Solving the Lunar lander environment with Reinforcement Learning

Neel Bhave and Winston Moh Tangongho
Khoury College of Computer Sciences
{mohtangongho.w, bhave.n } @husky.neu.edu

10 December 2019

Problem Statement

Rocket trajectory optimization is a classic topic in Optimal Control. Lunar Lander is an OpenAI environment which includes a Space-ship which tries to land safely on a Pad. This problem requires an Intelligent Agent which can learn to control the Space-ship and land it properly on the pad without crashing it. This environment is close to a real-world environment due to its continuous state-space. Hence it is not possible to get an Optimal policy using the traditional Q-learning method in Reinforcement Learning. To that effect, a different approach needs to be adapted to solve this problem.

Environment

The Lunar lander environment is provided for us in OpenAI Gym. It has a state space with 8 attributes which include:

- X - position
- Y - position
- X - velocity
- Y - velocity
- Angle of the Lander
- Angular velocity
- If the left leg has touched the ground
- If the right leg has touched the ground

And an action space of 4 attributes which include:

- Fire left engine
- Fire right engine
- Fire center engine
- Do nothing

The important thing about this environment was that the first six attributes in the state space of Lunarlander were all continuous values which ranged from -∞ to $+\infty$. This caveat made using regular Q-learning algorithms a lot harder and not that efficient. Hence, Deep learning algorithms were used. Next, we talk about setting up the OpenAI gym environment together with our collaboration IDE Google colab.
i) OpenAI Gym installation
We used OpenAI gym for our environment set-up together with the gym[box2d] module. First, we imported the gym module into our program and then created an environment variable by calling $gym.make(`LunarLander-v2')$. Next, we initialized our agent by calling an Agent class with the number of actions and state attributes as parameters. Finally, the Agent is trained using the DeepMind Deep Q Network algorithm till the mean scores of episodes in the replay window is greater than 200 and the Agent is able to land successfully 7 out of 10 times. At this point, a sample of the episodes can be executed and the Agent can be seen to act optimally.
ii) Google Colab - used for training Agent
To improve collaboration among team members, we used Google Colaboratory to train our Agents. We trained and tested our Agents using Google Colab and did some parameters tuning to get the most efficient Agents.

Approaches

We used the following methods to solve the Lunar Lander environment:
Q-Learning Approach - We tried to develop an optimal for the agent using regular Q-learning technique.Regular Q-learning technique works by creating a Q-table which maps each state to specific Q-value for each available action. This method is suitable for environments with finite state space representation with finite number of actions,as we have to create a Q-table in the memory,which needs to be of finite dimensions. In case of Lunar Lander,the state space is represented by continuous variables which have values ranging from -∞ to $+\infty$. Hence its not possible to create,store and operate on a matrix of such dimension. Even if the values are discretized to the first decimal, the dimension of the matrix are still not practical to be stored in the memory.
Approximate Q-learning approach - The approximate Q-learning technique is used when the possible state-space values are very large. In the approach, we have a set of feature functions and a weight for each feature function. The weighted sum of the feature function and the weights given a state and an action the feature functions give us the Q-value of that state. Here we use features to

generalize that experience to new but similar situations. We use linear-function approximation to update the value of the weights and try to develop the optimal set of weights,which would give us the accurate Q-value of the state given an action.
However, in the case of the Lunar-Lander, we do not have the set of feature functions. In order to develop reliable feature functions,we need to know the internal working of the environment,which is unknown, as we can only get the values representing the state and the reward at each step. Also the various properties of the environment are interdependent. Hence it is very difficult and time consuming to develop reliable feature functions for the Lunar-Lander environment.
***Deep Reinforcement Learning* -** Deep Learning has made it possible to extract high dimensional raw sensory data. The most successful approaches are trained directly from the raw inputs, using light-weight updates based on stochastic gradient descent. By feeding sufficient data into deep neural networks, it is often possible to learn better representations than handcrafted features [1]. Hence Deep Reinforcement Learning connects a reinforcement learning algorithm to a deep neural network and efficiently processes training data by using stochastic gradient updates. Hence Deep Reinforcement Learning seems to be a suitable approach for solving a complex environment like the Lunar Lander, which has a large number of possible states.The next section explains the different Deep Reinforcement Learning variants we used to train the agent and get the desired result.

