Solving Lunar Lander Problem Using DQN

Sudip Chowdhury
sudip4@live.com

***Abstract*—In recent years, deep reinforcement learning has seen major breakthroughs in solving the real world challenges in various domains. Deep Q-Network (DQN), implemented by Minh [1], is one of those early breakthroughs that achieved state-of-the-art results in different Atari Games without any handcrafted features or change in learning algorithm. This article demonstrates the effectiveness of DQN in solving OpenAI gym's Lunar Lander problem. Similar to Minh, I also used ANNs to approximate the state-action value function. Moreover, I used hyperparameter tuning to identify the best set of parameters that solves the problem with least number of episodes. During this experimentation, I found that the exploration strategy, architecture of the ANN and discount factor has the most impact on the agent's training process.**

I. Introduction

Controlling an agent's behaviour in a simulated environment to maximize it's long term expected reward has been greatly beneficial for the RL community in benchmarking performances of different algorithms. OpenAI's gym [2] is one such platform consisting many different environments. This project tries to solve OpenAI gym's LunarLander-v2 environment using Deep Q-Network (DQN). DQN is a variant of Q-learning algorithm, an off-policy Temporal Difference (TD) method. The main reason behind choosing Q-learning rather than an on-policy method like SARSA is the conservative approach of SARSA during training. In other words, if there is an optimal path with more potential of receiving a large negative reward along the way, SARSA tends to ignore it, whilst Q-learning tends to follow that path. This issue is mostly highlighted in a simulation where we do not care much about the rewards earned in the initial episodes of the training; but we mostly care about the rewards earned by a trained agent and the number of episodes it takes for the agent to solve the task. Adapting from the DQN paper, I have used Neural Networks as a state-action value function approximator which enabled the agent to solve the problem in 588 episodes and the trained agent obtained 221 rewards on average in 100 episodes.

II. Off-Policy TD Method

A. TD Learning

According to Sutton and Barto [3], if one has to identify the core idea that differentiates reinforcement learning from rest of machine learning, it has to be temporal difference (TD) learning. Instead of trying to calculate the complete reward of an episode, TD methods simply predict the combination of immediate reward and the value of the next timestep. In the next timestep, when new information is obtained, the new prediction is compared with what was predicted earlier. Based on this difference, the old prediction is adjusted towards the new prediction. By obtaining more and more actual reward from the environment and bringing the old prediction towards the new prediction, TD methods achieve convergence.

B. Q-Learning

Q-learning is an off-policy TD control algorithm where the target policy is different from the behaviour policy. In other words, it updates its state-action value $Q(S_t, A_t)$ of current state S_t and an action A_t (from behaviour policy) using the state-action value $Q(S_{t+1}, A_{t+1})$ of the next state S_{t+1} and an action A_{t+1} chosen, not by current policy, but using greedy policy.

$$\mathrm{Q}(\mathrm{S}_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \quad (1)$$

III. Function Approximation

With the increase of state space as in the case of LunarLander-v2, it becomes difficult for the TD methods to visit every state action pair of the environment and estimate the values accurately. As a result we cannot expect the agent to find an optimal policy, even in the limit of infinite time. To solve this issue, we can use a function approximator, parameterized with weight vector $w \in \mathbb{R}^d$, which takes the current state as an input and provides the estimated value as output. In this case, we can denote the state value as $\hat{v}(s, w) \approx v_\pi(s)$.

A. Loss Function

Although this function approximator has made the above problem tractable, it introduces some new issues. As we are using same weight vector to estimate the value of all the states, changes in one value of the vector will change the estimates of all the states. As a result, we have to decide which states we care about most by specifying the state distribution $\mu(s) \geq 0, \sum_s \mu(s) = 1$. If we know the true value of each state $v_\pi(s)$, then our loss function becomes,

$$\overline{VE}(w) = \sum_{s \in S} \mu(s)[v_\pi(s) - \hat{v}(s, w)]^2 \quad (2)$$

B. Optimization

We need to update the weight vector w, so that it can generalize all the states of the environment. Stochastic Gradient Descent (SGD) methods do this very efficiently in all machine learning by minimizing the loss function (2) and adjusting the weight vector to the true values by a small margin (α) after each example.

$$w_{t+1} = w_t - \frac{1}{2}\alpha \nabla [v_\pi(S_t) - \hat{v}(S_t, w_t)]^2 \quad (3)$$

$$= w_t + \alpha[v_\pi(S_t) - \hat{v}(S_t, w_t)] \nabla \hat{v}(S_t, w_t) \quad (4)$$

Here, $\nabla\hat{v}(S_t, w_t)$ denotes the partial derivative of value function with respect to w.

