GIT HASH for code: `b47fa9fba3fc712f0ef9b61a7bd1c7ef886b26bf`

Solving Lunar Lander problem using Deep Q-Learning with Experience Replay

Tharun Saranga
College of Computing
Georgia Institute of Technology
Atlanta, USA
tarun.saranga116@gmail.com

***Abstract*— In this paper the Lunar Lander problem from the OpenAI gym is discussed with the procedure to solve it using the Deep Q-Learning model as described in the paper [1], with some modifications according to the LunarLander problem. The model uses convolutional neural networks for value approximation and estimation. The model discussed in the paper is able to solve the LunarLander within 600 episodes.**

***Keywords*— Deep Q-Learning, IID, experience replay**

I. INTRODUCTION

A traditional Q-Learning method for reinforcement learning would require a set of state-action pairs through which the future estimation of the reward is calculated and the reward value is assigned to the corresponding state-action pair which would increase the overall reward. This is a simple task for a discrete environment. A Q-table with it rows as states and columns as actions and values as expected rewards is feasible. But, for a higher dimensional environment storing a Q-Table is not practical. If we discretize the environment into bins the advantage of convergence would disappear. Problems such as Lunar Lander have infinity dimensions and infinitesimal intermediate values. The agent in reinforcement learning learns gaining experience by exploration. The challenge is to simultaneously get a good estimation and a good performance of the model for the intermediate states. In Infinite dimensional environment the agent cannot possibly exhaustively explore the environment they are interacting with. The agent can only go to a finite amount of states. The other states' values have to be inferred from the already known states. This is called learning to Generalize. The remaining unknown state values are generalized using a positive feedback mechanism from previous experiences. This paper discusses one such method to solve complex high dimensional data using neural networks which are trained using Q-Learning algorithm.

II. PROBLEM DESCRIPTION

The Lunar Lander is an environment provided by the OpenAI gym, a virtual environment to test reinforcement learning algorithms. In Lunar Lander the agent has to learn to land a LunarLander in the predefined landing pad which is at coordinates (0,0). The environment consists of an 8-dimensional continuous state space and a discrete action space. The 8-dimentional state vector is (x, y, v_x, v_y, θ, v_θ, left-leg, right-leg). x and y are the coordinates of the lander's position. v_x, and v_y are the velocity components on the x and y axes. Theta is the angle of the lunar lander. v_θ is the angular velocity of the lander. The left-leg and right-leg are the binary values showing if the legs touch the ground. The agent can perform four actions: do nothing, fire the left orientation engine, fire the right orientation engine. The total reward for moving from the top of the screen to landing pad ranges from 100-140 points varying on the landing position of the lander on the pad. If the lander moves away from the pad it is penalized the amount of reward that would be gained by moving towards the pad. An episode is considered finish when the lander comes to rest or it crashes with reward of +100 on landing and -100 on crashing. Firing main engine incurs a -0.3 penalty for each occurrence. The problem is considered solved when achieving a score of 200 points or higher on average over 100 consecutive runs.
The environment has six dimensional continuous states with two discrete states.

III. METHODOLOGY

Inorder to solve the problem of LunarLander with continuous state space, neural networks are used for value approximation. Representing value function using a table is not feasible in this scenario. Neural Networks act as function approximators that can evaluate the relation between the known and unknown states automatically. A value function that the neural network uses to approximate has to solve the control problem. The action-value function solves this. The next action to take can be decided by knowing the values of the state action pairs which can gain information or maximize the reward. In this method we use an action-value function with a neural network. The architecture of the network has to be such that the time it takes to solve is small. A neural network with both states and actions as inputs would be complex with higher dimensional data. To fit this data would require several passes to map each action and each state to an output value. Hence, only the states are passed as input to the neural network which would calculate over all the possible action in that state. This would make the exploration strategy epsilon-greedy more efficient. The neural network is a 3 layer network with a hidden layer. First layer is the input layer which is the size of the observational space. The hidden layer is a 40 node network with relu (Rectified Linear Unit) activation. Output gives a single value. To optimize the loss, Huber loss is used with Adam Optimizer. Huber loss acts as both Mean Square Error and Mean Absolute Error. It acts as Mean Square Error in the region where the loss goes to zero and acts as Mean Absolute Error in the region after a specific threshold value of the loss. To optimize this loss function Adam optimizer is used which is a combination RMSprop and momentum optimizers. It works better than those optimizers individually. It moves in the direction of the velocity of the gradient, as in momentum and also scales update proportional to the moving average of the magnitude of the gradients, as in RMSprop. This combination makes Adam optimizer more efficient. Now, the actual Deep Q Learning algorithm. The concept is to make a value approximator which feedbacks with the positive experiences providing the agent to learning labelled data. This approach has mainly two problems. One, non-stationarity of the updates and two, the data is not Independent and Identically Distributed (IID). This can be resolved by freezing the targets for evaluating the next states and storing experiences and training with batches sampled. To achieve this a target network is used. A target network is a copy of the neural network frozen for a number of time steps. Then these weights of the frozen network are used as targets to train the actual neural network. In order to solve the IID problem a technique called Experience Replay is used. The experience collected by the agent is stored in form of tuple and stored in a data structure called reply buffer. The agent is trained on the mini batches sampled from this buffer. If the buffer is overflown the oldest experience is dequed. This would make the data conform to IID as the data is sampled from a uniformly sampled mini batches. The training phase uses epsilon-greedy strategy to explore and exploit the reward. The epsilon-greedy method has a variable epsilon whose value decreases with the number of episodes. At first high value of epsilon means the agent takes a random action inorder to explore the environment. As the epsilon value decreases the agent takes actions based on the experience it has gathered from the exploration.

