Reinforcement Learning Assignment 4

Hao Sun 516030910362
Xueyan Li 516030910373

May 2019

1 Introduction

In all the value evaluation methods we have learned before, the value of a pair of state and action is stored in a table. However, when the task becomes more complicated, such as the state space being combinatorial and enormous, it's impossible to store all the value of state-action pairs in a table. Therefore, it's necessary to using a value function to fit the values.

Deep neural network is characterized by the ability to automatically extract complex features, which makes it suitable for fitting complex functions. Therefore, the value under high dimension and continuous state space can be well fitted by deep neural network.

As reinforcement learning tends to be unstable, which means every time the neural network updates, the policy will be changed, directly applying deep neural network to it may leads to divergency. Therefore, deep Q-network (DQN)[1] makes two improvements on the basis of original deep neural network. Firstly, DQN uses experience replay to solve the correlation and non-static distribution problem. Experience replay uses a random sample of prior actions instead of the most recent action to proceed. The sample (s_t, a_t, r_t, s_{t+1}) generated by a step will be stored and randomly chose to train the network. This improvement can remove correlations in the observation sequence and smoothing over changes in the data distribution. Secondly, DQN uses a Q-network to fit the value function and a target network to decide the action. The Q-network will be updated after every steps while the target network updates every C steps. Such a target network fixed the policy when Q-network updates which leads to stability.

In our report, we will implement DQN algorithm and analysis the result of our experiment.

2 Enviroment: Mountain Car

[2] first described this problem, as shown in fig. 1, a car is on a one-dimensional track, positioned between two "mountains". The goal is to drive up the mountain on the right; however, since gravity is stronger than the car's engine, even at full throttle, the car cannot simply accelerate up the steep slope. The car is situated in a valley and must learn to leverage potential energy by driving up the opposite hill before the car is able to make it to the goal at the top of the rightmost hill. In this experiment we use Gym [3] to provides this environment.

The car has two-dimensional continuous state space including $Velocity$ and $Position$. And for each time step, the car has three discrete action space $Action$. The range of these sets are as below.

$$\begin{aligned} Velocity &\in (-0.07, 0.07) \\ Position &\in (-1.2, 0.6) \\ Action &\in \{-1, 0, 1\} \end{aligned}$$

where $-1, 0, 1$ in $Action$ respectively represents $left$, $neural$, and $right$. And the update functions for evert time step of these variables are that

$$Velocity = Velocity + 0.001 \times (Action) + (-0.0025) \times \cos(3 \times Position) \quad (1)$$

$$Position = Position + Velocity \quad (2)$$

At last, the termination condition is that $Position \geq 0.6$. In addition, $Reward$ of each step and Starting condition are often as hyperparameters.

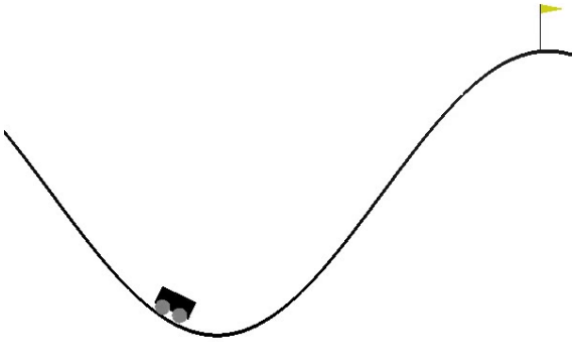


Figure 1: Mountain Car Environment

3 Algorithm

3.1 Deep Q-Learning Network (DQN)

In 2015, [1] proposed a method to combine Q-learning and deep learning, which is DQN. With Q being action value function, Instead of traditional Q-Table, DQN records and predicts Q in neural network, and updates continuously the weights in neural network to learn the best policy. The main reason of proposing DQN is that the states space in almost all environments are no longer discrete, which means the Q-Table would be infinitely large if we still recorded by Q-Table.

The DQN algorithm is shown in algorithm 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

Since the state is abstracted into car's position, convolutional layer is not necessary in our experiment. We build the fully connected neural network on the Tensorflow. The network structure is shown as fig. 2.

3.2 Network Structure

As fig. 2 shows, the neural network has two hidden layer and each hidden layer has ten neurons, which is . We input two features: The position and the velocity of agent (the car), and output the action, which has been abstracted into monitor set $\{-1, 0, 1\}$.

