Reinforcement Learning with Deep Q-Networks and Double Q-Learning

Wesley Liao
Georgia Institute of Technology
wliao63@gatech.edu

Abstract

Deep reinforcement learning is the practice of applying deep neural networks as a function approximation method for reinforcement learning problems. Function approximation is required when the state space or action space of a problem is very large or infinite, making tabular and exact methods infeasible. This report explores the Lunar Lander problem, which has a continuous state space and a discrete action space. The approach used will be both a straightforward Deep Q-Network as well as a Double Deep Q-Network. We will also examine the behavior and performance of the Deep Q-Network learner with respect to its various hyperparameters.

1. Introduction

1.1. Q-Learning

Q-Learning is a model-free reinforcement learning method that is based on the state-action value function $Q : S \times A \to \mathbb{R}$. The Q-value function aims to capture the total discounted future reward of taking some action a while in state s. The optimal policy is then derived by taking the action that maximizes the Q-value in every state. This formulation is captured in the Bellman Equation below.

$$Q(s, a) = r(s, a) + \gamma \max_{a'} Q(s', a') \tag{1}$$

To find the fixed-point solution, an iterative approach is usually taken where the Q-values are stored in a table and are updated from the previous iteration's Q-values using the Bellman Equation relationship. However, when the state space or action space become very large or infinite, this tabular method is no longer feasible.

1.2. Deep Q-Networks

Instead of maintaining the exact Q-values in a table, they can be approximated using function approximation. In other words, $Q(s, a)$ can be approximated by some function $f(s, a; \theta)$ where the θ represents the model parameters that minimize the error (usually mean squared error) between f and Q. This has many benefits over exact methods including tractability and generalization.

In the case where f is a neural network, it is called a Deep Q-Network (DQN) [3] [4]. The DQN learning rule uses a Temporal Difference (TD) approach where the error is the difference between the Q estimates of the current state and the Q estimates of the next state plus the current reward. This is also known as the TD(0) rule. The loss function is then defined as the expected mean square of those errors.

$$L(\theta) = \mathbb{E}[(r + \gamma \max_{a'} Q(s', a'; \theta) - Q(s, a; \theta))^2] \tag{2}$$

Although there is no guarantee of convergence for Q-learning with non-linear function approximation, there are stabilization techniques that have proven to be very effective empirically. The two techniques were used in this experiment are target networks and experience replay.

1.2.1 Target Networks

Instead of using the most recent Q-values in both evaluations in (2), the baseline $Q(s', a')$ can use an older version of the parameters that are updated less frequently.

$$L(\theta) = \mathbb{E}[(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta))^2] \tag{3}$$

The θ^- parameters are updated a regular frequencies but not at every iteration. This helps introduce stability in the learning process. A range of update frequencies for the target network were tested for this experiment and explored in the hyperparameters section.

1.2.2 Experience Replay

The training of the Deep Q-Network is through online supervised learning, where observations are provided to the network to do back-propagation on as the agent is interacting with the environment. One of the main assumptions of supervised learning is that each sample is independent and identically distributed. However, in the reinforcement learning scenario, the observation at each step is highly correlated to the previous observation. To help address this, we can use experience replay which randomly samples a batch of previous (s, a, r, s') observations from the agent's memory to train on [2].

Another benefit of experience replay is that each observation can be used multiple times by the learner. At each time step, the learner will randomly sample a batch of previous experiences and perform gradient updates with the sampled transitions.

1.3. Double DQN

Double Q-Learning was first introduced as a tabular method [6] to separate out the evaluation component from the selection component in the Bellman Equation (1). This can then be combined with DQN so that the target network is used for evaluation while the main network is used for action selection [5]. This approach is called Double DQN (DDQN). The resulting formulation for the Q-value approximation is shown below.

$$Q(s,a) = r(s,a) + \gamma Q(s', \underset{a'}{\operatorname{argmax}}\, Q(s',a';\theta);\theta^-) \quad (4)$$

This provides many of the benefits of Double Q-Learning such as the lower likelihood of overoptimistic value estimates. However, it does not train two networks separately and instead uses a lagging network that is updated with the online network's weights periodically. This allows the algorithm to be performed with very little additional overhead.

2. Lunar Lander Environment

The environment used for this experiment is Lunar Lander v2 from the OpenAI Gym [1]. The aim of the agent is to successfully land the spaceship given some stochastic starting point above the surface. The state space is continuous and is encoded in a 8-dimensional vector. The state vector contains the position, angle, velocity and the angular velocity of the lander as well as whether or not the lander's two legs are touching the ground. The action space is discrete; it can do nothing, fire the left engine, the main engine, or the right engine.

