Double Deep Q Learning (DDQN) applied on Lunar Lander System

Hongyu Zhou[1]

[1] hzhou311@gatech.edu; *Georgia Institute of Technology*

Introduction

In the current study, the environment from OpenAI Gym named LunarLander-v2 was applied to evaluate the reinforcement algorithm for continuous states problem. Specifically, Double Deep Q Learning was applied in the current project, and a remarkable result was achieved for the agent. Overall, for 100 testing trials, an average score (total rewards in an episode) of 210.3581 is accomplished, indicates the problem was successfully solved by the agent as good as the human-level control.

Investigation of Lunar Lander environments

The state space of Lunar Lander environment is eight dimensions. The first two dimensions indicate the coordinate of the lander. Next two dimensions include the velocity of the lander. After that, two dimensions are used to describe the angle of the lander and corresponding angular momentum. Last two dimensions indicated the contact of the leg with the ground. In the environment, the reward is calculated based on the states. For a given state, first I evaluate what's the 'shape' of the state based on velocity, leg contact, and coordinate as the following equation.

$$Shape = 10 * (D_6 + D_7) - 100 * (D_0^2 + D_1^2 + D_2^2 + D_3^2 + D_4)$$

The update rewards for the given action is the current 'shape' minus previous shape with some additional penalties on fuels. Lastly, if the lander successfully landed on the ground in certain steps, an addition +100 rewards was added otherwise -100 rewards. The lander has four actions available each step. The scores evaluated is the total rewards for an episode. The problem is considered as solved when the score is higher than 200.

Methods

The section of Methods is separated into three parts. Begin with a brief introduction to the concept of reinforcement learning, then the algorithm of the Deep Q Learning (DQN) and Double Deep Q Learning (DDQN) is briefly discussed. In the current study, DDQN is applied to solve the lunar lander problem.

Reinforcement Learning

The reinforcement learning algorithm is attempted to developed an agent which can interaction with environment overtime, and to maximum the expectation long-term return. At each time step, agent observed a state, and selected an action followed a policy, transited to the next state and received a scalar reward[2]. The object is to identify the optimal policy, which can be achieved by solving the Bellman Equation by dynamic programming (value/policy iteration) if the system model is available. Otherwise, TD Learning, SARSA, and Q Learning is available to solve the problem without model information, which is called model free learning. Among them, Q Learning is widely applied as an off-policy learning algorithm. The reinforcement learning problem can be reformulated as MDP when the environment satisfies the markov property. Although the above problem is defined for discrete state, it's relative easy to expended to continuous or infinite space using function approximations.

Deep Q Learning (DQN)

In 2015, a paper published by Mnih et al[1]. demonstrated Deep Q Learning methods, which ignited the field of deep reinforcement learning.

Before DQN, value function is hard to be predicted by non-linear network which can cause the RL to be unstable. DQN improved the field of Deep RL by using target network and experience replay to stabilize the training processes. Unlike some algorithm, DQN has no limitations on domain knowledge, and can be easily applied on many reinforcement learning problem (like Atari games). The algorithm stated in the original paper is referenced here[1].

```
Algorithm 1: deep Q-learning with experience replay.
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 $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$
    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
```

Algorithm 1: Deep Q Learning, adapted from Mnih et al[3].

Double Deep Q Learning (DDQN)

DQN use the same value to select and evaluate the action, which could possibly lead to over-optimistic problem[2]. One way to address this is proposed by Van et al[3], named Double Deep Q Learning (DDQN). The difference between DQN and DDQN is tiny, adopting the online network to evaluate the best action, but applying the target network to evaluate the value function for that actions. Specifically, the difference comes from y_j as the above equations shows, where Θ is for online network, Θ^- is for target network.

$$\boldsymbol{y_j = r_{j+1} + \gamma \hat{Q}(s_{t+1}, argmax_a Q(s_{t+1}, a; \Theta); \Theta^-)}$$

DDQN can find better policies in Atari games[3]. And therefore, in the current study, DDQN is implemented as shown in github source code.

Results

Agent training until meet criterion

The DDQN agent applied in the current study includes six different hyper parameters. Those hyper parameters includes replay memory length, mini-batch size, learning rate, epsilon decay rate, discount factor (gamma) and target neural network weights update frequency. After thorough tuning as demonstrating in the discussion section, those parameters are setting as 100,000 memory length, 30 batch size, 0.0003 learning rate, 0.993 epsilon, gamma 0.99 and 500 steps update frequency. Besides those parameters, the criterion for determining the agent was fully trained for best performance was also need to set. Here, because the problem is considered to be solved only when the average score for the last 100 trials is above 200. I also apply this criterion for training purpose.

During training, a score array is maintained for last 100 scores, and at each episode, the average score is printed (If the current episode is less than 100, then just average the past episodes). If the average score for last 100 episodes is greater than 215. Then, the agent is finished training and ready to be tested.

