AI Agent for Lunar Lander

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

December 14, 2018

1 Task Definition

The task accomplished by this project is to build an AI agent for the game of Lunar Lander defined by openAI gym in Box2D format. Here, a lunar lander needs to land with zero velocity between the flags on a landing pad as shown in the figure. with a constant high reward. This is accomplished by Reinforcement Learning, particularly by applying different Deep Q-learning techniques. This project has explored Full DQN, Double DQN, and Dueling DQN, to solve the game. We have considered the game as solved when the agent starts getting an average reward of 200 over 100 consecutive episodes. Moreover, performances of different DQN variants are compared with baselines and oracle.

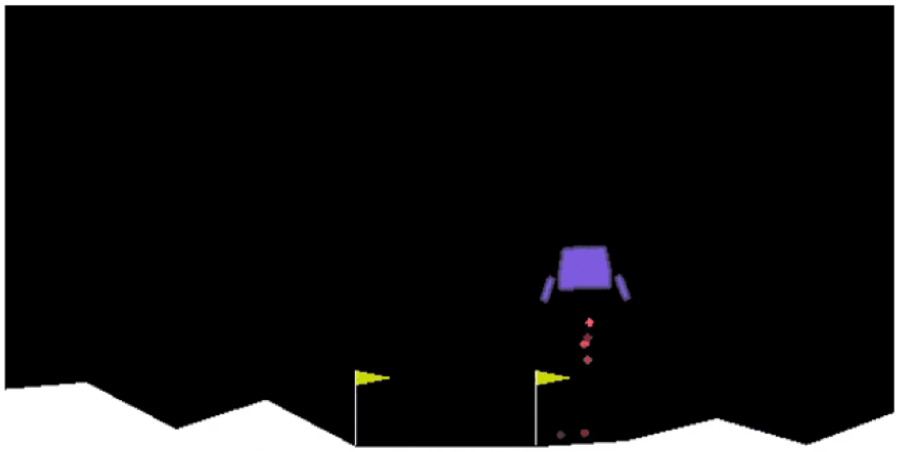

Figure 1: Game Environment Screenshot

Since this problem involves high dimensional and continuous state space, standard Q-learning cannot solve this problem unless some amount of discretization is done. Due to above-mentioned difficulties, Deep Q-Network (DQN) was the choice. We plan to explore different variants of DQN (particularly DQN, Double DQN, and Dueling network architecture).

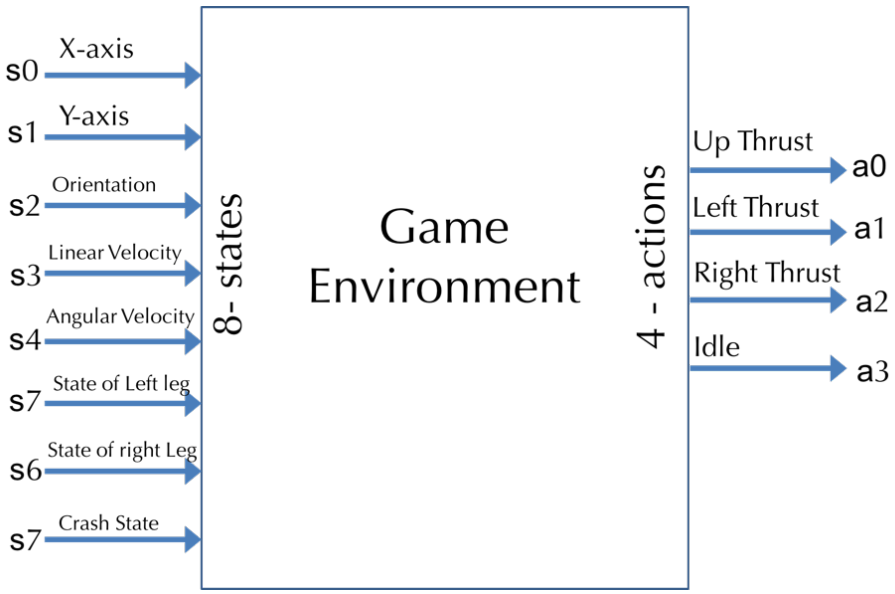

Figure 2: Game Environment

2 Infrastructure

We have used the OpenAI gym library to train our agent. Although some insights are provided in Box2D Lunar Lander on the OpenAI website, thorough exploration of actions, state space, environment etc. was 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:

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

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

3.1 Challenges

We dedicated time to understand how the openai-gym is working internally. For this specific problem, we understood how the rewards and environment is behaving on every episode, as described in infrastructure. Moreover, to get faster results, we focused on better machine and GPU settings on google cloud and getting docker environment setup for codalab and reproducibility across platforms.

