Lunar Lander Solver Based on DQN and DDQN Methods

Beixi Chen

Spring 2019
bchen372@gatech.edu
git hash: f264eefbb6b55e6a43c2ccab9ce3d19145176339

ABSTRACT

In "Playing Atari with Deep Reinforcement Learning", Mnih et al. presented the first deep learning model combined with reinforcement learning to successfully learn control policies directly from high-dimensional input. In this paper, Deep Q-learning Network (DQN) established by Mnih et al. [1] is implemented to learn control policy in an environment called Lunar Lander built by OpenAI gym and to achieve an average score of above 200 for last 100 experience within pre-specified maximum episode time. Yet Hado van Hasselt et al. [2] showed DQN algorithm suffers from substantial overestimations and proposed Double Deep Q-learning (DDQN) algorithm, a specific adaptation to the DQN algorithm and showed that the resulting algorithm alleviate the problem of overestimations, leading to much better performance. This paper further implements the DDQN algorithm to show the performance of it and the difference between these two models.

I. ENVIRONMENT AND PROBLEM ANALYSIS

A. Environment

Lunar Lander is an environment built in OpenAI gym, it consists of a 8-dimensional continuous state space.

B. Problem Analysis

1) High-dimensional input need DL to extract features: As stated above, the Lunar Lander has high-dimensional inputs data. Learning to control agents directly from high-dimensional inputs is challenging. Most Reinforcement Learning methods operate on these problems have relied on hand-crafted features combined with linear value functions or policy representations. [1] Obviously, the performance heavily relies on the quality of the feature selected. Yet significant advances in deep learning have made it possible to extract high-level features from high-dimensional raw data, leading to breakthroughs and making life much easier. Thus, it is natural to utilize Deep learning technique to solve Lunar Lander problem.

2) DL assumptions need to be satisfied: Firstly, most successful deep learning methods required a huge number of hand-labelled training data in order to get satisfactory results. Yet the reward which reinforcement learning would learn from is sparse and delayed, which means the training data could only get its true label after hundreds or thousands time steps during one episode learning. Another issue is that one of the assumption of deep learning is that the underlying distribution of data is fixed and no correlation among data samples meaning samples are independent. However, in reinforcement learning context, what the agent dealing with are sequences of highly correlated states, which to some extends, violate the assumption of deep learning algorithms. Mnih et al. [1] demonstrates that a convolutional neural network can overcome these challenges to successfully learn control policies from raw data in complex reinforcement learning environments:

- The convolutional neural network is trained with a variant of the Q-learning [3] algorithm, using stochastic gradient descent to update the weights.
- To alleviate the problems of correlated data and non-stationary distributions, Mnih et al. [1] used an experience replay mechanism which randomly samples previous transitions, and thereby smooths the training distribution over many past behaviors.

These two subsections stated above explain why this paper choose to use DQN to solve Lunar Lander problem. And the following content of this paper would show how DQN works and how it performs.

3) Overestimation problems in standard Q-learning: Q-learning, a form of temporal difference learning, is one of the most popular reinforcement learning algorithms. But because it includes a maximization step over estimated action values, it is known to learn higher action values, which tends to prefer overestimated to underestimated values. The overestimations can occur when the action values are inaccurate. [2] DQN algorithm also suffers from substantial overestimations in some games in the Atari 2600 domain, as demonstrated by Hado van Hasselt [2] Subsequently, Hado van Hasselt et al. constructed a new algorithm called Double DQN which they showed not only reduce overestimation, but lead to much higher scores on several games. The core of DDQN as follow:

$$Y_t^{DoubleQ} = R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\text{argmax}} Q(S_{t+1}, a; \boldsymbol{\theta}_t)); \boldsymbol{\theta}'_t)$$

The selection of the action, in the argmax, is still due to the online weights $\boldsymbol{\theta}_t$. Model still estimates the value of the greedy policy according to the current values, as defined by $\boldsymbol{\theta}_t$. However, model uses the weights $\boldsymbol{\theta}'_t$ to fairly evaluate the value of this policy.