Deep Reinforcement Learning

SARSA Agent

This Agent works on the SARSA algorithm [2]. Here we store the most recent observations, i.e. the State, Reward, Action, State, Reward tuple. We then select the most recent SARSA tuple and use it to update our network parameters. As we pick the most recent experience to update our network parameters, there is a high correlation between the samples which is not efficient. This may lead to unnecessary feedback loops and getting stuck in a local minimum. This happens because the current parameters determine the next data sample on which the parameters are trained on. Hence, SARSA Agent does not always converge to the optimal policy or may take a large number of episodes to get one.

DQN Agent

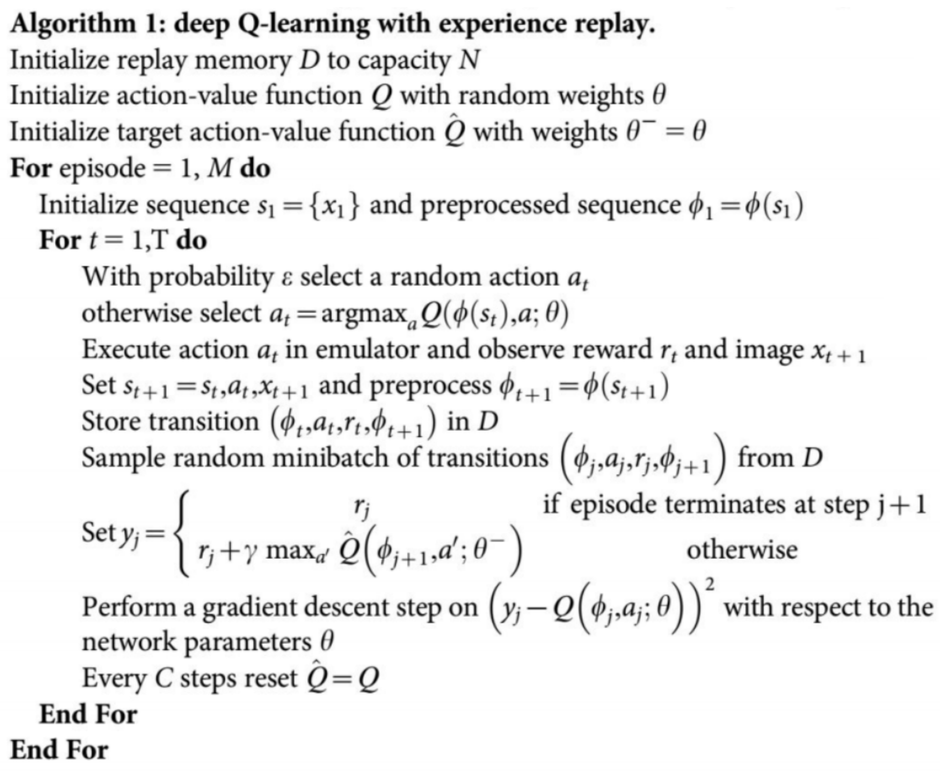
The goal here is to make Q-learning look like supervised learning. There are two main ideas for stabilizing Q-learning.
1) Apply Q-updates on batches of previous experience instead of online - This makes the data distribution more stationary. Experience replay has the largest performance improvement in the DQN.
2) Use an older set of weights to compute the targets(Target Network) - Keeps the target function from changing too quickly. We create two target networks,

θ^- and θ. The first one retrieves the Q-values while the second one includes the updates in the training. After an n number of updates to the network, we synchronize θ^- and θ to fix the Q-values temporarily so we don't have a moving target to chase. The gradient descent step equation on the network parameters is shown below in figure 1. Using both Experience replay and a target network

$$L_i(\theta_i) = \mathbb{E}_{s,a,s',r\sim D}\left(\underbrace{r + \gamma \max_{a'} Q(s', a'; \theta_i^-)}_{\text{target}} - Q(s, a; \theta_i)\right)^2$$

Figure 1: Target Network Update [3]

leads to a more stable input and output to train the network and behaves more like supervised training. The algorithm for Deep-Q learning with experience replay is shown in figure 2. An epsilon greedy policy was used to select a random



```
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, $M$ do
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t$ = 1,T do
     With probability $\varepsilon$ select a random action $a_t$
     otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
     Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
     Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
     Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
     Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
     Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
     Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
     network parameters $\theta$
     Every $C$ steps reset $\hat{Q} = Q$
  End For
End For
```