3.2 Modelling the problem

Basic framework is defined in Fig 1. The agent performs action(a) on the environment. Performing the action on the environment returns reward(r) and new state s'. Our agent then incorporates this information.

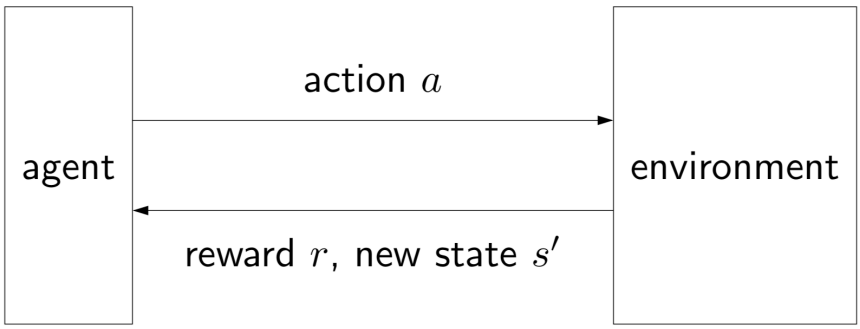

Figure 3: Neural Network

In such a model, for our problem, here is quick description of notations:

1. S is defined as all possible states, s is one particular 8 dimensional state
2. A is defined as all possible actions, a is one of the actions out of [idle, fire engine, rotate left, rotate right]
3. R is the reward distribution given (s, a)

4. P is the set of possible transitions and their probabilities given (s, a)
5. γ: the discount factor. How much we want our agent to discount future reward. It is a hyperparameter that we define ourselves.

Our model builds off of Q-learning algorithms by using a Deep Neural Network (DNN) for approximating the state-action Q-value, Q(s, a). Given the state s, our goal is to identify a policy π_{opt} that maps the states to actions in order to maximize the total reward we get.

$$\text{S} \rightarrow \text{A based on } \pi_{opt}$$

As we know, Q value is defined as the expected reward that we get following action a in a given state s and then following the policy π, our objective is to define a $Q_{opt}(s, a)$, which can be maximised over all possible actions at a state s, in order to find the π_{opt}.

3.3 Algorithms and Equations

3.3.1 DQN Introduction

Model-free based Deep Q Network algorithm was chosen specifically for the state size and complexity. DQN builds off of Q-learning algorithms by using a Deep Neural Network (DNN) for approximating the state-action value function, Q(s, a). Function Q(s, a) is defined such that for given state s and action a it returns an estimate of a total reward we would achieve starting at this state, taking the action and then following some policy. Let's call the Q function for the optimal policies Q_{opt}. Q_{opt} with discounting can be written as

$$Q_{opt}(s, a) = r_0 + \gamma r_1 + \gamma^2 r_2 + ... \tag{1}$$

Here, r stands for rewards. γ is called a discount factor and when set it to $\gamma < 1$, it makes sure that the sum in the formula is finite. The γ controls how much the function Q in state s depends on the future and so it can be thought of as how much ahead the agent sees.
The above equation can be rewritten in a recursive form.

$$Q_{opt}(s, a) = r_0 + \gamma max_a Q_{opt}(s', a) \tag{2}$$

This equation is proven to converge to the desired Q_{opt}, with finite number of states and each of the state-action pair is presented repeatedly. However, the Lunar Lander state space is real and continuous. We cannot store infinite number of values for every possible state. Instead, we approximate the Q function with a neural network. This network will take a state as an input and produce an estimation of the Q function for each action. This network with multiple layers is called Deep Q-network (DQN).

