Project 2: Lunar Lander

Matthew Molinare
mmolinare@gatech.edu
CS 7642 Reinforcement Learning

Abstract

I describe the Q-learning algorithm for model-free reinforcement learning as the foundation for deep Q-networks (DQNs), agents suited for domains with state spaces that are not fully observable. I implement a variety of DQN agents in Python capable of interacting with the Lunar Lander environment from the OpenAI Gym library. I am able to consistently achieve an average score of over 260. I present analysis on the effects of multiple network hyperparameters, including learning rate and exploration bias, on both training convergence speed and testing performance.

1 Introduction

The goal of the Q-learning algorithm in reinforcement learning is to teach an agent how to select actions in a way that maximizes the expected sum of rewards r_t discounted by $\gamma \in (0, 1]$ at each time-step t. That is, we wish to determine the policy π, mapping from a discrete set of actions $a_t \in \{1, ..., |\mathcal{A}|\}$ to states s_t, that maximizes the action-value function,

$$Q^{\pi}(s,a) = \mathbb{E}\Big[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...\Big| s_t = s, a_t = a, \pi\Big] \tag{1}$$

This can be achieved using the value iteration algorithm by representing the optimal action-value function $Q^*(s,a) = \max_\pi Q(s,a)$ as an iterative update,

$$Q_{i+1}(s,a) = \mathbb{E}\Big[r + \gamma \max_{a'} Q_i(s',a')\Big| s,a\Big] \tag{2}$$

whose convergence $Q_i \rightarrow Q^*$ as $i \rightarrow \infty$ has been proven [1]. Here, s' is the state encountered after taking action a in state s, and a' is the subsequent action. For discrete state spaces, a table of action-values for all combinations of states and actions can be stored and updated. This is intractable, however, for environments with continuous state spaces; instead, a neural-network function approximator with weights θ may be used to estimate the action-value function, $Q(s,a;\theta) \approx Q^*(s,a)$ [2]. The loss function at iteration i is defined as

$$L_i(\theta_i) = \mathbb{E}\Big[\big(y_i - Q(s,a;\theta_i)\big)^2\Big] \tag{3}$$

where y_i is the target. For deep Q-networks (DQNs), this takes the form

$$y_i^{DQN} = r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) \tag{4}$$

The weights from the previous iteration θ_{i-1} are used to optimize the loss function by stochastic gradient descent. A fixed and separate network $\hat{Q}$ for generating the targets, whose weights θ^- are set to θ every C number of iterations, can improve the stability of DQN by introducing a delay between updates to the online network Q and the targets [3]. The expression for the target becomes $r + \gamma \max_{a'} \hat{Q}(s',a';\theta^-)$; in the limit $C = 1$, this reduces to Eq. 4. To help mitigate overestimation bias, an extension to DQN known as double DQN (DDQN) [4] decouples the selection of the action from its evaluation in the max operator of the conventional Q-learning target. The modified target is

$$y_i^{DDQN} = r + \gamma \hat{Q}(s', \arg\max_{a'} Q(s',a';\theta_i);\theta^-) \tag{5}$$

Another version of DQN uses a dueling network architecture that splits the neural-network into two streams to separately estimate the state and advantages for each action [5]. The advantage function is defined as

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s) \tag{6}$$

where the state value $V^\pi(s) = \mathbb{E}_{a\sim\pi(s)}[Q^\pi(s,a)]$. The two streams are aggregated in the final layer of the network as

$$Q(s,a;\theta) = V(s;\theta) + \left(A(s,a;\theta) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a';\theta)\right) \tag{7}$$

