Using DQN to land on the moon

Joey Marten
Department of Computer Science
McGill University
Montreal, QC
joey.marten@mail.mcgill.ca

Abstract

We present an implementation of Deep Q Networks (DQN) to challenger OpenAI's Lunar Lander environment. Mnih et al. (2013) describe an algorithm that uses a deep neural network to approximate the Q values of a given continuous state. Training is aided by an experience replay buffer and an additional target network. We find that it performs well and learns quickly under the right hyper-parameters. The work was carried out entirely by a one-person team.

1 Introduction

Reinforcement learning attempts to train an agent to a specific environment by selecting an action depending on the current state and observing a reward. Many environments are simple to model with discrete states. However, many real-world applications use continuous state space representation. One successful solution to this has been to use bins to classify observations. Unfortunately, methods can be limited by the resolutions of the bins. With traditional Q-learning, observations are required to train the Q-values of each state-action pair.

RL has, for a long time, developed independently from multi-layer perceptrons. However, in the 2010s, many research papers started to demonstrate the power of neural networks in recognizing high level features from simple inputs. Labelled pictures were able to be identified with image recognition models such as AlexNet. The main challenge of using deep learning comes from the requirement of labelled data to train on. As a result, many online approaches to RL can't quite be used. Instead, the agent gathers data in stages by packaging states, actions, and rewards to train the underlying model periodically. The method lies somewhere in between online and offline. It's important to note that data collected this way is highly correlated. The input stored is a direct result of the previous states visited and the actions taken. This paper implements DQN which was designed to solve these issues. The performance of this method was tested on the Lunar Lander environment. The goal of this environment is to train an spacecraft to come to rest successfully on a specified landing pad. Although simple, this environment presents many interesting real-world extensions to the aeronautical and space industries. Further extensions to 3D, would be necessary but nonetheless, it provides the basis for future research and integration.

2 Background

Before delving into the details, some background knowledge is required. The following sections describe the environment and traditional Q learning.

2.1 Environment

OpenAI's Lunar Lander environment is a simplified 2D model for self-landing vehicles. The action space is discrete with 4 possible actions: do nothing, fire the left orientation engine, fire the

Submitted to 37th Conference on Neural Information Processing Systems (NeurIPS 2023). Do not distribute.

main engine, or fire the right orientation engine. Additionally, states are continuous with variables representing position, velocity, rotation, rotational velocity, and two supplementary booleans to indicate whether each leg is grounded. The reward function is somewhat abstract. Landing adds +100 while crashing results in -100. Each engine activation costs 0.3. The documentation indicates that a reward of 200 is considered solved. This reward function is sparse considering that the maximum number of steps is 1000.

2.2 Q-Learning

Q-Learning works by estimating the return from the current state and action under the current policy. This can be formulated as follows:

$$Q(s_t, a_t) = E[R_t|s_t, a_t] \tag{1}$$

Accordingly, actions are chosen greedily by choosing the action with the highest resulting Q value. The values can then be updated by assuming to use the optimal selections to maximize the current reward and the return of the next state.

$$Q^*(s, a) = E[r + max_{a'}Q(s', a')|s, a] \tag{2}$$

This methodology acts as a jumping-off point for DQN.

3 Methods

Deep neural networks allow for much more abstraction over data. By relying on past experience, an MLP can give accurate predictions for unseen data. As such, it works well on continuous input variables. Similar inputs can be mapped to similar outputs. This produces smooth results throughout where binning cannot. The main mechanism behind DQN is using past state transitions to train the model to estimate Q-Values. Past experience is stored in a replay buffer. Each sample includes the state, the action taken, the next state, the rewards, and a boolean indicating whether the episode terminated. It's easy to see how using only recent experience results in poor performance. A given transition is directly correlated with the previous state and action. As such, the network is trained on a batch sampled randomly from the entire buffer. Additionally, using a large enough buffer is critical to avoiding catastrophic forgetting.

Targets for the main network are aided by an auxiliary network as described by Mnih et al. (2015). For each sample in the batch, the target is calculated as follows:

$$y_i = r_i + \mathbb{1}(Terminal = 0) * \gamma * \max a' Q_{target}(s_{t+1}, a') \tag{3}$$

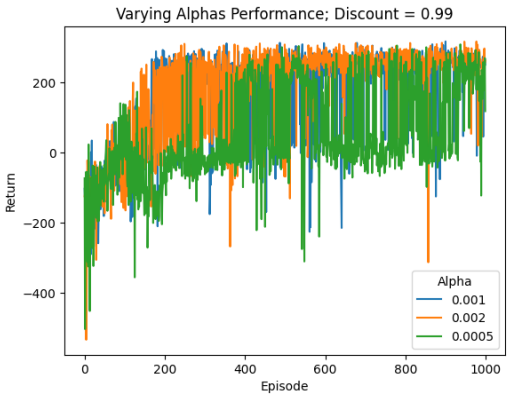
Gradient descent is performed on the difference between the target and the value predicted by the main networks. After a specified number of time steps, the main network's weights are copied to the secondary network.

To support exploration during training, the agent chooses actions using an ϵ-greedy algorithm. Additionally, the value of epsilon is decayed by multiplying with a coefficient until a minimum is reached. This results in high exploration at the beginning which decreases slowly towards the end.