Experience Replay

During each simulation step, the agent perform an action a in state s, receives immediate reward r and come to a new state s'.
There are two problems with online learning -
1. The samples arrive in order they are experienced and as such are highly correlated. This might cause overfitting.
2. Throwing away each sample immediately after we use it. This means we are not using our experience effectively.
The key idea of 'experience replay' is that we store these (s, a, r, s') transitions in a memory and during each learning step, sample a random batch and perform a gradient descend on it. This way we solve both issues. After reaching the finite allotted memory capacity, the oldest sample is discarded.

Exploration To find out that actions which might be better then others, we use ϵ greedy policy. This policy behaves greedily most of the time, but chooses a random action with probability ϵ. ϵ will be a hyper-parameter played with to make sure our agent learns fast enough while consistently performing better.

3.3.2 Full DQN

In DQN algorithm we set targets for gradient descend as:

$$Q_{opt}(s,a) = r_0 + \gamma max_a Q_{opt}(s',a) \tag{3}$$

We see that the target depends on the current network. A neural network works as a whole, and so each update to a point in the Q function also influences whole area around that point. And the points of $Q(s,a)$ and $Q(s',a)$ are very close together, because each sample describes a transition from s to s'. This leads to a problem that with each update, the target is likely to shift. This can lead to instabilities, oscillations or divergence.

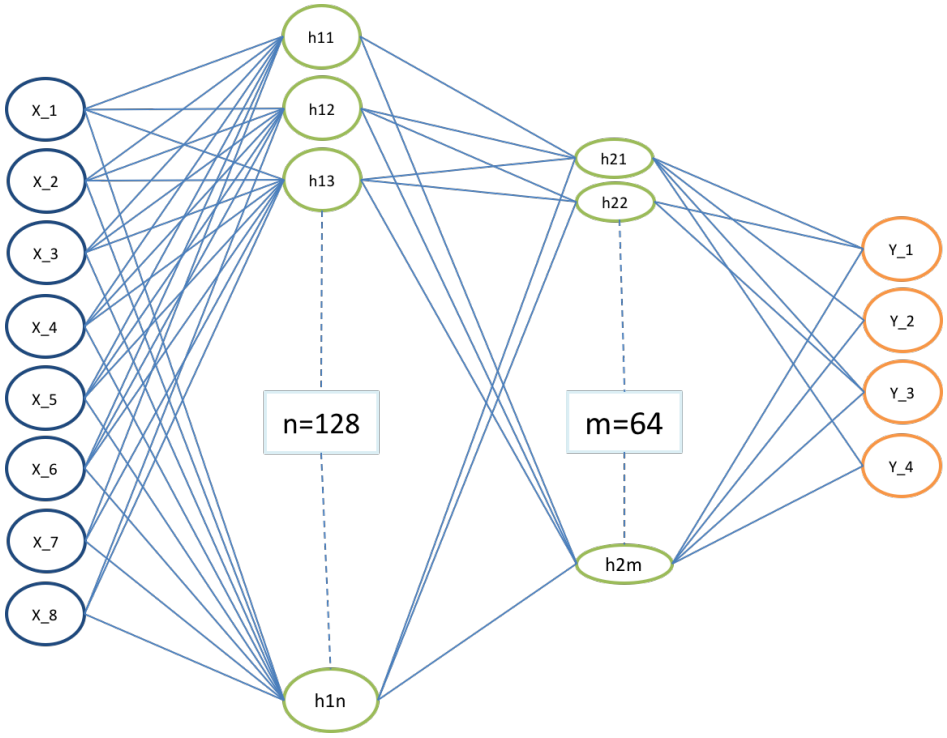

Figure 4: Neural Network

To overcome this problem, researches proposed to use a separate target network for setting the targets. This network is a mere copy of the previous network, but frozen in time. It provides stable $\tilde{Q}$ values and allows the algorithm to converge to the specified target:

$$Q(s,a) \rightarrow r + \gamma max_a \tilde{Q}(s',a) \tag{4}$$

After several steps, the target network is updated, just by copying the weights from the current network. To be effective, the interval between updates has to be large enough to leave enough time for the original network to converge.
For lunar lander, we update target model after every episode.