During the training of a Q-network, a limited data-set of the agent's past experiences is stored into a replay memory $\mathcal{D} = e_1, ..., e_N$. Each entry, $e_t = (s_t, a_t, r_t, s_{t+1})$, is a collection of the state, action, reward, and proximate state at some time-step. The Q-learning update is applied to a random sample, or minibatch, of the replay memory. This technique, known as experience replay, allows the efficient reuse of past experiences and reduces correlations between consecutive samples that may degrade the network's ability to generalize. The agent selects actions according to an ϵ-greedy strategy — it follows the greedy strategy $\arg\max_a Q(s,a;\theta)$ with probability $1-\epsilon$ and selects a random action with probability ϵ. The value of ϵ starts at 1 and decays to some asymptotic limit ϵ_∞,

$$\epsilon(t) = \epsilon_\infty + (1-\epsilon_\infty)e^{-\lambda t} \tag{8}$$

The decay rate λ determines how quickly the agent should favor exploitation of knowledge from past experiences in lieu of exploration of the state space.

2 Environment

In this study, the `LunarLander-v2` environment of OpenAI's Gym toolkit [6] was used as a testbed for the development of several Q-learning agents. The lander has an 8-dimensional continuous state space representing its position, velocity, orientation and whether or not its legs are in contact with the ground. The lander can perform 4 discrete actions: do nothing, fire the left engine, fire the main engine, or fire the right engine. The objective of the lander is to descend from its initial position and come to rest on the landing pad, for which it receives a positive reward inversely proportional to its final speed. The lander incurs a small negative reward for firing the main engine and a large negative reward for crashing.

My Python implementation of DQN uses the Keras deep learning library [7]. The network architecture consists of a single hidden layer of 512 neurons with ReLU activation. The Adam optimization algorithm is used instead of the classical stochastic gradient descent to update the network weights. The learning rate α is an adjustable parameter; it will be shown later that the best results were achieved for $\alpha = 0.0003$. The Huber loss function is used to compute the error between the predicted and true action-values for a given state-action pair. For small errors, it takes the form of the mean squared error (MSE) of Eq. 3; for large errors, it mimics the mean absolute error (MAE). That is,

$$L_i(x) = \begin{cases} \frac{1}{2}x^2 & \text{for } |x| \le 1, \\ |x| - \frac{1}{2} & \text{otherwise} \end{cases} \tag{9}$$

where the error term $x = y_i - Q(s,a;\theta_i)$. This formulation makes the network more robust to large errors.

Several versions of DQN were tested, including DDQN and dueling DQN. The aggregator in the dueling network was implemented as a `Lambda` layer following a dense layer with $|\mathcal{A}|+1$ neurons, incorporating the advantages for each action as well as the scalar state-value.

Hyperparameter	Optimal value
Learning rate (α)	0.0003
ϵ lower limit (ϵ_∞)	0.1
ϵ decay rate (λ)	0.001
Replay memory size (N)	100,000
Minibatch size	64
Discount factor (γ)	0.99

Table 1: Optimal hyperparameters for the DQN agent in the Lunar Lander environment. The learning rate, ϵ decay rate, and ϵ lower limit were found using a coarse grid search. Variations to the discount factor, replay memory size, and minibatch size had no marked effect on testing performance.