Figure 2: Deep-Q learning with Experience replay [4]

action during training. To design our DQN Agent, we used *Keras* to design

our Network Architecture. The model was a Sequential model with 8 inputs representing the state space, 2 fully connected hidden layers with 64 neurons each and Activation functions of type *relu*. The output layer had 4 neurons representing the number of possible actions with a linear activation function. We synchronized our target networks after 50000 updates and kept the window length at 1. The window length was 1 since we could get our observations in one time frame unlike Atari which needed about 4 time frames.
We trained our DQN Agent over 100,000 steps till it was able to land correctly more than 70% of the time. Finally, we saved our weights and tested our results on a sample of 10 episodes and analysed the results. The DQN Architecture is shown in figure 3.

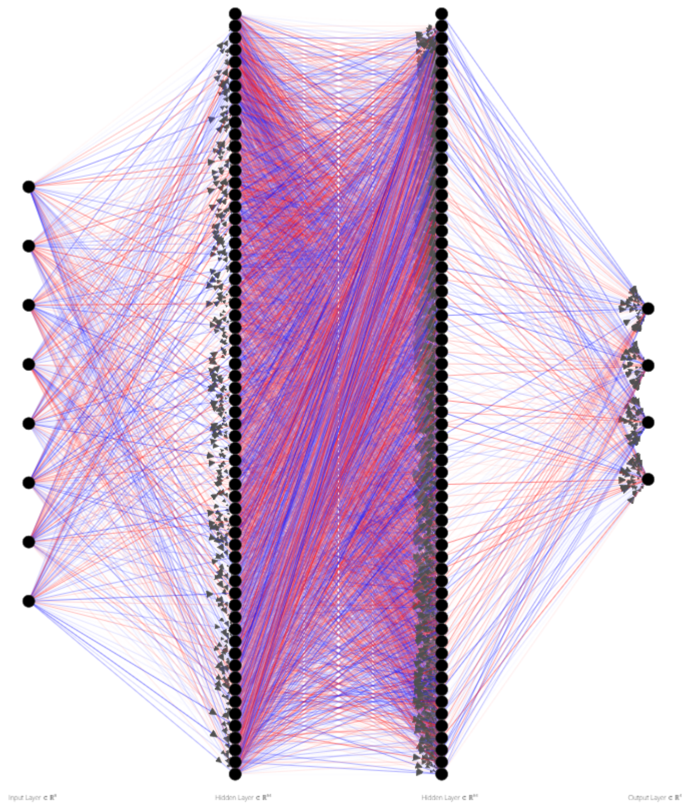


Figure 3: DQN Network Architecture in Keras [5]

Network Parameters

The network parameters consists of- *i) Neural Network Architecture* and *ii) Learning Rate*
We tweaked the above parameters to obtain the most suitable set of parameters for this environment.
i) Neural Network **-** Initially we choose a network architecture with the following configuration

- 8 input neurons
- 256 hidden neurons
- 256 hidden neurons
- 4 output neurons

All the layers were fully connected. This network had 269,316 trainable parameters. As a result the training was slow, since the number of parameters to be trained was very high. The train interval was set to the default value of 1 and target model update hyperparameter set to 4.
We observed the following result in figure 4 after training the model on this network configuration for DQN Agent- The agent took 831.22 seconds to train

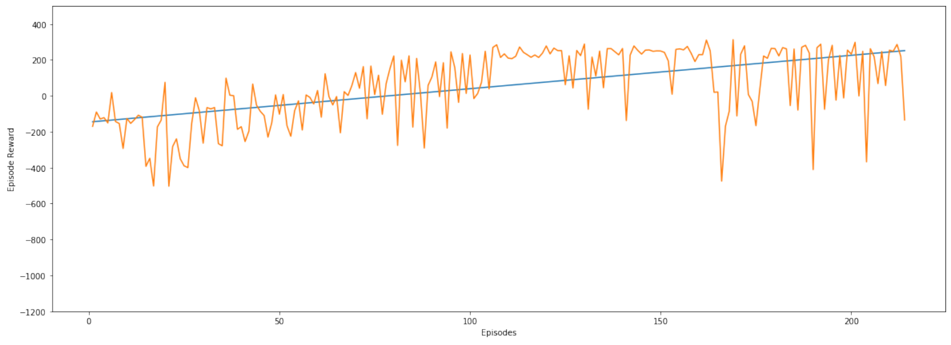


Figure 4: Episode Reward vs Episode for first configuration

with this configuration of the network. We then reduced the neurons in both the hidden layers to 64,this new network had 4,996 trainable parameters. This is about 50 times less trainable parameters,hence we observed that the network was training faster and more efficiently. With the following updated parameters-