3.3.3 Double DQN

One problem in the DQN algorithm is that the agent tends to overestimate the Q function value, due to the max in the formula used to set targets. Because of the max in the formula, the action with the highest

positive/negative error could be selected and this value might subsequently propagate further to other states. This leads to bias – value overestimation. This severe impact on stability of the learning algorithm.
In this new algorithm, two Q functions Q_1 and Q_2 – are independently learned. One function is then used to determine the maximizing action and second to estimate its value. Either Q_1 or Q_2 is updated randomly with a formula:

$$Q_1(s,a) \rightarrow r + \gamma Q_2(s', argmax_a Q_1(s',a)) \tag{5}$$

or

$$Q_2(s,a) \rightarrow r + \gamma Q_1(s', argmax_a Q_2(s',a)) \tag{6}$$

It was proven that by decoupling the maximizing action from its value in this way, one can indeed eliminate the maximization bias.
When thinking about implementation into the DQN algorithm, we can leverage the fact that we already have two different networks giving us two different estimates Q and $\tilde{Q}$ (target network). Although not really independent, it allows us to change our algorithm in a really simple way.
The original target formula would change to:

$$Q(s,a) \rightarrow r + \gamma \tilde{Q}(s', argmax_a Q(s',a)) \tag{7}$$

We could observe that Double DQN was more stable than Full DQN.

3.3.4 Dueling layer DQN

Q(s,a) represents the value of a given action a chosen in state s, V(s) represents the value of the state independent of action. By definition, $V(s) = max_a Q(s,a)$. Thus, A(s,a) provides a relative measure of the utility of actions in s. The insight behind the dueling network architecture is that sometimes the exact choice of action does not matter so much, and so the state could be more explicitly modeled, independent of the action. There are two neural networks — one learns to provide an estimate of the value at every timestep, and the other calculates potential advantages of each action, and the two are combined for a single action-advantage Q function. We can achieve more robust estimates of state value by decoupling it from the necessity of being attached to specific actions.FIXME

$$Q(s,a) \rightarrow A(s,a) + V(s) \tag{8}$$

The above equation is unidentifiable in the sense that given Q we cannot recover V and A uniquely. This lack of identifiability is mirrored by poor practical performance when this equation is used directly. To address this issue of identifiability, we can force the advantage function estimator to have zero advantage at the chosen action. That is, we let the last module of the network implement the forward mapping.

$$Q(s,a;\theta,\alpha,\beta) \rightarrow V(s;\theta,\beta) + (A(s,a\theta,\alpha) - max_{a'\epsilon|A|} A(s,a';\theta,\alpha)) \tag{9}$$

An alternative module replaces the max operator with an average. On the one hand this loses the original semantics of V and A because they are now off-target by a constant, but on the other hand it increases the stability of the optimization: with this following equation-

$$Q(s,a;\theta,\alpha,\beta) \rightarrow V(s;\theta,\beta) + (A(s,a\theta,\alpha) - \frac{1}{A}\sum_{a'} A(s,a';\theta,\alpha)) \tag{10}$$

the advantages (A) only need to change as fast as the mean, instead of having to compensate any change to the optimal action's advantage in (13)

3.4 Accuracy and Efficiency Trade-off

For our solutions, accuracy can be considered as consistency in getting more than 200 scores over a period of episodes. And efficiency would be how sooner we can get such weights for which the agent scores more than 200 scores.

When we train for more number of episodes (low efficiency), we get high accuracy (drop between consecutive episodes is lesser). And vice versa, if we stop early, we are highly efficient but accuracy is lower.

3.5 Implementation choices

The choice of state was based on the actual configuration of the lunar lander, and we learnt weights for each feature in the state. Another choice could have been training on image data, feeding images per episode and learning on gained rewards. Since this would have been computationally expensive, we chose the former approach to explicit definition of state space and learning it's weights.

For running different algorithms, we created separate classes so that the code is not only modularized, but can also be run easily on different openai environments, learning algorithms can be easily changed etc. Here's a quick description of different classes:

1. **Brain:** Class which contains keras models, updates the weights through train function and performs prediction based on learnt weights.
2. **Memory:** Class which appends observations until maximum memory length and samples based on given batch size hyperparameter
3. **Agent:** Our agent class which explores and exploits based on fixed hyperparameters(gamma, epsilon max, epsilon min and decay) and passed arguments. This is also the class where we are performing the replay action and training the agent's brain instance. It also contains another instance of memory class which is used in replay while sampling.
4. **Environment:** Class which runs the episode on given agent and asks the agent to observe and replay whenever the agent is trying to learn on episodes. It returns the information on how much reward was observed on each episode and for how long each episode ran

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.

