Training Lunar Lander with Deep Reinforcement Learning

Woojae Kim
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
jkim7657@gmail.com

Abstract

This work presents a Deep Q-network (DQN) to solve Open AI Gym's Lunar Lander environment. The trained DQN successfully controls the space fleet in a continuous state-space, enabling safe landings. The DQN utilizes three fully connected layers with rectifier nonlinearities to approximate the optimal action-value function. Experience replay and bootstrapping method through network duplication were utilized to improve the network's performance. Extensive grid search was performed to fine-tune the network's hyperparameters and study their effects on the agent's performance.

1 Introduction

Deep reinforcement learning was first introduced by Mnih et al. [1] in 2013. In their famous work, they demonstrated that a reinforcement learning agent can outperform humans in Atari 2600 games using Deep Q-network(DQN). Mnih et al. used deep convolutional neural network to approximate the optimal action-value functions $Q^*(s, a)$ for various Atari games, given raw pixel input data. Unlike other standard reinforcement learning algorithms that utilize handcrafted features [2], the DQN in [1] does not preprocess the raw input image to extract features. Just like humans, the DQN learned from nothing but the image data, reward, termination signal, and set of all possible actions. The network outperformed standard reinforcement learning algorithms, such as SARSA [2] and contingency [3] methods, in six of the seven games played and also expert human players in three of them.

In this work, a Deep Q-network was implemented to solve Open AI Gym's Lunar Lander environment. Sample screenshots of the agent and environment are provided in Figure 1.

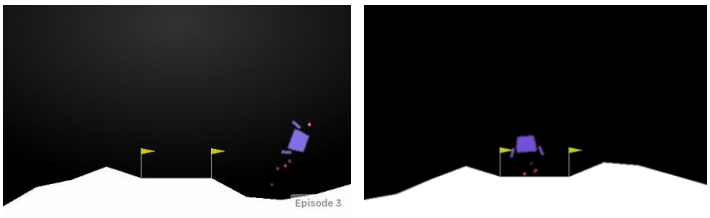

Figure 1a (left): Screenshot of a crash episode in Open AI Gym's Lunar Lander environment; figure 1b (right): Screenshot from a successful landing episode

The goal of the game is to safely land the Lunar Lander on the landing pad which is always centered at (0, 0). The agent is rewarded or penalized depending on the lander's placement on the pad. An episode finishes when the lander either crashes (-100 reward) or comes to rest (+100 reward). Although the fuel is infinite, firing the main engine incurs penalty. The environment is considered to be "solved" if the agent achieves a score of 200 points or higher on average over 100 consecutive runs.

The agent has four possible actions: do nothing, fire the left engine, fire the main engine, fire the right engine. At each time step, the agent is provided with a 8-tuple state vector—(x, y, v_x, v_y, θ, v_θ, left-leg, right-leg)—which indicates to x and y-coordinates, x and y component velocities, angle, angular velocity of the space fleet, and binary values to indicate whether either leg is touching the ground.

2 Background

2.1 Deep Q-Network

The DQN uses artificial neural network (ANN) for the optimal action-value function approximation task in a continuous state-space environment. As in other reinforcement learning algorithms, the goal of the agent using DQN is to maximize the future rewards, where the future discounted reward at time step t is defined as $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$, where T is the final time-step. Then the optimal action-value function is defined as:

$$Q^*(s,a) = max_\pi E[R_t|s_t = s, a_t = a, \pi]$$

where π is a poly mapping from state *s* to action *a*. This optimal action-value function obeys the Bellman equation, which can be expressed as:

$$Q^*(s,a) = E[r + \gamma\, max_{a'}\, Q^*(s',a')|s,a] \quad \textbf{(1)}$$

here *s'* is the next state and *a'* is all possible actions. Typical reinforcement learning algorithms, including tabular Q-learning, estimate the action-value function through an iterative update:

$$Q_{i+1}(s,a) \leftarrow E[r + \gamma\, max_{a'}\, Q_i(s',a')|s,a]$$

As $i \to \infty$, the approximation approaches the optimal action-value function $Q_i \to Q^*$. As pointed out previously and as in [1], however, this is impractical because the action-value function is estimated for each state without any generalization. Therefore, we will be using a neural network with weights *w*—which was first called Q-network in [1]—to approximate the action-value function $Q(s,a;w_i)$. A Q-network can be trained by minimizing a sequence of loss functions $L_i(w_i)$ at each iteration *i*:

