Solving LunarLander with Deep Q-Learning

Karan Jit Singh
College of Computing
Georgia Institute of Technology
karanjs@gatech.edu

***Abstract*—This paper presents a reinforcement solution for the OpenAI gym problem, LunarLander-V2. Using reinforcement leaning, we solve this problem with an average reward of greater than 250. We follow the famous algorithm of Deep Q-Learning presented in the paper "Human Level Control Through Deep Reinforcement Learning", Minh et al. (2015). We also discuss a general analysis of how this result was achieved with an analysis on hyperparameters and variations of DQN. Finally, we present some parallel experiments done for solving the same environment.**

Keywords—OpenAI, Lunar-Lander, reinforcement learning, Q-learning, DQN, neural networks.

I. INTRODUCTION

Lunar Lander is one of the random agent environments provided by OpenAI gym. Lunar lander presents a landing agent on a 2-dimensional physical space where the objective of the game is to control the agent such that it successfully lands between in the designated landing zone between the two flags as shown in Fig. 1.

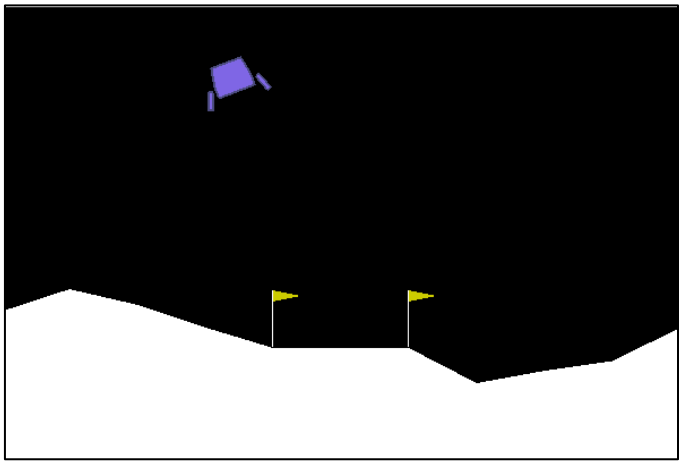

Fig. 1 A frame from the lunar lander environment.

A. *Environment Description*

The environment presents an 8-dimensional state space with 6 continuous variables and 2 discrete ones. The action space is discrete with 4 dimensions denoting the actions: Fire main engine, fire left engine, fire right engine, do nothing.

B. *Reward structure*

The agent is penalized if it moves away from the landing zone and is given back the same reward if it moves towards it. The reward for firing the main and orientation engines is -0.3 and -0.03 respectively. Agent is rewarded +100 if it lands safely and comes to rest and -100 if it crashes.

II. DEEP Q-LEARNING

A. *Q-Learning*

"Reinforcement learning is learning how to map situations to actions to maximize a numerical reward signal" [3]. With the provided environment, we attempt to solve this problem with an off-policy TD control algorithm, Q-learning [2]. We estimate $Q(s,a)$ using rule (1) as our basis for the algorithm.

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1},a) - Q(S_t,A_t)\right] \quad (1)$$

Where S_t and A_t represent the state and action pair at time t, α is the learning rate, γ is the discount factor and R_{t+1} represents the reward observed at time t. To estimate $Q(s,a)$ we use a multilayer perceptron with 2 hidden layers, that takes in S_t and outputs 4 values each corresponding to the Q-value of the respective state-action pair.

B. *Algorithm with Experience Replay and Target Network.*

For training the neural network, we use both the modifications as presented in the original deep reinforcement learning paper, Minh et al [3].

1. **Experience reply:** This technique stores each state transition and observation at time t in a fixed size FIFO queue known as the replay memory. While optimizing the function approximator we sample a small batch of transitions from this pool and use those observations to generate the loss that is backpropagated through the neural network.

2. **Target network:** The second modification separates the neural network used for generating the target $y = R_{t+1} + \gamma \max_a Q(S_{t+1},a)$. Every C updates the network Q is cloned to obtain the target network $\hat{Q}$.

Along with the above two mentioned modifications we introduce another modification to the algorithm that bounds the number of time steps of the episodes. While playing an episode, if this bound is exceeded, the episode is terminated immediately, and a new episode is started.