The following is the algorithm used for the Deep Q-Learning Network.

```
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights
for episode = 1, M do
      Initialise sequence s1 = {x1} and preprocessed sequenced φ1 = φ(s1)
      for t = 1, T do
         With probability     select a random action at
          otherwise select at = max Q∗(φ(st), a; θ)
          Execute action at in emulator and observe reward rt and image xt+1
          Set st+1 = st, at, xt+1 and preprocess φt+1 = φ(st+1)
          Store transition (φt, at, rt, φt+1) in D
          Sample random minibatch of transitions (φj , aj , rj , φj+1) from D
          Set yj = rj for terminal φj+1 / rj + γ maxa0 Q(φj+1, a’; θ) for non-terminal φj+1
          Perform a gradient descent step on (yj − Q(φj , aj ; θ))2
      end for
end for
```

IV. Experimental Setup

To implement this algorithm the python programming environment is used for which the gym environment is available. To implement the neural network tensorflow library is used. The Experience Replay is implemented using a namedtuple data structure. A target network which freezes for 500 time steps is used.

Training

The training is nothing but making the agent explore the environment as much as possible and gain new information to maximize the cumulative reward. In the beginning the agent has to select an action randomly from the action space and explore the environment, storing these experiences in the memory bank. The neural network is provided with this data to update its weights. As more number of episodes undergo the value of epsilon decreases, which decreases the amount of random actions. As epsilon decreases the agent chooses action from the experiences it has gathered through the neural network to approximate its actions for the unencountered states. Agent tries random actions at start which gives a negative reward as the agent is still learning. As the learning progresses the agent learns to increase the total reward slowly using the gained experience. As the training continues the agent learns to maximize the total reward to a positive level and reaching 200+ point.

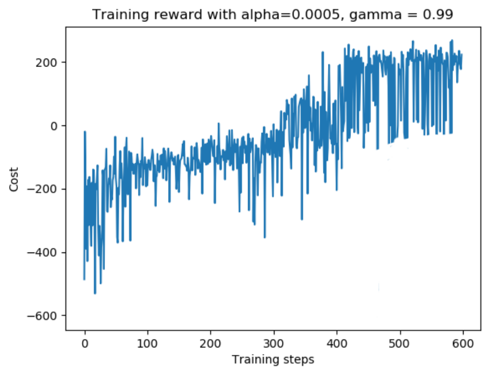

Figure 1

Graph showing the reward accumulated for each training episode for a total of 600 episodes with hyperparameters tuned to get an average reward of above 200.

1) *Observations during Training phase:*

The training has been started using the following hyper parameters. Learning Rate alpha = 0.0005, gamma = 0.99 and epsilon decay = 0.002/episode. The agent tries to hover the lander in the beginning as because of the high gamma value which makes the agent see the negative reward for crashing the lander. It doesn’t necessarily hover over the landing pad. Next the agent tries to learn moving towards the landing pad using the side thrusters, as it learns about the reward for moving towards the landing pad. Next it tries to land in the center without crashing. This learning behavior is mainly influenced by three parameters. The Learning rate alpha, gamma and epsilon. Tuning them to the optimal values maximizes the reward.

