Deep Q-Learning for Lunar Lander

Carlton Brady
University of Delaware
210 South College Ave. Newark, DE 19716
ctbrady@udel.edu

Abstract

Deep Q-learning and its variants are a family of deep reinforcement learning algorithms that are well suited for environments with a discrete action space. LunarLander-v2 is a relatively simple environment in OpenAI Gym that has a discrete action space and a continuous state space, so it can be solved using deep Q-learning. This project sought to implement deep Q-learning and double Q-learning in Python to solve LunarLander-v2. After implementation, different Q-network architectures and hyper parameters were tested with each algorithm to determine the combination that could solve the environment in the fewest training episodes. Each trained model was also evaluated by average score during testing. Though the number of training episodes required to converge on a good policy is partially dependent on exploration luck, deep Q-learning using the 8-64-128-4 architecture with 3 fully-connected layers had the best overall performance according to experiments.

1. Introduction

Reinforcement learning is an area of computer science that has received increasing attention in recent years due to the rise of deep learning and breakthrough achievements in AI involving deep reinforcement learning algorithms. Specifically, OpenAI Five and the DeepMind Alphastar project have shown that deep reinforcement learning at sufficient scale can be used to train agents that have super human performance in video games (Dota 2, StarCraft 2) that are much more complicated than the previous AI grand challenges of Chess or Go. OpenAI created Gym to help facilitate progress in deep reinforcement learning research by providing a set of standardized environments for testing and refining algorithms. The purpose of this project is to implement deep Q-networks to solve the LunarLander-v2 environment in Gym, then evaluate the effect that different hyper parameter and neural network architecture choices have on the number of training episodes required to solve the environment.

1.1. The LunarLander-v2 Environment

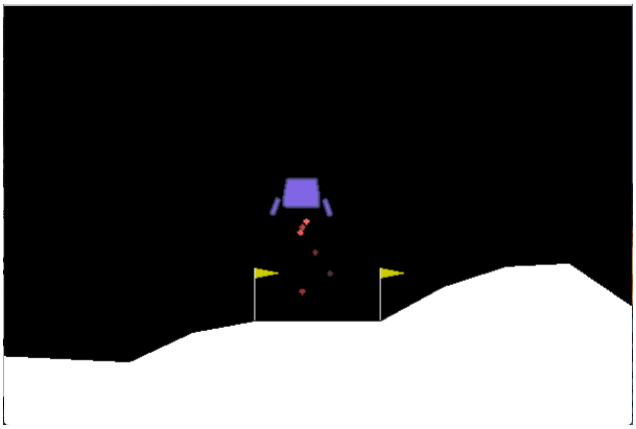

Figure 1: OpenAI Gym LunarLander-v2 Environment

In LunarLander-v2, the agent is tasked with guiding the landing craft to a gentle landing on the landing pad in between the 2 yellow flags in a short amount of time. For this project, the limit for each training episode was set to 1000 time steps. Training episodes terminate when the lander crashes, successfully lands, or the time step limit is reached. The environment has a discrete action space where the agent has 4 possible actions at each time step: do nothing (0), fire left thruster (1), fire bottom thruster (2), or fire right thruster (3). The state-space is continuous, represented by an 8-element vector containing the position, velocity, and acceleration of the lander in each of the 2 dimensions. The state vector also includes coordinates of the landing pad, which are always (0, 0) for this environment. The agent receives a reward at each time step as a single floating-point number, and rewards with larger magnitudes are given at the end of each training episode. Rewards can be positive or negative.

1.2. Deep Q-Learning

The code in this project was primarily based on the deep Q-learning algorithm detailed in [1] by Mnih, Kavukcuoglu, Silver, Graves, Antonoglou, Wierstra, and Riedmiller. In their paper, they applied deep Q-learning to seven Atari 2600 games and were able to train an agent to

super human performance on 3 of them.

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 ϵ 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

Algorithm 1: From [1], Deep Q-learning with Experience Replay by Mnih et al.

Two differences from [1] and this project are that they used images of the game as the state, and there was no epsilon decay. LunarLander-v2 provides a state vector which makes the task much easier than learning the value of the state based only on an image of the game.

The goal of deep Q-learning is to use a deep neural network to learn a function, *Q(s, a)*, that outputs the value of taking action *a* in state *s*. Given an accurate Q-function, an agent can take the action in each state with the largest Q-value, resulting in a policy that achieves good performance in the environment according to the reward function. The agent must record experiences in the environment to use as training data for the neural network that approximates the Q-function. Experiences are environment transitions stored in the following form: (s_t, a_t, r_t, s_{t+1}), where s_t is the state at time step *t*, a_t is the action taken, r_t is the reward received, and s_{t+1} is the state at *t+1*. Each environment transition experience is essentially 1 training example, and they are used in batches to performance training updates on the Q-network. The loss is computed between the actual reward observed by taking an action in a given state and the Q-value that the Q-network estimates for taking that action. An estimate of future reward discounted by *gamma* is also factored in for transitions that do not result in a terminal state. The agent takes random actions with probability *epsilon* to encourage exploration of the state space.

