OpenAI LunarLander Solved with Deep Q Learning

James Corwell

Git Commit ID:
`38ebb400ea3108398c561`
`0c03ec386d1eb954393`

***Abstract*—Deep Q Networks (DQN) are an extremely promising form of Deep Reinforcement Algorithm that are being used to efficiently solve a wide array of tasks. In this paper a DQN is used to solve the LunarLander-v2 environment from OpenAI gym.**

I. Introduction

The Lunar Lander problem is a reinforcement learning platform that is publicly available through OpenAI. The task is to get the agent to learn how to land an aircraft without it crashing. The agent has four actions: [nothing, thruster left, thrust right, thruster down], as well as 8 state variables. The agent is considered successfully trained with it can achieve a score >= 200 for 100 episodes of the LunarLander.

While initially researching this problem, it became clear that deep reinforcement learning was required. The open source solutions typically used one of the following algorithms;

- Deep Q Learning, (DQN, DDQN, etc.)
- Policy Gradients (REINFORCE)
- Actor-Critic Algorithm

After researching the problem, it seemed that Deep Q Learning (DQN) would be an interesting way to solve this problem, as Q Learning is already a familiar Reinforcement Learning topic. While traditional Q-Learning relies on developing a Q-table for every possible action at a given state, a DQN uses a Neural Network to predict the best Q values for an action based on a given state. . The original algorithm was developed by DeepMind (7) (Appendix II).

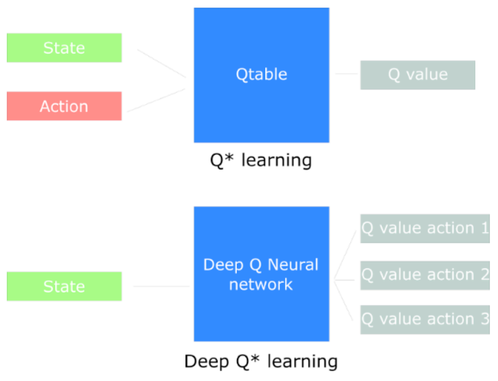


Fig 1: Q Learning vs DQN (1)

II. Methods and Implementation

There are a couple of things in particular that differentiate a DQN from traditional Q learning, namely the Replay Memory. This is essentially a memory store for our agent, where the agent then randomly pulls samples from to learn from. We store these past experiences as a tuple of the shape (action, state, reward, next state, done). To represent this, an entire class for handling and storing memories is designed.

We also have to implement a class for our Neural Network. PyTorch makes it very easy for us to design a basic neural network (2). The neural network was designed with 2 hidden layers -- any less and it's not really deep learning. The number of nodes is designed with 8 input nodes (number of states) as well as 4 output nodes (number of actions) as is customary in designing neural networks. The Hidden layers were both designed with 64 nodes and ReLU activation potential.

The DQN agent is a smart agent has the following abilities:

- Make an action
- Learn (update NN)
- Save a memory (Replay Memory)
- Recall a Memory (Replay Memory)

The Training process that the agent goes through in an environment is as designed in most basic terms as:

```
For every episode: {
For every timestep:
{
    *Agent acts according to state & epsilon
    *Update environment according to action
    *Store memory / Learn from action
    *Terminate when landed/crashed
    }
    *Store Episode Reward
    *Decay Epsilon
    Repeat until agent is 'smart'
    }
```

A 'trick' to enhance the training process is to set the maximum time an agent is allowed to exist in an episode. If it goes around flying all day in circles it is accumulating a large reward deficit. The agent should stop training and begin a new trial. As a human cognitive connection – one wouldn't want to learn from an outlier experience either, and should tune it out.

III. Results

The results were very positive. The agent was able to be considered trained at 668 episodes, when supplied with the hyperparameters from the top solution on OpenAI (Appendix1)(5).