Computation for 600 episodes took an average of 9630 seconds on Intel i3 machine. Tuning parameters and waiting for the network to train to find the results is very time consuming process. After 600 episodes of training with epsilon-greedy exploration algorithm the agent is finally able to get an average reward of more than 200. In this phase the agent almost always lands on the landing pad with occasional bumps around for the unknown states. To further increase the accuracy of the lander the parameters have to we have to play with the huber_loss delta value which changes the loss function from quadratic to linear. The values of epsilon and learning rate also affect the final outcome of the agent. The noise in the agent is part due to encountering hugely varied unknown states and partly because of non-optimal hyperparameters which overshoot the optimal rewards. If there was enough computing power and more time in hand the parameter tuning would become easy and reaching optimal solution would be practical. The random seed value also has an effect in the training process.

V. Test Run

After training phase the agent is tested using the gain experience. In this test phase the agent chooses actions based on the states values it learnt during the training phase. If the agent encounters an unknown state it extrapolates the data from the similar states that it has seen during the training session. Different seed values produce a different test result. The following result is with a different seed value from the training phase. The average reward over 100 test runs is 204.239. A huge negative spike during the test run might be due to largely new state in the environment. On individual test runs the agent could achieve reward as high as 261.45.

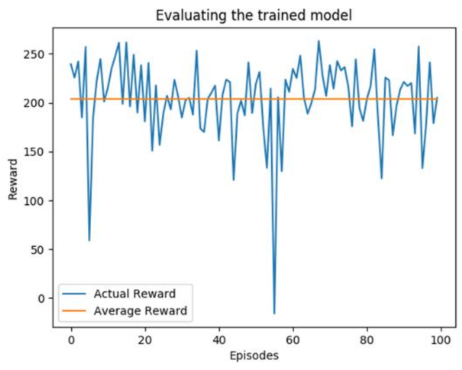

Figure 2
Graph showing the reward per run during the test phase acting on the gained experience. The orange line shows the average reward over the 100 test runs.

VI. Effect of Hyperparameters

A. Effect of Alpha (Learning Rate):

The value of alpha has significant effect on the training phase. The range of alpha values selected for the experiment are from (0.005, 0.0005, 0.00005). The values are 10 fold different from the current optimal values. A larger learning rate alpha of 0.005 learned the environment but overshot and was delearning the experience as shown in the Figure 3. Alpha with 0.0005 seems to work fine maximizing the reward above 200 as shown in Figure 4. Alpha with 0.00005 fails to learn the environment without reaching the average of 200 reward as shown in Figure 5. This hyperparameter requires very fine tuning to get an optimal response.

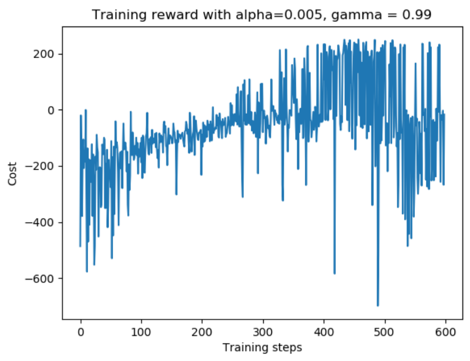

Figure 3
Graph showing the effect of a larger Learning rate alpha. The agent reaches the 200 reward states but overshot and was delearning with more training.

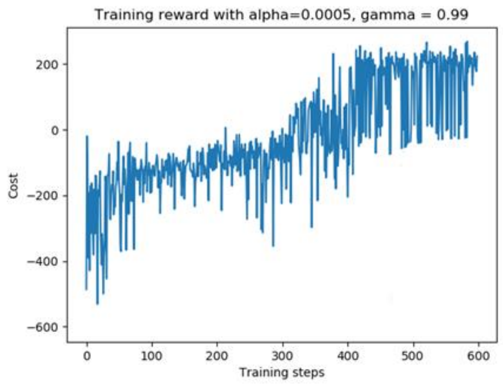

Figure 4
Graph showing the effect of near optimal learning rate. The agent could successfully reach the 200 reward states.

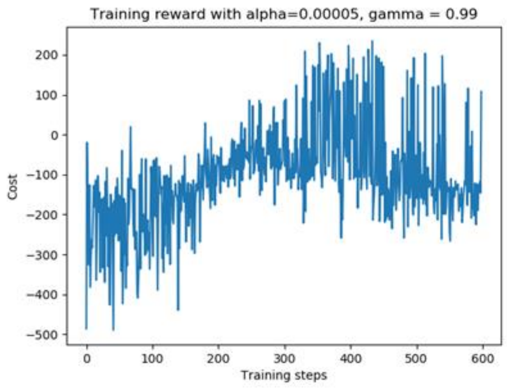

Figure 5
Graph showing effect of very low learning rate. The agent could not stay learning in the 200 reward states. The average reward fell below 0.

