AU 332 Artificial Intelligence: Principles and Techniques

By: ZheHao Huang (518021910660)

HW#: 2

October 25, 2020

1 Reinforcement Learning in Maze Environment

1.1 Game Description

Suppose 6 × 6 grid-shaped maze in Figure 1.

- The red rectangle represents the start point.
- The green circle represents the exit point, and finding the exit will reward +1 and terminate current iteration.
- The black rectangles represent the traps, and falling into traps will cause a reward −1 and terminate current iteration.
- The golden diamond represent the treasure, and finding it will get a bonus reward +3.

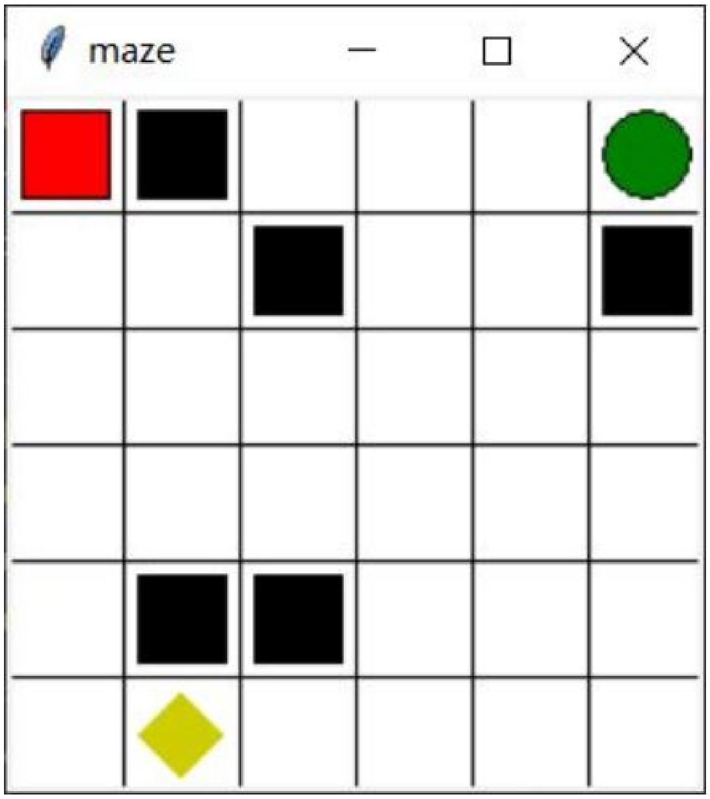


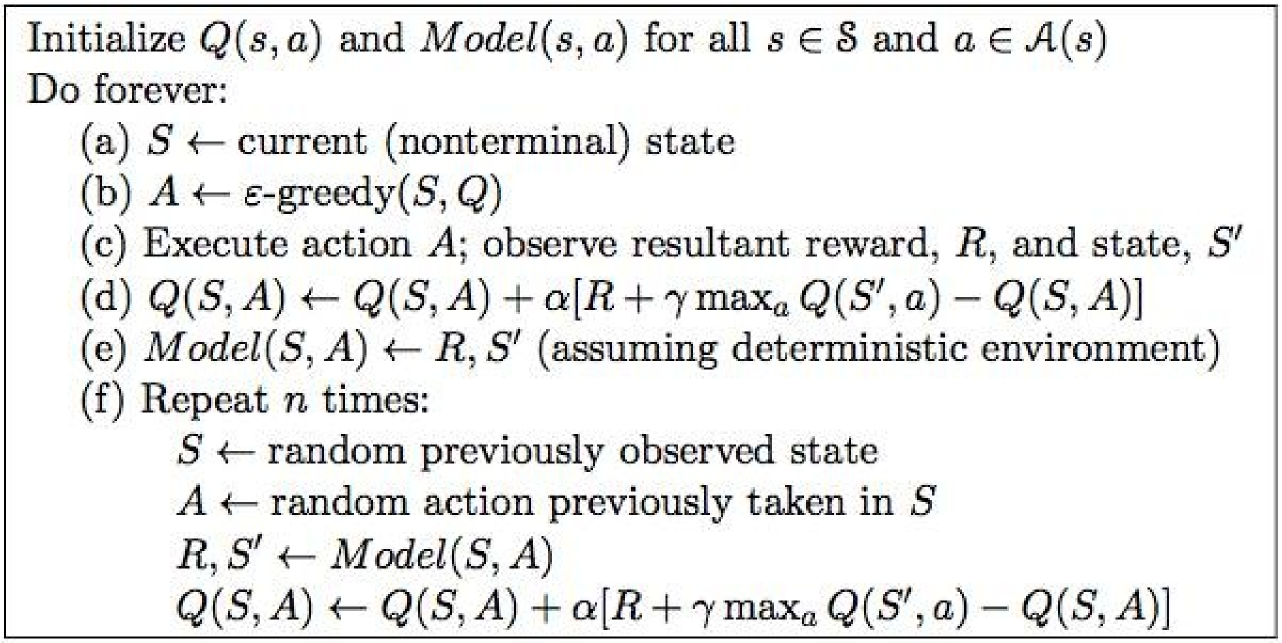
Figure 1: Maze environment

Have four legal actions that we can move upward, downward, leftward and rightward. And the **goal** is to exit after finding the treasure and to avoid falling into the traps.

- **State(5-dimension)**: The current position of the agent(4D) and a bool variable(1D) that indicates whether the treasure has been found.
- **Action(1-dimension)**: A discrete variable and 0, 1, 2, 3 respectively represent move upward, downward, rightward and leftward.

1.2 Dyna-Q Learning and Epsilon Greedy

I use the Dyna-Q Algorithm in Figure 2 to help find a way getting the treasure and finally getting out as quickly as possible.



```
Initialize $Q(s, a)$ and $Model(s, a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Do forever:
    (a) $S \leftarrow$ current (nonterminal) state
    (b) $A \leftarrow \varepsilon$-greedy$(S, Q)$
    (c) Execute action $A$; observe resultant reward, $R$, and state, $S'$
    (d) $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
    (e) $Model(S, A) \leftarrow R, S'$ (assuming deterministic environment)