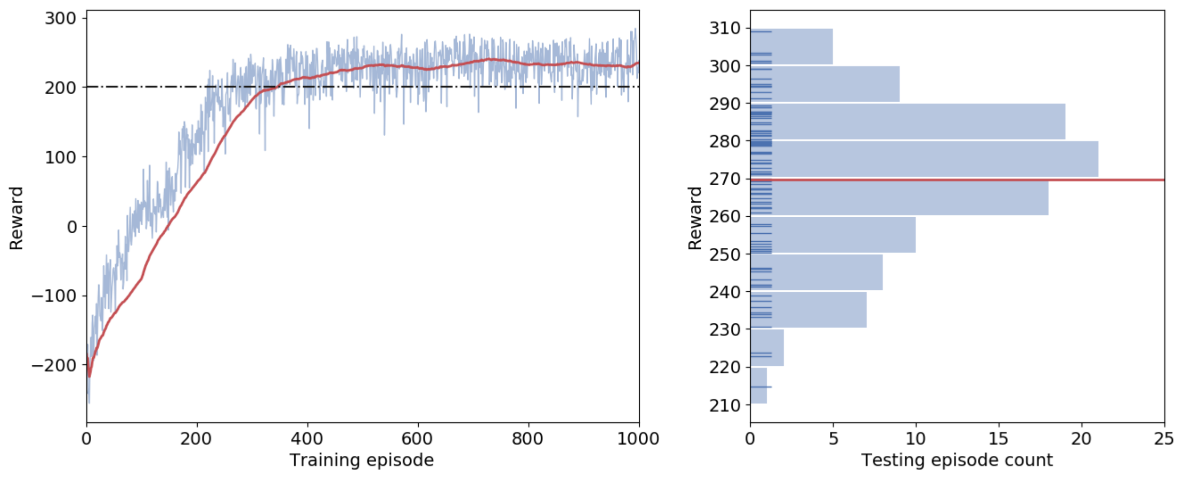
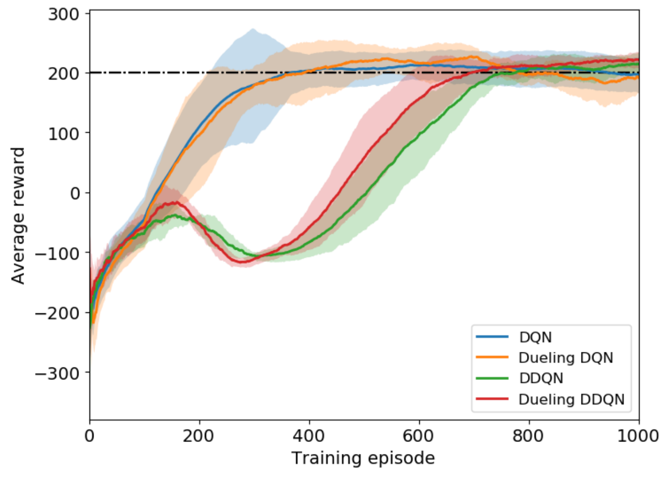


Figure 1: Left: Reward per training episode. Results are averaged over 9 repeated runs. The red line indicates the running average reward over the previous 100 training episodes, attaining a value of 200 after 345 episodes and converging to approximately 235. Right: Distribution of rewards per testing episode of a trained agent. The horizontal red line indicates an average reward of 269 over 100 testing episodes.

3 Results

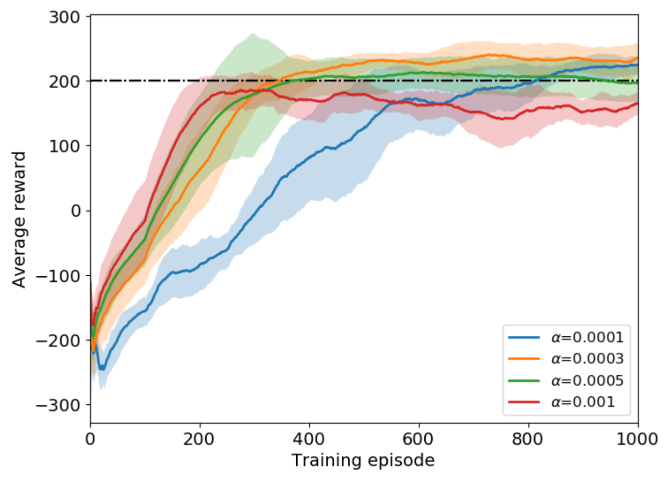
The DQN agents were trained over 1,000 episodes. Each episode consists of a single play-through of the Lunar Lander game environment. The game was terminated early if the accumulated rewards fell below -500 or if the number of time-steps exceeded 1,000. A set of hyperparameters was adopted from Mnih et al. [3] and optimized for the Lunar Lander environment by performing a coarse grid search over various learning rates (α), ϵ decay rates (λ) and ϵ lower limits (ϵ_∞). Unless otherwise stated, the agents discussed in this section use the optimal hyperparameters listed in Table 1. The neural network weights are saved in regular intervals to a library of weights. These weights are reloaded by a new instance of the agent who behaves according to a greedy policy. The weights yielding the highest running average reward over 100 training episodes are selected for testing. A test is deemed successful if the average reward over 100 episodes is at least 200.