Now that we have established the loss function (2) and optimization technique (4) for function approximation, we need to fix one issue with true value of each state $v_\pi(s)$ that we assumed earlier. In RL, it is generally an approximation ($U_t \in \mathbb{R}$) of $v_\pi(s)$.

$$w_{t+1} = w_t + \alpha[U_t - \hat{v}(S_t, w_t)]\nabla\hat{v}(S_t, w_t) \tag{5}$$

If we use Monte Carlo method where the loss is calculated after each episode, U_t becomes an unbiased estimate of $v_\pi(s)$ and w_t converges to a local optima using (4). But in case of Bootstrapping targets like n-step return which depends on the current value of w_t, U_t becomes an biased estimate and w_t does not converge as robustly as gradient based methods. One way to look at this is that it is possible to derive (4) from (3) if the target is independent of w_t. These methods are called semi-gradient methods. This enables the target for one step TD methods be, $U_t = R_{t+1} + \gamma\hat{v}(S_{t+1}, w)$.

C. Non-Linear Function Approximation

Although linear function approximation has some nice property, like data efficiency, fast to compute, and convergence guarantee, require handcrafted features for every task. Another limitation is that it does not provide interaction between features. In some cases, increase in one feature value can affect other features; this is not captured in linear forms. As a result, non-linear function approximators like Artificial Neural Networks (ANNs) have become very popular. This sudden surge in popularity can be attributed to it's universal function approximator property. In theory, a feed-forward network with one hidden can approximate any function given enough data and computation power. Another benefit of ANNs is the parallel computation capability on multi-core processors. As each neurons for one layer is connected with all the neurons of next layer, ANNs allow interaction between features as well. Apart from these benefits, ANNs also introduce some challenges like vanishing or exploding gradient during training using backpropagation, overfitting to the training data etc. Though, these issues can be alleviated through various techniques like regularization, batch normalization, skip connection, weight sharing etc. But the most important issue of non-linear function approximator (ANNs) arises when combined with bootstrapping method (Q-learning) in a off-policy training. This combination is called the deadly triad. It makes the semi-gradient methods unstable and divergent.

IV. Deep Q-Network & Lunar Lander

In the previous sections II and III, we have discussed all the building blocks of Deep Q-Network (DQN). In this section, we will delve deeper into the DQN algorithm and discuss how this algorithm is adapted to solve the LunarLander-v2 problem.

DQN, developed by Minh at DeepMind[1], is a major breakthrough in reinforcement learning research. This paper identified the issues that causes non-linear function approximators to be unstable in a RL setting. These are correlation between the sequence of observations, small change in Q may significantly change the policy and data distribution, and the correlation between state-action values and target values. To solve these problems, they added two key improvements in the existing Q-learning algorithm. The first one is experience replay. It stores the agent's experience (s, a, r, s') in the memory (D). At each time step t a mini batch of these experiences are drawn uniformly ($(s, a, r, s') \sim U(D)$) from the memory and Q-learning updates are performed. This technique has several benefits. First of all, because of storing experiences, DQN is able to learn more efficiently, thereby increasing sample efficiency. Second, as the mini batches are drawn randomly, correlation between sequence of observations are automatically removed and the instability in data distribution also gets improved. The other improvement is that they periodically update the target network using behavior network's parameters, thereby reducing the correlation between target and the state-action values. The loss function at iteration i for DQN is as follows:

$$\mathrm{L}_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}[(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i))^2] \tag{6}$$

where γ is a discount factor, θ_i are the parameters of Network at iteration i, and θ_i^- represents the target network's parameters. θ_i^- are only updated with θ_i after every C steps and are fixed in between.