4.2 Oracle

We defined our oracle to be human playing score and the leaderboard scores. In order to create a simulation where humans could actually play the game and collect data, we used openai gym's implementation code. We recorded observations for the same after playing the game 10 times each.

We also compared our results with the ones on OpenAI Gym Leaderboard Wiki. In general, the human oracle scores are less because speed of the LunarLander doesn't affect AI agent however it makes the game hard to play for humans.

Iteration	Prabhjot	Abhishek	Amey
1	180	80	−90
2	200	100	170
3	−10	−90	180
4	150	−90	160
5	20	180	30
6	10	120	20
7	120	60	150
8	130	200	−100
9	−90	210	90
10	−80	−80	30
AVG	63	69	82

Table 1: Human Oracle Scores

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 2: Comparision of Results for different model implementations

4.3 Literature Review

Since the game of Lunar Lander is available on openAI gym platform, we came across various implementations of it online. We also found a wiki-driven leaderboard is available for a small amount of comparative benchmarks. [FIXME add ref]. We found [FIXME Allan Reyes ref] work to be well documented. Here the AI agent was trained using simple DQN. We started with the same approach of simple DQN, and after experimenting with hyperparameters we could achieve better results of getting reward of 200 within 525 episodes. Later two more algorithms Double DQN and Duel DQN further enhanced the results. So far stand 2nd place on the leaderboard. [FIXME ref]

4.4 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 choosen based on covergence of the loss, while batch size was choosen to be relatively small to get the benefits of stochasticity. We tried different epsilon decay rates to get the best results.

4.5 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 and 128). Of all combinations, we saw best score with batch size of 64 across all models.

