CS 7642 - Reinforcement Learning
Deep Q-Learning on Lunar Lander

Prasanna Venkatesan Srinivasan
MS in Computer Science at Georgia Tech
psrinivasan48@gatech.edu
Git Hash: 5d1718553b2768203cee44a48162a6e5bcdd3ca3

1 Introduction

This report explores the implementation details and the results obtained from the application of Deep Q-Learning on Open AI gym "LunarLander-v2" experiment. Various model and hyper-parameters have been tuned and tested and the impact of those parameters on the performance of the agent has been elaborated.

2 Background

2.1 Q-Learning

Q function (Q(s,a)) computes the long term expected reward for an agent being in a particular state and executing a particular action. The action that maximizes the Q value has to be selected by the agent at every time-step. The main advantage of Q learning is that it enables itself to be applied to problems that are dynamic, ie., it doesn't require the agent to be fully aware of the environment. It lets an agent explore and update its belief about the environment from time to time.
Q-Learning algorithm is an off-policy Temporal Difference Control algorithm which is defined by the following equation:

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

Where Q(s, a) is the $quality$ of executing an action a from state s by the agent. As the agent takes the long term expected reward into account, there is a discount factor γ which can be fine tuned according to the expected behavior of the agent. Smaller values of γ lead to short sighted policies where the agent only worries about the expected rewards in the near future, as against larger values that lead to the agent being too proactive. Sutton[1] demonstrates that the Q function updated according to (1) ultimately converges to Q_*, the optimal action-value function.

2.2 Function Approximation

Q-Learning algorithm, described in the previous section talks about the application of fixed point iteration to learn the optimal Q function (Q*). This requires the use a compact representation to store the successive Q values in memory. This is quite difficult to achieve in problems with large and continuous state spaces. Different approaches to discretize the state spaces can be employed (like binning), but are quite complex and hard to implement. Alternatively, function approximation methods come handy, where the requirement is just to learn a parametrized function with a d-dimensional weight vector that would serve as a representation of Q(s,a).
Supervised machine learning algorithms learn to construct a model of d-parameters having a d-dimensional weight vector. These algorithms, over multiple iterations learn the weights so as to mimic the mapping from input to output over the dataset it has been trained on. The weights are updated over multiple iterations, one for every data point in the dataset, repeated over multiple time-steps, where each step of the iteration would move the weights learned towards the target weights. Similarly, function approximation methods start by assuming an arbitrary weight vector and aim to move the weights to an appropriate target over multiple episodes, based on the rewards observed by the agent at every step. Ultimately, the learning of weights converges such that the learned Q function is an approximation to Q*.

3 The Problem

This section discusses the Lunar Lander Problem and the DQN implementation of the same.

3.1 The Lunar Lander Problem

Lunar Lander is a problem in Open AI Gym where the objective is to land the spaceship safely on the area between a pair of flags placed on the ground. The following is the description of the LunarLander-v2 problem:

- S: The state space - The state space is continuous comprising of the following set of 7 parameters:
  - Horizontal position of the agent x
  - Vertical position of the agent y
  - Horizontal velocity of the agent v_x
  - Vertical velocity of the agent v_y
  - Angle swept by the agent with the horizontal (θ)
  - Angular velocity v_θ
  - Whether the left leg of the lander has touched the ground
  - Whether the right leg of the lander has touched the ground
- A: 4 actions can be taken
  - Do nothing (and let gravity do its job)
  - Fire left orientation engine
  - Fire main engine
  - Fire right orientation engine
- R: The reward function: Landing on the ground between the flags gives out a reward of 100 to 140, which gets deducted as the agent drifts away from the goal area. Agent touching the ground with a leg is rewarded with +10. An episodes either times out after 1000 steps or gets terminated as the agent either crashes or lands on the ground, procuring a reward of -100 or +100.
- The game is assumed to be ended if the average reward obtained by agent over a hundred continuous episodes has reached a value of 200.

3.2 Deep Q Learning

3.2.1 Artificial Neural Network

Deep Q Learning is a non-linear function approximation method. It involves the use of artificial neural network (ANN) to construct a model to map the input to the output. Figure 1 shows an example of ANN from [1] consisting of 4 input units to feed in 4 parameters, 2 output units and two hidden layers.