$$L_i(w_i) = E_{s,a}[(y_i - Q(s,a;w_i))^2] \quad \textbf{(2)}$$

where $y_i = E_{s'}[r + \gamma\, max_{a'}\, Q^*(s',a';w_{i-1})|s,a]$ is the target for time-step *i*. When optimizing the loss function $L_i(w_i)$, the parameters from the previous iteration w_{i-1} are held fixed. Although it is possible to calculate the full gradient of the loss function with respect to the weights through expectation, the DQN is typically optimized through stochastic gradient descent.

As pointed out in [1], this algorithm is model-free; it uses experience samples to directly approximate the action-value function without constructing models for transitions and rewards. It is also off-policy; while learning, it ignores past state-action decisions.

2.2 Implementation and Model Architecture

The Q-network designed to solve the Lunar Lander environment resembles that of the original network in [1] in terms of its algorithmic implementation with minor modifications. As in the original paper, experience replay [4] and bootstrapping through the use of duplicate network [1] were adopted. In order to promote exploration in the beginning phase of the learning, epsilon-greedy exploration method with linearly decreasing ε was implemented. The details of the deep Q-learning algorithm is described in detail below.

For the neural network architecture, the original implementation in [1] included convolution layers for image processing purposes. Since the input to the Lunar Lander environment is not image data, fully connected hidden layers were selected in place. The input to the Q-Network was 8×n (where n=size of the random batch), and the output was 4×n. The network consisted of three, fully-connected hidden layers of 64 units with ReLu activations. All codes were written in Python, and for the backpropagation calculation, PyTorch was utilized.

Algorithm: Deep Q-learning with Experience Replay and ε-greedy Exploration

-Initialize replay memory D (queue of capacity=$1E5$)
-Initialize DQN $\widehat{Q}(s, a; w)$ and with Xavier initialization
-Set the duplicate network $\widetilde{Q} = \widehat{Q}(s, a; w)$
for episode = 1 to M:
    -Initialize s_1
    for t = 1 to T:
        -Select a random action a_t with probability ε
        -Otherwise, $a_t = max_a \ \widehat{Q}_i(s_t, a; w_i)$
        -Take a_t in the environment and observe reward r_t and the next state s_{t+1}
        -Store the experience (s_t, a_t, r_t, s_{t+1}) in D
        -Sample a random minibatch (n=64) of experiences from D
        -Set the target $y_i = \begin{cases} r_i & for\ terminal\ state\ s_{i+1} \\ r_i + +\gamma\ max_{a'} \widetilde{Q}_i(s_i, a'; w_{i-1}) & for\ non-terminal\ state\ s_{i+1} \end{cases}$
        -Calculate the loss
$$L_t(w_i) = \mathrm{E}_{s,a}\left[(y_i - \widehat{Q}(s, a; w_i))^2\right]$$
        -Perform minibatch stochastic gradient descent (with learning rate α) on the loss function to update w_i using backpropagation
        for every $C=4$ updates:
            -Set $\widetilde{Q} = \widehat{Q}$
        end
    end
end

3 Experiments

Effect of three hyperparameters—learning rate α, discount factor γ, and random action probability ε—on the performance of the agent was studied. In addition to the aforementioned hyperparameters, there were additional parameters that had to be fixed. To determine these parameters, an extensive grid search was performed on various combinations model parameters. Once a reasonable performance level was achieved, the found parameters were fixed. Additional grid search was performed on the combinations of the three hyperparameters to determine their working ranges. Then the effect of varying each hyperparameter (while fixing the other two constant) was studied.

The training of the agent was stopped once a mean score of 200 points or higher was achieved over 100 consecutive runs, and the problem was considered solved. In order to test the agent performance on "unseen" data, reward per trial for 100 trials using the trained agents were also plotted.

3.1 Discount Factor

The discount factor is a measure of how far ahead in time the agent will value the reward. Although there has been some research arguing for the importance of a dynamic discount factor [5], it is arbitrarily chosen to be near 0.9 in most researches and in applications. For this experiment, values between 0.7 and 1 were experimented.

The learning rate was fixed at 5E-4. The random action probability ε was initialized at 1 and was linearly decreased to 0.05 over 5000 time-steps. During the initial grid search, it was found that these values worked the best and solved the problem in most cases.