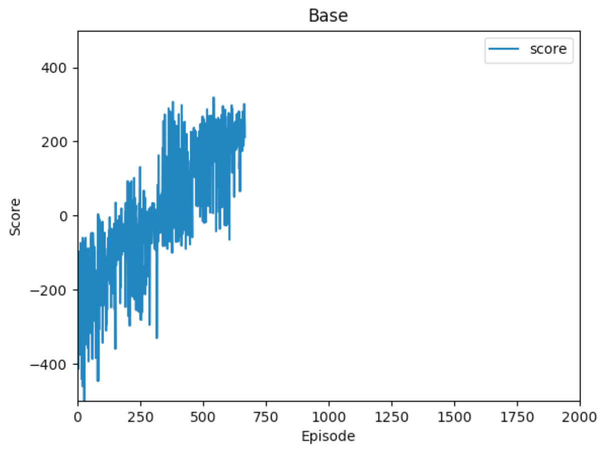


Fig 2. Base learning curve

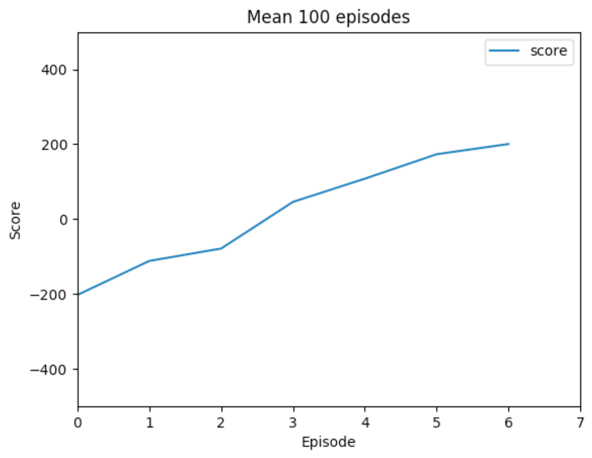


Fig 3. Showing the mean of learning per 100 episodes. Shows very smooth learning.

After we have established that the agent performs well, we can look at the effects of learning based off of varied hyperparameter values.

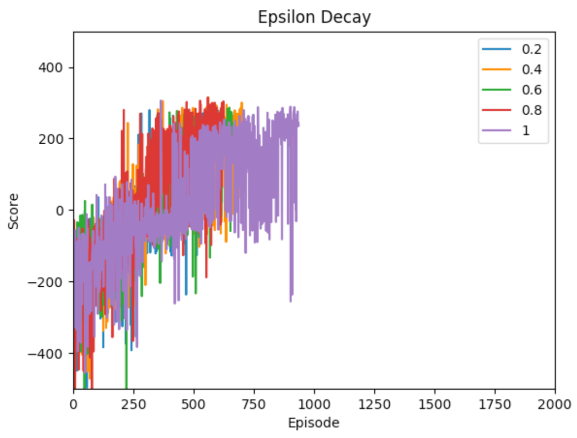


Fig 4. Varied Epsilon Decay.

Figure 3 shows that the Epsilon Decay hyperparameter isn't too important in determining the convergence of our model. For every value, the DQN converged within 1000 iterations.

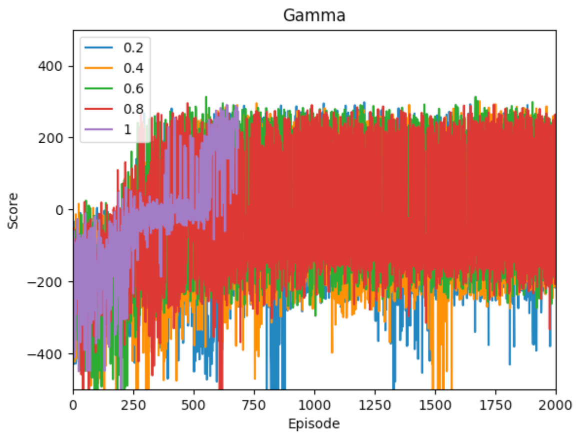


Fig 5: Varied Gamma

Figure 4 shows that small gammas are extremely unreliable, quickly learning to oscillate rapidly about 0.