- 8 input neurons
- 64 hidden neurons (Relu activation)
- 64 hidden neurons (Relu activation)
- 4 output neurons (Linear activation)

We observed the results in figure 5 after training the agent on the above configuration-The agent took 600 seconds to train with the second configuration, which is

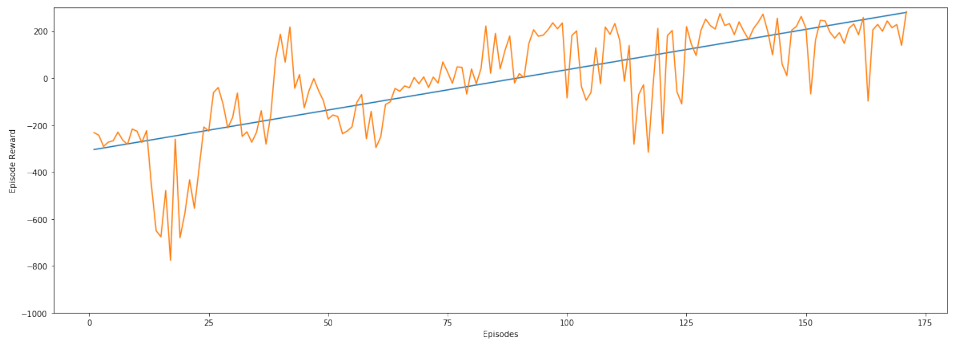


Figure 5: Episode Reward vs Episode for updated configuration

27.8% faster than the previous configuration. Hence we decided to select this as the final configuration for the neural network.

***ii) Learning Rate* -** We initially started with a learning rate of 10^{-5}. However, we observed that this learning rate was resulting in slower learning. As you can see from figure 6 that when the model was trained on this learning rate, even after training for 1750 episodes the policy had not converged and the reward for the episodes was still very low. We increased the learning rate to 10^{-3} in order

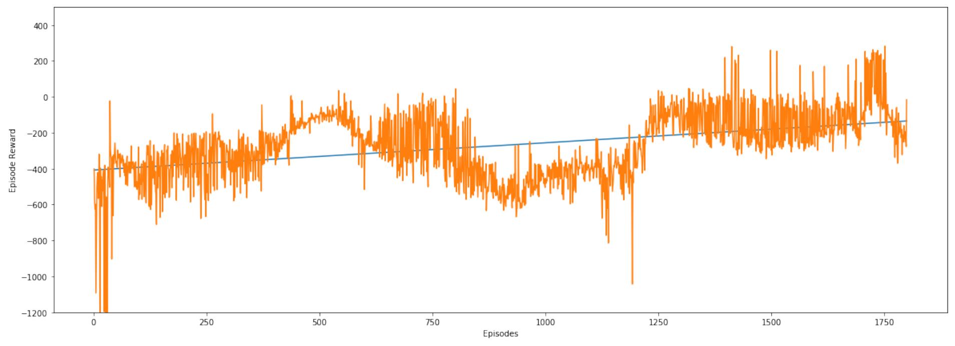


Figure 6: Episode Reward vs Episode with learning rate 10^{-5}

to overcome this. As seen in figure 5, we notice that this resulted in faster and better learning of the policy as the episode rewards crossed 200 after training for about 175 episodes. Hence we decided 10^{-3} as our final learning rate.

Tools Used

Different software tools were used to solve the Lunar Lander Environment.

- We used ***Keras***(a Python Deep Learning Library) to train our DQN and SARSA Agents.
- We used ***wandb*** to observe the learning process of the model on graphs. It logs metrics from our script so we can visualize the results in real-time.