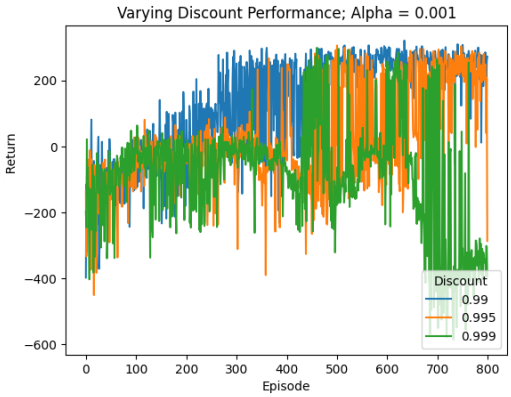
One important implementation detail is the model architecture and training. Initially, I used the action of a sample as an additional parameter to the input of the neural network. This resulted in slow training times. To solve this issue, I used only the sample as input but with 4 outputs (one per action). This greatly improved training times. Furthermore, I experimented with using Huber as the loss function. Training times were accelerated using a gradient tape method detailed by Sanz Ausin (2020).

4 Results

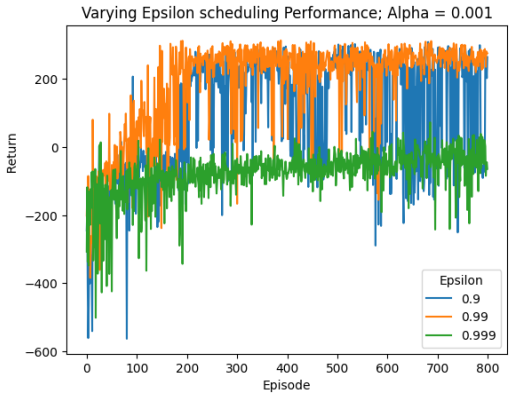
To test the algorithm's performance, I varied several hyper-parameters then plotted the episode returns and analyzed the convergence. Additionally, the results allowed me to tune the agent to the environment and use the best values for the video demonstration. Each test used a replay buffer with a size of 300000 transition states and 128 samples were used in each batch. The neural networks used



(a) Effects of discount



(b) Effects of discount



(c) Effects of discount epsilon

Figure 1: Results of tuning

3 hidden layers each with 64 nodes and ReLu activations. Furthermore, the target network's weights were updated every 1000 steps while the main model updated every 4 steps. The epsilon minimum was set to 0.01

To start, I varied the alpha between 0.001, 0.002, and 0.0005 using a discount of 0.99 while the epsilon was annealed with a factor of 0.99 each step as seen in figure 1a. Using 0.001 and 0.002 gave very similar results. They began to see good results by 200 episodes but converged around 500 episodes with a very small variance. On the other hand, 0.0005 was too small to converge within 1000 episodes. It did achieve several successful episodes starting from 400 but with a quite high variance.

Next, I chose to measure the effects of using different discount rates shown in figure 1b. The learning rate and epsilon decay factor for this experiment were 0.001 and 0.99 respectively. A discount of 0.99 results in faster convergence. We see that it starts to succeed after approximately 300 episodes and converges between 400 and 500 episodes. In contrast, the other epsilons perform similarly until 400 episodes. Using 0.995, the agent starts to land more consistently and slowly converges at 750 episodes. Interestingly, the higher discount of 0.999 begins to perform well but rapidly diverges and begins to perform worse.

Lastly, in figure 1c the value of ϵ was annealed at different rates. An epsilon of 0.99 performed the best. It converged faster and with lower variance than the other two. Using an epsilon of 0.9 learned at a similar pace, however, it had a high variance throughout. An epsilon of 0.999 did not converge to a valid solution but had a relatively low variance. It's possible that the agent got stuck in a local minimum.

5 Conclusion

In this work, we proved that DQN is an effective algorithm for complex control policies by solving the Lunar Lander environment. We presented the fundamental elements that make this model powerful such as the dual neural network and the experience replay buffer. Our agent managed to land successfully and consistently within 300 episodes of training when using an alpha of 0.001, a discount of 0.99, and an epsilon of 0.99.

6 Future Work

I believe expanding on the existing algorithm by finding or creating additional methods would make for interesting work. The lunar lander environment has very sparse rewards. As such, reward shaping could be a valid method to accelerate convergence. Additionally, I think a comparison to other algorithms based on the total time needed to train on this environment would produce informative results for practical applications.

References

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *arXiv e-prints*, Article arXiv:1312.5602 (Dec. 2013), arXiv:1312.5602 pages. https://doi.org/10.48550/arXiv.1312.5602 arXiv:1312.5602 [cs.LG]

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Kirkeby Fidjeland, Georg Ostrovski, Stig Petersen, Charlie Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature* 518 (2015), 529–533.

Markel Sanz Ausin. 2020. Introduction to Reinforcement Learning. Part 3: Q-Learning with Neural Networks, Algorithm DQN. (2020). https://markelsanz14.medium.com/introduction-to-reinforcement-learning-part-3-q-learning-with-neural-networks-algorithm-dqn-1e