    (f) Repeat $n$ times:
            $S \leftarrow$ random previously observed state
            $A \leftarrow$ random action previously taken in $S$
            $R, S' \leftarrow Model(S, A)$
            $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
```

Figure 2: The Dyna-Q Algorithm

And in the following I will introduce my algorithm in the order of the Model, the Q-table, and the training process of the agent.

1.2.1 Model

In the Dyna-Q Learning ,the Model plays a role as a memory which stores pairs (**key:**(S, A), **value:**(S', R)), assuming the environment is deterministic, so each pairs of (S, A) and (S', R) is certain and won't be changed after assigned. When we need to retrive a pair of key and value, we just take (S, A) as an input and output (S', R).

- S represents a previously observed state.
- A represents a action previously taken in S.
- R, S' respectively represent the reward and the next state got by the environment after taking action A in the state S.

So I use a **dictionary** to build up the memory for quick storing and sampling. And I use **tuple** to concatenate the state S and the action A taken in S as (S, A) and also the next state S' and its reward R as (S', R) because firstly list is a unhashable type and can't be used as the key of a dictionary, and secondly after assigning the key and the value of a pair in the dictionary, we don't need to change them.

Each time we take action A according to Q-table and the current state S to interact with the environment and get the next state S' and the reward R, we need to store them into the Model. And in each training epoch, we also need to randomly retrieve a pair of previously observed state, its action, the next state and the reward to update the Q-table.