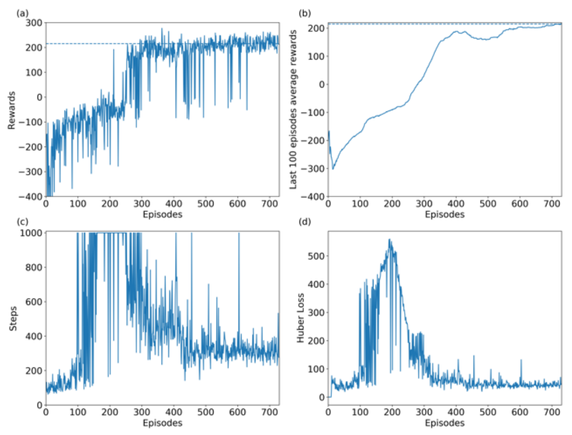

Figure 1: Training Agent results until meet the criterion (a) Scores for each episode (b) Last 100 episodes average score (c) steps for each episode (d) Huber Loss for each episode

The reason for using average last 100 episodes score instead of directly using current score is for validation purpose. Only one score for the current episode is less significant to have some statistical meaning. Instead, average score value for last 100 episodes is much more statistical significant, and the trends are also easier to be captured as shown in Figure 1.

Per episode and Average score during training

Figure 1 (a) and (b) indicates the score for each episode and the average score for last 100 episodes. 1(a) has some fluctuations, but it's obvious that after 400 episodes, the average score is more than 200. 1(b) is clearer. During training, the average reward is continuous increasing until meet the finishing criterion (>215). The total training episode is 729, with the average score for last 100 episodes (629-729) is 216.1472. The improvement from the initial average ~-250 score to the last average 216 score is significant, indicates DDQN is able to solve the problem.

Optimization Loss during training

Anther interesting observation is the steps in each episode and the Huber loss. For both the training processes and parameter tuning processes (will discussed later). In order to reach the >200 scores, the agent must need to pass the stage that performing 1000 steps without landing. And after this stage, the agent will start gradually reach the ground. If the hyper parameters are not well tuned, sometimes the agent will continue 1000 steps without reach the ground. The Huber loss express the trends that decreasing after increasing which can indicate the target network is reaching its' optimal value.

100 trials results for trained agent

After the agent is fully trained, the network is only used to predict action, and each time, the best action is selected to see the scores in each episodes.

The results are remarkable as shown in Figure2. No trial in the 100 episode received a negative rewards and most of them can get a score around 200. The average scores for the 100 episode is 210.3581, which can be considered as problem solved. This results indicate our agent was able to solve the problem successfully.

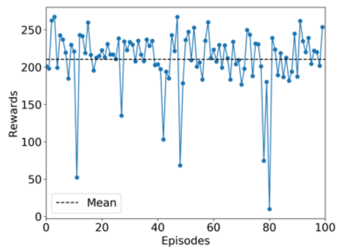

Figure 2: Agent tested in 100 trials

Discussion

The current section will be separated into three subsections. First, I will further examine the convergence of the algorithm when trained for a long time (10, 000 episodes). Second, thoroughly discussion about the hyper parameters tuning applied in the current model. Third, short discussion about other issues concerned.

Convergence of the training

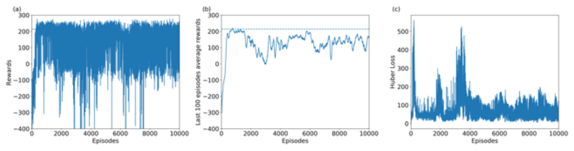

Figure 3: Training Agent in 10,000 episode (a) Scores for each episode (b) Last 100 episodes' average score (c) Huber Loss for each episode

Although from the results part, it shows after 729 episodes, the agent was fully trained to solve the problem with average 210 scores in 100 trials. However, it would be interesting to know whether or not the algorithm can be converged to an upper bound if we continue training. Nevertheless, the results show an opposite direction. The average score for the last 100 trails tends to be fluctuated. After reached the highest value, the average score will begin to oscillate, first go down and then increase again, then decrease and so forth as shown in Figure 3(b). The Huber Loss also shows the same trends, the loss values increase then decrease again and so on. One possible reason is the bad score in the latter episodes could adjust the well-trained network to the wrong way and decrease its' performance. This may further lead to more bad scores and worse performance until the good score increase the performance again. Therefore, the oscillation problem can be the result of constant learning rate, and the higher the learning rate, the oscillation problem would be more significant. Parameter tuning results also support this idea. From the above discussion, we know the longer training doesn't necessary lead to better performance. A good criterion to finish training is necessary for a DQN model.

Parameters Tuning

Parameters tuning is always an essential part for machine learning and reinforcement learning problem. In the current model, six parameters needed to be tuned. Even each parameter only considers 10 possibilities, the whole possibilities would be 1000000, which is incredibly computational cost. However, if each parameter is adjusted by itself, the correlation between different parameters would be ignored. In order to address this problem, the parameters correlated with each other will be adjusted together, including replay memory length with batch size, learning rates with epsilon. The discount factor and network update frequency is adjusted by themselves. Overall, around ~500 different parameters combinations simulations were tested and shown in the following parts.