1.3. Epsilon Decay

Balancing exploration and exploitation is a core issue in reinforcement learning. Epsilon decay is a simple method for reducing the amount of exploration over time as an agent learns more. In the beginning of training, *epsilon* is set to 1.0 so the agent explores totally randomly. After each training update, *epsilon* is multiplied by a decay rate less than 1, so the probability of taking a random action decreases as training progresses. A minimum *epsilon* of 0.1 was also enforced for this project so that exploration would still happen with a small chance even after many training updates.

1.4. Experience Replay

As specified in Algorithm 1, experiences accumulate sequentially in replay memory. However, consecutive experiences are highly correlated so they are likely to contain similar information about the environment. Experience replay uses a randomly sampled batch of experiences to ensure that a wide range of uncorrelated environment transitions are used in each training update, making each update more effective. Additionally, using a random sample of experiences allows a particular experience to be used in more than one training update, which makes the algorithm more sample-efficient.

1.5. Double Q-Learning

The code for the second algorithm that was implemented in this project is based on [2] by Hasselt, Guez, and Silver. Double Q-learning uses an additional "target" neural network of the same architecture as the Q-network to help stabilize training and reduce Q-value overestimation.

Algorithm 1 : Double Q-learning (Hasselt et al., 2015)

Initialize primary network Q_θ, target network $Q_{\theta'}$, replay buffer $\mathcal{D}$, $\tau << 1$
for each iteration **do**
  for each environment step **do**
    Observe state s_t and select $a_t \sim \pi(a_t, s_t)$
    Execute a_t and observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$
  for each update step **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
    Compute target Q value:
      $Q^*(s_t, a_t) \approx r_t + \gamma\, Q_\theta(s_{t+1}, argmax_{a'} Q_{\theta'}(s_{t+1}, a'))$
    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
    Update target network parameters:
      $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$

Algorithm 2: From [2], Double Q-learning by Hasselt et al.

Double Q-learning separates action selection and action evaluation. The target network is used for action evaluation, while the regular Q-network is used for action selection. The target network parameters are updated according to a weighted update rule using an additional hyper parameter *tau*. *Tau* is a value less than 1 which specifies how much the regular Q-network parameters are weighted in the update rule. As a result, the target network parameters are updated with only a fraction of the magnitude that the regular Q-value parameters are, which reduces fluctuations in Q-value estimates and stabilizes training.

2. Experiments

A baseline Q-network architecture was chosen according to some existing solutions since it was known that such a small network could solve LunarLander-v2. The baseline network had 3 fully-connected layers, 8

inputs, 32 units in the 2nd layer, 64 units in the 3rd layer, and 4 outputs. The 8 inputs correspond to the 8 values in the environment state vector, and the 4 outputs correspond to the 4 possible actions in the discrete action space. 2 additional network architectures were tested that varied from the baseline network, one of which was wider than the baseline network, and another that was deeper than the baseline network. The wider network had 3 fully-connected layers, 8 inputs, 64 units in the 2nd layer, 128 units in the 3rd layer, and 4 outputs. The deeper network had 4 fully-connected layers, 8 inputs, 32 units in the 2nd layer, 64 units in the 3rd layer, 128 units in the 4th layer, and 4 outputs. All networks tested in this project were created using PyTorch and trained only using a 2017 MacBook Pro with a 3.1 GHz Dual-Core Intel Core i5 CPU.

To evaluate each network architecture, training runs were done for each of the two aforementioned Q-learning algorithms and various *epsilon* decay rate values. Training would terminate once the agent received an average score of 200 in the most recent 100 training episodes, as that is the criterion defined as solving LunarLander-v2. A fixed learning rate of 0.0001 was used since larger learning rates failed to converge to a policy that solved the environment. After training termination, models were tested on 100 episodes with *epsilon* set to 0 so no random actions were performed.

2.1. Results

A total of 18 training runs were performed, 6 for each neural network architecture, 3 for each architecture and algorithm combination. Only architecture, algorithm, and, and *epsilon* decay rate were varied. It was not possible to establish a strong statistical case for which architecture was best due to time constraints and the randomness inherent in the training from both the environment and agents' actions.

