a classical RL control environment provided by OpenAI.

2 Experimental Principle

In this section, I will give a brief introduction to the principle of the two algorithms we mentioned above, DQN and Double DQN.

2.1 DQN

Traditional Q-learning reinforcement learning algorithms store the values of all state-action pairs in a table for state evaluation. However, this algorithm encounters difficulties when dealing with high-dimensional or continuous state spaces.

DQN, short for Deep Q-Network, is a **deep learning-based** reinforcement learning algorithm. Its background lies in addressing the limitations of traditional reinforcement learning algorithms in handling high-dimensional state and action spaces, as well as the difficulties of traditional Q-learning algorithms in dealing with continuous state and action spaces.

The core of the DQN algorithm is to **use a deep neural network to approximate the Q-value function.** Unlike traditional reinforcement learning algorithms that need to maintain a Q-value table, the DQN algorithm uses a neural network to learn the Q-value function, with the current state as input and outputting the Q-value for each corresponding action. This approach can handle problems in high-dimensional state and action spaces, as well as continuous state and action spaces.

The training objective of the DQN network is to minimize the following loss function:

$$L(\theta) = E_{(s,a,r,s')\sim U(D)}[(y - Q(s,a;\theta))^2]$$

where s represents the current state, $a$represents the action to be taken, r represents the reward, s' represents the next state, $U(D)$ represents the samples in the replay buffer, $y = r+\gamma \max_{a'} Q(s',a';\theta^-)$ represents the expected Q-value, θ represents the parameters of the neural network, θ^- represents the parameters of the target neural network, γ represents the discounting factor.

When training a DQN network, two tricks, namely Experience Replay and Target Network, are usually used to improve the stability of the algorithm and speed up the convergence. In the following two sections 2.1 and 2.1.1 I will give a brief introduction of these two methods.

The pseudo code of the DQN algorithm is as shown in figure 1.

Algorithm 1: Deep Q-learning Algorithm with experience replay

Initialize replay memory D to capacity N;
Initialize action-value function Q with random weights θ;
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$;
for *episode = 1, M* **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$;
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$;
    Execute action a_t in emulator and observe reward r_t and image x_{t+1};
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess ϕ_{t+1} in D;
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{Otherwise} \end{cases}$;
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D;
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ;
    Every C steps reset $\hat{Q} = Q$;
  end
end

Figure 1: Pseudo code of DQN Algorithm

2.1.1 Experience Replay

Experience Replay is an important technique in the DQN algorithm that enhances the training efficiency and stability. Experience Replay stores the agent's experience in the environment (usually stored as a tuple of (state, action, reward, next_state)) and randomly samples from it to train the neural network, thus breaking the correlation and temporal structure of the data and reducing variance during training.

Specifically, Experience Replay includes the following steps:

1. Storing experiences: At each time step, the tuples of experience in the agent-environment interaction are stored in a replay buffer. The buffer is a fixed-size circular buffer, and when it is full, the earliest experience is overwritten.

2. Sampling experiences: During neural network training, a batch of experiences is randomly sampled from the buffer. These experiences are independent of each other, reducing correlation between samples and improving training efficiency.

3. Training the network: The neural network is trained using the sampled experience tuples, updating the network parameters by minimizing the mean squared error of the Q-value function. During network updating, the DQN algorithm uses two networks, the target network and the current network, to reduce jitter during the updating process.

In a word, Experience Replay has the following advantages:

- Enhancing training efficiency and stability by reducing the correlation between samples
- Decreasing variance
- Allowing for better use of experiences
- Preventing over-fitting and over-remembering
- Utilize experiences better to avoid losing useful information during training

2.1.2 Target Network

The Target Network is an important technique in the DQN algorithm that improves the stability and convergence speed of the algorithm. The core idea of the Target Network is to use a fixed neural network with the same structure as the current network but with fixed parameters to calculate the target value of the Q-value, thereby reducing jitter during the update process.

Specifically, the Target Network includes the following steps:

1. Regularly updating the Target Network: In the DQN algorithm, the parameters of the Target Network are fixed and periodically copied from the current network. The interval at which the parameters are copied is called the update interval, which can be adjusted according to the specific situation.

2. Calculating the target Q-value: During neural network training, the target value of the Q-value is calculated using the Target Network, rather than the estimated value of the Q-value calculated using the current network. This reduces jitter during the update process and improves the stability of the algorithm.