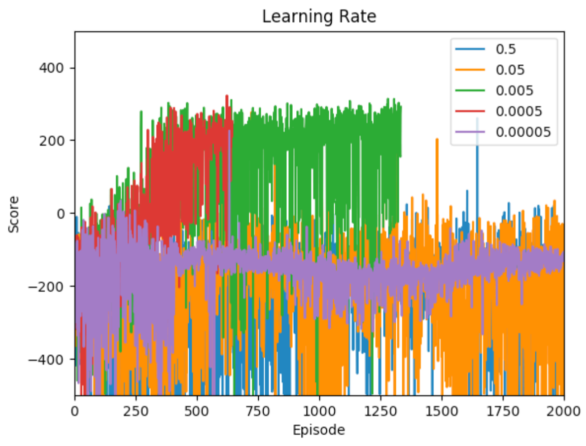


Fig 6. Varied Learning Rate

A varied learning rate produced interesting results in that it shows that both small (<0.0005) and large (>0.05) give poor solutions, while larger Learning Rates have a wide range of score. This shows that we have made a decent case that the learning rate should be within the magnitude of .00005 and .0005.

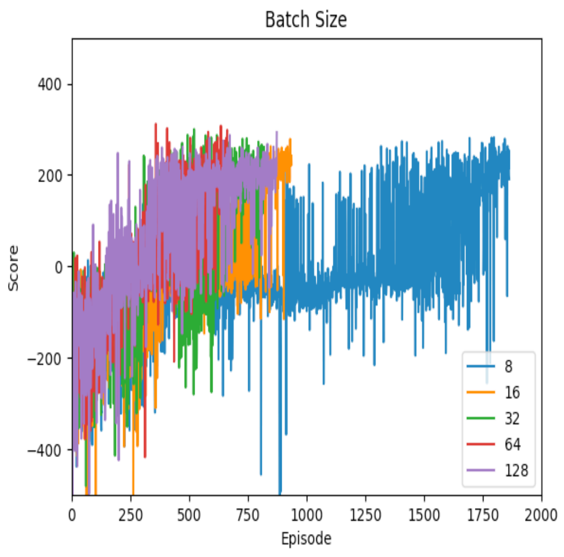
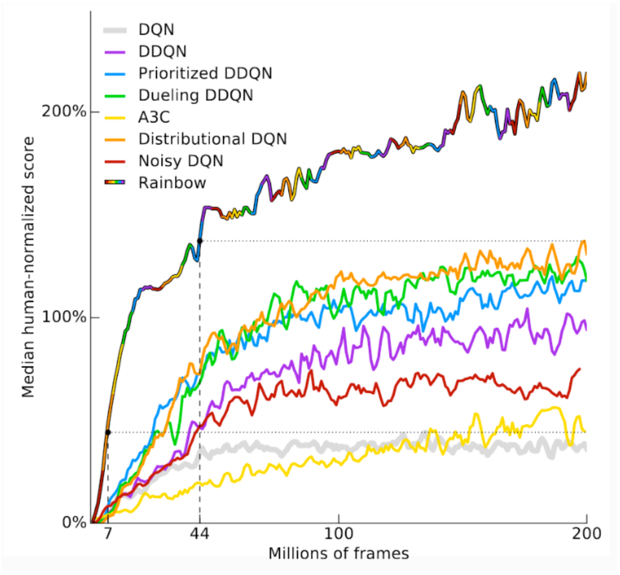


Fig 6. Varied Batch Size

Batch size was largely similar in results, though a batch size of 8 is obviously too small.

IV. Conclusion/further

DQN has shown itself extremely promising in the realm of functional deep reinforcement learning. With more time, I would have tried tom implement further types of DQN. DeepMind released the following graph showing how the strength of other variants of the DQN algorithm in reinforcement learning tasks:



Plot depicting efficacy of different variants of DQN (2)

It would be interesting to examine different buffer sizes in future experiments as well. A new paper from Zhang and Sutton details that while using 10^6 is standard practice when selecting a buffer size, this isn't necessarily the best value for this parameter. They offer several ideas to determine this. (6)

There were a couple notable challenges in the generation of a DQN agent. Initially I tried to implement the REINFORCE algorithm, but found it too slow at higher iterations, and eventually gave up on trying to get it to converge. Also, Deep Learning frameworks are still young and developing, the most popular being Tensorflow (developed by Google) and PyTorch (developed by Facebook). Tensorflow has a large syntax barrier, but a larger community behind it. PyTorch is more 'pythonic', but still young and has less of a community.