B. Effect of gamma:

Gamma value makes the agent see the discounted future reward from a particular state. A low value of gamma heavily discounts the reward and makes the agent move around without any significance to the task to be performed. A high gamma value will discount the reward vey less forcing the agent to move towards the goal state. In this experiment the effect of gamma value is observed by using 3 values (0.2, 0.5, 0.99). With less gamma value of 0.2 the agent fails to achieve the goal state as it cannot see the final goal reward which is large as shown in the Figure 6. A larger value of 0.5 gamma value results in partially observing the reward of the final state. The agent tries to move towards the goal state but couldn't achieve it continuously as shown in Figure 7. By using a value of 0.99 as shown in the Figure 8, the agent could finally see the large reward of reaching the final goal state of landing on the pad. As the reward is no longer discounted the large reward at the landing state is visible to the agent. This motivates the agent to move towards the goal state.

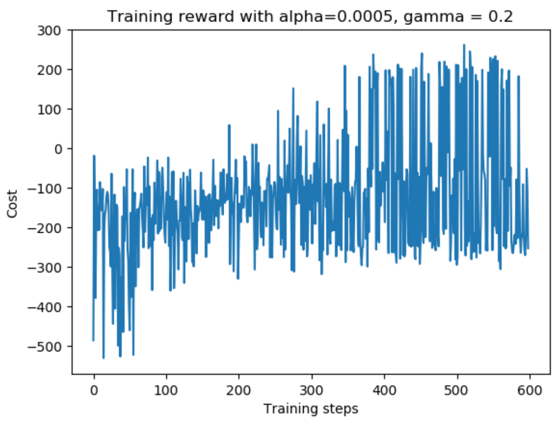

Figure 6

Graph showing the effect of gamma on the training phase. The agent could not learn to sustain the states to 200 reward

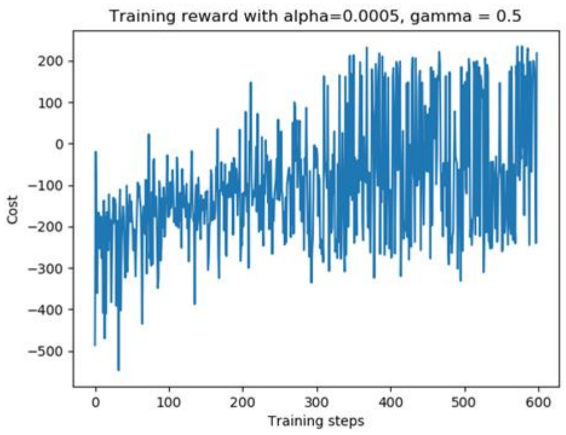

Figure 7

Graph showing the effect of gamma on the training phase. The agent could partially achieve the 200 reward states.

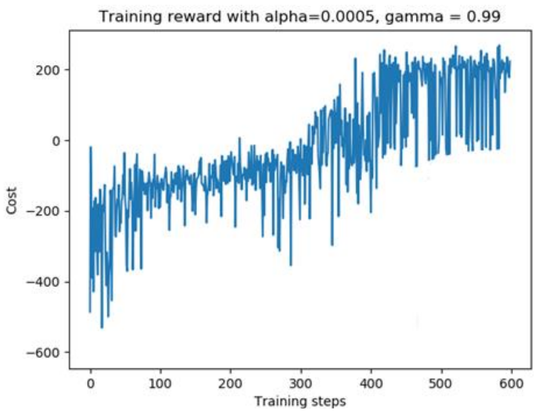

Figure 8

Graph showing the effect of gamma on training phase. With high enough gamma the agent could achieve the 200 reward states.