3. Updating network parameters: The parameters of the current network are updated using the difference between the calculated target Q-value and the estimated Q-value of the current network. During network parameter updating, the DQN algorithm uses techniques such as experience replay and gradient clipping to further enhance the stability and convergence speed of the algorithm.

In a word, Target Network has the following advantages:

- Reducing jitter during the update process

- Improve the stability and convergence speed of the algorithm
- Reduce the number of update iterations, improve the training efficiency of the algorithm

However, the disadvantage of the Target Network is that it may **lead to delayed updates** in the algorithm because the parameters of the Target Network are fixed, while the Q-values in the actual environment are constantly changing. Therefore, the update interval needs to be set according to the requirements of the specific problem to balance the stability and convergence speed of the algorithm.

2.2 Double DQN

Double Deep Q-Network (Double DQN) is a reinforcement learning algorithm that is an improved version based on the Deep Q-Network (DQN) to solve the over-estimation problem in DQN algorithms.

2.2.1 Over-estimation in DQN

When choosing an action, the DQN network uses greedy policy to select the action with a maximum Q-value and the target Q-value is calculated based on this selected action. Though using such a greedy policy can help the network to converge towards the potential optimal target Q-values, it will also lead to over-estimation on the Q-values for that each time the model estimate a Q-value, it may over-estimate it a little bit and after thousands of iterations, the gap between the ground-truth Q-values and the estimated Q-values will grow wider and wider. Besides, the instability and noise of the neural network may lead to overestimation of the target Q value.

In a word, there are two main reasons for overestimation:

1. The estimation of the target Q value is based on the maximum Q value under the current policy, rather than the maximum Q value under the optimal policy, which may lead to overestimation
2. The instability and noise of the neural network may lead to error accumulation and amplification, resulting in overestimation

2.2.2 Double DQN

The Double DQN algorithm is an improved version of the Deep Q-Network (DQN) algorithm designed to address the estimation bias problem in DQN. The core idea of Double DQN is to use two neural networks to estimate the Q-values of each action in the current state and the Q-value after selecting the optimal action, respectively, in order to reduce estimation bias and improve learning performance.

Specifically, the Double DQN algorithm includes two neural networks: one called the main network, which is used to select the optimal action, and the other called the target network, which is used to estimate the Q-value of the optimal action. The parameters of the main network are copied to the target network every certain time to make the estimation more accurate.

The pseudo code of the DQN algorithm is as shown in figure 2.

2.3 Dueling DQN

The Dueling Deep Q-Network (Dueling DQN) algorithm is an improvement over the Deep Q-Network (DQN) algorithm, which was proposed by DeepMind to address the instability and high variance of Q-learning in reinforcement learning.

The core idea of the Dueling DQN algorithm is to decompose the Q-value function into two parts, a state-value function and an advantage function.

- The state-value function represents the expected return in that state
- The advantage function represents the additional return that can be obtained by taking action a compared to other actions.

In this way, the algorithm can better estimate the Q-value of each action without increasing the computational cost, thereby improving learning efficiency and stability.

In the Dueling DQN algorithm, the network's output is divided into two parts: the state-value function $V(s)$ and the advantage function $A(s, a)$. Then, the Q-value of each action is calculated by combining them:

Algorithm 1: Double DQN Algorithm.

input : $\mathcal{D}$ – empty replay buffer; θ – initial network parameters, θ^- – copy of θ
input : N_r – replay buffer maximum size; N_b – training batch size; N^- – target network replacement freq.
for *episode* $e \in \{1, 2, \ldots, M\}$ **do**
  Initialize frame sequence $\mathbf{x} \leftarrow ()$
  for $t \in \{0, 1, \ldots\}$ **do**
    Set state $s \leftarrow \mathbf{x}$, sample action $a \sim \pi_{\mathcal{B}}$
    Sample next frame x^t from environment $\mathcal{E}$ given (s, a) and receive reward r, and append x^t to $\mathbf{x}$
    if $|\mathbf{x}| > N_f$ **then** delete oldest frame $x_{t_{min}}$ from $\mathbf{x}$ **end**
    Set $s' \leftarrow \mathbf{x}$, and add transition tuple (s, a, r, s') to $\mathcal{D}$,
      replacing the oldest tuple if $|\mathcal{D}| \geq N_r$
    Sample a minibatch of N_b tuples $(s, a, r, s') \sim \text{Unif}(\mathcal{D})$
    Construct target values, one for each of the N_b tuples:
    Define $a^{\max}(s'; \theta) = \arg\max_{a'} Q(s', a'; \theta)$
    $y_j = \begin{cases} r & \text{if } s' \text{ is terminal} \\ r + \gamma Q(s', a^{\max}(s'; \theta); \theta^-), & \text{otherwise.} \end{cases}$
    Do a gradient descent step with loss $\|y_j - Q(s, a; \theta)\|^2$
    Replace target parameters $\theta^- \leftarrow \theta$ every N^- steps
  end
end