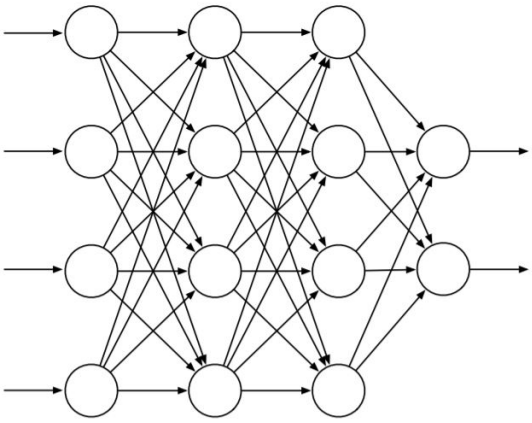

Figure 1: ANN with 4 input nodes, 2 output nodes and 2 hidden layers

The training on ANN is done by multiple passes of forward propagation and back propagation. The forward propagation in hidden layers involves application of a linear function, precisely a dot product of weight vector, with the parameters from obtained from the previous layers, followed by the application of a non-linear activation function. The difference between the expected and the actual output is computed and the error is propagated back to the previous hidden layers, where each hidden layer uses the error information to correct its weights. A optimization function governs the procedure with which the weights are updated during every iteration.

3.2.2 Replay Memory Buffer

Reinforcement Learning through non-linear function approximators can lead to model divergence. To ensure model stability, the following approaches have been implemented in addition to DQN algorithm.

- Training using only the latest observations can lead unstable results due the presence of correlation among a continuous sequence of observations. One approach to solve this problem is to use a fixed size buffer called replay buffer and training the model only on a randomly sampled subset of replay buffer. Every observation made in this way does not directly make it to the model at the same iteration, but goes into the replay buffer and may ultimately get used to learn the Q function.

- We maintain two neural networks:
  - One neural network learns the action-value function
  - Another neural network is used to make prediction.

  The training model weights are updated on to the prediction model once in every C iterations. This is to ensure that the model doesn't overfit to chunks of the mini-batches being provided.

3.2.3 Exploration-Exploitation Dilemma

One of the important problems to be considered in Q-Learning in general is the resolution of exploration-exploitation dilemma. The agent has to explore during the initial phases to learn the environment and has to exploit its belief once it is done learning. At any given point of time the hyperparameter ϵ dictates the probability with which an agent should explore the environment. As the time progresses, the model should explore less and exploit more. To achieve this, three hyperparameters are used.

- ϵ_{max} - The maximum ϵ value that the agent should start with.
- ϵ_{decay} - The factor by which ϵ should be reduced during every episode
- ϵ_{min} - The minimum value of ϵ that can be allowed post which, not further decay is done.

3.2.4 Other factors

The following are the other factors that are to considered and fine-tuned in a deep Q learning algorithm:

Mini-batch size: The size of the sample that was drawn at every time-step from the replay memory buffer that is fed into the ANN to train. Very small mini-batch size leads to the overfitting of the model towards the mini-batch, leading to multiple episodes dealing with unlearning and relearning of weights and larger batch size causes under-fitting, where the weight updates are insignificant.

Replay Memory Buffer Size: Smaller replay memory buffers place too much of emphasis on the recent observations and larger replay memory buffers make the agent take into account the observations that are too far behind on the timeline.

Learning rate α: The learning rate of the optimization function employed in the ANN.

The discount factor γ: Same as the γ employed in the Q learning algorithm (1). Lesser values of γ lead to short-sightedness of the agent and larger γ values (≈ 1) leads to long-sighted policies.

Number of nodes in the hidden layer of ANN

Algorithm 1: DQN algorithm with experience replay

Input: Environment, Model and hyperparameters
Result: Model for Q(s,a)

Initialize the ANN model and replay memory buffer;$\epsilon \leftarrow \epsilon_{max}$;
repeat
  repeat
    Choose one randomly a from the action space with probability ϵ and choose a such that $\max_a Q(s,a)$ with probability $1-\epsilon$;
    Execute $s, a \rightarrow_r s'$ and add s, a, r, s' to replay buffer;
    if *Replay Buffer Overflow* **then**
      Remove the oldest entry;
    end
    Sample a mini-batch of tuples(s, a, r, s') from the replay memory and compute

$$q = \begin{cases} r + \gamma max_a \hat{Q}(s', a) & \text{if s' is non-terminal} \\ r & \text{if s' is terminal} \end{cases}$$

    Train the ANN to learn the Q(s,a) on the mini-batch.
  until *End of the episode*;
  if $\epsilon > \epsilon_{min}$ **then**
    $\epsilon = \epsilon * \epsilon_{decay}$
  end
  if *episode % C == 0* **then**
    Copy the weights from Q to
  end
until *Convergence*;

4 Experiments and Results

4.1 Parameter values used

The following parameters have been used for the results discussed in this section.
Learning rate $\alpha = 0.001$
$\epsilon_{max} = 1$
$\epsilon_{decay} = 0.995$