Only the agent with discount factor γ=0.99 converged— at around 400 training episodes—within 1500-episode training window (Figure 2a). The other agents with various discount factors could not solve the environment in a given number of episodes. It was interesting that even the agent with γ=0.95, which was very close to 0.99, could not solve the problem. It was also observed that the agent with γ=1, which corresponds to the infinite horizon sum case, exhibited drastically different behavior, with increasing large oscillations. This oscillation was likely due to action-value function values diverging, since in theory the total discounted rewards may sum up to infinity (if episodes goes on forever).

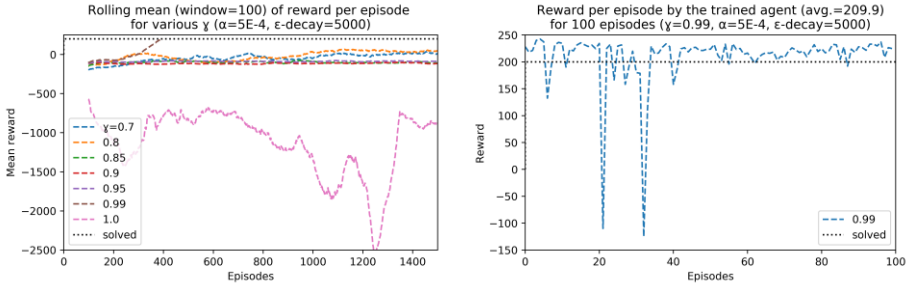

Figure 2a (left): Rolling mean of reward per episode during the training for various discount factors; figure 2b (right): reward per episode by the trained agent

In figure 2b, the reward per episode by the trained agent (γ=0.99) for 100 episodes were plotted. In majority of cases, the reward per episode exceeded 200 points (with the mean of 209.9). A few cases with large negative reward values were likely due to episodes where the agent had to go through state spaces that were unencountered during the training.

3.2 Random Action Probability

For the random action probability experiment, previously found discount factor of 0.99 was used. The learning rate was also fixed at 5E-4. Agents with linearly decreasing ε, from 1 to 0.05 over ε-*decay* number of time steps–between 100 and 2E5–were studied (figure 3a). Likewise, figure 3b and Table 1 summarize the trained agent's performance over 100 episodes.

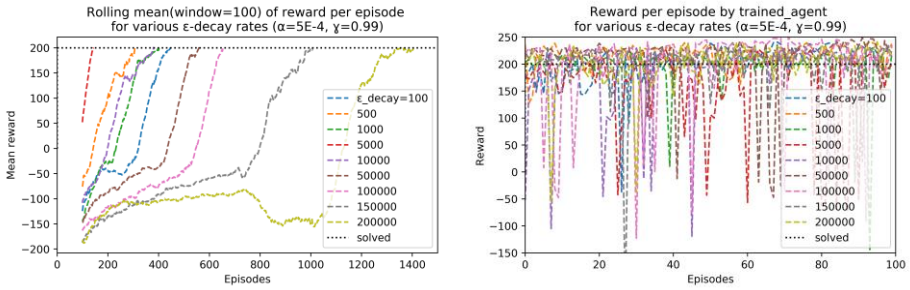

Figure 3a (left): Rolling mean of reward per episode during the training for linearly decreasing ε; figure 3b(right): Reward per episode by trained agent

	ε-decay=100	500	1000	5000	1E4	5E4	1E5	1.5E5	2E5
Avg. Reward	202.5	**217.9**	197.7	163	190.5	209.6	193.8	200.1	200.7

Table1: Mean reward per episode by the trained agent with linearly decreasing ε

All agents were able to solve the environment within the training window. The agent with ε-*decay=5000* solved the problem in the shortest amount of time. During the testing phase, however, the trained agent with ε-*decay=500* scored the highest average. The agents with 5E4≤ε-*decay*≤2E5 exhibited more consistent performances. This is likely due to the fact that the agents with higher ε-*decay* had more chances to explore. In smaller ε-*decay region*, the agent performances were more inconsistent which can be also attributed to less chances of exploration and more exploitation.

3.3 Learning Rate

For the learning rate experiment, γ and ε-*decay* were fixed at 0.99 and 1E5 as these values were found to work in the previous analyses. Figure 3a and 3b plot the agent performance per episode for various α during training and testing, respectively. Table 2 summarizes the

average performances of the trained agents over 100 episodes.