Figure 2: Pseudo code of Double DQN Algorithm

$$Q(s,a) = V(s) + A(s,a) - 1/|A| \cdot \sum A(s,a')$$

where $|A|$ represents the number of available actions, and $sum(A(s,a'))$ represents the average of the advantage function for all other actions in state s.

The advantages of the Dueling DQN algorithm include:

1. Improving the instability and high variance of Q-learning, which enhances the learning efficiency and convergence speed.
2. By decomposing the state-value function and the advantage function, it can better estimate the Q-value of each action, thereby improving the accuracy and stability of learning.
3. It can be combined with other deep reinforcement learning algorithms such as Double DQN, Prioritized Experience Replay, Rainbow, etc., to further improve the performance and robustness of the algorithm.

3 Experimental Content

The directory structure of my code is like:

- main.py: The main file to be run, in which I create an instance of the Agent class I implemented to complete the task of this assignment.
- model.py: I implement a class named Agent to provide the functionality of the two algorithms we mentioned in the last section, a class named ReplayBuffer to serve as the replay buffer in the DQN algorithm and two extra network classes named QNetwork1 and QNetwork2 to serve as neural networks in the model.

Here I will list the methods of each classes and the further details will not be explained here (Please refer to the detailed comments I added in the source code)

3.1 Agent

The methods of Class Agent are shown as below:

- train(episodes,batch_size,lr,alpha): Train the model.
- train_model(self,batch_size,loss_fn,alpha): Called in the train method to train the Q-network to update the parameters.
- test(test_episode): Test the best model obtained in the training process for test_episode episodes.

- getReward(state,reward): Return rewards based on different reward type(I design two more rewards different from the original one)
- epsilon_greedy(action): Conduct epsilon greedy method to provide randomness for action selection
- plot_loss(): Plot the loss curve of the training process.

3.2 ReplayBuffer

The methods of Class ReplayBuffer are showed as below:

- push(state, action, reward, next_state, done): Store a 5-tuple of a transition in the replay buffer.
- sample(batch_size): Randomly sample batch_size records from the buffer for training.

3.3 Networks

There are three neural networks implemented with pytorch.

- The Q_Network1 has only two linear forwarding layers with a hidden dim of 16, connected with a relu activation function, which stands for a simple network.
- The Q_Network2 has three linear forwarding layers with hidden dim of 8 and 16, connected with relu activation functions, which stands for a relatively complicated network.
- The DuelingDQN is the network used in Dueling DQN algorithm, the output of the first forwarding layer is put into two sub-networks: the state-value function $V(s)$ network and the advantage function $A(s, a)$ network. The output of the network is calculate based on the output of the two sub-networks.

3.4 Reward

The original reward given by the environment is straightforward which gives a reward of -1 each step until the agent reaches the terminal state. The original reward contains little information so I designed two more types of rewards tried to improve the performance of the algorithm.

We need to give some details about the environment first. The environment of Mountain-Car is shown as in fig 3. The parameters scale of the environment state is: $x \in [-1.2, 0.6]$, $v \in [0, 0.07]$, and the x coordinate of the terminal state is 0.5 while the x coordinate of the valley state is -0.5.