Minh used this DQN network to train an agent on 49 different Atari 2600 video games without changing the network architecture and hyperparameters. With random initialization of Network weights, the agent demonstrated state-of-the-art results in 43 of those games and identified different policies for each game. The only input to the agent was the the raw pixels of the screen, the game specific score, and the number of actions available. They used deep CNNs to automatically extract features from the visual images. During training, they used RMSProp optimizer with mini-batches of size 32. The behaviour policy was ε greedy, with ε being annealed from 1 to 0.1 in the first one million frames, and fixed at 0.1 thereafter.

They used another trick that helped in improving the stability of the algorithm. They clipped the error $(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i))$, so that it resides in [-1, 1]. They also clipped the positive rewards to 1 and negative rewards to -1, leaving 0 rewards unchanged. This helped them to use same learning rate in all the games.

A. Environment

The main objective of LunarLander-v2 of OpenAI gym is to settle the lander safely on the landing pad at coordinates (0, 0). The shape of the surface is not constant, varies from episode to episode. At each time step, the environment outputs a 8-dimensional vector $(x, y, \dot{x}, \dot{y}, \theta, \dot{\theta}, leg_L, leg_R)$ that represents the state of the lander. Here, x and y denotes the coordinates (x, y) of the lander; $\dot{x}$ and $\dot{y}$ denotes the velocity in x and y axis; θ is the angle with respect to vertical axis; $\dot{\theta}$ is the angular velocity; and leg_L, leg_R shows whether left or right leg of the lander has touched the ground respectively. First 6 of these values are continuous and the last 2 values are of

[1] www.deepmind.com

Algorithm 1: deep Q-learning with experience replay

```
Init memory D to capacity N;
Init Q(s, a) with random weights θ;
Init target action-value function Q̂ with weights θ⁻ = θ;
for episode = 1, 2000 do
    Reset state;
    for t = 1, 1000 do
        Select random action with probability ϵ a_t
         Otherwise select a_t = argmax_a Q(φ(s_t), a; θ);
         Execute action a_t in environment, observe r_t
         and next state s_{t+1};
        Store experience (s_t, a_t, r_t, s_{t+1}, done) in D
        Sample random minibatch of experiences from
         D
        for (s_t, a_t, r_t, s_{t+1}, done) in minibatch do
            if done then
                y_j = r_t
            else
                y_j = r_t + γ * max_a' Q̂(s_{j+1}, a'; θ)
            end
        end
        Perform gradient descent step;
        if done then
            break;
        else
    end
    if episode % C == 0 then
        Reset Q̂ = Q;
    else
end
```

type binary (0,1). Based on the state, Lander can choose to perform 4 actions (do nothing, fire left orientation engine, fire main engine, fire right orientation engine).

The reward system is as follows: Reward for moving from the starting point to the pad with zero velocity is from 100 to 140. If the lander moves away from the landing pad, it will incur the same amount of negative reward that would be gained by moving toward the pad. This will discourage the agent to roam around. If the lander crashes to the ground, it will receive -100 points, and +100 if successfully lands; and the episode will finish. For each leg contact with the ground, it will receive +10. For each firing of main engine and orientation engines, it will incur -0.3 and -0.03 points. Though fuel is infinite, this will encourage the agent to use as little fuel as possible and find the shortest path toward the landing pad, and the large rewards at the end will make sure the agent will learn to touch down the pad safely.

B. Solving Lunar Lander

In the DQN paper, the authors used deep CNNs to extract features from the raw pixels of the screen. But in the case of LunarLander, gym already provides a well structured feature representation of the lander's state, so I did not need to use CNN algorithms. Instead, I have used a shallow feedforward network with one hidden layer to estimate the state-action values. This helped me to train the network much faster and need not worry about vanishing or exploding gradients. When an agent performs an action in gym's environment, the emulator returns the next state, reward and whether the episode has ended or not. When storing the experiences of the agent in memory for experience replay, I have also stored whether the next state is the last state or not (s, a, r, s', e). Here, e is the boolean denoting the end of episode. It helped me in performing Q-learning updates on the mini-batch. Please refer to algorithm 1 for more details.