C. *Effect of Epsilon decay:*

Epsilon is the decay factor for the epsilon-greedy algorithm which dictates the amount of exploration vs exploitation during the training stage. A small epsilon decay value would make the agent stay in exploration stage more. A large epsilon decay value would make the agent move to exploitation state without exploring and without gaining enough information about the environment. Hence, an optimal epsilon decay value would provide just enough information about the environment and then the agent exploits this information to gain access to new states from the previous experience. This experiment is to observe the effect of epsilon decay value on the training phase of the agent. (0.0002, 0.0015 and 0.00225) are chosen for the hyperparameter values. These values were chosen during tuning phase. Since, the environment state space too large for the lunar lander, thinking that a smaller decay value would provide more information about the environment and then exploit this information to gain reward. This proved to be wrong. With small epsilon decay value of 0.0002 the agent never reaches the state of exploitation as shown in Figure 9. It just randomly moves around the environment exploring and never actually gaining significant reward. The reward doesn't reach even a positive value. To mitigate this the epsilon decay value is increased to 0.0015 which is calculated such that the epsilon value reaches 0.05 from 0.95 in 600. Now the agent could reach 200 reward mark but the average reward was still low as shown in Figure 10. The agent was didn't have enough time to act upon the learned experience. So, the epsilon decay value was further increased to 0.00225 such that the whole exploration stage would complete in 400 episodes and the agent uses this in the next 200 episodes to exploit the reward as shown in the Figure 11. This made the agent reach its optimal reward range within 600 training episodes.

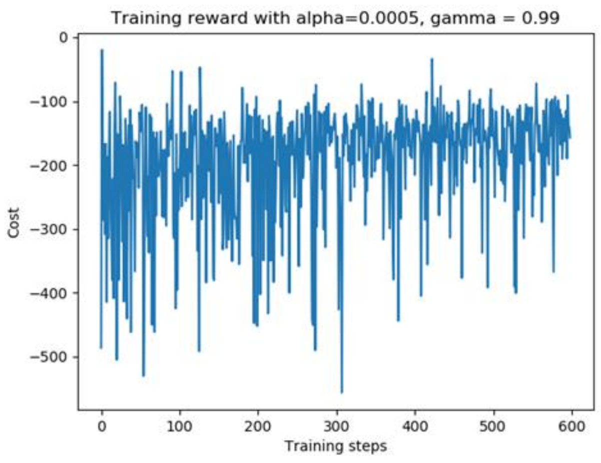

Figure 9

Graph showing the effect of epsilon decay value. With epsilon decay value of 0.0002. The agent continued with random moves exploring the environment with ever exploiting the experience. It could never reach a positive reward.

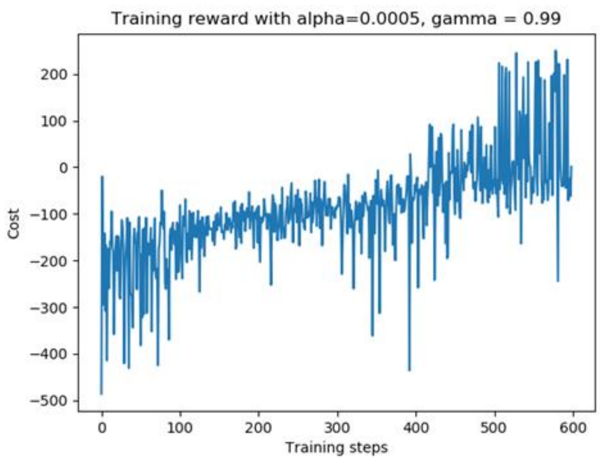

Figure 10
Graph showing the effect of epsilon decay value on training phase, with an epsilon value of 0.0015. The agent could partially exploit the earned experience before the termination of the episodes.

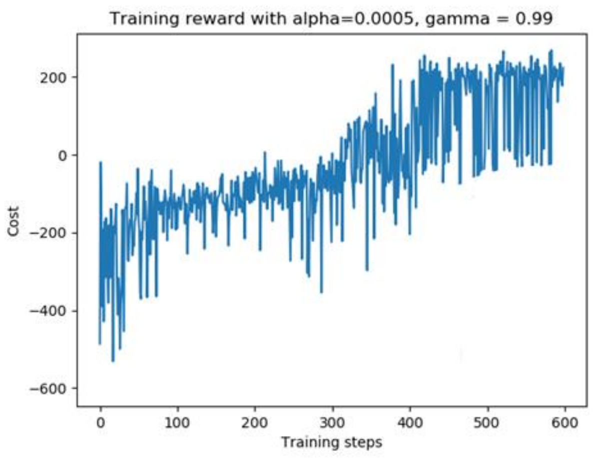

Figure 11
Graph showing the effect of epsilon decay value on the training phase. With epsilon decay value of 0.00225, the agent could achieve required states with reward of 200.

VII. Conclusion

The Deep Q-Learning method which uses neural networks as function approximator was able to solve the lunar lander problem with an average reward of 204.239. The problem can be solved to get more reward by more fine tuning the hyperparameters which is time consuming and computationally expensive on small machines. If had more time and computational power I would try different loss functions, and even try tuning the huber loss delta parameter.