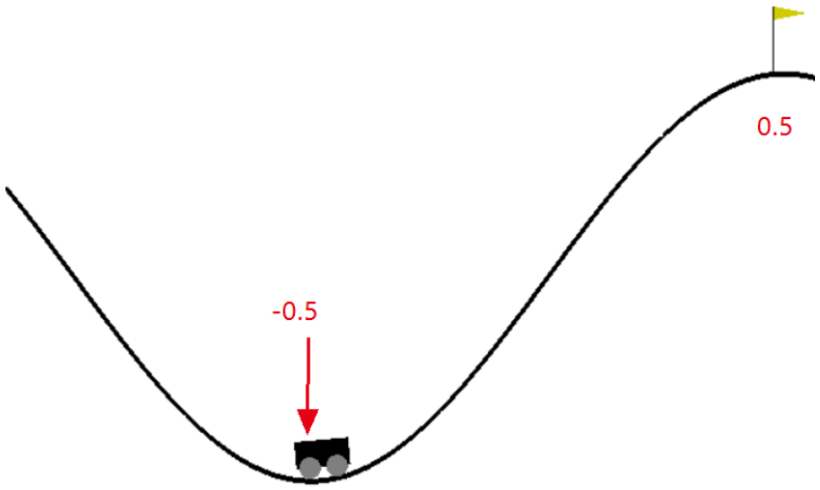


Figure 3: Illustration of Mountain-Car environment

3.4.1 Reward1

The first type of reward that I came up with takes the x coordinate of the state into account. The value of the reward is based on the x coordinate value of the state and the calculating formula is like:

$$Reward(x) = \begin{cases} 5(x+0.5)^2, & \text{if } x <= -0.5 \\ 10(1-(x-0.5)^2), & \text{if } x > -0.5 \end{cases}$$

The reason for this design is:

1. When x is less than -0.5, as the car moves further away from the lowest point, although the distance to the endpoint also increases, the height increases as well. At this time, **the potential energy of the car is relatively high, which is beneficial for subsequent acceleration.** Therefore, an appropriate reward is given for the distance away from the lowest point.

2. When x is greater than -0.5, the closer the car is to the endpoint (x=0.5), the greater the reward should be given, encouraging the car to move towards the endpoint.

3.4.2 Reward2

Unlike the Reward1, the second type of reward that I designed takes both the x value and the velocity of the state into account, for that the velocity of the car is also of greater importance in the car's climbing process. The value of the reward is based on the x coordinate value and the velocity value of the state, and the calculating formula is like:

$$Reward(x) = \begin{cases} 5(x+0.5)^2 + 100(|v|), & \text{if } x <= -0.5 \\ 100 - 100v, & \text{if } x >= 0.5 \\ (100-(10(x-0.5))^2) + (100v)^2 & \text{otherwise} \end{cases}$$

The reason for this design is:

1. When x is less than -0.5, compared with Reward1, I further add a velocity term in the formula to encourage a velocity with greater absolute value At this time, **the kinetic energy of the car is relatively higher with a velocity with greater absolute value, which is beneficial for subsequent acceleration.** Therefore, an appropriate reward is given for the distance away from the lowest point and a velocity with greater absolute value.

2. When x is greater than 0.5, the car is very close to the terminal point so I always give a great reward to such a state. The velocity term here is to punish a velocity with a wrong direction which leads the car goes right and moves further away from the terminal state.

3. When x is less than 0.5 and greater than -0.5, I give a greater reward to a state with x value closer to 0.5(the terminal state) and a greater velocity value.

3.5 Program Running

To run the codes, you only need to type in the command like:

python main.py

and the program will run and save the running result in a file.

Further more, the program are implemented to takes several arguments and you are able to tune the hyper-parameters flexibly.

- '-i','–iteration': the minimum iterations to run the two methods.
- '-m','–model_type': argument to indicate whether using DQN, Double DQN or Dueling DQN.
- '-b','–buffer_capacity': the capacity of the replay buffer.
- '-e','–epsilon': the epsilon in ϵ-greedy exploration.
- '-o','–output_file': the output file path to save the results.

- '-a','–alpha': the step size in the two algorithms when updating the Q-values.
- '–lr': learning rate when training the neural network.
- '-s','–step': step size in Fixed Target Network method, the step gap to update the target model with the trained model's parameters.
- '-r','–reward': argument to indicates the reward type, should be in [0,1,2].
- '-n','–network': argument to indicates the network type, should be in [1,2,3], respectively represents Q_Network1, Q_Network2 and DuelingDQN.

4 Experimental Result

In this section, I conducted several experiments and all the models are tested for 5 episodes on the Mountain-Car environment.

4.1 DQN and Double-DQN

In the experiment, I compare the performance of DQN and Double-DQN.