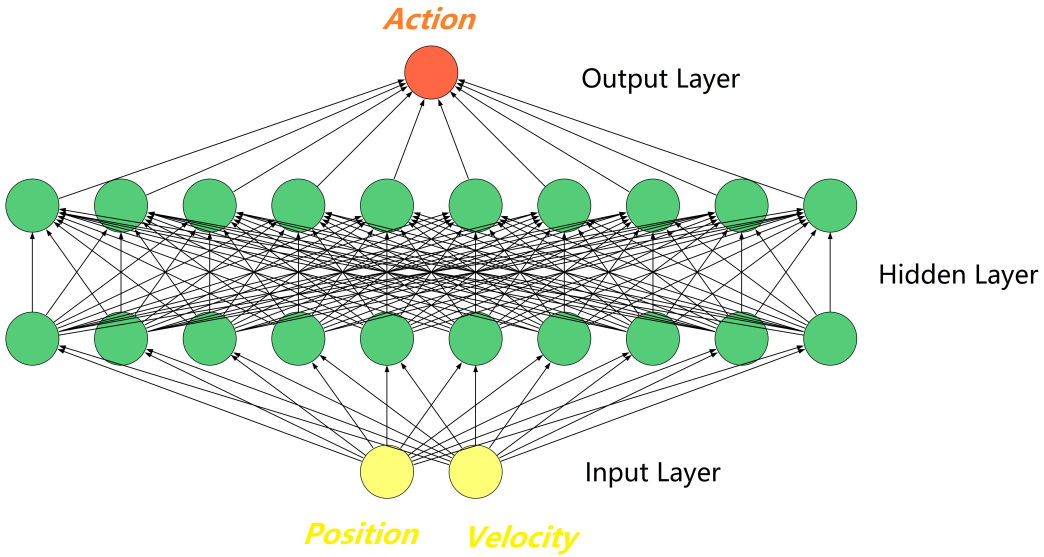


Figure 2: Our neural network structure in this experiment

4 Setting

In the origin DQN paper, the author used 3 convolution layer and 2 fully connected layer. In our experiment, we simplify the neural network according to the complexity of problem. We use a neural network with 2 fully connected layer and 10 neurons in each hidden layer. We set $Reward = |Position - (-0.5)|$ to encourage the car to reach a higher position. As a comparison, we also set another reward like $Reward = -0.5$ for each time step while $Reward = 100$ if car arrives the flag. We set the probability of choosing the greedy strategy $\epsilon = 0.9$. which gets larger to 0.9 with initial value 0 in training process, we set a hyperparameter $\Delta\epsilon$ as the increment step. The reward discount we set is $\gamma = 0.9$ and the learning rate is 0.01. Every time time step counter gets $replace\_target\_iter$, which is set as 300, the counter recounts and save the parameters in training neural networks and reset action-value function Q.

5 Result

We **made a video** to record the training progress of mountain car, which will be also put in the same directory, hoping that you will have a look.

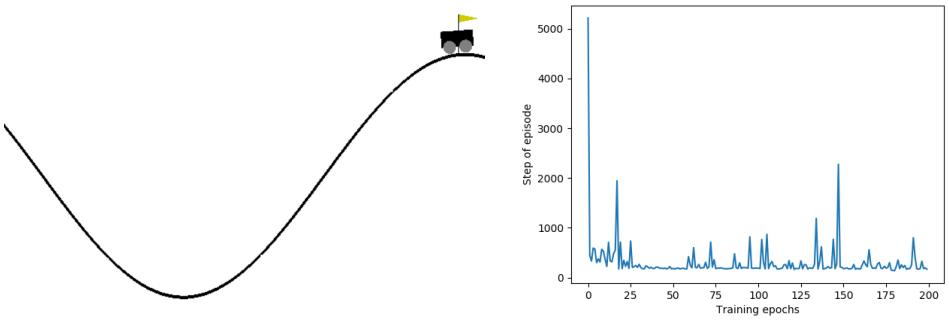
In our experiment, we run 200 episode with the ending state like fig. 3(a). We record the steps of each episode and the cost in learning to evaluate DQN. As shown in fig. 3(b), each point is the steps using in an episode. At the beginning of the training, an episode needs more than 5000 steps to reach the ending state. Subsequently, the number of steps required by an episode fluctuated around 500 times. After about **25** episodes, the number of steps required is maintained at 200-300 with outliers less than half of the episodes at the beginning. Therefore, it proves that DQN can converge to an optimized policy.

What's more, we also observed the change of cost with training steps. As shown in Fig.4, unlike other model, the cost of DQN won't decrease smoothly but fluctuate because in different states, the observed data changes which will influence the input of DQN, leading to a fluctuation in cost.

6 Discussion

From the above experiments, DQN can finally converge to an optimized policy. We changed some parameters in the model to observe its performance.

As fig. 4 shows, we have gotten that the cost of DQN won't decrease smoothly but fluctuate because in different states, the observed data changes which will influence the input of DQN, leading to a fluctuation in cost. The sudden increment of cost is caused by the input and out data



(a) Mountain Car Ending state of one episode (b) Number of steps in each Episode

Figure 3: The result of DQN

changing, which happens when time steps gets the hyperparameter *replace_target_iter*. If we want the cost curve become more smooth, we can set the probability of choosing greedy strategy $\epsilon = 1.0$ after we think the car performs well enough.