Replay memory Length and Batch size

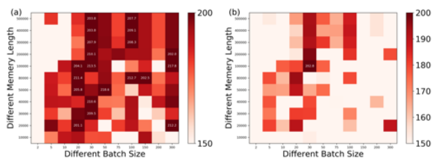

Figure 4: Replay Memory Length and Batch Size Tuning (a) Maximum average last 200 episodes value during training (b) The last 500 episodes average value

The first parameters tuned is replay memory length and batch size. Those parameters can affect the running time and training results. The parameter ranges tuning for those two parameters are from 10000 to 500000 for replay memory length, and from 2 to 300 for mini batch size. The ranges for the tuning is selected to be widely cover all the possible combinations. Total 11*11=121 tuning training are conducted.

As discussed before, for a given hyper parameter, the performance of agent could be oscillated. Therefore, the results are evaluated in two aspects, both from the highest performance and the stability as shown in Figure 4 (a) and (b). The results from Figure 4 indicates the larger batch size or memory length do not necessary lead to better performance. Instead, batch size 30 provides the best performance with all range of replay memory length. Considering the computational cost, the highest performance from Figure 4a, and the stability from 4b, batch size 30 with memory length 100000 were selected.

Learning rates (α) and Epsilon

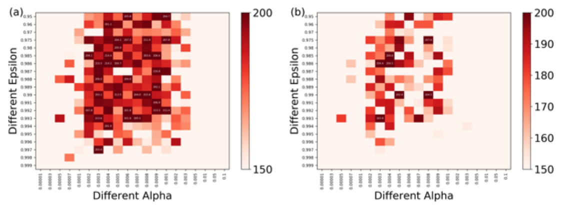

Figure 5: Learning rates and epsilon

Second tuning parameters is learning rates and epsilon. Some as Figure 4, the highest performance and the stability were examined during training. The range of alpha value is from 1e-5 to 0.1, and the range of epsilon is from 0.999 to 0.95. The epsilon controls the exploration-exploitation dilemmas, and alpha controls the speed of the learning. From figure 5, it indicates that the small alpha value and the large alpha value will both lead to terrible results and small epsilon value will also lead to bad results. Detail trends for alpha value 0.0003 and 0.003 are shown in Figure 6.

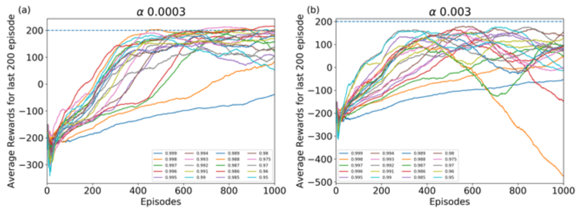

Figure 6: (a) Tuning results in alpha 0.0003 and (b) alpha 0.003

It's clear that with larger alpha value, the results will have higher fluctuation as discussed before. Smaller alpha value lead to a slower learning speed which possibly cannot solve the problem in 1000 episode. Smaller epsilon value also leads to slower training because the exploration is higher compared with large epsilon value.

Discount Factor (γ)

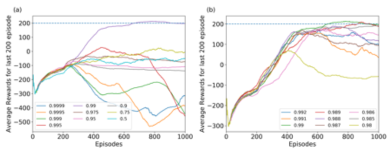

Figure 7: Discount factor tuning from different ranges (a) 0.9999 to 0.5 (b) 0.98 to 0.992

After finish tuning alpha and epsilon, the tuned parameters are fixed and the impacts of discount factor are investigated in Figure 7. First, I examine a large range of gamma, range from 0.9999 to 0.5. Interestingly, the gamma has significant impacts on the overall average scores. Both larger and smaller gamma will lead to bad scores. Therefore, I carefully examine the smaller range of gamma from 0.98 to 0.992. The results indicate 0.99 is the best choice for gamma value, and applied in the current study.

Target Network Update Frequency

Update frequency has less impacts than other parameters, as shown in Figure 8. The frequency ranged from 300-1000 all provides the good results. Only frequency less than 100 will lead to bad results overall. The frequency chosen in current study is the one with the best performance, frequency 500.

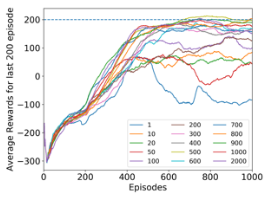

Figure 8: update frequency tuning

Other Issues

Some other issues needed to be concerned here including the vectorization. For each batch fitting, the setup need to be vectorized well in order to obtain higher speed (significant faster about 10 times). Only after vectorization, the computational cost for ~500 tunings are affordable. Another issue is the randomizations, even same hyper parameters are applied, the results can be totally different. In order to address this, the numpy, random and the environment seed is set before training or tuning in order to be reproducible.

Conclusions

In the current study, DDQN was applied and successfully solve the lunar lander problem with 210.3581 average scores in 100 testing trials. The hyper parameters are fully tuned through ~500 tuning simulations. Overall, DDQN is a suitable algorithm for solving complex reinforcement learning problem without any limitations on the domain knowledge.

References

[1]: Mnih, Volodymyr, et al. *Nature* 518.7540 (2015): 529.

[2]: Li, Yuxi. *arXiv:1701.07274* (2017).

[3]: Van Hasselt, Hado, Arthur Guez, and David Silver. *AAAI*. Vol. 16. 2016