Results

After training, we tested the train agents on the environment for 10 episodes each. The actions of the Lunar Lander in order to land on the landing pad tell us about the policy it has learned. We observed the following for the two agents-

i) SARSA Agent

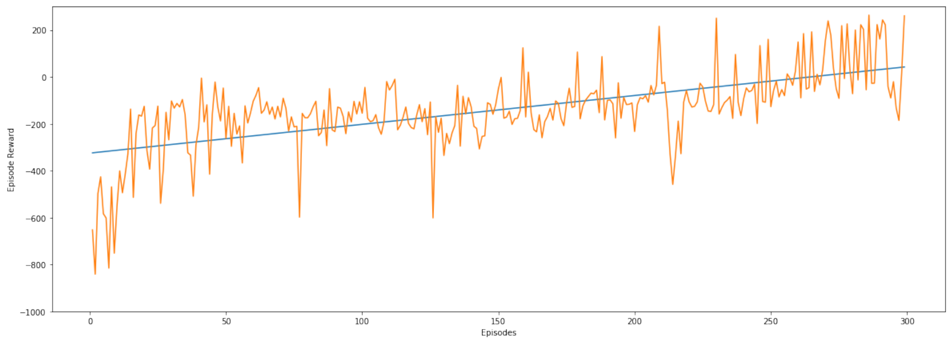


Figure 7: Episode Reward vs Episode

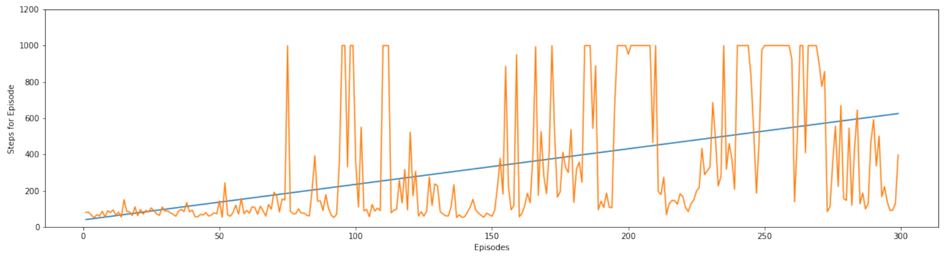
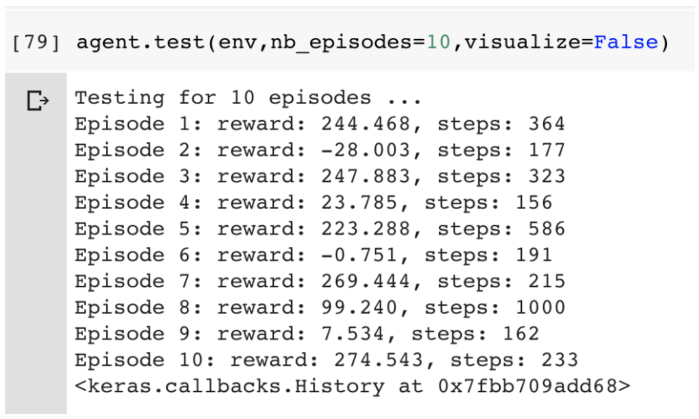


Figure 8: Number of steps/episode vs Number of training episodes

Policy learned by SARSA Agent- It is evident from figures 7 and 8 that the SARSA agent takes more steps to land. On observing the test simulations



```
[79] agent.test(env,nb_episodes=10,visualize=False)

Testing for 10 episodes ...
Episode 1: reward: 244.468, steps: 364
Episode 2: reward: -28.003, steps: 177
Episode 3: reward: 247.883, steps: 323
Episode 4: reward: 23.785, steps: 156
Episode 5: reward: 223.288, steps: 586
Episode 6: reward: -0.751, steps: 191
Episode 7: reward: 269.444, steps: 215
Episode 8: reward: 99.240, steps: 1000
Episode 9: reward: 7.534, steps: 162
Episode 10: reward: 274.543, steps: 233
<keras.callbacks.History at 0x7fbb709add68>
```