II. DQN AND DDQN METHODS

A. Background

1) Standard Q-function: In the reinforcement learning context, the ultimate purpose is to find the optimal policy of each state, one way to realize this objective is to find the true action-value function(also known as Q-function) for each state-action pairs. Q-function can be denoted as the maximum sum of rewards r_t discounted by c at each time step t, achievable by a behaviour policy $\pi = P(a|s)$, after making an observation (s) and taking an action (a).

$$Q^*(s,a) = \max_x \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...|s_t = s, a_t = a, \pi]$$

2) Deep Q-learning Network: As mentioned in the problem analysis section, due to Lunar Lander problem's complex continuous space state, this paper would apply deep neural networks to make non-linear function approximation, hence, flexible function approximation on Q-function and to further build up progressively more abstract representations of high-dimensional input data. Now, the task is to learn a parameterized value function $Q(s, a; \boldsymbol{\theta}_t)$. The standard Q-learning update is then:

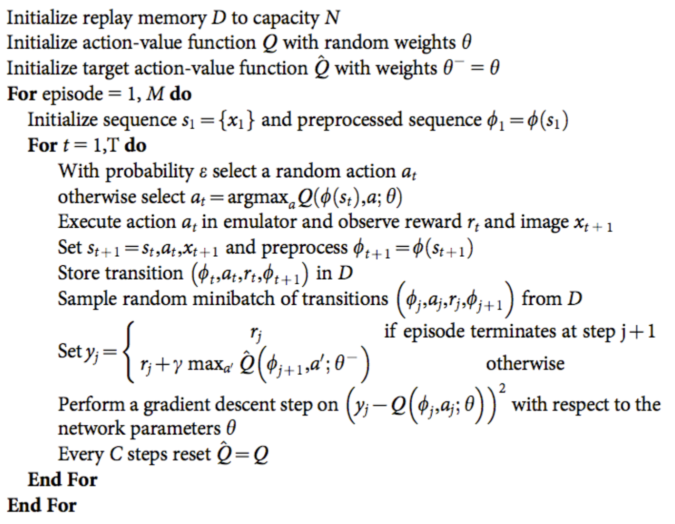
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha * (Y_t^Q - Q(S_t, A_t; \boldsymbol{\theta}_t))\nabla_{\boldsymbol{\theta}_t} Q(S_t, A_t; \boldsymbol{\theta}_t) \quad (1)$$

3) Experience replay: However, the correlations present in the sequence of observations would lead to the consequence that small updates to Q may significantly change the policy π and target Y and therefore change the data distribution. This non-stationarity would induce upward bias and divergence in the course of reinforcement learning when a nonlinear function approximator including a neural network is used to represent the Q-function. To address this problem, Mnih et al. [4] utilized a biologically inspired mechanism termed experience replay that randomizes over the data and they managed to reduce correlations in the state sequences and to smooth changes in the data distribution. In this paper,the experience replay would also be used to show its power to overcome problem of correlated sequences.

B. Implementation

1) Algorithm Explanation: The algorithm shown in Fig.1 is a full reference from Mnih et al.'s "Human-level control through deep reinforcement learning". [4] This paper would implement this algorithm closely for the DQN part. The following content would explain in detail how this paper implements this algorithm and what adaption was made.

- An experience replay buffer of 2 million size is created.
- There are two models being created, the first one in this paper is called "Training Model", the second one is called "Target Model".
- Next is the Q-learning Network, author of this paper utilized keras to construct the conventional neural network. As shown in Mnih et al.'s paper, networks are constructed by 3 layers. The first hidden layer convolves 64 filters with the input state of size 8 and applies a rectifier non-linearity. Note that rectifier non-linearity is 'relu' activation function in keras. The second and third hidden layer convolves 64 filters followed by a rectifier non-linearity. The output layer is a fully-connected linear layer with output for all actions, meaning the output is a Y vector of size 4 (we have 4 actions in Lunar Lander).



```
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
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 $\varepsilon$ 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 $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
        network parameters $\theta$
        Every $C$ steps reset $\hat{Q} = Q$
    End For
End For
```

Fig. 1: Algorithm 1: deep Q-learning with experience replay.

- At the beginning of each episode, Training and Target networks are initialized with same uniformly randomized weights.
- When a current state is given, action is being chosen either randomly with probability of ϵ or deterministically from the Training Model which has the maximum y-value. The value of ϵ would be discussed in hyper-parameters part later.
- According to the current state and the action that agent choose, the Lunar Lander environment would return tuples of <current state, action, reward, next state, done>, which then be stored in replay buffer.
- At each step, a random batch of previous experiences would be drawn from buffer. When a terminal state is reached, target vectors Y_j is computed as reward, otherwise, as reward plus discounted future values using Target Model as follows, where the discounted rate is known as γ.
- Training Model would update with these previous experience tuples and their corresponding Y_j.
- Loss function is computed as MSE with Adam optimizer.
- After every C steps, Target Model would be updated with weights of Training Model to reduce overestimation. Training episode would be forced to end when agent get average reward of over 200 for the last 100 episodes.