I was able to successfully train a DQN agent to achieve an average reward of 269 (see Figure 1). The use of a target network did not significantly improve the performance of the agent in terms of either the number of training episodes until convergence or average testing reward. Figure 2 shows the difference between the



Agent type	Testing reward
DQN	$\mathbf{230.1 \pm 30.8}$
DDQN	226.2 ± 36.1
Dueling DQN	204.4 ± 33.7
Dueling DDQN	219.8 ± 28.6

Figure 2: Running average training reward for various agent types. Here, a learning rate of $\alpha = 0.0005$ is used for all agent types. Note that agents using a target network, such as DDQN and dueling DDQN, experience a temporary decline in training rewards starting around episode 200 and lasting around 100 episodes. Additional analysis on the impact of replay memory size on training performance may shed light on this phenomenon.



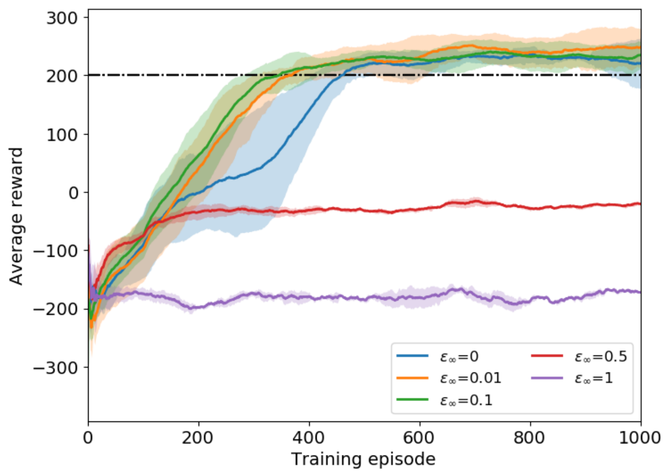
Learning rate (α)	Testing reward
0.0001	218.6 ± 20.1
0.0003	$\mathbf{248.0 \pm 33.5}$
0.0005	230.1 ± 30.8
0.001	203.4 ± 47.9

Figure 3: Running average training reward for various learning rates α. The number of training episodes until convergence decreases as α increases. For $\alpha = 0.001$, the agent fails to reliably attain a running average training reward of at least 200. At $\alpha = 0.01$, the agent takes more than 800 training episodes to converge. The best performance was observed for $\alpha = 0.0003$.

single-network DQN and its various extensions. For this and similar figures included in this report, results are averaged over 6 or more repeated runs. On the left, the shaded regions indicate the standard deviations about the average values. The accompanying table on the right lists the average rewards over 100 testing episodes using the first set of weights corresponding to an average running training reward of 200. An entry of N/A indicates that a value of 200 was not attained over the first 1,000 training episodes.

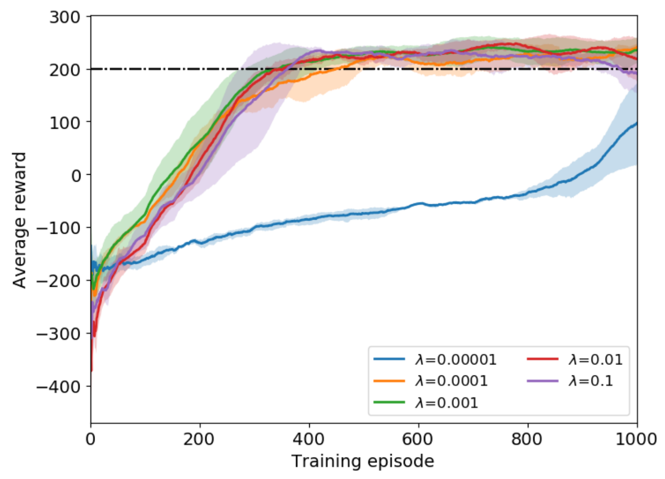
As expected, analysis of the learning rate reveals that the agent trains more quickly, i.e. converges in fewer training episodes, as α increases (see Figure 3). Additional training episodes are required to determine if there are any significant benefits to testing performance on lower learning rates ($\alpha \leq 0.0001$). Figure 4 shows the high sensitivity of testing performance on the ϵ lower limit. It may be beneficial to conduct a finer search in ϵ_∞-space between 0.01 and 0.5 to find a value that would further improve testing performance. The effect of ϵ decay rate on convergence was also examined (see Figure 5), however it is inconclusive if variations within the range $0.0001 \leq \lambda \leq 0.1$ have a significant impact on testing performance in the Lunar Lander environment.