The reward structure is as follows: -0.3 points per time step for firing the main engine, +10 points for each leg ground contact, -100 points for crash, and +100 points for a successful landing. The problem is considered solved if the agent can score +200 points or more on average across 100 consecutive episodes. This is an interesting problem for testing DQN and DDQN since the continuous state space makes tabular methods infeasible and allow approximate methods to shine.

3. Hyperparameter Search

The evaluate the effect of the different hyperparameters, the learner was trained using a range of hyperparameters. The results were then compiled and graphed with a 30 period moving average smoothing to reduce overall volatility of the series.

Due to time and space constraints, hyperparameter testing was only done for the DQN learner. In the future, it would be interesting to see the effect of varying hyperparameters on Double DQN as well as other DQN-based algorithms.

3.1. Epsilon Decay Rate

To ensure that enough of the state space is explored before behaving greedily, an ϵ-greedy approach was used. The ϵ was initially set to 1 and decayed after each time step / update. A number of decay rates were tested and the results are shown in the figure below.

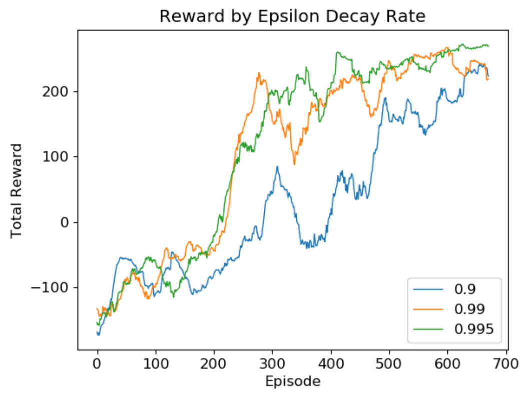

Figure 1. Reward curves for different ϵ decay rates. Higher rates allowed for more exploration and ultimately produced better convergence results.

Slower decay proved to be better as the agent was able to explore more of the action-state space before acting greedily. The ϵ was floored at 0.05 so that the agent could still do some exploration in the limit, albeit minimal. The final decay rate picked for the learner was 0.992. This ensured that the agent explore enough of the state space to solve the problem but can converge and hit the ϵ floor well within 700 episodes.

3.2. Neural Network Learning Rate

The learning rate for the target network, also called the α hyperparameter, determines how large of a step the learner takes per update. Larger values allow the learner to progress faster but risk overshooting the goal and could be unstable. Three different order of magnitudes were tested for the learning rate: 10^{-3}, 10^{-4}, and 10^{-5}.

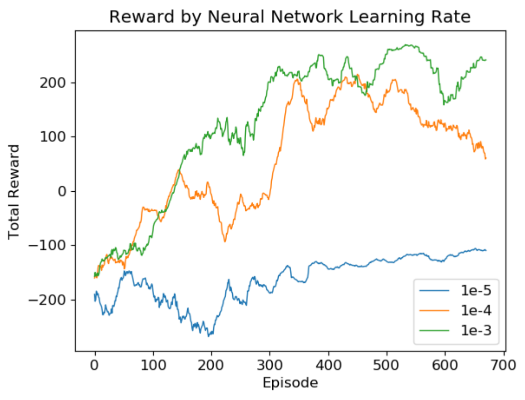

Figure 2. Reward by α, the learning rate hyperparameter for Q-value neural network.

Larger learning rates seem to perform better as the estimators can approach the true parameters at a faster rate. It is surprising that the smallest step size was not able to converge at all. One possible explanation is that the learner was stuck in some local basin and was unable to escape due to small step sizes. The learning rate that was ultimately used was 2×10^{-4}, as that was sufficiently large to reach convergence but small enough to be reasonably stable.

3.3. Target Network Update Frequency

The target network parameters θ^{-} are updated periodically but not at every step. The number of steps between updates proved to have material effect on the stability of the learner.

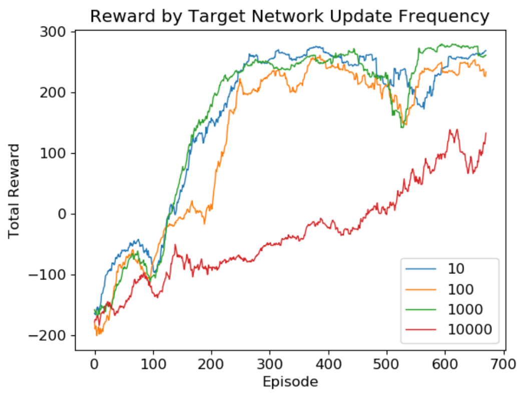

Figure 3. Total reward per episode by the target network update frequency. The frequency indicates the number of steps that the target network weights are frozen for before being set to the main network's weights.