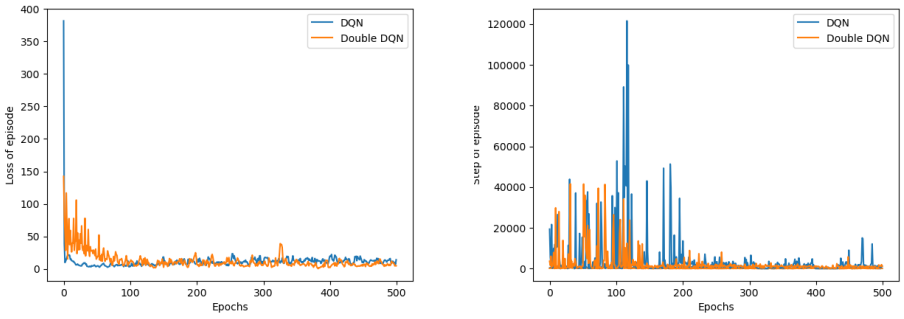
The arguments are set as: $buffer\ capacity$: 1000, $step$: 300, $iteration$: 500, $batch\ size$: 128, $alpha$: 0.9, $reward\_type$: 0

I run the two algorithms and the result are as shown in Table 1

Algorithm	Training time(s)	Testing Steps
DQN	1226.909667491913	139.0
Double DQN	1263.342220067978	147.4

Table 1: DQN and Double DQN Performance

More intuitive, I plot the loss curve and the step curve of each episodes for the two algorithms and here I take reward0 as examples, which are shown in Fig 4



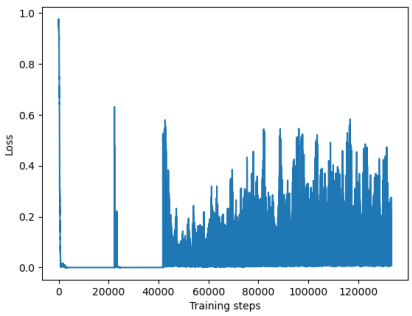
(a) Loss curve

(b) Step curve

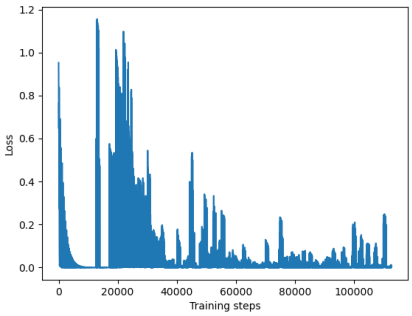
Figure 4: Loss curve and Step curve of DQN and Double DQN using reward0

From the result, we can see that:

1. After being trained for 500 episodes, both DQN and Double DQN converged and the car guided by the agent is able to reach the hill successfully within a little amount of steps in a short time.
2. According to the curves, the convergence speed of Double DQN to approximate convergence is slower than that of DQN(indicated from the loss curve), but its stability after convergence is significantly stronger than that of DQN(for that the DQN agent would sometimes take more steps than usual to reach the terminal even after convergence).



(a) Loss curve in DQN



(b) Step curve in Double DQN

Figure 5: Loss curve and Step curve of DQN and Double DQN using reward0

Furthermore, I plot the loss curve for each step during the training process, which is shown as Fig 5

Analyzing on the loss curves, we can see that:

1. The loss of both two models doesn't decline smoothly as expected, but fluctuates over a large range.
2. The fluctuations in loss may due to the fact that in same states, the observed data changes under the strategy of experience replay, which affects the input of DQN and DDQN resulting in fluctuations in loss.
3. A sudden increase in cost regularity can be caused by updates to the target network.
4. Moreover, the loss of DQN fluctuates greater while the loss of Double DQN becomes much stable in the later stages, which also confirm the fact that Double DQN is more stable and converge better than DQN.

4.2 Experiment on network types

When using DQN or Double DQN, I design two types of Q-network, one of which is simple(2 forwarding layer) and the other one is relatively deeper(3 forwarding layer). As we all know, a deeper network is able to fit more complex functions and bring better performance to the model while as well requiring more training time.

In the experiment, I compare the performance of DQN and Double-DQN using two types of Q-network.