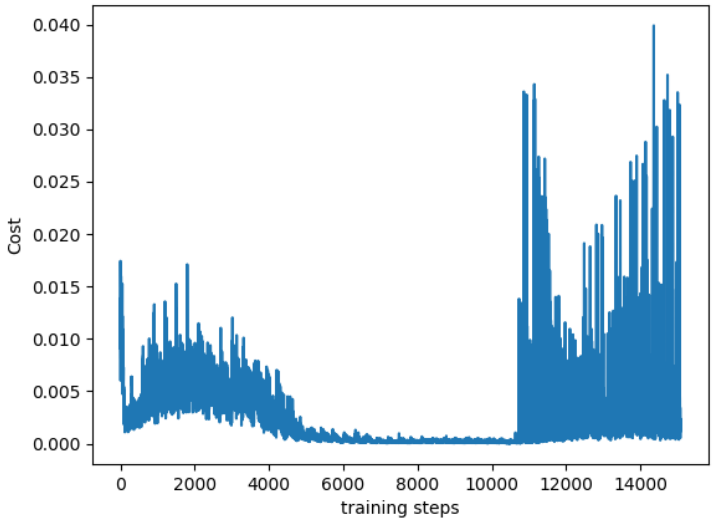


Figure 4: Loss of neural network in the whole training progress

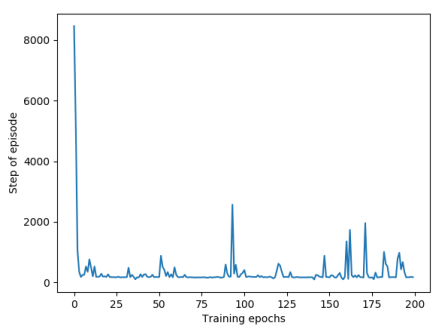
Meanwhile, we wonder some hyperparameters will cause great influence on the result. Therefore we did some experiments with modified hyperparameters to compare the performance.

6.1 Learning Rate

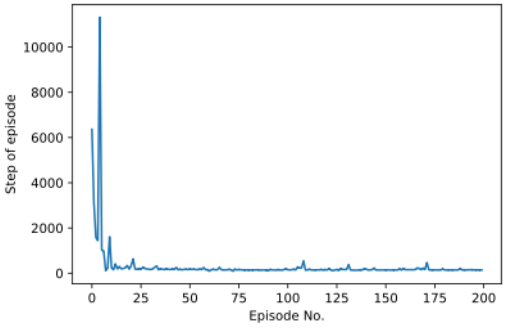
In the above experiment, we set learning rate as 0.01. In this part, we set it to 0.001, of which the result is shown in fig. 5(a). From the result, we can see that with a lower learning rate, the steps an episode need to reach the ending state is much higher, like about 5000 under learning rate of 0.01 but more than 8000 under learning rate of 0.001. However, there's no much difference between the result of two learning rate. And both of them can converge. Therefore, the learning rate only affect the first few episodes.

6.2 Reward Function

In the original experiment, we set $Reward = |Position - (-0.5)|$. This time we set reward function as $Reward = \begin{cases} -1 & \text{Each time step} \\ 100 & \text{If the car reaches the flag} \end{cases}$



(a) Number of Steps in each Episode with the learning rate of 0.001



(b) Number of Steps in each Episode with new reward function

Figure 5: The number of steps in each episode with modified hyperparameters

Then the result is shown as fig. 5(b), and we can see that if we choose this reward, the result is much **more stable** but the converge speed is also much **slower**.

From the above two experiments, we can see that DQN is not much sensitive to hyperparameters, especially in this simple environment. Therefore it is not key about adjusting parameters compared with other deep learning models.

6.3 Limitations of DQN

From the experiments we have conducted, we can sum up these limitations of DQN.

- **Long time to explore in the beginning.** However we change the hyperparameters, the first several episode cost the agent much time to explore (changing reward may perform a little better). This is a simple environment actually, we guess that DQN has a low effeciency problem faced with many difficult problems.
- **Not stable enough.** After the car finds the optimal way to get the flag, chances are that car still spends much time, getting trapped in the local solution. We have tried that changing ϵ, which brings up another problem that the car needs much more time to explore in the beginning.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015. [Online]. Available: http://dx.doi.org/10.1038/nature14236

[2] A. W. Moore, "Efficient memory-based learning for robot control," Tech. Rep., 1990.

[3] http://gym.openai.com/envs/MountainCar-v0/.