

AI Agent for Lunar Lander

Prabhjot Singh Rai (prabhjot) Abhishek Bharani (abharani)
Amey Naik (ameynaik)

December 13, 2018

1 Task Definition

The task accomplished by this project is to build an AI agent which would land with constant high rewards on a landing pad, defined by Box2D Lunar Lander available on OpenAI gym. This is accomplished by Reinforcement Learning, particularly by applying different Q-learning techniques. This project has explored Full DQN, Double DQN and Dueling Network Architecture (Dueling DQN), their performances in "solving" the game. We have considered a game to be solved when the agent starts getting average reward of 200 over 100 consecutive episodes. Moreover, performances have also been compared with baselines and oracle.

2 Infrastructure

We have used OpenAI library as our infrastructure. Although some insights are provided in Box2D Lunar Lander on OpenAI website, but thorough exploration of actions, state space, environment etc. were done before starting to solve the problem. Following is the description:

2.1 Actions

In this game, four discrete actions are available to the playing agent at any time frame:

- (a) Do nothing
- (b) Fire left orientation engine (rotates the lunar lander clockwise)
- (c) Fire main engine (provides upward thrust)
- (d) Fire right orientation engine (rotates the lunar lander anti-clockwise)

The agent can choose only one action among the given actions at a given time frame.

2.2 Terrain

The terrain is a combination of 10 points, and the helipad(landing zone) is fixed between 5th and 6th points towards the center. The values of the height of the landing zone(5th and 6th points on the terrain) are viewport height divided by 4, and the rest of the points are randomly sampled between 0 to $H/2$ using *numpy* random and smoothened (averaging 3 continuous points).

2.3 Initial State

The state is 8 dimensional values of different parameters of the lunar lander at any given time. The starting state is randomly initialized (the lunar lander takes a step in the world through the "idle" action) with certain bounds based on the environment.

Inspecting code of open_ai lunar lander, we see that the initial states are defined by simulating the environment in one frame (calling box world simulation using time step as $1/FPS$). Using this simulation, elaborating the initial state as follows:

1. Position X (Initial Position X: final x which changed from half of viewport width to a value after taking "idle" action before the simulation)
2. Position Y (Initial Position Y: final y which changed from from the viewport height to a value after taking "idle" action before simulation)
3. Velocity X (Initial Velocity X: final velocity x changed from 0 to a value after taking "idle" action before simulation)
4. Velocity Y (Initial Velocity Y: final velocity y which changed from 0 to a value after taking "idle" action before simulation)
5. Current lander angle (Initial lander angle: final lander angle after simulation on "idle" action from 0 degrees)
6. Angular velocity (Initial angular velocity: final angular velocity after simulation on "idle" action from 0 angular velocity)
7. Left leg contacted the surface (Initial value: False, since there's is no probability that the lunar lander's leg will touch the moon surface when at the top)
8. Right leg contacted the surface (Initial value: False)

2.4 End State

The episode ends in the following scenarios:

1. When the lunar lander goes outside of the viewport bounds, the game is over with -100 is negative reward.
2. When the lunar lander touches the ground with a high velocity
3. When the lunar lander touches the ground with body part except the legs
4. When the lunar lander stabilises on the moon's surface (change in shape of lunar lander is constantly 0 for a number of frames)

2.5 Rewards and Transitions

Before defining the rewards, let's define the shape of the lunar lander which decides the rewards. The shape of the lunar lander is a function of position coordinates (x, y) , linear velocities (v_x, v_y) , lander angle θ and contact of both the lander legs. We are interested in finding the change of shape at every step for the lunar lander to calculate the rewards for each given action. Shape change is given by subtraction of previous shape and current shape. Formally, shaping at time frame t :

$$\begin{aligned}
\text{shaping}_t = & -100 * (x^2 + y^2) \\
& -100 * (v_x^2 + v_y^2) \\
& -100 * \text{abs}(\theta) + 10 * (\text{Left leg contacted}) + 10 * (\text{Right leg contacted}) \\
\text{shape change} = & \text{shaping}_t - \text{shaping}_{t-1}
\end{aligned}$$

The rewards are defined as follows:

1. If the lunar lander crashes, or goes out of the bounds: -100
2. If the lunar lander is not awake anymore (stabilises at 0 shape change): $+100$
3. Doing nothing: shape change
4. Firing the engine: shape change - 0.3
5. Rotating: shape change - 0.03

The total reward will automatically be a sum of all the rewards at each time frame, and if the lander touches the ground with it's legs, will add those rewards to the total rewards earned during an episode. Transition probabilities are unknown, we get next states by simulating the lunar lander in the box environment given the current state and action taken.