The arguments are set as: $buffer\ capacity$: 1000, $step$: 300, $iteration$: 500, $batch\ size$: 128, $alpha$: 0.9, $reward\_type$: 0

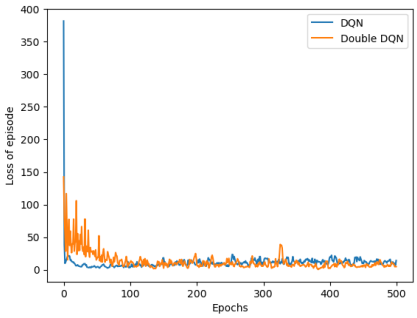
The result are as shown in Table 2

Network type	Algorithm	Training time(s)	Testing Steps
1	DQN	1226.909667491913	139.0
	Double DQN	1263.342220067978	147.4
2	DQN	291.5291895866394	128.4
	Double DQN	797.3040602207184	134.2

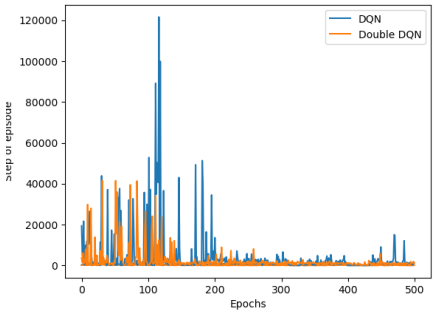
Table 2: DQN and Double DQN Performance with different networks

More intuitive, I plot the loss curve and the step curve of each episodes for the two algorithms and here I take reward0 as examples, which are shown in Fig 6

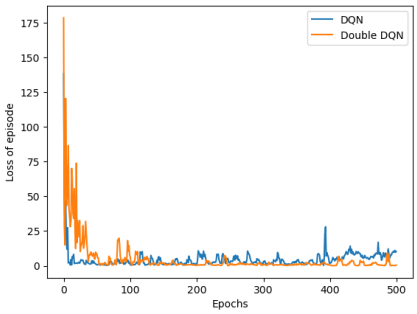
From the result, we can see that:



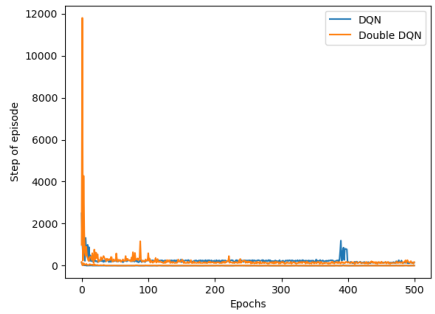
(a) Loss curve with Network1



(b) Step curve with Network1



(c) Loss curve with Network2



(d) Step curve with Network2

Figure 6: Loss curve and Step curve of DQN and Double DQN using reward0

1. The average testing accuracy of DQN and Double DQN using a deeper network(network2) is higher than the ones that using a simpler network(network1), which validates that a deeper network really help with the DQN and Double DQN agent.

2. The training time of DQN and Double DQN using a deeper network(network2) is significantly less than that with a simpler network(network1), and the reason for this is that a deeper network can better fit the task and **converges faster in the early episodes than a simpler network**, which saves a lot of time exploring in the wrong direction.

3. The Double DQN algorithm is still more stable than the DQN algorithm using a deeper network.

4.3 Experiment on reward types

In the experiment, I run the two algorithms with different setting of reward types and the result are as shown in Table 3

The arguments are set as: $buffer\ capacity$: 1000, $step$: 300, $iteration$: 500, $batch\ size$: 128, $alpha$: 0.9, $network\_type$: 1

Reward type	Algorithm	Training time(s)	Testing Steps
0	DQN	1226.909667491913	139.0
	Double DQN	1263.342220067978	147.4
1	DQN	19391.556564331055	191.0
	Double DQN	5988.864568471909	220.0
2	DQN	370.44224214553833	140.4
	Double DQN	835.775927066803	152.2

Table 3: DQN and Double DQN Performance with different reward types

From the result, we can see that:

1. Reward1 did not provide any help for the task and led to increased difficulty in model training and decreased test performance. Reward2 provided some help for the task, although there was a slight decrease in test performance, the training time of the model was significantly reduced.

2. Reward1 is a reward that only considers the position coordinate (x value) of the car. Compared to the original reward provided by the environment, it provides more information to the agent and ideally should bring some performance improvement. However, the reason for the actual performance decline may be that **the design of the reward is not reasonable enough**. For example, it provides significant rewards for the area to the left of x=-0.5, which may mislead the car to move to the left of that point, away from the endpoint, ultimately leading to a decrease in performance.