The results show that generally more frequent updates are better, as the gradient calculated from more recent weights will likely reflect the true direction of improvement better. However, it is surprising that even the 10 step update worked so well. One possible explanation is that with relatively large batch sizes, such as 64, the variance associated with each gradient descent step is already dampened sufficiently. The update frequency that was chosen for the experiment was 100 steps. This had a good balance of being able to converge quickly while still having some of the stability benefits of having a target network in the first place.

3.4. Replay Batch Size

The size of the batch sampled from the agent's memory dictates how much learning is done by the agent at each update. Small batch sizes are faster but the agent will not learn as much during each time step. Conversely, a large batch size will allow the agent to learn a lot but it could be slow and provide only marginal benefit over smaller batch sizes.

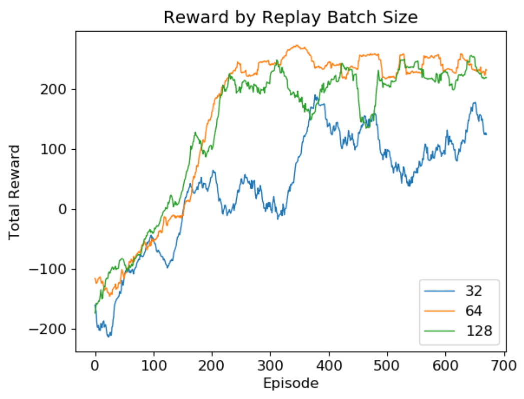

Figure 4. Reward by experience replay batch size. The size indicates the number of transitions that are sampled from the agent's memory during each update step.

Out of the three batch sizes tested, the smallest batch size (32) produced the worst result. This is expected as the agent was learning significantly less during each update step. However, the largest batch size of 128 produced about the same results as the the 64 batch. This shows that beyond some point there are diminishing returns to having larger batch sizes. One reason could be that once each observation in the agent's memory is explored enough times on average, there is limited additional information to be gained if it were sampled more times. As a result, the batch size used for the the final agent training and trials was 64.

3.5. Maximum Replay Memory Size

The agent keeps a limited number of its most recent observations for experience replay. The size of the memory determines how far back it can sample from during each batch update. Three orders of magnitudes were tested for the agent's memory size, and the results are shown in the figure below.

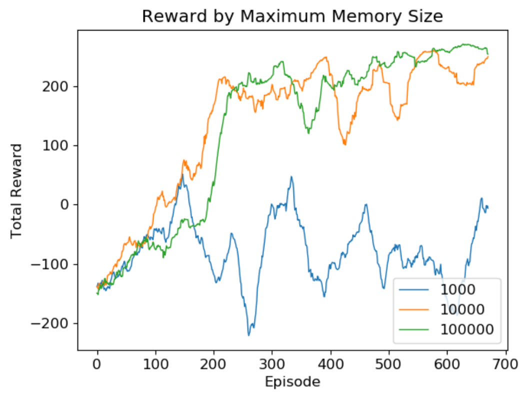

Figure 5. Total reward earned by agent of different memory sizes. The agent keeps the most recent transitions in memory; in other words, its memory has a first-in, first-out policy.

As expected, a small memory size was detrimental to the learning ability of the agent. The 1,000 observation memory agent was not able to learn to consistently land. The 10,000 and 100,000 memory size agents did not see much of a difference but the 100,000 memory size agent did take significantly longer to run. Much like the batch size, there is a point of diminishing returns. Beyond 50,000 stored transitions, the benefits of experience replay is fully realized and additional sampling of very old observations is not helpful for the learner.

Ultimately, a memory size of 50,000 was used. This was a good balance between having enough observations to learn from and maintaining relevance in the sampled batches.

4. Results

The DQN and DDQN learners were trained with the hyperparameters discussed above for 700 episodes. The training procedure was then repeated 7 times. The learning curves for those runs were aggregated and graphed.

After training, the saved weights were used to do a trial run with $\epsilon = 0$ to understand the quality and consistency of the learner. This was also repeated 7 times to assess variance of the trial performance.

4.1. DQN

The learning curve for the DQN agent was fairly stable, with steady improvements in its policy until hitting the +230 reward mark around episode 280. At this point the learner plateaus and consistently produces an average reward of +235. The average reward of the last 100 episodes of training is approximately +250.

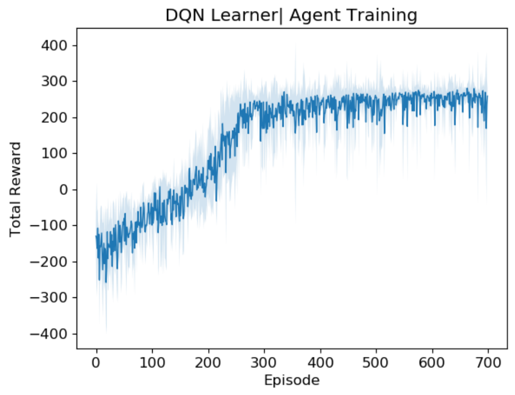

Figure 6. Training curve for DQN learner for 700 episodes. Upper and lower bounds reflect one standard deviation calculated from 7 separate training sessions.