With the above modification we use get the loss function defined by (2) for the Q-learning update iteration i [1].

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\left[\left(r + \gamma \max_{a'} \hat{Q}\,(s', a', \theta_i^-) - Q(s, a; \theta_i)\right)^2\right] \quad (2)$$

Where $Q(s, a; \theta_i)$ represents a parameterized value approximator, r is the observed reward, γ is the discount factor. $(s, a, r, s')\sim U(D)$ represents the sampled mini batch from the replay memory on which the loss is accumulated. With the loss function defined, we now present Algorithm 1, for training the reinforcement-learning agent.

Set TLimit as the max episode timesteps
Set C as number of steps to update target
For episode = 1 to M:
  Initialize environment
  Observe initial state vector s_0
  Initialize $t = 0$
  While $t < TLimit$:
    With probability ε:
      select random action a_t
    Else:
      select $a_t = argmax_a Q(s_0, a, \theta)$
    Perform action a_t and observe the next state vector s_{t+1} and reward r_t
    Store transition $<s_t, a_t, r_t, s_{t+1}>$ in replay memory
    Sample random minibatch of transitions $<s_j, a_j, r_j, s_{j+1}>$ from replay memory
    Set $y_j = \begin{cases} r_j & if\ episode\ ends\ at\ j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a', \theta^-) & otherwise \end{cases}$
    Perform gradient descent step on $\left(y_j - Q(s_j, a_{j;\theta})\right)^2$ with respect to network parameters θ
    Set $\hat{Q} = Q$ for every C steps
    If s_t is terminal:
      Break
  End While
End For

Algorithm 1. The deep reinforcement learning algorithm for training Lunar Lander

III. Training Initialization

To train our model, we used a 4-layer neural network that takes in an 8-dimensional input and outputs 4 values corresponding to each action, with each hidden layer containing 100 units. We initialize the parameters of the neural network with random weights from a uniform distribution ranging between -0.5 and +0.5.

Along with the neural network, the replay memory was also initialized to fill 20% of its capacity with transitions from episodes played on a random policy.

For the exploration and exploitation problem we start off with epsilon, $\varepsilon = 0.9$, and decay the value with the rule (2) as training steps t increase in the overall training.

$$\varepsilon = 0.05 + (0.85) \cdot e^{-t/d} \quad (2)$$

IV. Results

After tuning for hyperparameters for the described model, we were consistently able to train majority of random seeds for the neural network to converge to an optimal policy. Fig 2(a). Shows cumulative reward per episode at each step of the training and Fig 2(b) Shows the cumulative reward achieved on the game with the trained agent.

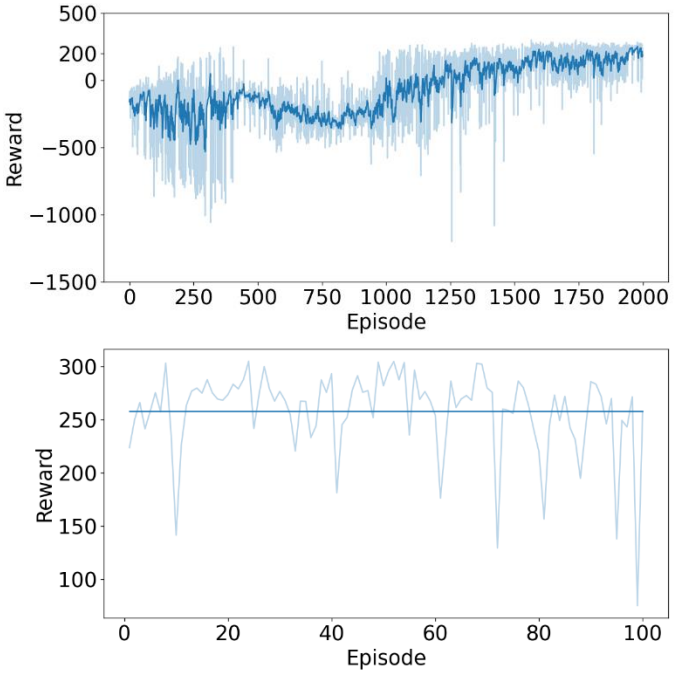

Fig 2 (a). Cumulative reward per episode while training the model. (b) Cumulative reward per episode for 100 episodes played with trained agent, the dark line represents the total average reward.