1.2.2 Q-table

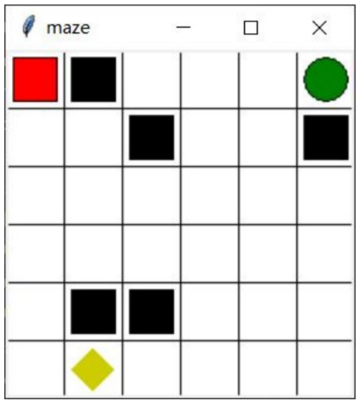
Q-table is used to get the policy $\pi(S)$ according to the current state S.

Firstly let's consider how to build up our Q-table. Taking into account that the state is **5-dimension** of which first four represent the current position and the last one indicates whether the treasure has been found. And actually the first one and the third one in the state are linearly dependent, so we can simplify them into one feature by (1) and (2), which at the same time can be applied on the second one and the fourth one:

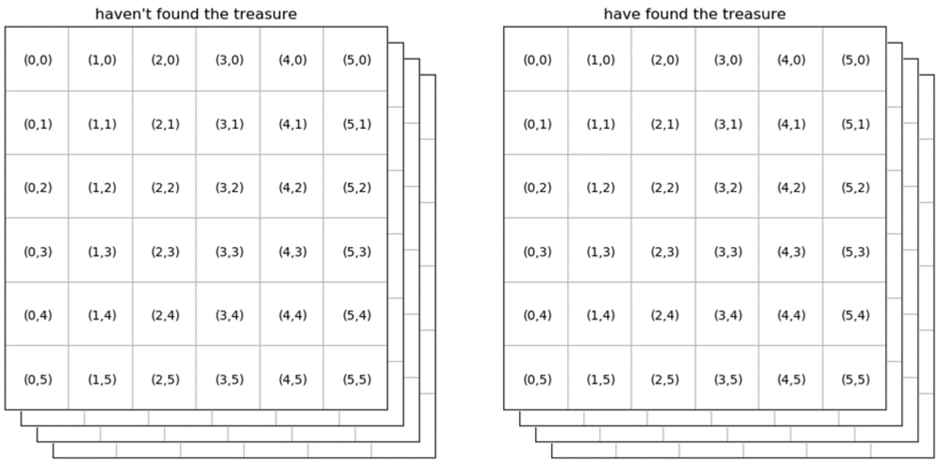
$$x = \frac{state[0] - 5}{UNIT} \tag{1}$$

$$y = \frac{state[1] - 5}{UNIT} \tag{2}$$

So finally I use a **4-dimension** matrix to store my Q-table of ($state\_found \times x \times y \times action\_size$) in Figure 3.



(a) Maze environment



(b) Q-table matrix

Figure 3: relationship between Maze environment and the Q-table matrix

Secondly let's see how Q-table update itself when get inputs(state, action, next state, reward):

$$Q(S,A) \leftarrow (1-\alpha)Q(S,A) + \alpha[R(S,a) + \gamma max_a Q(S',a)] \tag{3}$$

where α is the learning rate and γ is discount factor. From the formula, it's obivous that the larger α is, the less result trained before wil be preserved, and the larger γ is, $max_a Q(S',a)$ plays a more significant role in the updating. And actually $max_a Q(S',a)$ is the benefit in memory. So the larger γ is the agent will pay more attention to the previous experience. They can be fine tuned to help our agent converge more quickly.

Finally the agent should properly return action according to the current state to get as much reward as possible. In this part I use **ϵ-greedy** action selection which means it will choose random actions in an epsilon fraction of the time, and follows its current best Q-values otherwise. But if there are more than one actions sharing the same max value, the agent will randomly select one of them in order to avoid getting frozen at some positions especially at the corner of the grids.

And I use a gradually decaying method to reduce α and ϵ per episode to help the agent explore the maze more efficiently.

1.2.3 Training process

The training process is showed in **Algorithm 1** which is almost the same as the original Dyna-Q Learning algorithm.