2) Hyper-parameters:

- α: Learning rate of Neural Network
- γ: Discount rate of future reward
- $sample\ size$: Sample batch size when drawn from buffer

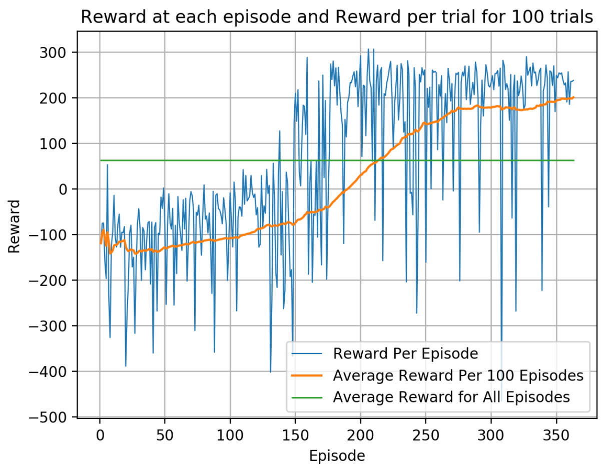


Fig. 2: Reward per episode and reward for last 100 episodes

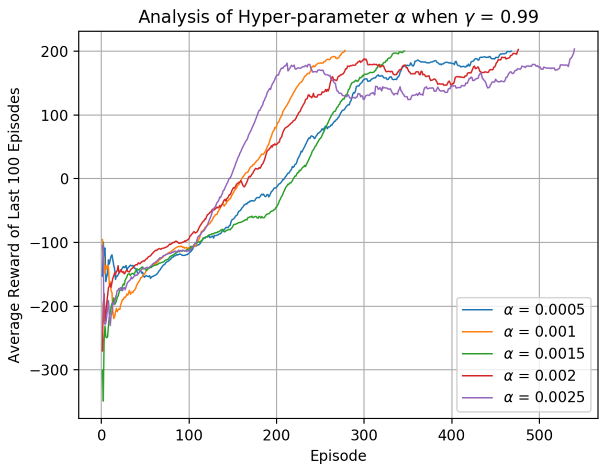


Fig. 3: Analysis of Hyper-parameter α

- C: Target Model updates every C steps
- $Epsilon\ decay$: ϵ decreases linearly from 1 to 0.001 over $Epsilon\ decay$ episodes.

$$\epsilon = \epsilon_{start} - \frac{\epsilon_{start} - \epsilon_{end}}{decay}$$

What about the value chosen for these hyper-parameters? In this paper, γ would be fixed at 0.99 in order to compliance with Mnih's paper [4]. α would be assigned with value from 0.0005 up to 0.004 since too small α would require much time to converge and to reach the target average reward, likewise if α is too large, the loss function is highly unlikely to be minimized. As for $sample\ size$, since one episode generally contains 70 to 120 steps, so a reasonable $sample\ size$ could round the number of steps per episode, so author of this paper set the range of $sample\ size$ from 20 to 80. Next is the Update Step C, which cannot be too small since Target Model would then be updated frequently, hence impair the purpose of reducing overestimations. As for $Epsilon\ decay$, after several trials of different value, author found that the reasonable value range of it would be from 100 to 300, and higher $Epsilon\ decay$ means agent would explore more.

III. Experiments

A. Experiment One: DQN Model

The author used a set of hyper-parameters: $\alpha = 0.0015$, $\gamma = 0.99$, $Epsilon\ Decay = 150$, $sample\ size = 50$, $Update\ Step = 2$, to train the DQN model. And the resulting reward per episode and average reward are shown in Fig.2. As illustrated in Fig.2, the average reward of last 100 episode reach the target at about 350 episode, and according to the time information the author stored at .txt file, whole training used about 19 min. Although the DQN model finally reach the target, the reward per episode is very volatile, leading to a relatively low average reward over all episodes as illustrated by green horizontal line.