Note: The transition probabilities and rewards are unknown to our agent, which it will try to figure out through exploration and incorporate in learning.

3 Models and different Approaches

3.0.1 Basic Understanding and Motivation

3.1 DQN

3.2 Double DQN

3.3 Dueling layer DQN

3.3.1 PER DQN

4 Experiments and Evaluation

In this section, we will present experimental results for all three different models presented in previous section as well as the visualization of training process.

4.1 Baselines

The first baseline is purely a random approach. The Agent was taking random actions and this is just to make sure our agent outperforms a random one. The second baseline is a simple linear classifier.

Model	Avg. Score	Number of episodes to reach 200 score
Baseline	-200	never
Linear Baseline	-150	never
DQN	123	525
Double DQN	220	500
Dueling DQN	225	435

Table 1: Comparison of Results for different model implementations

4.2 Training

In the most proposed models an initial learning rate of 10^{-4} was used to initialize training. Adam Optimizer is used for full duration of number of episodes. All models can be trained within 30-40 mins on CPU as input to the neural network is a state vector. All the models were trained for 800 episodes with batch size of 32 or 64. The number of episodes was chosen based on convergence of the loss, while batch size was chosen to be relatively small to get the benefits of stochasticity. We tried different epsilon decay rates to get the best results.

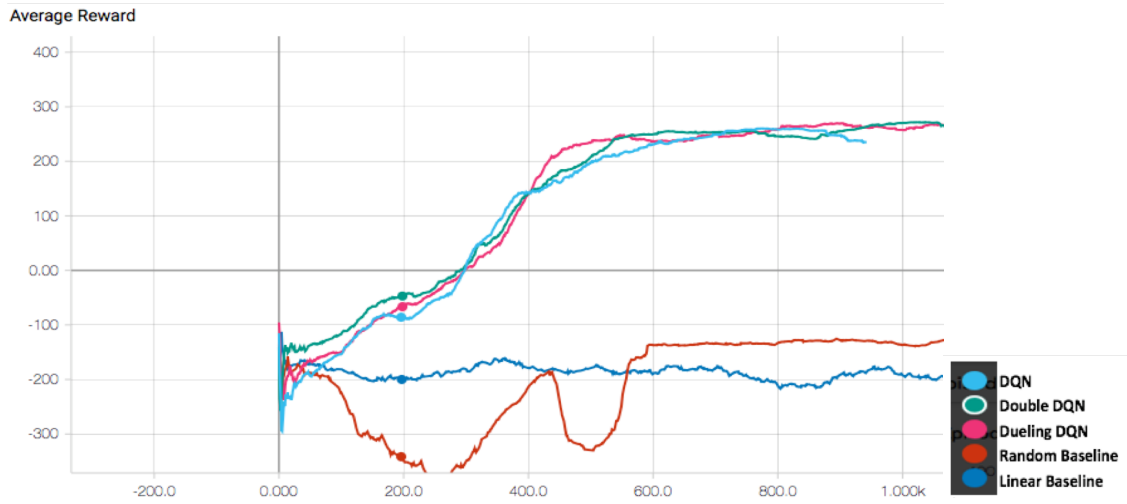


Figure 1: Rewards for different approaches on tensor board

4.3 Hyperparameter Tuning

- We tested with different learning rate of 0.01, 0.002, 0.005 and 0.001. We noticed our model gave best result with 0.001
- We also tried different epsilon decay and got the best results at 0.995
- We tried various combinations of batchsize(32, 64 128). Of all combinations, we saw best score with batch size of 64 across all models.

4.4 Analysis

Error analysis: After doing hyper-parameter tuning, our Agent was able to achieve average score of more than 200 quickly (at 435 episodes). However for some episodes after 435 the rewards were not consistently more than 200 for 100 iterations and we observe the variation is more in DQN and Double DQN as compared to Dueling DQN. We got the best performance on Set 4 with Dueling DQN model.