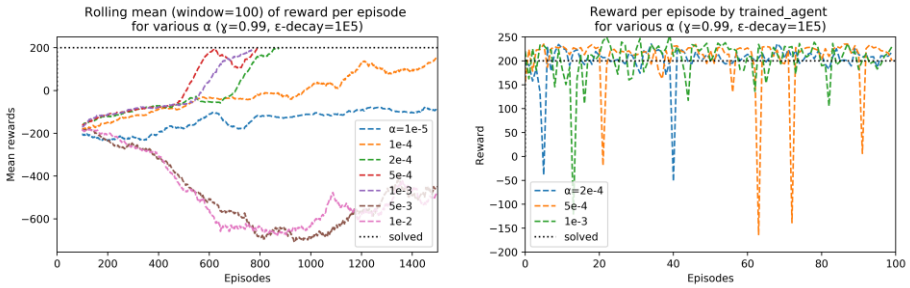

Figure 4a (left): Rolling mean of reward per episode during the training for various α; figure 4b(right): Reward per episode by trained agent

	α=1E-5	1E-4	2E-4	5E-4	1E-3	5E-3	1E-2
Avg. Reward	-134.8	198.5	201.2	**205.1**	203.7	-458.7	-995.4

Table 2: Mean reward per episode by the trained agent with various α

For the learning rate, a range of 2E-4≤ α ≤1E-3 seemed to work the best (with the peak performance at α=5E-4) in terms of wall clock time and performance during the testing phase. In terms of wall clock time, the agent with α=1E-3 was the quickest to solve the problem. As expected, lower learning rates resulted in slower learning agents. The agents with high learning rates exhibited divergence with oscillations. The agents that were able to solve the environment during the training also exhibited consistent performances during the testing.

4 Conclusion

In this work, deep Q-learning was implemented to solve Open AI Gym's Lunar Lander environment. The implementation details of the deep Q-learning algorithm and its network architecture were presented. The effects of varying learning rate, discount factor, and random action probability on the agent performance were explored.

Among many challenges encountered during the development and analysis of the DQN, selection of memory queue size (for experience replay) was perhaps the biggest pitfall. The key idea behind the experience replay is to store many transitions in memory and randomly sample them at each time-step to make the distribution of training samples as iid as possible. The selection of the experience memory size is tricky; if it is too small, Q-network will not have enough data to generalize. If it is too large, the algorithm takes up more memory and may take longer to converge. Initially, I had set the memory size=1E3, and the agents were unable to solve the environment in the given number of episodes, regardless of other parameter settings. After many experiments, a working memory size was found to be 1E5.

I would also like to note that I had initially tried tabular Q-learning with quantized states. Nonetheless, the algorithm did not work very well. The presence of continuous state variables made it very hard to work with, as I did not have any idea on appropriate clipping ranges for quantization. Absence of experience replay probably made the Q-learning much harder to converge since the input data was much less iid.

For future experiments, I would come up with other convergence threshold, such as minimum reward difference between episodes, and let the agents to run until convergence. This will allow us to see if the agents with different hyperparameters will converge to the same action-value functions. For most runs, I had stopped the training of DQN once the average training reward exceeded 200 over 100 consecutive episodes (which is considered "solved" as per the instruction). As a starting point, I trained an agent with set of best-performing hyperpameters (α=5E-4, γ=0.99, ε *(fixed)=0.05)* and ran it over 1500 episodes without stopping. The agent's average reward oscillated between 240 and 260. For another setup with ε *(fixed)=0.1*, the mean reward value oscillated between 200 and 230.

References

[1] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602,.

[2] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. (2013). The arcade learningenvironment: An evaluation platform for general agents. Journal of Artificial Intelligence Research, 47:253–279.

[3] Marc G Bellemare, Joel Veness, and Michael Bowling. (2012). Investigating contingency awareness using atari 2600 games. In AAAI.

[4] Long-Ji Lin. Reinforcement learning for robots using neural networks. (1993). Technical report, DTIC Document.

[5] Francois, Vincent & Fonteneau, Raphael & Ernst, Damien. (2015). How to Discount Deep Reinforcement Learning: Towards New Dynamic Strategies.