B. Experiment Two: Hyper-parameter α

The author then try to use the DQN model to test a set of α while remain other hyper-parameters unchanged. The value of other fixed hyper-parameters are: $Epsilon\ Decay = 150$, $Sample = 30$, $Update = 1$.

We can observe that the $\alpha = 0.001$ reach the target using less than 300 episodes, while $\alpha = 0.0025$ using more than 500 episodes. Also, a slightly change in α from 0.001 to 0.0005 makes the episodes need for reaching target much longer. This phenomenon demonstrates that α should not be too large or too small to get decent reward in reasonable time. Thus, hyper-parameter tuning is very important.

C. Experiment Three: Other Hyper-parameters

Likewise, this paper trained a set of different hyper-parameters, including Epsilon Decay, Sample Size, Update Step. In each experiment, only one parameter has a range of value while value of other hyper-parameters remained unchanged. The results are shown in Fig.3-6, note that one line in a figure only represents one experiments, thus if hyper-parameter has a value which ourperforms others, it is not guarantee to be better than other value due to the randomness from environment and ϵ-greedy algorithm.

D. Experiment Four: Test Hyper-parameters

The forth experiment has an interesting result. When author picked the value of hyper-parameter which requires less episodes than others, like $\alpha = 0.001$ in Fig.3, Sample Size $= 50$ in Fig.5 and Update Step $= 5$ in Fig.6, then tested them using their model weights. Author later found that, as shown in Fig.7-10, the average reward of these model actually performed poorly, with reward around 100. Yet when author picked the value of parameters which requires more episodes, like Epsilon Decay $= 200$ in Fig.4, and tested it, the average reward of this model, as shown in Fig.10 surprisingly above 200! This phenomenon somehow show that an algorithm requiring more time is not necessarily a bad one.