3. Reward2 is a reward that considers both the position coordinate (x value) and velocity coordinate (v value) of the car. At this point, **the composition of the reward is more reasonable**, guiding the behavior of the car more reasonably and promoting the convergence rate of the model. The slight decrease in test performance may be due to the randomness of the test (only 5 episodes were conducted), and the model training in the environment itself also has some randomness (the ϵ greedy algorithm), so it is uncertain whether the use of Reward2 will result in a decrease in performance.

4.4 Dueling DQN

In this section, I conducted an experiment using the Dueling DQN I implemented to solve the MountainCar problem with different setting of learning rate and reward types. The result are as shown in Table 4

lr	Reward Type	Training time(s)	Testing Steps
1e-2	0	(didn't converge)	-
	1	(didn't converge)	-
	2	9762.501041173935	135.0
5e-3	0	(didn't converge)	-
	1	13870.960108041763	115.8
	2	4803.310376882553	120.4
1e-3	0	(didn't converge)	-
	1	9390.200968503952	116.6
	2	3838.691004753113	132.4

Table 4: Dueling DQN Performance with different reward types and learning rate

More intuitive, I plot the loss curve and the step curve of each episodes for the Dueling DQN algorithms using Reward1 and Reward2, which are shown in Fig 7

From the result, we can see that:

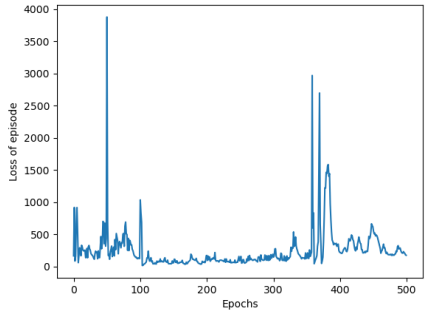
1. The performance of Dueling DQN on this task is generally inferior to that of DQN and Double DQN. When using the original reward (Reward0), the training speed is so slow that **the model cannot even converge**.

2. After convergence, the Dueling DQN model is very stable (indicated from the Fig 7).

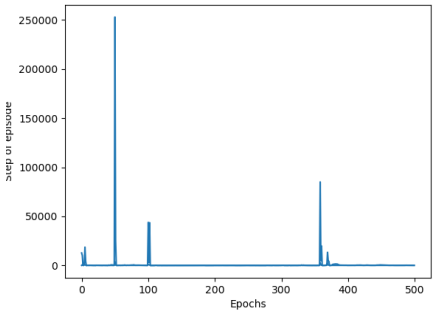
3. The choice of learning rate is also crucial for the model training and convergence. A learning rate that is too large can also prevent the model from converging.

4. **The effectiveness of the self-designed Reward1 and Reward2 has been validated** in this experiment, especially the promoting effect of Reward2 on the model convergence.

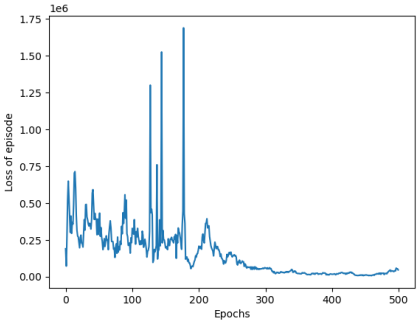
5. The possible reason for the poor performance of Dueling DQN algorithm may be **due to improper hyper-parameter settings**. The network structure (number of layers and hidden layer dimensions), learning rate, discount factor, and other hyper-parameters have a significant impact



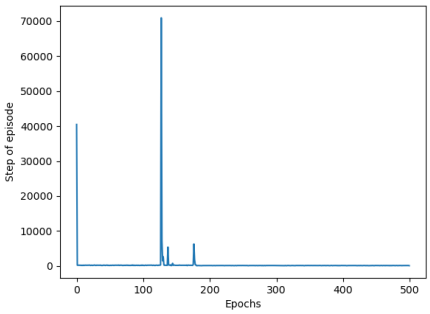
(a) Loss curve with Reward1



(b) Step curve with Reward1



(c) Loss curve with Reward2



(d) Step curve with Reward2

Figure 7: Loss curve and Step curve of Dueling DQN using Reward1 and Reward2

on the algorithm's performance. However, in this experiment, there was insufficient exploration of hyper-parameters, and the chosen hyper-parameters may not be optimal.

5 Conclusion

During the process of this experiment, I first systematically learned the relevant knowledge of three kinds of DQN algorithms, the original DQN, Double DQN and the Dueling DQN, on the basis of the course content, and then implemented the three algorithms on my own to better understanding the implementation details.