Figure 9: Testing the SARSA Agent on 10 episodes

in figure 9, we found that the lander had a very slow descent rate towards the ground. It seemed like the SARSA agent was only able to learn how to fly and descend slowly, i.e. how to avoid crashing in 100 episodes. It had not learnt how to position itself above the landing pad while minimizing the descent time. In case if the agent starts far away from the landing pad in some horizontal offset position, the agent prioritises stabilizing itself in the air and not crashing while slowly aligning itself with the landing pad, which would result in landing without crashing but away from the landing pad. So, even though the SARSA agent didn't crash, it had not developed the optimal policy to land.

ii) DQN Agent

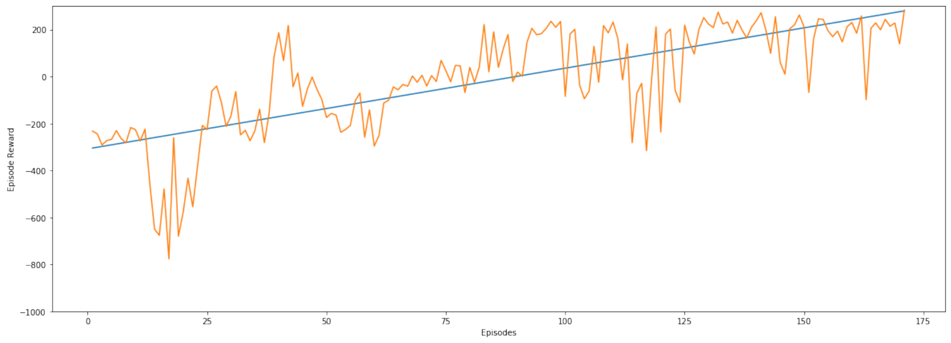


Figure 10: Average reward/episode against number of training episodes

Policy learned by DQN Agent- From the results in figures 10 and 11, we see that the DQN required lesser number of steps and also had more average reward per episode than the SARSA agent. On observing the test simulation in figure 12, we observed that,the DQN agent had learned the optimal strategy to control its position and also the descent rate towards the landing site. If the DQN Agent starts far away horizontally from the landing site, it still fires the appropriate engines to control the descent rate and also align itself with the landing site as much as possible. Hence we concluded that the DQN agent had developed the optimal policy to land on the site without crashing.

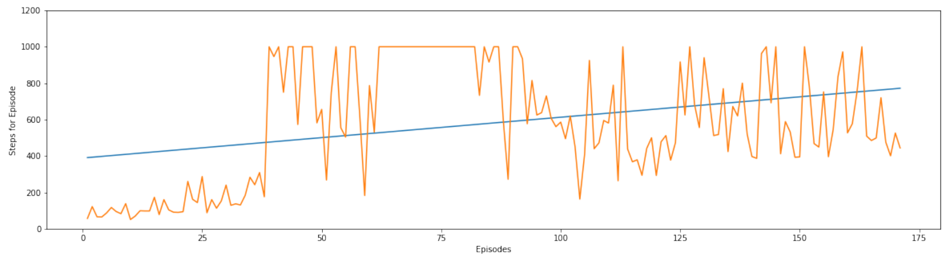
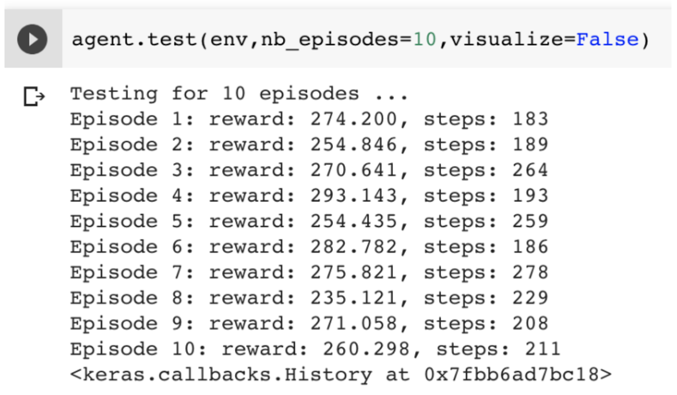