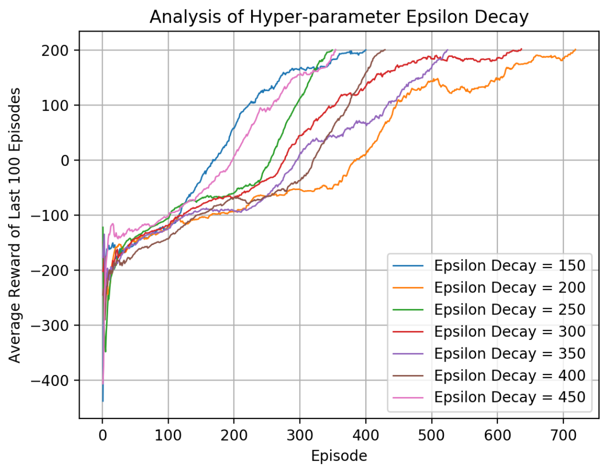


Fig. 4: Analysis of Hyper-parameter: Epsilon decay

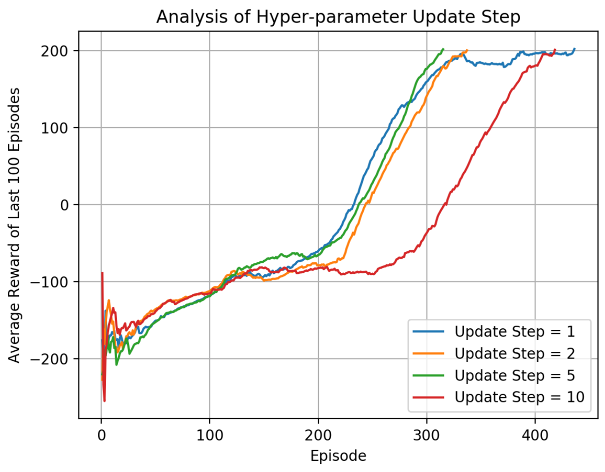


Fig. 6: Analysis of Hyper-parameter: Update Step

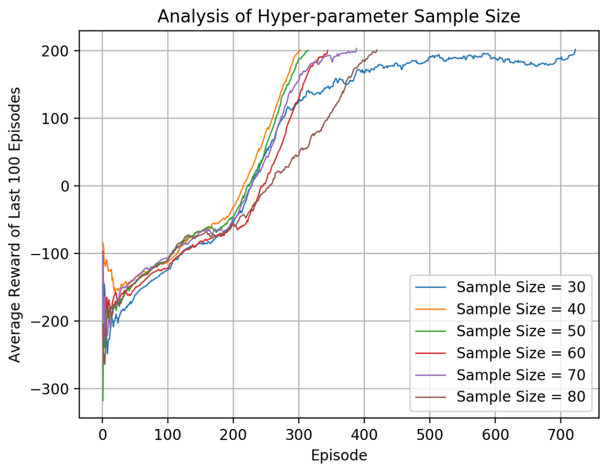


Fig. 5: Analysis of Hyper-parameter: Sample Size

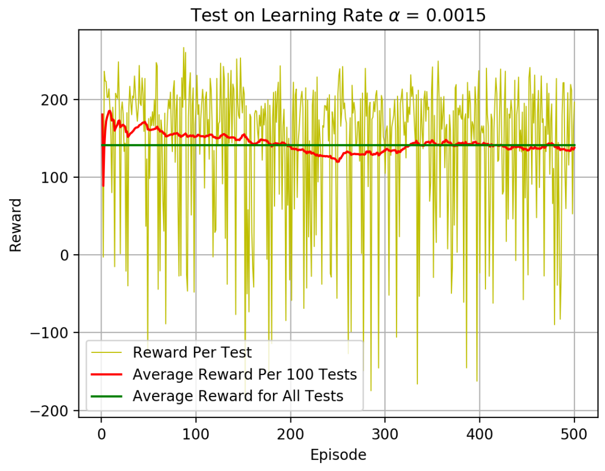


Fig. 7: Test model with $\alpha = 0.0015$

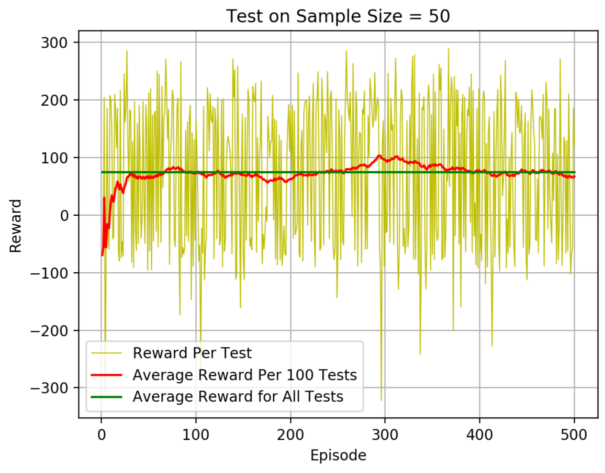


Fig. 8: Test model with sample size = 50

E. Experiment Five: DDQN method

The DDQN method's selection of actions changes from DQN's max to DDQN's argmax, only a very little modification is needed in DQN model to become DDQN.

$$Y_t^{DoubleQ} = R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\text{argmax}} Q(S_{t+1}, a; \boldsymbol{\theta}_t)); \boldsymbol{\theta}'_t)$$

After several trial, author managed to build a DDQN model which could reach the target reward but it needs a really long time. As shown in Fig.11, it takes almost 2500+ episodes to reach the objective bar. Yet when author tested on weights saved from DDQN model, it actually performs very well, average reward is above the 200 reward bar, as demonstrated in Fig.12. As Hasselt's paper [2] shown, DDQN alleviates the ubiquitous overestimation problem and leads to better performance.