From Fig 2(a) we observe that as the agent begins to learn, since initially the randomness for the ε-greedy policy is high. The agent mostly attains rewards randomly. After a couple hundred episode the agents slowly begins to learn to not crash in the initial time steps of the episode and therefore learns the value of hovering. This can be explained with the first spike in number of steps as shown in in Fig 3. On continued training with a small epsilon value, the randomness in action selection causes the agent to explore during hovering and eventually learn to attempt to land. This can be seen from the correlated increase in reward along with a decrease in number of steps.

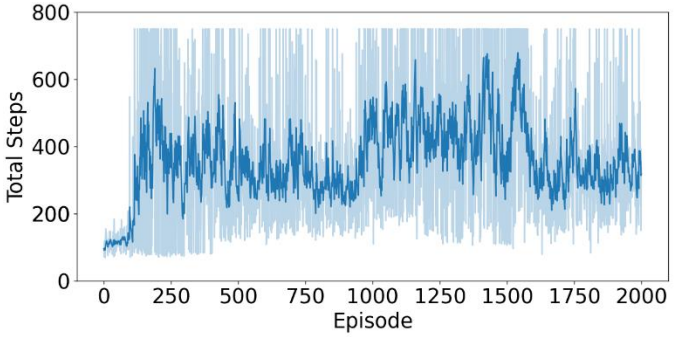

Fig 3. Total steps per episode while training the model.

As training progresses, the agent begins to perfect its landing and decrease crash rate after which we again observe a spike in the average cumulative reward to around 200. After the agent has successfully mastered landing, on continual training, the agent begins to optimize on number of steps for episode to maximize the reward. This again can be observed on the steps-to-episode plot given by Fig 3.

V. Training Analysis

A. *Picking a Suitable Learning Rate and mini-batch size*

To observe the effect of the learning rate alpha, we trained the model with a range of alpha values and plot the cumulative reward for each alpha Fig 4. We also observed that for very high values of alpha, the loss explodes after a few episodes. On the contrary, as we decrease alpha, the model learns a better policy and is able to converge. We also observe that as we lower the values of alpha, we require more time for the model to converge to a better policy. For very low values of alpha, some models almost never learnt the optimal policy.

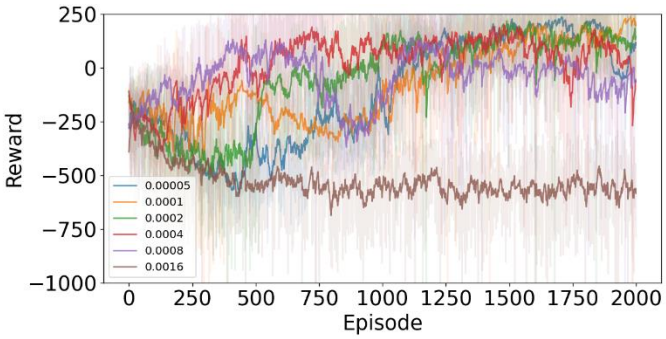

Fig 4 Cumulative reward per episode for the training for varied values of alpha.

The hyperparameter, mini-batch size, for the optimization step had almost similar affects as compared to the learning rate. Perhaps because the accumulated gradient is directly proportional to the size of the mini-batch, similar to alpha.

B. *Effect of replay memory size*

We tested different values of replay memory size to find its optimal capacity. With replay memories initially filled with experiences played from random policies up to 20% of their capacities, for each memory size, we plot the cumulative reward experienced in each episode in Fig 5.

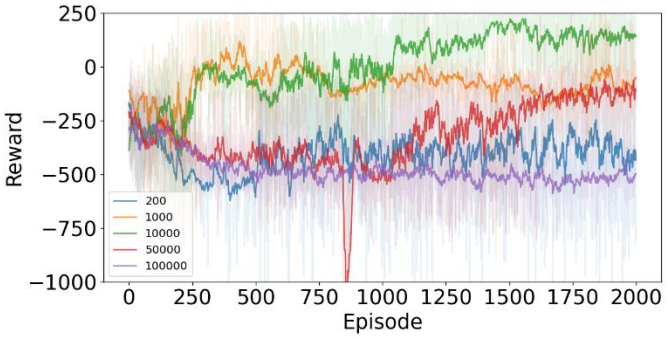

Fig 5 Cumulative reward for varied values of replay memory size.

