Dive into Deep Q-Network Algorithm[*]

Ngoc-Tan Pham[†] **Tri-Dzung Bui**[†] **An Khanh Vo**[†]

ngctn3701@gmail.com btrdung1621@gmail.com vokhanhan25@gmail.com

December 2020

Abstract

Deep Q-Network is an important algorithm, used to solve sequential decision making problems. It involves training a Deep Neural Network called a *Deep Q-Network (DQN)*, to approximate a function associated with optimal decision making, the *Q-function. Deep Q-Network* has been reaching its peak by defeating former go champion - Lee Sedol, and became both famous and empirically successful all around the world in the field of Computer Science. Its theoretical foundation make it relatively difficult to understand completely, however, through the combination between *Neural network* [5] and *Q-Learning* [1]. In this work, we make our first attempt to theoretically understand the *Deep Q-Network algorithm* [2] from both algorithmic and mathematical perspectives. Besides, we shall take experiments to clarify the advantages of *Deep Q-Network* in comparison with *Double Deep Q-Network (Double DQN)* by means of detailed figures.

1 Introduction

Previously, computer scientists optimize decision-making problems by interacting with the environment and learning from the experiences, which in details, they have to find an optimal policy in the sense of maximizing the expected value of the total reward over any and all successive steps, starting from the current state. From this point of view, they came up with a brand new idea and they called it *Q-Learning. Q-Learning* is a model-free reinforcement learning algorithm to learn equality of actions telling an agent what action to take under what circumstances. It requires a non-model of the environment, or in another sayings, it uses model-free method to handle problems with stochastic transitions and rewards without requiring any adaptations. The main limitations for this kind of algorithm, however, are not having sufficient enough data sets to fast executive real world problems and having to learn a huge steps of very poor performance in simulation initially. This accidentally leads to the slowly and low-productively training process.

In such situations, computer scientists came up with another training method, which the agent had good online performance from the start of learning. They proceed pre-training agent so that it is capable of performing perfectly in the task from the start of learning, and then improving continually from its own self-generated data. They propose, in details, a new deep reinforcement learning algorithms, called *Deep Q-Learning.* We can make it easy by considering it a method to leverage even very small amounts of demonstration data to massively accelerate learning. With this breakthrough in computer science, agents now achieve tremendous success in solving highly challenging problems, the game of Go, for instance, which we have mentioned above. In *Deep Q-Network*, the value of policy functions is often represented as deep neural networks and the related deep learning techniques can be applied.

2 Background

2.1 Reinforcement learning

Before starting, let imagine a gold digger. He works in a mine. He usually have to dig in different places each time. Once he digs a gold, he will get 100 dollars. In an *Markov decision processing* (MDP), the mine is the environment, the gold digger is the agent, 100 dollar is the reward. Each time he goes through the mine or each time he digs a gold, we called it an action. Each step he goes, we called it the state. Our problem here is how that gold digger can achieve as much cumulative money as possible. In other words, it is the agent's goal to maximize the cumulative rewards.

[*]This is our report for CS115 - Mathematics for Computer Science (Fall 2020) at University of Information Technology - Vietnam National University HCMC

[†]Faculty of Computer Science, University of Information Technology - Vietnam National University HCMC

Key notion:

The figure below shows us a visualization of MDP's cycle:

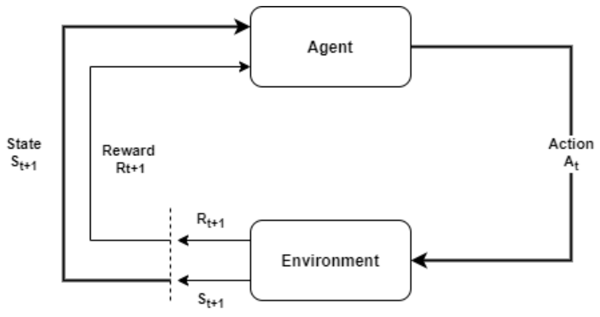

Figure 1: Markov decision process' cycle

Verbally, we can say:

- **Step 1**: At time t, the environment has the state of S_t
- **Step 2**: The agent decides which action A_t to take next, based on current state
- **Step 3**: The environment will change its state, from S_t to S_{t+1}. The agent will achieve the reward R_{t+1} afterwards
- **Step 4**: We assign t by $t+1$, then we return to *step 1*

At *step* 2 above, the agent decides to choose the action A_t based on the transition probabilities.
Let denote this probability by:

$$p(s', r \mid s, a) = \mathbb{P}\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\}$$

The p function returns the transition probability from state s to s' after doing the action A and get the reward of R. We assign $\mathbb{P}$ as the probability from now on.