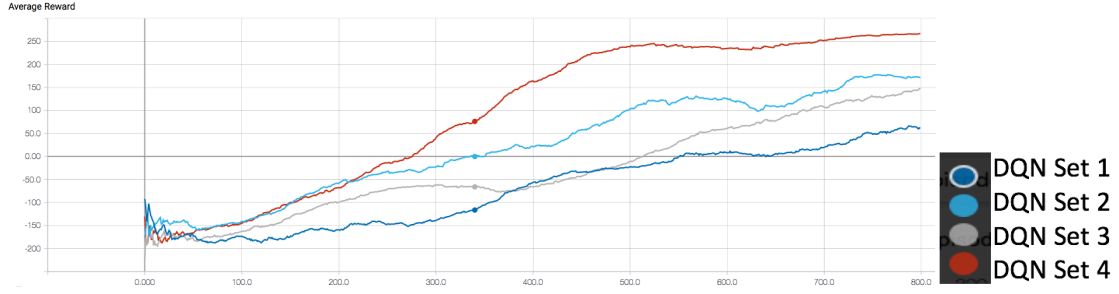


Figure 2: Different Sets of hyperparamerts for DQN Model

Hyper-parameter	Set 1	Set 2	Set 3	Set 4
gamma	0.99	0.99	0.99	0.99
Epsilon(max,min,decay)	(1,0,.998)	(1,.01,.995)	(1,.01,.998)	(1,.01,.998)
Learning Rate	0.001	0.0001	0.0001	0.0001
DNN layers	[32,32]	[128,32]	[128,64]	[128,64]
Loss Function	MSE	MSE	MSE	MSE
Batch Size	32	32	64	64
Replay Memory Size	2^{16}	2^{16}	2^{16}	2^{16}

Table 2: Different set of hyper-parameters were tried

4.5 Model Evaluation

All aforementioned models are evaluated for 100 episodes after 450 episodes.

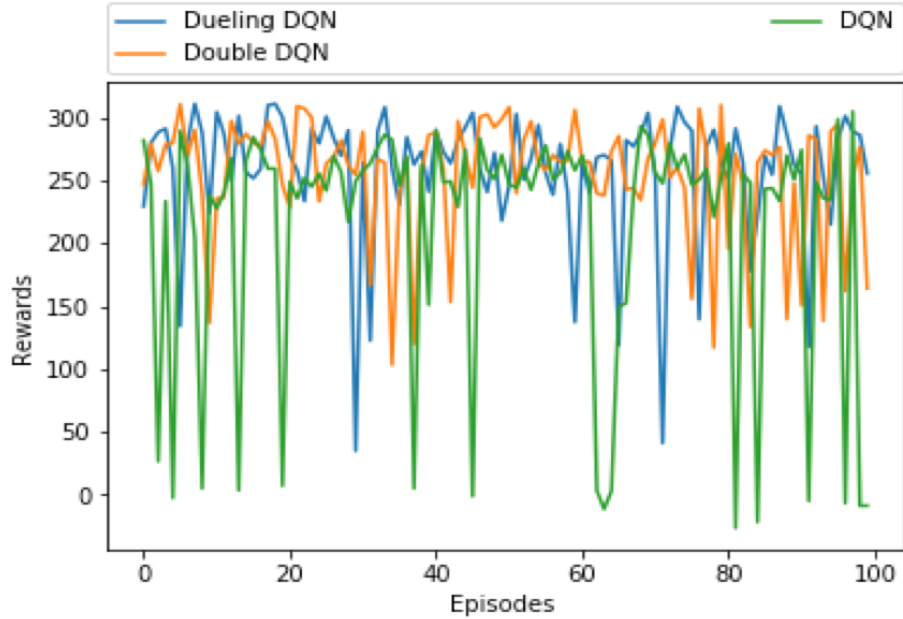


Figure 3: Evaluating Performance of different DQN Networks

4.6 Regularization

5 Conclusion

6 Codalab Link

<https://worksheets.codalab.org/worksheets/0x3f15e8eba6af45828a26df3bc0e1f490/>
command to run

```
cl run :brain.py :hyperparameters.py :run.py :agent.py :environment.py :memory.py `python3 run.py` -n  
sort-run --request-docker-image prabhjotrai/openai-gym:v1
```