Algorithm 1 Training Dyna-Q Learning Agent

```
1: Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
2: while Q-table not converge do
3:     $s \leftarrow$ current state
4:     $a \leftarrow \pi\epsilon_i(s,Q)$
5:     Execute action $a$; observe resultant state $s'$ and reward $r$
6:     $Q(S,A) \leftarrow (1-\alpha_i)Q(S,A) + \alpha_i[R(S,a) + \gamma max_a Q(S',a)]$
7:     $n \leftarrow 0$
8:     repeat
9:         $n \leftarrow n+1$
10:        $s \leftarrow$ random previously observed state
11:        $a \leftarrow$ random action previously taken in $s$
12:        $s', a \leftarrow Model(s,a)$
13:        $Q(S,A) \leftarrow (1-\alpha_i)Q(S,A) + \alpha_i[R(S,a) + \gamma max_a Q(S',a)]$
14:    until n=N
15:    $\alpha_{i+1} \leftarrow \alpha_i - \Delta\alpha$
16:    $\epsilon_{i+1} \leftarrow \epsilon_i - \Delta\epsilon$
17: end while
```

1.3 Experiment result

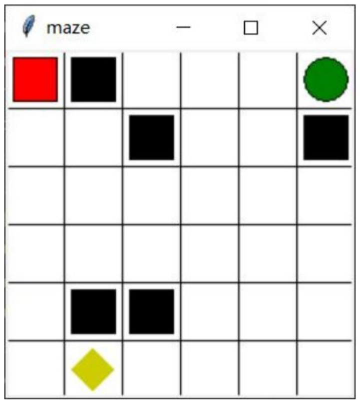
In the experiment part, I set different random seeds to help test how many interaction times different combinations of hyper-parameters required to train the Q-table to convergence.

Find $N = 1500, \gamma = 0.9, \alpha = 0.5, \epsilon = 0.1$ combining to train the Dyna-Q Learning agent can get performance that the least interaction times is **11**.

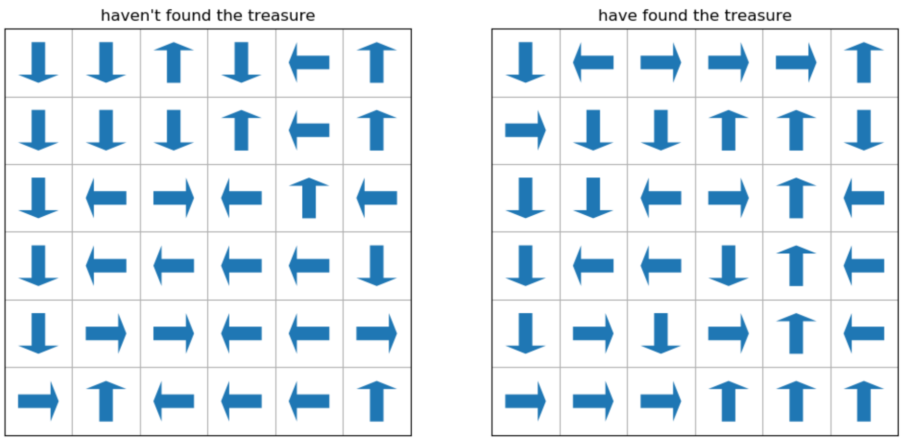
The converged policy table is showed in Figure 4.

Table 1: Different hyper-parameter(γ) experiments on Maze environment about how many episodes required to converge the Q-table.

Experiment	α_{min}	ϵ_{min}	α decay steps	ϵ decay steps	γ	random seeds	0	1	2	3	4
1	0.5	0.1	5	5	0.9	–	20	11	21	19	21
2	0.5	0.1	5	5	0.8	–	20	18	21	15	21
3	0.5	0.1	5	5	0.7	–	20	18	21	15	21



(a) Maze environment



(b) Q-table matrix

Figure 4: relationship between Maze environment and the Q-table matrix

2 Reinforcement Learning on Atari Game

2.1 Game Description

Pacman is one of the classic and leading games. Our goal is to guide the pac-Man to eat all the dots and avoid the ghosts. And the DQN agent is provided to be utilized to learn a control policies directly from the visual information of the game.