Expected return is used to measure either how good a state is or how worth an action will be. *Future reward*, or *return*, is the sum of the *discounted reward* multiplied by the reward at time t.

$$G_t = R_{t+1} + \gamma R_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{1}$$

The *discounted factor*, or the $\gamma \in [0;1]$, penalizes the rewards in the future. We need that formula to avoid divergence and result in infinity loop. In other hand, this can be seen as the uncertainty for future reward, e.g., weather forecast, and the further the reward is, the less immediate benefit we achieve.

We can transform (1) into:

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^4 R_{t+4} + \ldots \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \ldots) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned}$$

This transformation shows us the correlation between continuous time steps

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi\left[G_t \mid S_t = s\right] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right] \end{aligned} \tag{2}$$

Equation (2) can be understood as the value of state $S(v_\pi)$ followed by policy π is the expected return from state s at time t followed by policy π thereafter.

$$\begin{aligned} Q_\pi(s, a) &= \mathbb{E}_\pi\left[G_t \mid S_t = s, A_t = a\right] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right] \end{aligned} \tag{3}$$

Q-function, denoted by *action-value function* Q_π, has the return result for any given actions-states pair called *Q-value*. *Q-function* can be understood as the value of action a in state s followed by policy π, calculated by expected return from the beginning of state s at time t taking action a followed by policy π thereafter.

$$V_\pi(s) = \sum_{a \in A} Q_\pi(s,a)\pi(a \mid s)$$

Besides, since we follow policy π, we can get *state value* by multiplying probability distribution of possible actions at a state by *Q-values*.

$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s)$$

Should we take the difference between *Q-value* and *state value*, we shall get a new value, which is *A-value*, used to recognize how precious an acted action at a certain state is.

$$V_*(s) = \max_\pi v_\pi(s) \text{ for all s} \in \mathbf{S} \tag{4}$$

The *optimal state-value function* V_* shows us the *maximum expected benefit* we can achieve by any policy π for each state.

$$Q_*(s,a) = \max_\pi Q_\pi(s,a) \text{ for all s} \in \mathbf{S} \text{ and for all a} \in \mathbf{A}(\text{s}) \tag{5}$$

The *optimal Q-function* Q_* brings us the *maximum expected value* by any given policy for each possible state - action pair.
From (4) and (5), we get the optimal-value function:

$$\begin{aligned}\pi_* &= \arg\max_\pi \, V_\pi(s) \\ &= \arg\max_\pi \, Q_\pi(s,a)\end{aligned}$$

Bellman Equation

Bellman equation is a necessary condition for optimizing associated with mathematical optimizing method, called *dynamic programming*. It is a set of equations to analyze the value functions into the sum of immediate rewards and future discounted values. This leads to dividing optimized problems into a series of simpler sub-problems.

$$\begin{aligned}Q(s,a) &= \mathbb{E}[R_{t+1} + \gamma V(S_{t+1}) \mid S_t, A_t = a] \\ &= \mathbb{E}[R_{t+1} + \gamma \mathbb{E}_{a\sim\pi} Q(S_{t+1}, a) \mid S_t = s, A_t = a]\end{aligned}$$

$$\begin{aligned}V(s) &= \mathbb{E}[G_t \mid S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \mid S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots) \mid S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma V(S_{t+1}) \mid S_t = s]\end{aligned}$$

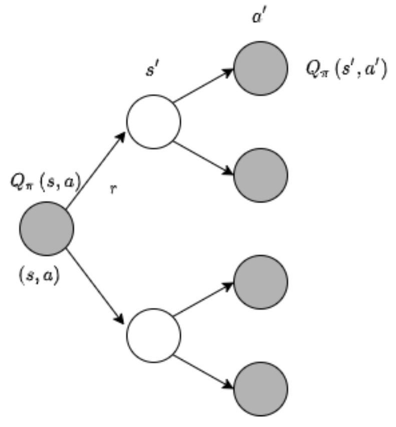

Figure 2: Bellman tree-building.