$\epsilon_{min} = 0.01$
Discount factor $\gamma = 0.9$
Replay Memory Buffer size = 10,000
Mini-batch size = 64
Optimizer - Adam's optimization function
The following is the reward vs episodes plot for the above set of parameters: It can be seen that the

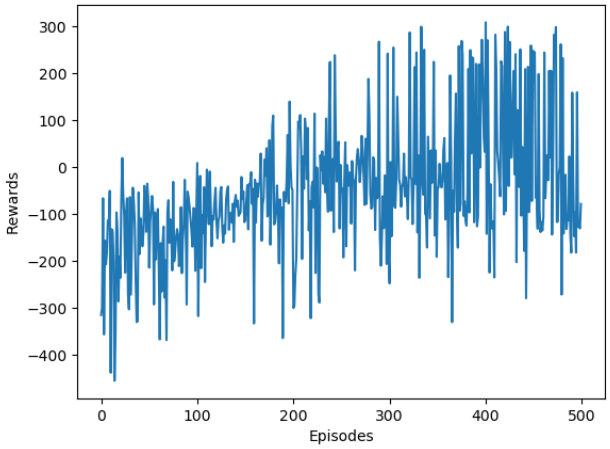

Figure 2: Episodes vs Rewards for different learning rates

rewards increase as time progresses. The training procedure was timed-out before the convergence was reached. Following figure presents the rewards obtained from running the converged model for 100 episodes. From the above plot it is evident

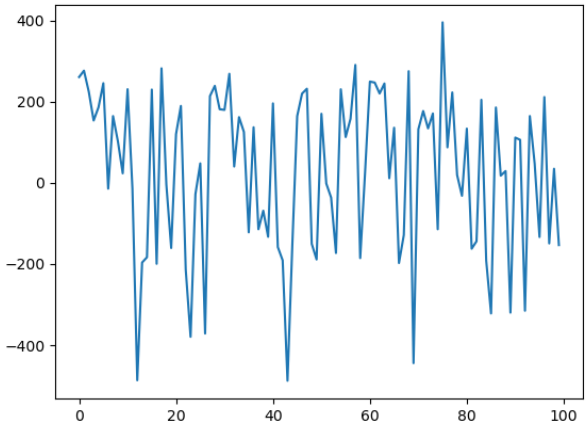

Figure 3: Episodes vs Rewards for different learning rates

that the model has not coverged to its optimum, but reward values in the range of 100-200 are observed in several episodes.

4.2 Experiments on parameters

All the following experiments have been run on 500 episodes.

Learning rate α: It can be seen that the higher

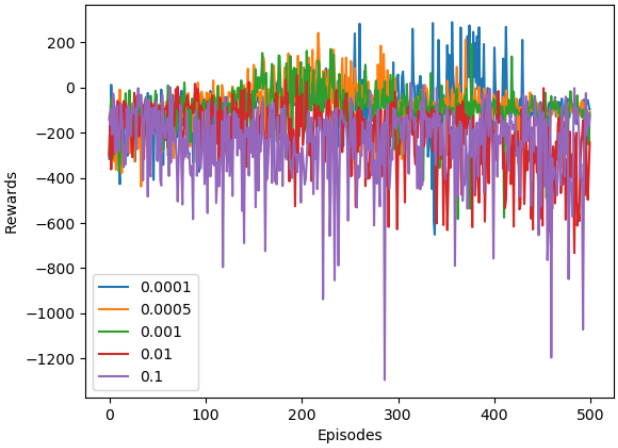

Figure 4: Episodes vs Rewards for different learning rates

learning rates produce the worst performance. Higher learning rates cause larger updates to the weights, causing the agent to unlearn and re-learn the weight vector for every mini-batch. This is the reason behind huge spiking in rewards vs episodes plot of Figure 2 for $\alpha = 0.1$. It can also be seen that the reward for $\alpha = 0.0005$ peaks earlier than $\alpha = 0.0001$. This is due to the fact that larger learning rates lead to faster convergence. $\alpha = 0.0001$ has be chosen for training the Lunar Lander agent as it positive reward spikes are consistently seen.