The 'MsPacman-ram-v0' gym environment is utilized as the training environment. This environment

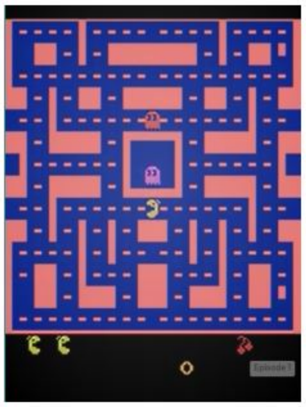


Figure 5: Atari Breakout

provides the ram(128 bytes) of the atari console as model input. Each time, the agent should choose an action from 9 available actions, corresponding to the 8 buttons on the handle and ”do nothing”.

Finally, we implement our Tricolor Rainbow to achieve the average score of 1100 and use LSTM-based model to get the highest score of 4900 in the ’MsPacman-ram-v0’.

2.2 Deep Q-Learning algorithms

The following parts will be organized in the order that:

1. We will briefly go through how to complete the training part of the Nature DQN agent to make our experiment kick off and some preliminary attempts and improvements.

2. We will introduce our implemention of double DQN, prioritized experience replay, dueling DQN, and how we combine these three methods to form a more powerfull agent.

3. We add a LSTM layer to take into account multiple continuous frames of the game instead of just one frame each time, expecting our agent learning from the previous several states to make better decision.

2.2.1 Kick off

Actually, our original DQN agent is the implemention of Nature DQN(2015). A deep Q network(DQN) is a multi-layered neural network that for a given state s outputs a vector of action values, and we need to train the neural network to make it perform better.

Nature DQN uses two Q networks one of which is used to choose action, interact with the environment and update its weights, and the other is exactly the same as the former one and it’s used to evaluate the current Q value of the target. And the former one is eval-network and the latter is target-network. The weights of the target-network is not trainable and the weights of the eval-network will be copied to the target-network once a few episodes, which is meant for reducing the correlation between the Q value of the target and the current one.

Another trick which originates from the pioneered DQN(2013) is the experiment replay. In the training part, the DQN randomly samples the previously experienced state and the other information of that state for the training of the deep Q network. When the deep Q network has not converged, there is a gap between the Q values of the targets derived from the experience replay and the Q values of the targets predicted by the neural network, and we can use SGD method updating the weights in the neural network to narrow that gap, which finally forms our policy to make actions.

In the Nature DQN, if we have m samples $\{S_j, A_j, R_j, S'_j, is\_end_j\}, j = 1, 2, ..., m$, we can use (4) to get the Q values of the current targets and use gradient descent and back propagation algorithm to update the weights w of the neural network to minimize the Loss(5).

$$y_j = \begin{cases} R_j & is\_end_j \ is \ true \\ R_j + \gamma max_{a'} Q(S'_j, A_j, w) & is\_end_j \ is \ false \end{cases} \tag{4}$$

$$\mathcal{L} = \frac{1}{m} \sum_{j=1}^{m} (y_j - Q(S_j, A_j, w))^2 \tag{5}$$

Besides finishing the training part of the Nature DQN agent, we attempt to give penalty when the agent dead, tune the structure(Figure 6) of the Q network and normalize the model input. The input of the model is one state and it is shown that every elements in the state is range from 0 to 255, so we shrink the input 255 times to make them range from 0 to 1.