Geometrically, the Bellman tree-building process can transform equations into state-value functions and action-value functions. As we reach further, we can extend the tree as we mentioned before, according to policy π

$$Q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} Q_*(s',a')]$$

The above equation is called *Bellman optimality equation* for Q_*. This equation can be understood as the *Q-value* is the expected reward R_{t+1} we achieve when executing action a at state s, plus the *maximum expected discounted return* that can be achieved from any possible next state-action pair (s', a'). Since the agent is following an optimal policy, the state s' will be the state from which the next best action a' will be executed at time $t+1$.
When we have Q_*, we can identify the optimal policy. It is due to the fact that for any given state s, the *reinforcement learning algorithm* can find action a to maximize $Q_*(s,a)$.

2.2 Q-learning

Q-learning is a model-free reinforcement learning algorithm to learn the quality of action a to tell the agent which actions to execute at a certain state s. It does not require a model of a certain environment. It can handle dynamically environmental problems and achieve rewards without any adaptations. The development of *Q-learning* is a breakthrough in early days of reinforcement learning.

The interesting point in *Q-learning* is the independence of the current policy to choose the second action a_{t+1}. It, essentially, estimates Q_* among the best *Q-values*, but it does not matter which action leads to this maximum *Q-value* and in the next step, this algorithm does not have to follow that policy.

Updating the Q-Value

Algorithm 1: Updating the Q-value

```
1 $t = 0$
2 while $t \leftarrow 0$ not converged do
3     At time step $t$, we pick the action according to $Q$ values, $A_t = \mathbf{argmax}_{a \in A} Q(S_t, a)$ and $\epsilon$ -
      greedy is commonly applied.
4     After applying action $A_t$, we observe reward $R_{t+1}$ and get into the next state $S_{t+1}$.
5     Update the Q-value function: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \mathbf{max}_a Q(S_{t+1}, a) - Q(S_t, A_t))$
6     $t = t + 1$
7 end while
8 return $Q$
```

We want to create a *Q-value* for action-state pair as close to the right hand side of the *Bellman equation* as possible to make *Q-value* converge on optimal *Q-value* Q_*. Essentially, this can be made by iterating comparison the loss between *Q-value* and the optimal Q-value Q_* for each state-action pair. We update *Q-value* over and over every time we meet this same state-action pair so as to reduce the loss, after that.

$$\begin{aligned} Loss &= Q_*(s,a) - Q(s,a) \\ &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} Q_*(s',a')\right] - \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\right] \end{aligned} \tag{6}$$

We have the equation to calculate the new *Q-value* for new state-action pair (s,a) at time t:

$$Q_{new}\left(s,a\right) = \left(1-\alpha\right) \underbrace{Q\left(s,a\right)}_{\text{old value}} + \alpha \overbrace{\left(R_{t+1} + \gamma \max_{a'} Q\left(s',a'\right)\right)}^{\text{learned value}} \tag{7}$$

Our new *Q-value* is equal to a weighted sum of our old value and the learned value. In the first time, the old value is assigned to 0 and the agent experiences this particular state-action pair. Then, we multiply the old value by $1-\alpha$.

The *learning rate* α is a real number between 0 and 1. It can be thought as how quickly the agent changes to abandon the previous *Q-value* in *Q-table* for a given state-action pair to perceive the new *Q-value.*

Our *learned value* is the reward from which our agent achieves moving right from the starting state, plus with the discounted estimate of future *optimal Q-value* for the next state-action pair (s',a') at time $t+1$. We afterward multiply all the learned value with our learning rate α.

Our newly calculated *Q-value* will be stored in our *Q-table* for this particular state-action pair, noticeably.

We have completed the single time step. Not until the end of episode will this process happen continuously. Once *Q-function* converges on the *optimal Q-function*, we receive the *optimal policy.*

3 Deep Q-Network

Theoretically, we can store Q_* for every state-action pair in *Q-learning* into an enormous table. This can make the storage space in computer infeasibly, however, when the number of state-action pairs become gigantic. Thus, we shall find a new method to create a specific function for approximating the Q-value. Luckily, we are in the spring of the advances in deep learning techniques, which enable us to apply *neural network* [5] into this case. This combination of deep learning of reinforcement learning is the significant discovery, and this combination is known as *Deep Q-network* (DQN). Ideally, we can use a neural network with current state s as input layer and the execute action as output layer instead of using an *Q-table* to store huge state-action pairs as in *Q-learning.*

Unfortunately, *Deep reinforcement learning* also get into a variety of trouble. Reinforcement learning becomes more unstable when neural network is used to represent action value. Training a neural network requires such a great deal of data, but even if we get such an amount, it is not guaranteed neural network will converge on the optimal value function. In practice, there are conditions in which network's weights might be either diverged or oscillate due to the high correlation between actions and values.