Discount Factor γ: For this experiment, ϵ_{decay} was set to 0.995. It can be seen from from figure 3 that the γ values 0, 0.2, 0.4 do not perform well at all. Mean rewards for 100 iterations increase up to a particular point, after which the average rewards stop dropping. The dip can be correlated to the decrease in the ϵ value. The discount factor of 0.6, increases upto a point and saturates at a mean rewards of around -75. $\gamma = 0.8$ performs well, where steady raise in the mean reward value is seen. This is due to the fact that the resulting policy is proactive and emphasis is more on the importance of expected rewards to be received on a long run. It can also be seen that the $\epsilon = 1$ causes the model to be way too proactive about the future. This leads to steeper increase in the mean rewards received, up to a point, post which there is a steep

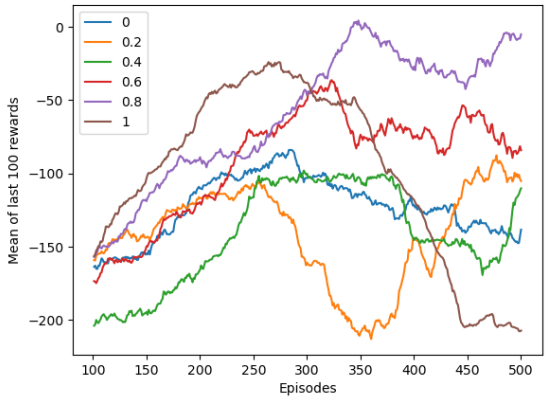

Figure 5: Episodes vs Mean Rewards over 100 episodes for different discount factors

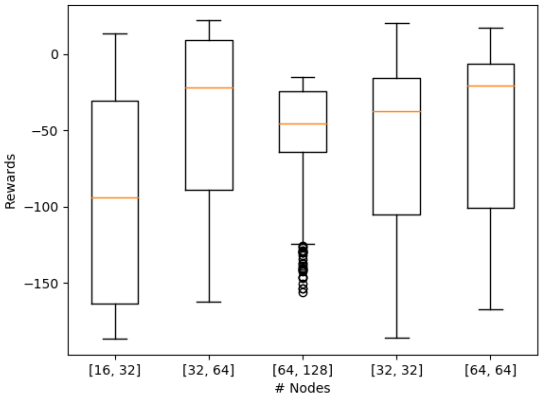

Figure 7: Box Plot - Nodes vs Mean Rewards over 100 episodes

decrease in the reward an agent receives.

ϵ **- Decay rate** ϵ_{decay} plays a vital role in the context of an agent resolving the exploration and exploitation dilemma. ϵ is set to an initial value of 1 and its value is decayed in successive episodes until, ϵ_{min} is reached. It can be seen in Figure 4 that $\epsilon_{decay} = 1$ causes a flat curve which means that it does not let the agent to exploit from its learning from its exploration. Smaller values of ϵ_{decay} rapidly decay the ϵ value, causing it to quickly wrap up the exploration process and start exploiting more. It can be seen that $\epsilon_{decay} = 0.995$ leads to a steady increase the mean reward over 100 iterations and hence can be assumed to aptly balance the exploration vs exploitation dilemma of the agent.

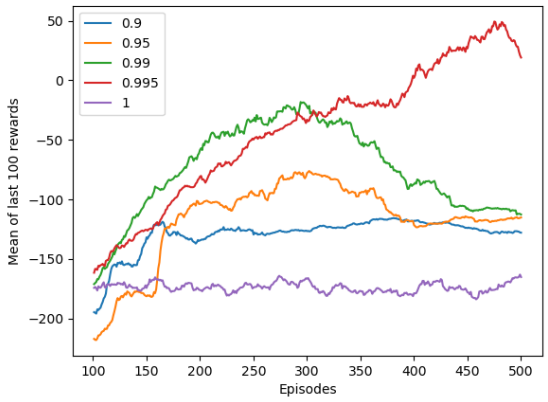

Figure 6: Episodes vs Mean Rewards over 100 episodes for different ϵ_{decay} values

Number of nodes in hidden layer: For the purposes of this experiment, a neural network with 2 hidden layers was constructed. The number of nodes in each layer of the neural network was fine-tuned and the rewards received were measured. The (64, 128) network provides the least variance in the expected rewards, buts it's mean is much lesser most of the other neural network constructed, making it an unattractive choice. The range of rewards obtained from the (16, 32) is much lesser as compared to other networks.

5 Conclusion

This report explored the implementation of DQN on OpenAI gym's Lunar Lander problem, discussed the possible range of some of the model parameters and hyperparameters and the impact of these parameters on the expected reward. To further extend the presented model, approaches like Double-DQN and DRQN can also be applied to this problem. Double-DQN can be used to solve the problem of possible maximization bias that can result out of a DQN model. DRQN will be able to make optimum use of the correlated nature of temporally successive observations.

6 References

[1] Richard S. Sutton and Andrew G. Barto. Introduction to Reinforcement Learning (2st. ed.). MIT Press, Cambridge, MA, USA. [2] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." arXiv preprint arXiv:1312.5602 (2013). [3] Mnih, V., Kavukcuoglu, K., Silver, D. et al. Human-level control through deep reinforcement learning. Nature 518, 529–533 (2015). https://doi.org/10.1038/nature14236