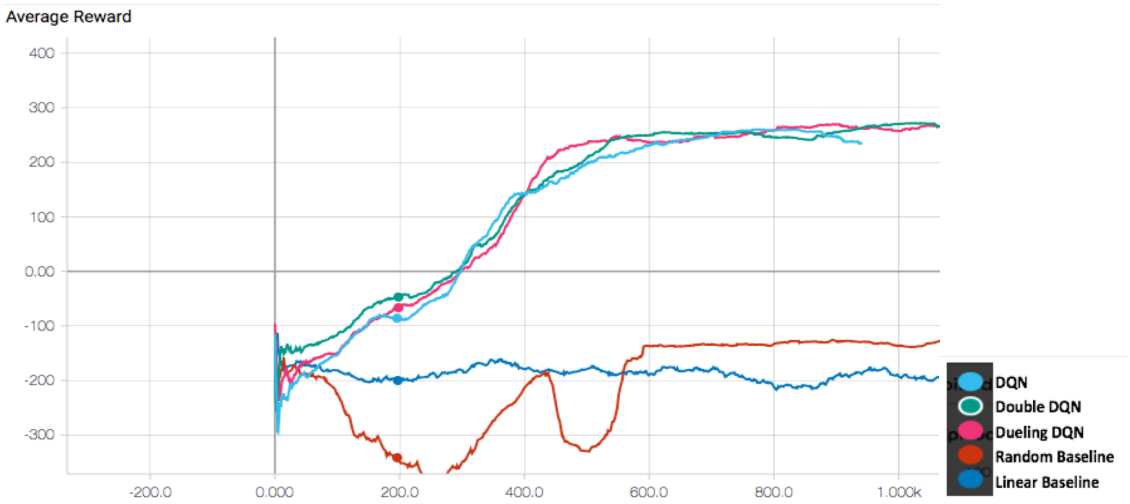

Figure 5: Rewards for different approaches on tensor board

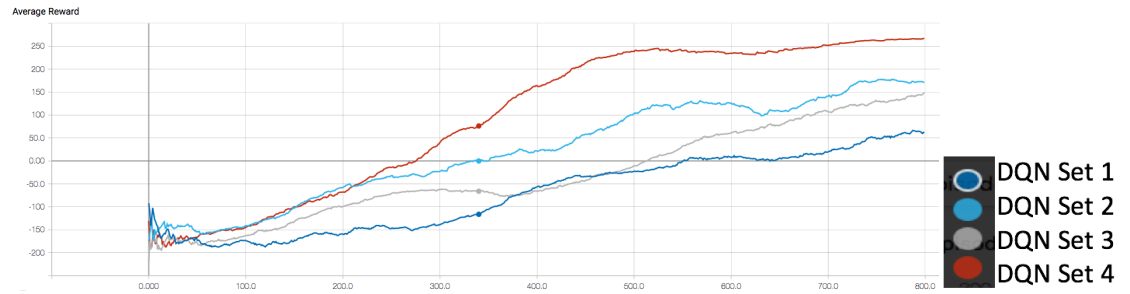

Figure 6: Different Sets of hyperparamerts for DQN Model

4.6 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.

4.7 Model Evaluation

All aforementioned models are evaluated for 100 episodes for a fixed weights.

5 Conclusion

We were successful in implementing an AI agent which was able to score an average reward of more than 200 for 100 consecutive episodes and compared different variants of DQN. DQN was applied effectively on this specific problem, and produced successful results. The success strengthens the use and generality of this algorithm on other problems. We tried it on "CartPole-v1" game defined on openAI gym platform. Our AI agent successfully learnt the game and consecutively won it. This proves the generality of the algorithm.

An agent with too high of a discount was unable to credit actions to success, far enough in the future. Simply put, it was too myopic. As a result, agents learned how to hover, but never learned how to land.

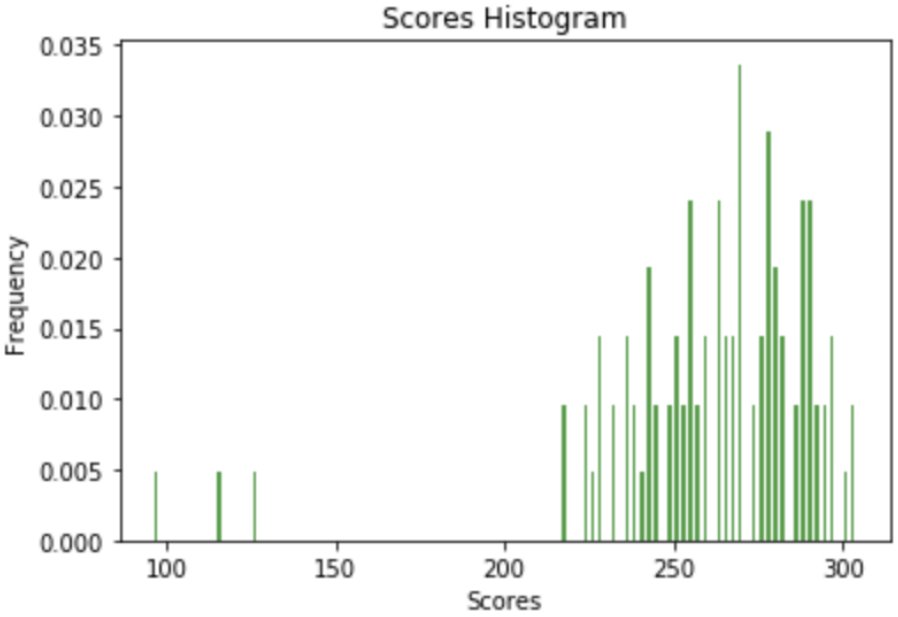

Figure 7: Histogram Distribution of 100 scores after exploration (ϵ) is set to 0

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 3: Different set of hyper-parameters were tried

We also observed effect of having smaller replay-memory size. Sometimes a drop in the rewards after model learns is because after many consecutive successes, the replay buffer won't have many failure cases to train on. So, it used to 'forget' how to recover from many failure cases.

The DQN variants we tried gave very good results once the hyperparameters were tuned correctly. When implemented the algorithm code in a modularized format, we could easily change the parameters and play with it. Also, minimal tweaks were required for each the DQN variant. Not much of hyperparameter tuning was required across DQN variants. The more advanced Duel Q-learning learner undoubtedly increased the accuracy of the reinforcement learning agent. Given more time and resources, the agent could have been tuned via a more exhaustive grid search. After some literature review, we tried out DQN with Prioritized replay as well. Unfortunately, the results were not promising in comparison to rest of the three approaches.