When tuning for hyperparameters it's difficult to get a convergent system by pure experimentation, as the graphs show. To do this would require many trials and much time, hence the reliance on an outside parameter to develop the base case. Potentially a grid search would be ideal to determine parameters, but on a slow machine this difficult.

V. References

[1] HTTPS://MEDIUM.FREECODECAMP.ORG/AN-INTRODUCTION-TO-DEEP-Q-LEARNING-LETS-PLAY-DOOM-54D02D8017D8

[2] HTTPS://PYTORCH.ORG/TUTORIALS/BEGINNER/BLITZ/NEURAL_NETWORKS_TUTORIAL.HTML

[3] HTTPS://ARXIV.ORG/PDF/1710.02298.PDF

[4] HTTPS://ARXIV.ORG/PDF/1712.01275.PDF

[5] HTTPS://GITHUB.COM/CPOW-89/EXTENDED-DEEP-Q-LEARNING-FOR-OPEN-AI-GYM-ENVIRONMENTS/BLOB/MASTER/CONFIGS/LUNAR_LANDER_V2.JSON

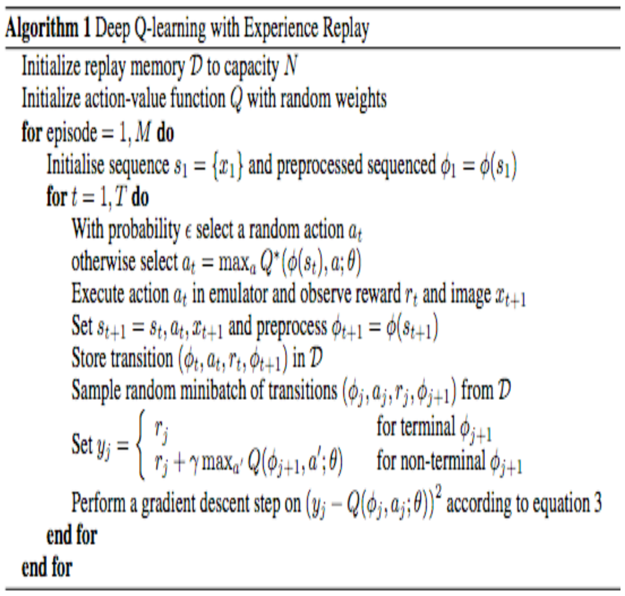
[6] https://arxiv.org/pdf/1712.01275.pdf

[7] https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

VI. Appendix

Training Episodes	***2000***	***Gamma***	***.99***
Max Time to Train	***1000***	***Tau***	***.001***
Epsilon initial	***1.0***	***Learning Rate***	***.00005***
Epsilon end	***.1***	***Update Interval***	***4***
Epsilon Decay	***.995***	***Batch Size***	***64***
Buffer Size	***100000***		

I. Table showing the parameters of the base case. Initially used from the #1 set from the #1 solution to LunarLander-v2 on OpenAI gym.(5)



```
Algorithm 1 Deep Q-learning with Experience Replay
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode = 1, $M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        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 $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
    end for
end for
```

II. Algorithm for Deep Q Learning with replay buffer from original DeepMind paper (7)

Various Terminology:

The following terminology is commonly used, but not well defined, and outside of the scope of Reinforcement Learning lectures. This is in no way comprehensive..

Batch-Size: How many samples the neural network will see before updating its weights.
Epsilon Greedy Exploration: When an action is selected in training, it is chosen as the action with the highest q-value or a random action. Choosing between these is random and based on the value of epsilon. During the course of training,

initially lots of random action is taken (exploration) and towards the end of training maximal q-value actions are taken (exploitation)
Replay Memory: stores the transitions that the agent observes, allowing us to reuse this data later. By sampling from it randomly, the transitions that build up a batch are decorrelated. It has been shown that this greatly stabilizes and improves the DQN training procedure.
Soft Update: Instead of updating the entire model(Hard Update), gradually adopt changes
Adam Optimizer: Adam stands for Adaptive Moment Estimation. It also calculates different learning rate. Adam works well in practice, is faster, and outperforms other techniques. Method used to update neural network weights.