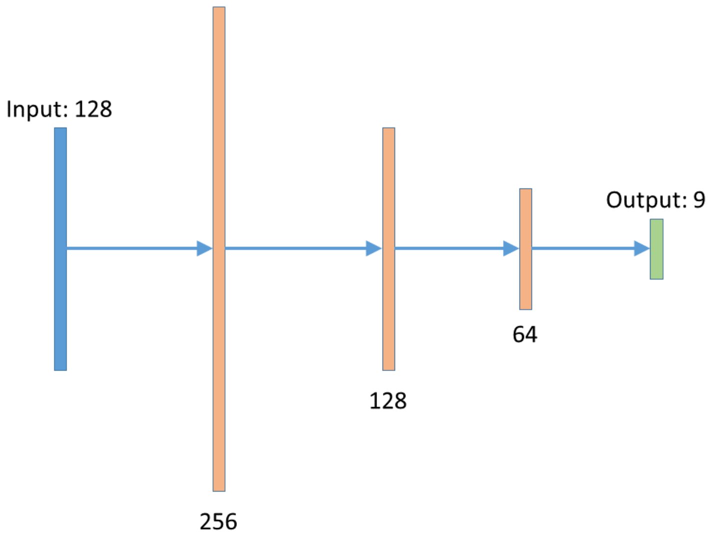


Figure 6: the structure of the Nature DQN

2.2.2 Tricolor Rainbow

Rainbow combine 6 deep Q network algorithms to get the state of the art in deep reinforcement learning. And we implement three of there algorithms which are double DQN, prioritized experience replay and dueling DQN to build up our Tricolor Rainbow DQN model.

Firstly, before DDQN appeared, the Q value in most DQN model derived from the epsilon-greedy method and its Q value is updated by (4). Using the maximum may make the network converge more quickly, but that may result in over-estimation of the truth Q value and leave high bias in the final model prediction. To solve this problem, DDQN decouple the selection of the action and the computation of the Q value of the target to eliminate over-estimation. In DDQN get the Q value by (6), while the rest part of the algorithm is the same as Nature DQN.

$$y_j = R_j + \gamma Q'(S'_j, \arg\max_{a'} Q(S'_j, a, w), w') \tag{6}$$

Secondly, in the previous DQN, we use experience replay to randomly sample training data to update the weights of the neural network. And all samples in the experience replay pool are selected in the same sampling probability. But noticing that the influence of different samples on the update of the weights varies from their temporal difference. It is obvious that the larger temporal difference, the more significantly the sample update the weights. So we can use the TD to represent the sampling probability of the data. If a sample has high TD, its sampling probability should be high, too. And we use SumTree structure to build up our experience replay pool for more efficient storage and sampling. What's more, the loss function is also improved as (7). The other part of algorithm in priority experience replay is the same as double DQN.

$$\mathcal{L} = \frac{1}{m}\sum_{j=1}^{m} TD_j(y_j - Q(S_j, A_j, w))^2 \tag{7}$$

Finally we utilize the dueling DQN the improve the performance by modify the structure of the neural network. Dueling DQN splits the Q network into two parts, the first part is only related to the state S as value function, and the other is both related to the state S and the action A as the advantage function and the prediction of the Q value can be reconstructed in the sum of these two parts as (8)

$$Q(S, A, w, \alpha, \beta) = V(S, w, \alpha) + A(S, A, w, \beta) \tag{8}$$

where w is the weights of the common part, α is the weights uniquely possesed by the value function and β is the weights in the advantage function.

But the equation above can not exclusively split the value function and the advantage function from the output and can not distinguish their roles in the prediction. So (8) can be developed as (9) in order to identify these two parts from the final result.

$$Q(S, A, \alpha, \beta) = V(S, w, \alpha) + (A(S, A, w, \beta) - \frac{1}{\mathcal{A}}\sum_{a'\in\mathcal{A}} A(S, a', w, \beta)) \tag{9}$$