We note that as we exponentially increase the size of the memory the agent fails to learn anything in first 2000 episodes. This is primarily due to the fact that memory gets flooded with bad transitions, and the probability that a good transition is selected as part of minibatch optimization becomes extremely low. The bigger the capacity, more time the agent takes to collect good transitions and learn from them.

Given enough time, with higher memory size, the agent might perhaps converge to a better policy with more probability once its able to fill its memory with only good transitions. Exponentially smaller memory size has similar problems in that good transitions get flushed out too quickly before they can be sampled for leanring.

C. *Effect of other parameters*

We also measured the effects of other learning parameters such as the discounting factor γ, epsilon decay, and epsilon final values.

- **Gamma**: For lower to medium range of values for gamma, we note that the agent fails to learn any optimal policy.
- **Epsilon decay**: We found that if we decayed epsilon too quickly the agent does not have enough initial exploration to reach any kind of policy and fails to improve at all, for slower epsilon decay we found that either the model diverged after a while or with random chance found good transitions to learn from and converged after epsilon became significantly small.
- **Asymptotic value of epsilon**: We noted that if we decayed epsilon to 0, the agent learns a sub-optimal policy and fails to learn after a point with an average cumulative reward from -10 to 50. Any experiments for higher final values of epsilon do not make sense since it introduces too much stochasticity into the model and the model either fails to learn the optimal policy or diverges.

D. *Experiments with neural networks*

We experimented with multiple combinations of properties for neural networks such as network size, loss functions, and activation functions.

- **Network size:** To optimize on the network, we performed limited experiments with respect to the size of the neural networks. We found that the network generally converged with single and double hidden layers with 100 units each. While any increase in the size of the network caused it to be too slow to converge.
- **Loss functions:** Apart from Mean Squared Error, we experimented with two different loss functions, Mean absolute error and Huber loss. Even with a large range of alpha values, model failed to converge with both the latter losses.
- **Activation functions:** We experimented with two activation functions for the experiment, Relu and

Sigmoid and found that with sigmoid function, we neither diverged nor converged and the agent never learnt an optimal policy. In terms of Q-learning, Relu makes more sense since the output of the network is supposed to estimate the Q-value of the policy, which unlike sigmoid, is not bounded to 1. During our experiments we also noted that Relu due to its unbounded nature, is more sensitive to changes in alpha as compared to Sigmoid.

VI. Pitfalls and Solutions

While training the agent, we encountered several problems that caused the model to result in sub-optimal policies. We talk about two of the most prominent problems that were posed.

A. *Hovering Problem and Episode Bounding*

The hovering problem occurs when, after certain amount of learning, the agent learns to successfully hover but after continued learning the model either diverged, converged with high negative reward, or failed to learn further. We noted this problem in two different situations.

1. **Episode bounding:** we found that if we do not limit the maximum number of time steps in an episode, large unbounded episodes would fill the replay memory with bad transitions and the agent would eventually start to overfit on one action and catastrophically fail. To solve this, we end the episode if number of times steps exceed a limit, we call this episode bounding. We, however, did encounter some random seeds for the neural network that managed to converge to optimal policy with unbounded episodes. Such models were also highly prone to unlearning as training persisted.
2. **Decaying epsilon to zero:** In the initial model of the experiment, we designed for the epsilon to eventually decrease to zero. For the few initial episodes of the training, the agent successfully learns to hover over the launching pad trying to minimize the large negative reward that is incurred due crashing. Once epsilon becomes negligibly small, the agent loses the ability to explore other actions in the hovering states and is doomed to hover over the landing pad and eventually crash. With epsilon decaying to a small value, we give the agent enough randomness to explore the action space.