Architecture: 8-32-64-4			
Algorithm	**Deep Q-learning**	**Deep Q-learning**	**Deep Q-learning**
Initial epsilon	1.0	1.0	1.0
Minimum epsilon	0.1	0.1	0.1
Epsilon decay rate	0.99	0.999	0.995
Learning rate	0.0001	0.0001	0.0001
Gamma	0.99	0.99	0.99
Tau	1.0	1.0	1.0
Training episodes required to solve	**433**	841	1035
100 episode test score avg.	**226.68**	214.97	197.27

Figure 2: Baseline architecture with deep Q-learning results

The baseline architecture had its best deep Q-learning training run using the highest *epsilon* decay rate, solving the environment in 433 training episodes while achieving an average score of 226.68 on the 100 testing episodes.

Architecture: 8-64-128-4 (wider)			
Algorithm	**Deep Q-learning**	**Deep Q-learning**	**Deep Q-learning**
Initial epsilon	1.0	1.0	1.0
Min epsilon	0.1	0.1	0.1
Epsilon decay rate	0.99	0.999	0.995
Learning rate	0.0001	0.0001	0.0001
Gamma	0.99	0.99	0.99
Tau	1.0	1.0	1.0
Training episodes required to solve	464	**346**	1012
100 episode test score avg.	**256.57**	195.25	217.93

Figure 2.1: Wider architecture with deep Q-learning results

The wider architecture using deep Q-learning with the largest *epsilon* decay rate had the highest test score average out of all models that were trained, and it only took an impressively low 464 training episodes.

Architecture: 8-32-64-128-4 (deeper)			
Algorithm	**Deep Q-learning**	**Deep Q-learning**	**Deep Q-learning**
Initial epsilon	1.0	1.0	1.0
Min epsilon	0.1	0.1	0.1
Epsilon decay rate	0.99	0.999	0.995
Learning rate	0.0001	0.0001	0.0001
Gamma	0.99	0.99	0.99
Tau	1.0	1.0	1.0
Training episodes required to solve	1215	**238**	2501
100 episode test score avg.	186.74	169.82	**235.09**

Figure 2.2: Deeper architecture with deep Q-learning results

The deeper architecture with deep Q-learning showed massive variance in the number of training episodes required to solve the task. It appears to be more difficult to train without much of a testing performance benefit in LunarLander-v2. The environment is likely simple enough that additional layers are not needed or helpful.

Architecture: 8-32-64-4			
Algorithm	**Double Q-Learning**	**Double Q-Learning**	**Double Q-Learning**
Initial epsilon	1.0	1.0	1.0
Minimum epsilon	0.1	0.1	0.1
Epsilon decay rate	0.99	0.995	0.999
Learning rate	0.0001	0.0001	0.0001
Gamma	0.99	0.99	0.99
Tau	0.001	0.001	0.001
Training episodes required to solve	1148	708	**689**
100 episode test score avg.	176.82	**246.48**	202.2

Figure 2.3: Baseline architecture with double Q-learning results

The baseline architecture with double Q-learning achieved the second highest test score average of 246.48, but it required 708 training episodes.

Architecture: 8-64-128-4 (wider)			
Algorithm	**Double Q-Learning**	**Double Q-Learning**	**Double Q-Learning**
Initial epsilon	1.0	1.0	1.0
Minimum epsilon	0.1	0.1	0.1
Epsilon decay rate	0.99	0.995	0.999
Learning rate	0.0001	0.0001	0.0001
Gamma	0.99	0.99	0.99
Tau	0.001	0.001	0.001
Training episodes required to solve	**608**	643	616
100 episode test score avg.	197.26	209.52	**212.17**

Figure 2.4: Wider architecture with double Q-learning results

The wider architecture with double Q-learning was consistently able to solve the environment with roughly

600 training episodes, but the test score averages were underwhelming.

Architecture: 8-32-64-128-4 (deeper)			
Algorithm	**Double Q-Learning**	**Double Q-Learning**	**Double Q-Learning**
Initial epsilon	1.0	1.0	1.0
Minimum epsilon	0.1	0.1	0.1
Epsilon decay rate	0.99	0.995	0.999
Learning rate	0.0001	0.0001	0.0001
Gamma	0.99	0.99	0.99
Tau	0.001	0.001	0.001
Training episodes required to solve	**576**	1073	867
100 episode test score avg.	167.77	213.33	**245.38**

Figure 2.5: Deeper architecture with double Q-learning results

The deeper architecture with double Q-learning achieved the third highest test score average of 245.38, but it took 867 training episodes.