It is *experience replay* and *target network* that DQN uses to stablize and improve significantly the training process.

Experience replay

In DQN, we often use the technique, known as *experience replay.* By this means, we can store the experience of our agent at each time step in our data set, or can be denoted as *replay memory.* We express the experience at time t as tuple e_t:

$$e_t = (s_t, a_t, r_{t+1}, s_{t+1})$$

All the experience at one time step in all episodes are all stored in *replay memory.* In practice, our *replay memory* has its own limitation, nevertheless. Thus, we are able to store $\boldsymbol{N}$ last experience.

In this *replay memory data set*, we retrieve random samples for training our neural network. The action of experience collecting and experience sample extracting from stored replay memory is called *experience* *replay.* Our main reason for using replay memory is to break the correlate relation between continuous samples. Only if our neural network learns from continuous experience as they occurred sequentially in the environment do our samples get the highly correlate relation, which will lead to the inefficiency in learning.

Target Network

Bellman equation provides us the value of $Q(s, a)$ via $Q(s', a')$. There is just one step among s and s', which leads to its high correlation between them. This is also a barrier and neural network will get difficulty in distinguishing them, as the result.

$Q(s, a)$ is as close as our desired output once we update neural network's parameters, we can indirectly change the parameters created for $Q(s', a')$ and those close by states. This accidentally leads to unstable neural network's training process.

To make our training process more stable, there is a technique called *target network.* Through which, we can keep the neural network and use it for $Q(s', a')$ value in Bellman equation.

Q-value, which is predicted in our copy of Q-network, is called *target network.* It will be used to backpropagate through and train the main Q-network. The importance which needs to be clarify is the untrained neural network's parameters, but they will be synchronized periodly with the main Q-network's parameters. The idea of using Q-value of target network to train main Q-network will increase the stability in training process.

Deep Q Network

There are 2 main interleaved phases in Deep Q Network. One is we sample environment by means of taking actions and storing the observed experienced tuples in a replay memory. The another is we take a small batch of the tuple from the memory randomly, and learn from those batches through SGD to update steps [3].

Those 2 phases do not depend directly to each other and we can sample our data through many steps, and then one learning step, or even many learning steps with many random batches. In practice, we can not executive learning step immediately. Not until tuples of experience are not sufficient in replay memory $\mathfrak{D}$ will we have to execute.

Algorithm 2: Deep Q Network Algorithm

```
1  Initialize network $Q$, target network $\hat{Q}$, experience replay memory $D$
2  Initialize the Agent to interact with the Environment
3  while not converged do
4      $\epsilon \leftarrow$ setting new epsilon with $\epsilon - decay$
5      Choose an action a from state $s$ using policy $\epsilon - greedy\,(Q)$
6      Agent takes action $a$, observe reward $r$, and next state $s'$
7      Store transition $(s, a, r, s', done)$ in the experience replay memory $D$
8      if enough experiences in $D$ then
9          Sample a random minibatch of $N$ transitions from $D$
10         for every transition $(s_i, a_i, r_i, s'_i, done_i)$ do
11             if $done_i$ then
12                 $y_i = r_i$
13             else
14                 $y_i = r_i + \gamma max_{a' \in A} \hat{Q}\,(s'_i, a_{'})$
15             end if
16         end for
17         Calculate the loss $\mathcal{L} = 1/N \sum_{i=0}^{N-1} \left(Q\,(s_i, a_i) - y_i\right)^2$
18         Update $Q$ using the SGD algorithm by minimizing the loss $\mathcal{L}$
19         Every C steps, copy weights from $Q$ to $\hat{Q}$
20     end if
21 end while
```

We will intialize our main Q-network and target network, at first, and initilize empty replay memory $\mathfrak{D}$. Since we have to notice is the finity of our memory, we are able to use pseudo code circular queue to store $\mathfrak{N}$ near experience tuple. We will also initialize the agent, which will interact with the environment.

One noticable thing is that we will not clear our memory after episodes, which leads to our recall and mass memory building from episodes. This data map is what maps our algorithmic installation.

Our *loss function* looks like:

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s')\sim U(\mathfrak{D})}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^{-}) - Q(s,a;\theta)\right)^2\right] \tag{8}$$

in which $U(D)$ is the uniform distribution on replay memory $\mathfrak{D}$ and θ^{-} is the parameter of target Q-network.

Double Deep Q-Network

We have discussed about basic *Q-Learning* and *Deep Q-Network* above.

Motivation

DQN is proposed to solve overestimation of Q-value in vanilla Q-Learning.