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

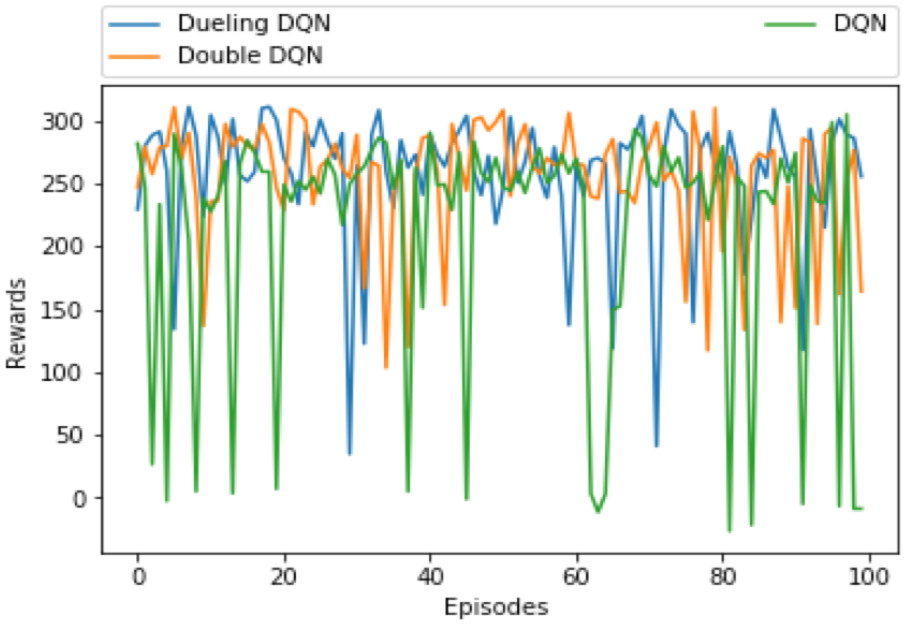

Figure 8: Evaluating Performance of different DQN Networks

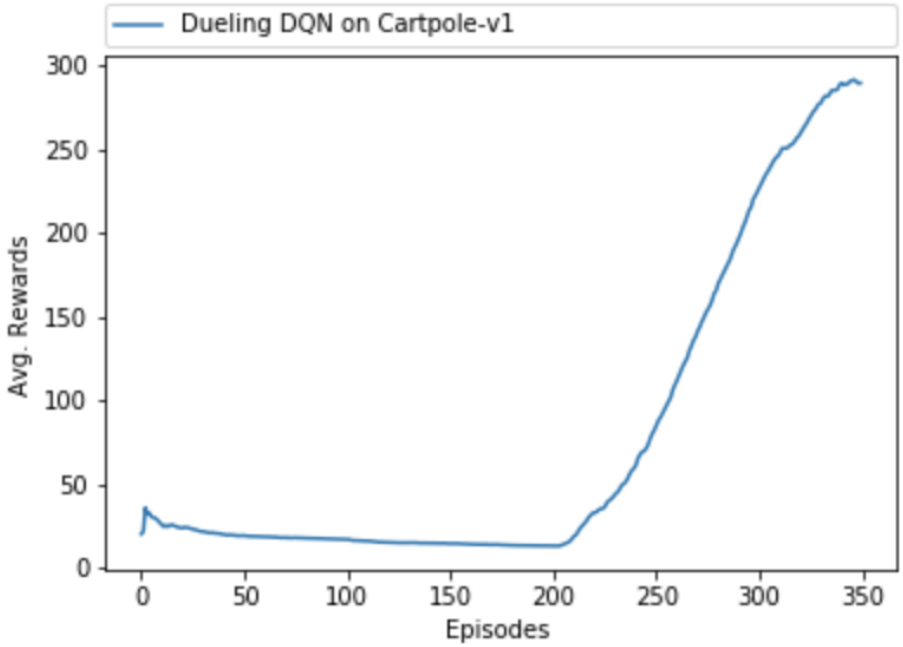

Figure 9: Agent learning CartPole Game

sort-run –request-docker-image prabhjotrai/openai-gym:v1

References