This demonstrates that with the appropriate hyperparameters, DQN is able to converge consistently and maintain stability.

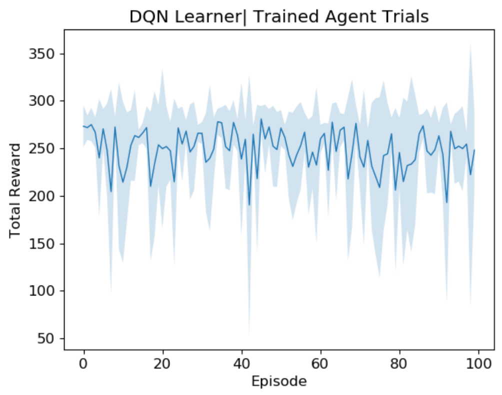

Figure 7. Total reward from trial runs with trained DQN agent.

Across the 7 trial runs, the DQN agent scored an average of +249 with standard deviation of 58, and a median of +264. There is moderate variance where some episodes dip below 0 and some go above +300. For the most part the agent learned the ability to land consistently and achieve reward well above the +200 threshold.

4.2. Double DQN

The DDQN learner was run with the same hyperparameters as the DQN learner for a direct comparison. The DDQN learner was not as fast to converge and exhibited greater variance in both training and testing.

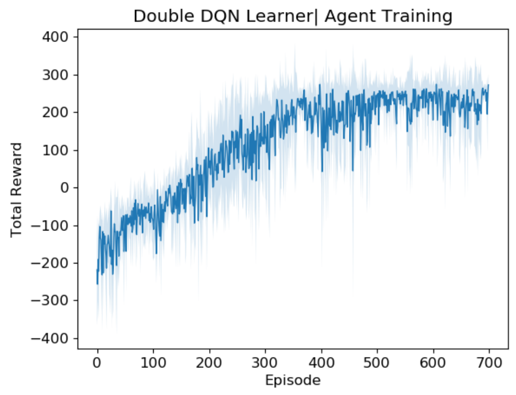

Figure 8. Training curve for Double DQN learner run with the same hyperparameters as the DQN learner. Results are aggregated across 7 separate training sessions.

The results are surprising given research has shown that the decoupling of selection and evaluation should result in faster and more stable learning However, this may not be the case of this run as the hyperparameters, especially the target update frequency, may not be fully optimized for the Double DQN learner.

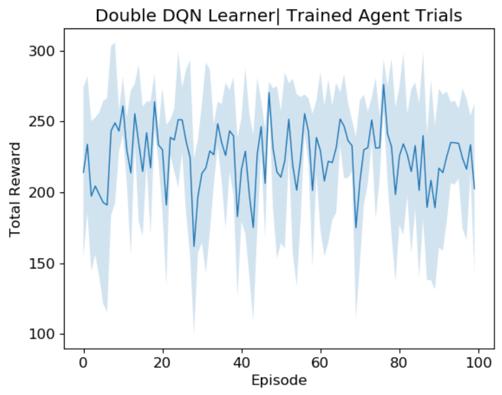

Figure 9. Total reward from trial runs with trained Double DQN Learner. Results were averaged over 7 trial runs with 100 episodes each.

Trial results show overall worse performance than the DQN learner. Across the 7 trials, the agent scored an average of +224 points with a standard deviation of 51, and a median of +234 points. This is lower than the DQN learner by about 25 points, which is likely due to the fact that the hyperparameters may not have been optimal. With more time for experimentation, it would have been good to include a rigorous exploration of hyperparameters for the Double DQN learner.

5. Discussion

Through these experiments, we've explored how function approximation can be used to solve reinforcement learning problems with continuous state spaces. In particular, the Lunar Lander environment could be solved by DQN and DDQN with appropriate hyperparameters. We also examined the effect of various hyperparameters on the learning speed and convergence of the DQN algorithm.

In the future, it would be interesting to implement and test other DQN-related approaches such as prioritized replay and dueling networks. It would also be interesting to compare all the DQN-based algorithms to policy gradient and actor-critic algorithms.

References

[1] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym, 2016. 2

[2] L.-J. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 1992. 2

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *NIPS Deep Learning Workshop*, 2013. 1

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 02 2015. 1

[5] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015. 2

[6] H. P. van Hasselt. Double q-learning. *NIPS*, 2010. 2