In vanilla Q-learning, the optimal policy of agent is always choosing the best action in every given action. The theory behind this idea is which the best action has the largest estimated Q-value. Nevertheless, our agent does not know anything about the environment, at the beginning, it has to estimate $Q(s,a)$ and update at each loop through. Such Q-values have a huge noise and at no times will we make sure whether the action with maximum estimated Q-values is the really dominated.

Unfortunately, the best action tends to have smaller Q-values opposed to the non-optimal one's in most of the cases. According to the optimal policy of vanilla Q-learning, the agent has a tendancy of take the non-optimal action in any given states, just for the the maximum Q-value. This problem is termed as *overestimations of Q-value.*

When such problem happens, noises from estimated Q-values shall the big causes for *positive biases* in update process. Consequently, learning process becomes sophisticated and messy.

Because of the noisy of current $Q(s,a)$ used to estimate the best $Q(s,a)$, *positive biases* happen. The difference between the best $Q(s,a)$ and the current one in *loss function* is also messy and contains positive biases, which are happened by different noises. Positive biases impact a high volume in the updating process

$$\begin{aligned}
Loss &= Q_{*}(s,a) - Q(s,a) \\
Q_{*}(s,a) &= R_{t+1} + \gamma \max_{a'} Q_{*}(s', a') \\
Q_{new}(s,a) &= (1-\alpha) Q(s,a) + \alpha \left(R_{t+1} + \gamma \max_{a'} Q(s', a') \right)
\end{aligned}$$

More specifically, shall the noises of all Q-values have the *uniform distribution* (Q-values have are equally overestimated), overestimation is not the problem since those noises do not affect to the difference between $Q(s', a)$ and $Q(s,a)$. This is originated from H. van Hasselt 2015 [4]

Double Q-Learning

Double Q-Learning uses 2 different action-value functions, Q and Q' are estimators. Even if noises are noisy, they are still viewed as uniform distribution. That is, this algorithm solves the problem of estimations of action values.

The update procedure is different slightly in comparison to the basic one:

Basic Q-Learning:

$$Q_{(s,a)} = (1-\alpha)\, Q\,(s,a) + \alpha\Big(R_{t+1} + \gamma \max_{a'} Q\,(s',a')\Big)$$

Double Q-Learning:

$$Q_{(s,a)} = (1-\alpha)\, Q\,(s,a) + \alpha\Big[R_{t+1} + \gamma Q'\,\Big(s', \max_{a'}(s',a')\Big)\Big]$$

Q-function is used to choose the best action a with maximum Q-value of the next state.
Q'-function is used to calculate the expected Q-value by using the chosen action a above.
We update *Q-function* by using expected Q-value of *Q'-function*

Algorithm 3: Double Q-learning

```
1  Initialize $Q^A, Q^B, s$
2  repeat
3      Choose $a$, based on $Q^A(s,.)$ and $Q^A(s,.)$, observe $r, s'$
4      Choose (e.g. random) either UPDATE(A) or UPDATE(B)
5      if UPDATE(A) then
6          Define $a^* = argmax_a Q^A(s',a)$ $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)\big(r + \gamma Q^B(s',a^*) - Q^A(s,a)\big)$
7      end if
8      else
9          Define $b^* = argmax_b Q^B(s',a)$ $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)\big(r + \gamma Q^A(s',a^*) - Q^B(s,a)\big)$
10     end if
11     $s \leftarrow s'$
12 until;
```

Nevertheless, this method also have the same drawbacks as Q-learning for its tabular way. We shall resolve this problem with deep neural network.

Double Deep Q-Network

Inspired by Double Q-Learning, Double DQN splits *deep neural network* in DQN into 2 seperation: *Deep Q-Network* and *target network.* In DQN, the target Q-Network selects and evaluates every action resulting in an overestimation of Q value. To resolve this issue, **DDQN proposes to use the Q-Network to choose the action and use the target Q-Network to evaluate the action**. In details:

1. Deep Q Network chooses the best action a with maximum Q-value of next state.
2. Target network uses to evaluate Q-value with the selected action a above
3. Update Q-value of Deep Q-Network based on evaluated Q-value from target network
4. Update target network's parameters based on deep Q network's parameters in each iteration
5. Update deep Q network's parameters with *Adam optimizer*