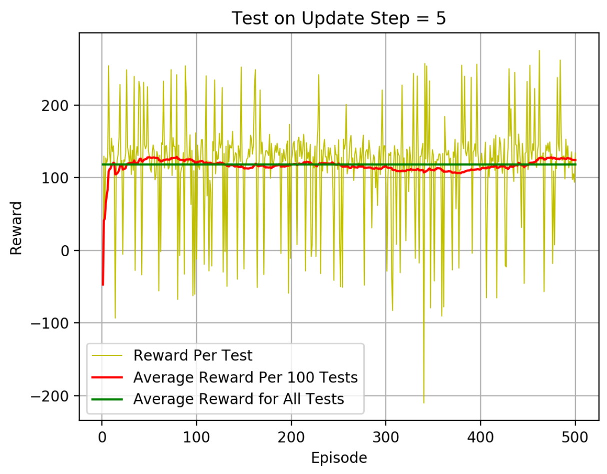


Fig. 9: Test model with update step = 5

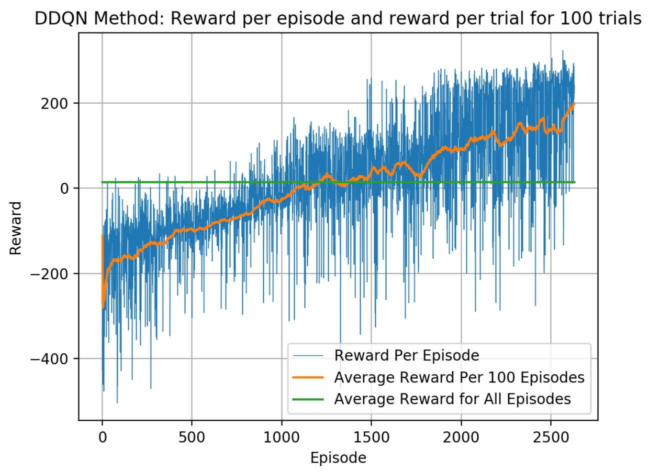


Fig. 11: DDQN method: reward and average reward

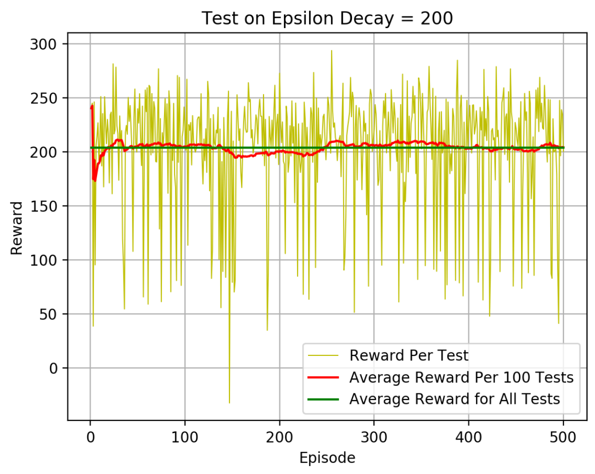


Fig. 10: Test model with epsilon decay = 200

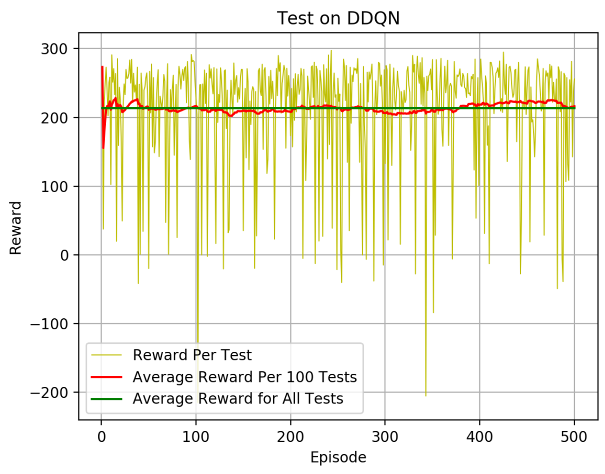


Fig. 12: Test on DDQN model

IV. Problems encountered

- When author first implemented DQN model, author assigned discounted rate γ with value of 0.59, which never converge even over $2k$ episodes even for all reasonable learning rate α. Then author read Mnih [4] carefully and fond that γ should always be 0.99 in these cases, and then everything worked.
- In "Human-level control through deep reinforcement learning", Mnih et al. didn't specify the size of buffer, when author first implemented their algorithm, buffer size was assigned with value of 3000, and the average reward also never reach the 200 bar even after many times of hyper-parameters value tuning. Then author realized that the size of 3000 was obviously not enough, since each tuple stored in buffer represents only one step within one episode, and generally there are 70 to 100 steps in one episode, so buffer of size 3000 would only remember last 30 episodes which is clearly not enough and would impair the purpose of making good use of previous experience.

V. Conclusion

This paper shows how DQN, combination of deep learning techniques and reinforcement learning algorithm, can efficiently solve challenging RL assignments including Lunar Lander in reasonable time. Tuning hyper-parameters such as α, ϵ is also an indispensable part for DQN to play its role. Overestimation is a prevalent problem in RL context, yet DDQN is a powerful algorithm which could reduce this problem and leads to better performance.

References

[1] K. K. Volodymyr Mnih, "Playing atari with deep reinforcement learning," 2015.
[2] D. S. Hado van Hasselt, Arthur Guez, "Deep reinforcement learning with double q-learning," 2016.
[3] C. J. Watkins and P. D. Q-learning, *Machine learning*, 1992.
[4] K. K. Volodymyr Mnih, "Human-level control through deep reinforcement learning."