Unbounded episodes were also victim to another problem, where the agent would safely land on a trough just beside the landing pad and eternally fire its orientation engine to try and move closer to it. To analyze this, we compare successful training sessions of bounded and unbounded models by plotting the time steps for each episode during the training in Fig 6. We see that bounded model ends up converging to small number of steps per episode while unbounded model although converges to an optimal policy, persists with high number of steps even after matured training. Episode bounding solves this with the fact that it does not pollute the replay memory, and any small random action that the agent took to either stop the episode or move closer to the landing pad has a higher probability to get picked up in the mini-batch of the optimization step.

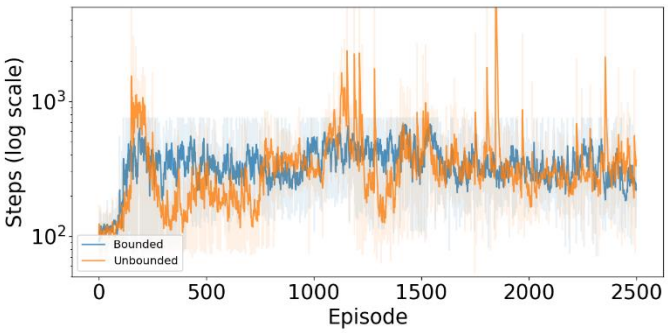

Fig 6. Total steps per episode throughout the training for bounded and unbounded models. The total steps are plotted on the log scale for a viewable comparison.

B. *Unlearning optimal policy problem*

For most of the models we observed that if we continued training after achieving an optimal policy, the agent would unlearn certain "good" actions and perform slightly worse than optimal. Fig.7 plots the cumulative reward per episode for the length of the training. We see that after episode 2000 the agent starts to perform worse and has a dip in the cumulative reward. On continued training the agent starts to oscillate between a suboptimal and an optimal policy. For higher learning rates and continued training, sometimes, the models diverged. This behavior can be explained with two reasons: (1) The behavior of gradient descent, wherein, after encountering the optimal policy and continued learning, we start to oscillate on the global maxima for the network parameters. (2) Bad transitions introduced due to exploration randomness.

To solve this problem, we store the trained model at every small interval of the training session and pick the best trained model for benchmarking. Another solution is to decay the learning rate after certain number of episodes in which the model is ensured to converge to optimal.

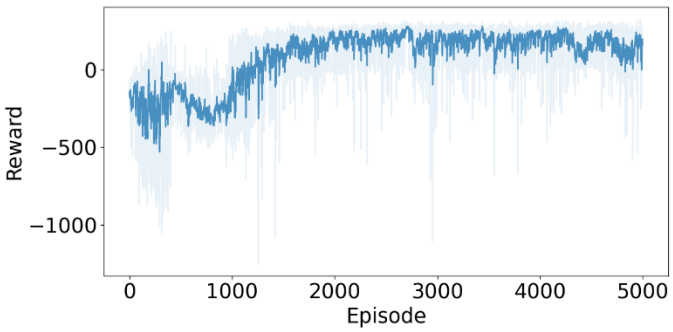

Fig 7 Cumulative reward per episode for continued training after achieving an optimal policy.

VII. Other Experiments

A. *Repay Memory and Target Network*

As done in [1], we also compare the results of our model without the two modifications to the Q-learning algorithm. (1)

With replay memory, without target network (2) Without replay memory, with target network and (3) without replay memory or target network. We plot the cumulative rewards per training episode on Fig 6.

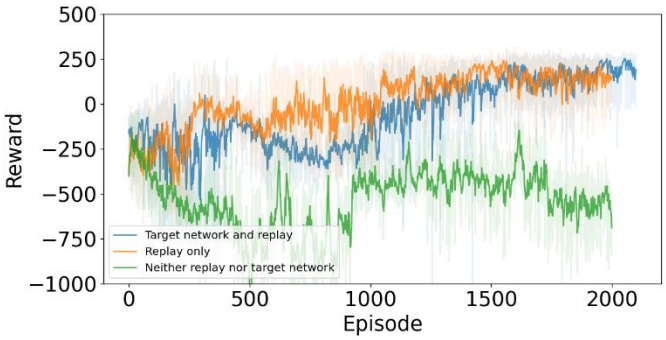

Fig 8 Cumulative reward per training episode for all combinations with replay memory and target network

We note that with replay memory the agent learns a fairly optimal policy and although without target network the model converged faster, we noted with multiple training samples, that model with target network was more reliable over the random seed space. Without replay memory or target network, we found that the agent rarely learns even a suboptimal policy.