Algorithm 4: Double Deep Q-Network (Hasselt et al., 2015)

```
1  Initialize primary network $Q_\theta$, target network $Q_{\theta'}$, replay buffer $D$, $\tau \ll 1$
2  for each environment step do
3      for each environment step do
4          Observe state $s_t$ and select $a_t \sim \pi(a_t, s_t)$
5          Execute $a_t$ and observe next state $s_{t+1}$ and reward $r_t = R(s_t, a_t)$
6          Store $(s_t, a_t, r_t, s_{t+1})$ in replay buffer D
7      end for
8      for each update step do
9          sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim D$
10         Compute target $Q$ value: $Q^*(s_t, a_t) \approx r_t + \gamma Q_\theta(s_{t+1}, argmax_{a'} Q_{\theta'}(s_{t+1}, a'))$
11         Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
12         Update target network parameters: $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$
13     end for
14 end for
```

Comparison between Deep Q Network and Double Deep Q Network

In the game of *Lunar Lander*:

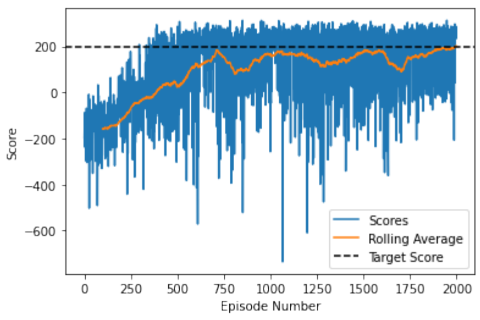

Figure 3: Deep Q Network

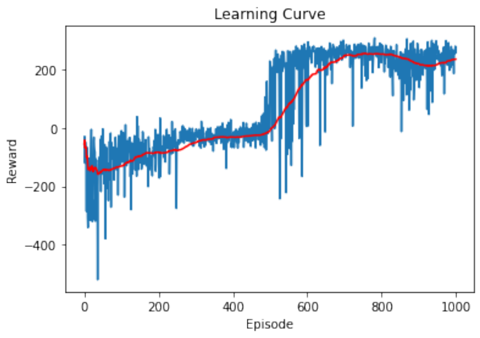

Figure 4: Double Deep Q Network

In the game of *CartPole*:

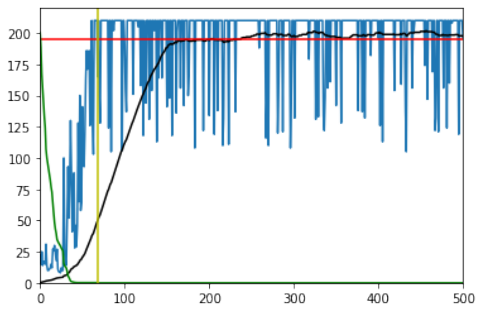

Figure 5: Deep Q Network

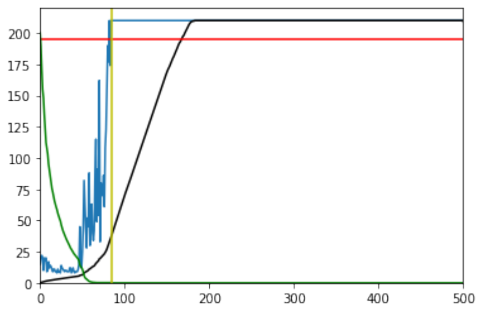

Figure 6: Double Deep Q Network

4 Summary

In this report, we have discussed about the theoretical and mathematical base of Deep Q Network and Double Deep Q Network, which are among the most important base of Reinforcement Learning. We then visualize to compare performance between DQN and DDQN.

References

[1] A. A. Ezhov and D. Ventura. *Quantum Neural Networks*, pages 213–235. Physica-Verlag HD, Heidelberg, 2000. ISBN 978-3-7908-1856-7. doi: 10.1007/978-3-7908-1856-7_11. URL https://doi.org/10.1007/978-3-7908-1856-7_11.

[2] 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*, 518(7540):529–533, Feb. 2015. ISSN 00280836. URL http://dx.doi.org/10.1038/nature14236.

[3] P. Nakkiran, G. Kaplun, D. Kalimeris, T. Yang, B. L. Edelman, F. Zhang, and B. Barak. Sgd on neural networks learns functions of increasing complexity, 2019.

[4] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015. URL http://arxiv.org/abs/1509.06461.

[5] N. Yadav, A. Yadav, and M. Kumar. *History of Neural Networks*, pages 13–15. Springer Netherlands, Dordrecht, 2015. ISBN 978-94-017-9816-7. doi: 10.1007/978-94-017-9816-7_2. URL https://doi.org/10.1007/978-94-017-9816-7_2.