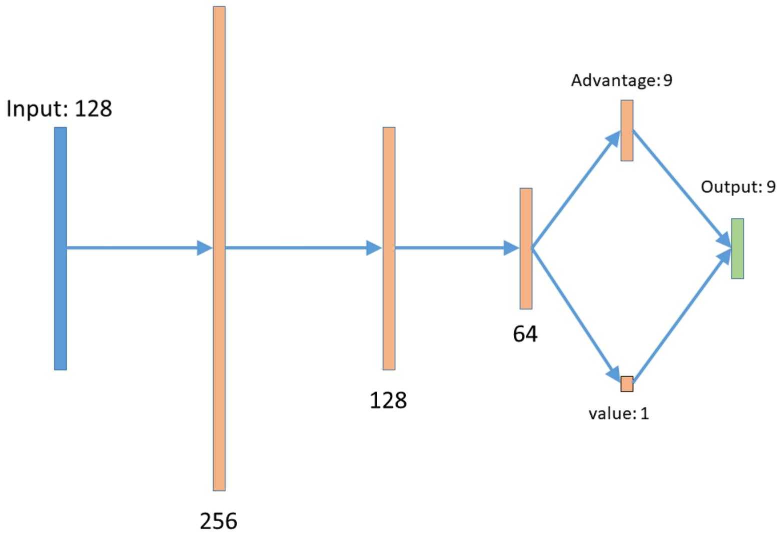


Figure 7: the structure of the dueling DQN

It is obvious that double DQN, priority experience replay and dueling DQN can be implemented independently without interference on each other. So our team combine these three DQN algorithms to form our Tricolor Rainbow model to run on the atari game 'MsPacman-ram-v0' gym. The experiment result is shown in the Section 2.3 Result.

2.2.3 LSTM

We notice that in the atari game, the agent live in a dynamic environment and the continuous frames of the game may give a litte bit more clue to guide our agent perform better comparing with justing using a single current frame as the input. But using a continuous frames as the input resulting in the problem that states in a sample are correlated and hardly using a simple fully neural network to solve it. So Long short-term memory(LSTM) which is a recurrent neural network designed to be able to not only process a single data input but also entire sequential data with various lengths, which is suitable to be utilized to learn the continuous frame states in the atari games.

We only modify the structure of our neural network(Figure 8) and the rest parts all inherit from the Tricolor Rainbow. We remain the structure of dueling DQN and also use LSTM to predict the Q value of the current state according to the sequential of a few of previous states.

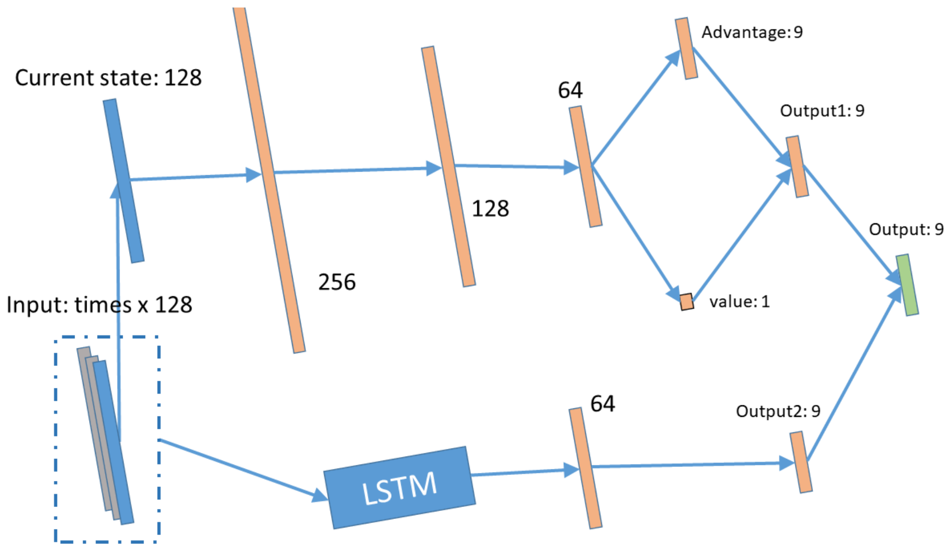


Figure 8: the structure of the dueling DQN

2.3 Result

In this part, we will show our experiment results of some of the above methods to make a comparison and to summarize our experiment on DQN.

2.3.1 Some details of the experiment

We set the hyper parameters that the max training episode is 2000, we train our model once the pacman move 5 steps, and the discount factor is 0.99, the ϵ in the epsilon-greedy method of action selection is at last 0.02. considering the size of the experience replay pool determines variety of the training data, so we at most store 20000 samples in our priority experience replay pool and our model will begin training once the size of the pool reach 10000.