2.2. Conclusion

It is clear from the experiments in this project that LunarLander-v2 is simple enough to be solved by deep Q-learning and double Q-learning with small neural networks. Randomness of the agent's actions and environment starting states played a large role in determining how many training episodes were needed to converge on a good policy. Double Q-learning appears to reduce the effect of that randomness and stabilize training, resulting in a more consistent number of training episodes required to solve the environment. The volatility in training is illustrated well by a plot of training episodes and score in figure 3 below.

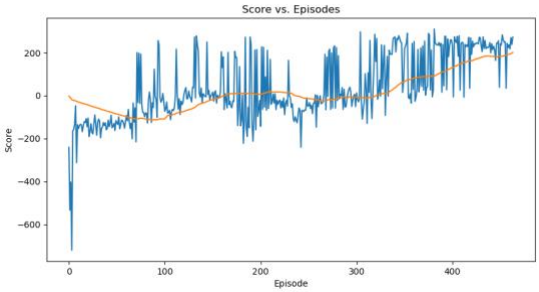

Figure 3: Training plot of the best performing architecture

It is difficult to draw conclusions about what the best architecture, algorithm, and hyper parameter combination really was due to limited experiments, randomness, and volatility in the training, but the wider architecture with deep Q-learning and a 0.99 *epsilon* decay rate subjectively had the best performance.

2.3 Q and A from the Presentation

Q: What was the best performing architecture?
A: The training run with the minimum number of training episodes to solve the task occurred using the 8-32-64-128-4 deeper architecture, and the model with the highest testing score was the 8-64-128-4 wider architecture. Although limited experiments were performed due to time constraints and there is randomness involved, the 8-64-128-4 wider architecture had the best combination of training episodes required and testing score across all experiments.

Q: What are the benefits of double Q-learning?
A: Double Q-learning reduces the Q-value overestimation that happens in traditional deep Q-learning. Q-value overestimation is a result of the bootstrapping nature of Q-learning, where estimates are learned from estimates. If Q-values are overestimated, the agent has a distorted view of the value of actions relative to one another, which can lead to unstable training and poor policy. This is shown in [2].

Q: Why not do a training update every few timesteps as opposed to every single time step?
A: Using the hardware I had for this project, training was not converging in a reasonable amount of time without doing more frequent training updates.

Q: Why not give a success rate metric in addition to average score to help illustrate model performance?
A: That would be a good addition. I did not think of it. Average score does give a more detailed view of agent performance, but it hides instances where the agent may be failing catastrophically. Using a score over 200 as a success and under 200 as failure, a success rate metric would help depict the robustness of an agent to different landing situations.

Q: How long did it take to reach a solved level of performance typically?
A: Using a MacBook Pro with 3.1 GHz Intel Core i5 CPU, training a single agent would usually complete in 30 minutes to 1 hour, depending on the size of the network and how many training episodes it took to converge on a good policy.

Q: How robust are the agents?
A: The best agents were very robust, and would never receive a reward of less than 200 in the 100 testing episodes.

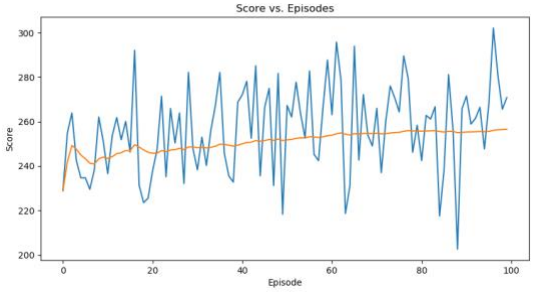

Figure 4: Testing plot of a robust agent

However, some agents had a good average score in testing, but it was achieved with mostly high scoring episodes interspersed with massive failures where the agent received a large negative reward.

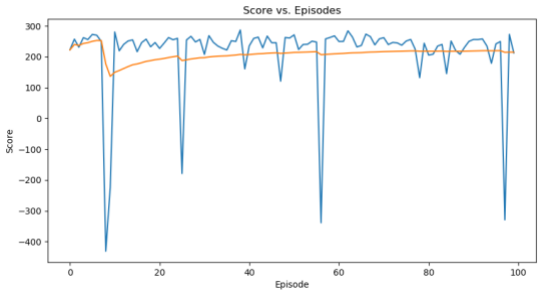

Figure 4.1: Testing plot of a non-robust agent

This is likely due to the agent not being robust to certain regions of the state space that it encountered during testing. This may have been caused by poor or unlucky exploration.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, & Martin Riedmiller. (2013). Playing Atari with Deep Reinforcement Learning.

[2] Hado van Hasselt, Arthur Guez, & David Silver. (2015). Deep Reinforcement Learning with Double Q-learning.