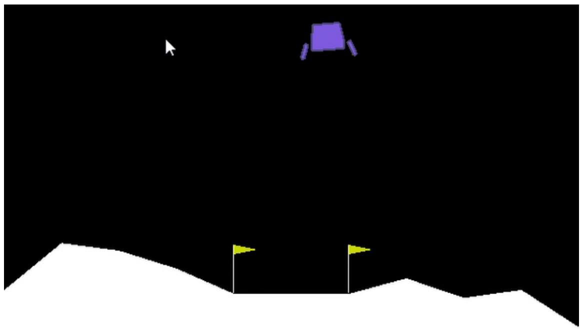
Figure 11: number of steps/episode vs number of training episodes



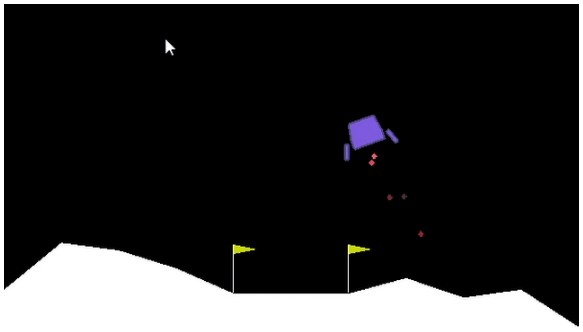
```
agent.test(env,nb_episodes=10,visualize=False)

Testing for 10 episodes ...
Episode 1: reward: 274.200, steps: 183
Episode 2: reward: 254.846, steps: 189
Episode 3: reward: 270.641, steps: 264
Episode 4: reward: 293.143, steps: 193
Episode 5: reward: 254.435, steps: 259
Episode 6: reward: 282.782, steps: 186
Episode 7: reward: 275.821, steps: 278
Episode 8: reward: 235.121, steps: 229
Episode 9: reward: 271.058, steps: 208
Episode 10: reward: 260.298, steps: 211
<keras.callbacks.History at 0x7fbb6ad7bc18>
```

Figure 12: Testing DQN Agent on 10 episodes.

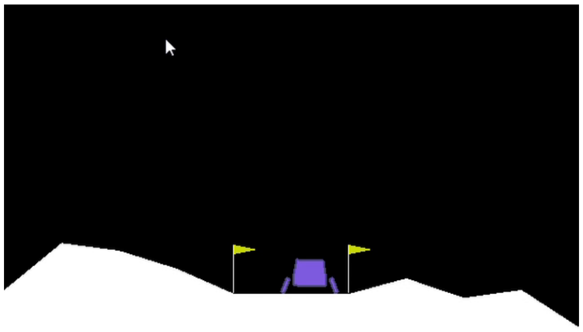
Comparison of SARSA and DQN Agent- The percentage increase in average reward from SARSA to DQN was 61% and the percentage decrease in steps was 241%. This proves that the DQN Agent is a better Agent to train the Lunar Lander space-ship. We were able to capture three frames from the Lunar Lander DQN Agent trying to land and they are shown in figures 13(a), 13(b) and 13(c).



(a) Frame 1



(b) Frame 2



(c) Frame 3

Figure 13: DQN Agent Landing using the learned policy

Conclusion

Using regular Q-learning algorithms is sufficient for training an RL Agent when the state space and/or action space isn't infinite or very large. But with the advent of Deep-Learning methods, such as Deep-Q Networks, Double Deep-Q Networks and others, agents can be trained even in very large state and action spaces in a reasonable time with very optimal results. This has greatly improved the application of Reinforcement learning in solving many problems previously seen as impossible to solve by an Artificial Agent.
Using a DQN Agent to solve the Lunar Lander led to a huge improvement in the average reward obtained by the space-ship after training on an average of 240 episodes. Using Deep learning for our project was a huge learning experience and it also reinforced some of the ideas we had learned from class.

Future Scope

Future enhancements include testing our DQN Agent on the Lunar Lander continuous environment which has both a continuous state and action space. This version will require a more advanced model which will mimic real-life scenarios where there are infinitely many states and actions.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller. *Playing Atari with Deep Reinforcement Learning.* arXiv:1312.5602 [cs.LG].

[2] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction, 2018.*

[3] RL — DQN Deep Q-network - Jonathan Hui - Medium." 16 Jul. 2018. *Accessed 9 Dec. 2019.*

[4] Playing Atari with Deep Reinforcement Learning - University ...." *Accessed 9 Dec. 2019.*

[5] NN SVG - Alex Lenail." https://alexlenail.me/NN-SVG/. *Accessed 9 Dec. 2019.*