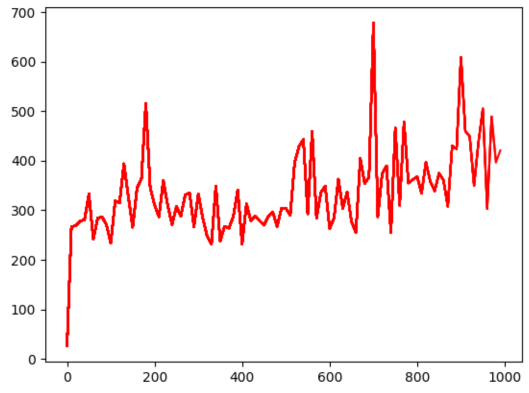
We also gradually decrease the learning rate in the training process in order to make our model converge better.

2.3.2 Comparison

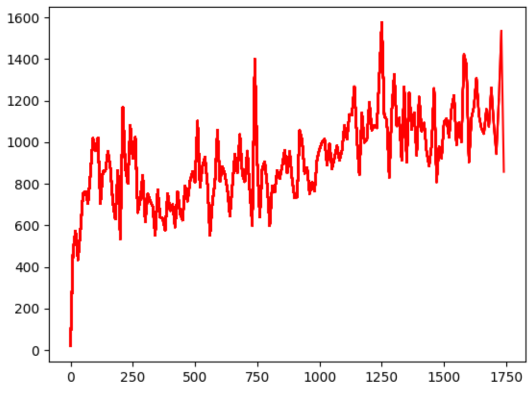
We run the Nature DQN, Tricolor Rainbow, and the LSTM-based model to compare their performance on the 'MsPacman-ram-v0'. It turns out that Tricolor gets the highest average score of 1100 and the highest best score of 4900.

Table 2: Performance of different models on the 'MsPacman-ram-v0'

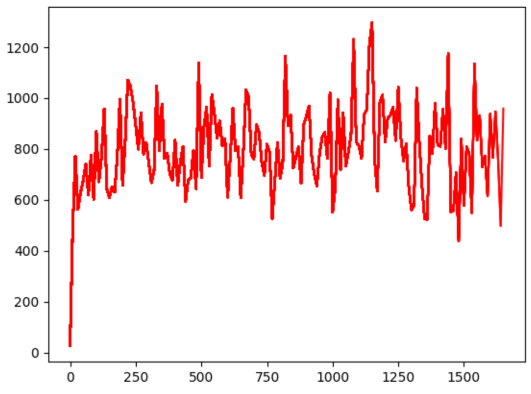
Method	average score	best score
Nature DQN	500	3440
Tricolor Rainbow	1100	4900
LSTM-based model	850	4390



(a) Nature DQN



(b) Tricolor Rainbow



(c) LSTM-based model

Figure 9: Training trajectory of three methods

3 Conclusion

In first part we use Dyna Q Learning algorithm to solve the deterministic environment of a maze and aim to converge our final policy with as little interaction as possible, which significantly enhances our theoretical knowledge of the basic reinforce learning and gives an opportunity to utilize it to solve some simple problems. And in the practice, we not only get more familiar with the basic idea of Q learning but also acquire more impression of some details of the Dyna Q learning.

In the second part, in which the training part is really time-consuming and our first thing to learning from is that we should properly schedule our training and developing plans. And there are also some tricks in how to effectively tune our deep Q network to perform better. But as for my experience of the second part of the homework 2, actually, it's quite rewarding to learn from the code provided by the TA ,widely search for more material of deep Q network learning to get some inspiration and grasp several open source code to improve our model. Meanwhile, we can add our novel idea into the trial. And finally do systematic wide range of experiment to tune the hyper parameters of the model. The whole process almost covers an academic research, which lays the foundation for future research.