One pitfall I encountered was the premature termination of the training process before the agent had



ϵ lower limit (ϵ_∞)	Testing reward
0	154.1 ± 83.4
0.01	232.9 ± 17.6
0.1	$\mathbf{248.0 \pm 33.5}$
0.5	N/A
1	N/A

Figure 4: Running average training reward for various ϵ lower limits ϵ_∞. For $\epsilon_\infty = 1$, the agent selects random actions only and is unable to learn a sequence of actions that yields high rewards. For $\epsilon_\infty = 0.5$, as $t \to \infty$, the agent has no preference between exploration and exploitation and converges to a value that is only a moderate improvement over an purely exploratory strategy. When $\epsilon_\infty = 0$, the agent converges to high value by exploiting the experiences from early time-steps; however, it may fail to generalize well to novel scenarios due to over-fitting. Its testing performance is highly variable due to its dependence on a limited, initial set of observations.



ϵ decay rate (λ)	Testing reward
0.00001	N/A
0.0001	246.0 ± 9.1
0.001	$\mathbf{248.0 \pm 33.5}$
0.01	207.5 ± 63.9
0.1	229.8 ± 31.7

Figure 5: Running average training reward for various ϵ decay rates λ. For too slow a decay rate ($\lambda = 0.00001$), the agent spends many episodes exploring random actions, curbing its ability to execute strategies requiring multiple time-steps. Otherwise, the dependence of convergence rate and testing performance on λ is insignificant.

truly converged. Originally, I had defined the termination criterion as the whether or not the agent had achieved a running average training reward of 200. In many cases, however, the agent had not ceased learning. Allowing training to occur for 1,000 episodes regardless of its score, I was able to extract weights corresponding to rolling averages of 230-270 that produced testing scores with lower variance. Whereas before it was not atypical to see upwards of 20% of the testing scores fall below the 200 threshold, now less than 5% of tests fail.

4 Conclusion

I was able to implement a DQN reinforcement learning agent to successfully land the Lundar Lander after only a several hundred episodes of training. I studied the effects of several hyperparameters on average testing reward to obtain a set of optimal hyperparameters. There remains, however, more searching of the hyperparameter space to be done. For example, the negative slope in training reward of agents utilizing a target network may be caused by an insufficient replay memory size or some combination of other hyperparameters. Still, these results demonstrate the applicability of deep reinforcement learning to highly complex, sequential decision-making problems.

References

[1] Watkins, Christopher JCH, and Peter Dayan. *Q-learning.* Machine Learning 8.3-4 (1992): 279-292.

[2] Mnih, Volodymyr, et al. *Playing Atari with deep reinforcement learning.* arXiv preprint arXiv:1312.5602 (2013).

[3] Mnih, Volodymyr, et al. *Human-level control through deep reinforcement learning.* Nature 518.7540 (2015): 529.

[4] Van Hasselt, Hado, Arthur Guez, and David Silver. *Deep reinforcement learning with double Q-learning.* Thirtieth AAAI Conference on Artificial Intelligence. 2016.

[5] Wang, Ziyu, et al. *Dueling network architectures for deep reinforcement learning.* arXiv preprint arXiv:1511.06581 (2015).

[6] OpenAI Gym. https://gym.openai.com/

[7] Keras: The Python Deep Learning library https://keras.io/