B. On-Policy Learning

The Q-learning algorithm we present is an off-policy algorithm. We compare our results to an on-policy algorithm known as Sarsa. Sarsa also forms its basis from the Q-learning update (1). On-policy methods attempt to evaluate or improve the policy that is used to make decisions, whereas off-policy methods evaluate or improve a policy different from that used to generate the data [3].

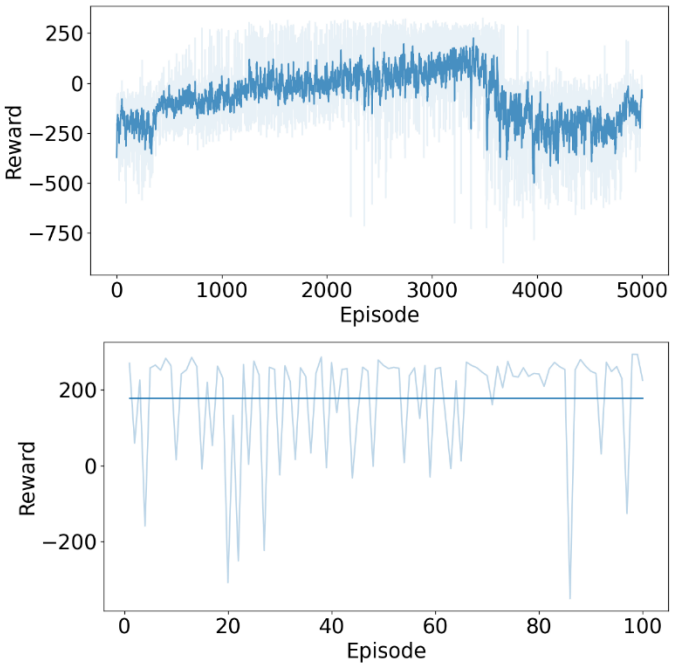

Fig 9(a) Cumulative reward per episode while training for on-policy. (b) Reward per episode on the best trained agent.

We found that the on-policy method successfully converged to a near-optimal policy. Perhaps its optimality would be at par with the off-policy algorithm with better hyperparameter tuning. In Fig 9a. we plot the cumulative reward per episode for the training. In Fig 9b, we plot the reward per episode on the best agent and observe that the average reward per 100 episodes is near-optimal. By comparing the two models we note the following key points about on-policy:

1. On-policy on an average took more time to achieve a near-optimal policy.
2. Was more sensitive to unlearning as compared to off-policy and in-fact showed catastrophic forgetting on continued training.
3. Just like off policy, this method was also victim to the hovering problem, which was solved by bounding the episodes.

It is possible that perhaps our hyperparameter tuning was completely off due to the sheer dimensionality of hyperparameters when training the presented algorithms. With both on-policy and off-policy converging to near optimal policies, the off-policy method, however, yields a more robust model over the space of random seeds and converges to optimal policy with higher probability.

VIII. Future Experiments

Our agent for Lunar Lander was at a significant disadvantage due to its limitation of the state space. The state space provided by the gym environment only provides the positional and movement parameters for the lander along with two extra parameters that convey whether the lander legs are touching the ground or not. A further experimentation of the DQN algorithm would be to train this agent using state space made from the images of the game as originally done in [1]. This would provide the agent an advantage of surveying the land, and perhaps would prevent the eternal engine firing problem on landing on troughs besides landing pad.

One of the experiments performed, not mentioned in Section VII, was punishing the agent with a negative reward if it exceeded the episode bounding limit, we however with limited hyperparameter tuning were not successful in getting the model to converge optimally. An interesting future experiment for this model would be tweaking the reward function to get the model to possible converge faster.

References

[1] Mnih, V., Kavukcuoglu, K., Silver, D. et al. Human-level control through deep reinforcement learning. Nature 518, 529–533 (2015).

[2] Watkins, C.J.C.H., Dayan, P. Q-learning. Mach Learn 8, 279–292 (1992).

[3] Sutton, R.S, Barto A.G, Reinforcement Learing-An Introduction, 2rd ed., 2020

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep Learning. 2016. MIT Press