On the other hand, they clipped the errors to [-1, 1]. The main reason behind this clipping was to make sure that values between -1 and 1 would use squared error loss function, and outside -1 and 1 absolute error is used. Squared error loss is not good for large values or outliers, as it would exaggerate the values by squaring them. Similarly, absolute error is bad for small values, the error would be too small. So, instead of clipping the errors I used Huber Loss[2] which combines best of both worlds. I also didn't need to clip the rewards, as I was training the agent for only one task.

Another important change from the DQN paper is the use of Adaptive Moment Estimation (Adam) instead of RMSProp. RMSProp is good at identifying the importance of each features and based on that it performs smaller or larger updates to the weights. But it does take into account of the slope to speedup the learning. Adam does both of those. It is also less fiddly than RMSProp, and it's default parameters in PyTorch works pretty well.

V. Experiments

For computational restrictions, I have set a limit of 2000 episodes for training the agent. The problem is solved if the agent receives 200 or more reward points on average in 100 episodes. Agent following parameters from Table I took around 45 minutes to train in Google Colab's[3] CPU and the trained agent received around 221 points on average in 100 episodes with standard deviation of 94.6; as shown in figure 2. Figure 1 shows the rewards accumulated during the agent's training, and also the rate of epsilon decay. Although the agent solved the problem during training in 588 episodes, it kept on training until the max number of episodes are reached. Initially, I trained the agent until it accumulated average 200 or more rewards in 100 successive episodes with max episodes of 2000; but the trained agent was not able to generalize well and received only 160 rewards in the testing scenario. This led me to believe that as the state space is huge in Lunar Lander problem, agent was not able to learn the approximate true values of many states that are necessary for successful landing. Moreover, as the shape of the landing ground changes in each episode, it became difficult for the agent to generalize to new states. Even after training for 2000 episodes, we can see in figure 2 that there are few episodes with negative rewards;

[2]https://en.wikipedia.org/wiki/Huber_loss

[3]https://colab.research.google.com/

TABLE I
TRAINING PARAMETERS

γ	0.99
Batch Size	128
ANN Hidden Size	32
Epsilon Maximum Value	1
Epsilon Decay Rate	0.995
Epsilon Minimum Value	0.005
Target ANN Update Frequency	4

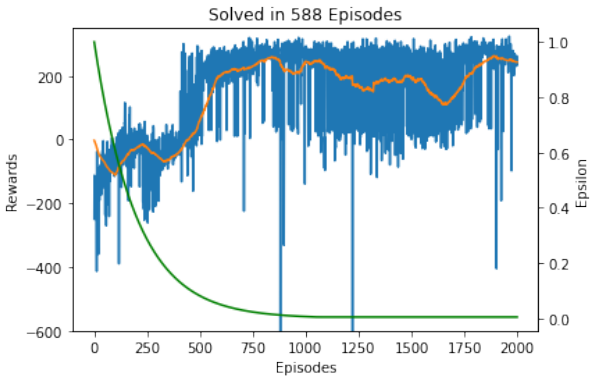

Fig. 1. Rewards and Epsilon value during training an agent based on the parameters of Table I. Agent solved the problem in 588 episodes.Total number of episodes 2000.

that means this problem still persists to some extent. I have also tested the trained model's performance using 10 different random seeds and achieved 232 mean rewards with standard deviation of 6.7.

To understand more about the model performance, I plotted the Q-values at each time step in figure 3. While looking both at the rendered gif file and Q-values, it became clear that when the lander entered the screen fully it became aware of it's surrounding and as a result we can see a spike in q-values in frames 40 to 100. We can see two more such spikes around frames 250 and 300. In the first case lander came horizontally near the landing area and in the last case, lander was about to land on the surface.

A. Hyperparameter Tuning: Epsilon Decay

During hyperparameter tuning, epsilon decay proved to be the most important parameter, mostly because it directly affects the agent's policy. From figure 1, it is clear that during the initial stages of training when epsilon value was higher, agent was taking actions more randomly and was accumulating experiences. But, after 400 episodes when epsilon was 0.1 or lower agent started using the previous experiences extensively and gained more and more positive rewards which helped the agent to solve the problem in 588 episodes. I have used a threshold 0.005 for minimum epsilon value.

I used 6 different epsilon decays as shown in figure 4. First of all, I trained multiple agents with different epsilon decays, then during testing to produce consistent results I used 10 different random seeds. Mean value (Blue Line) of those 10

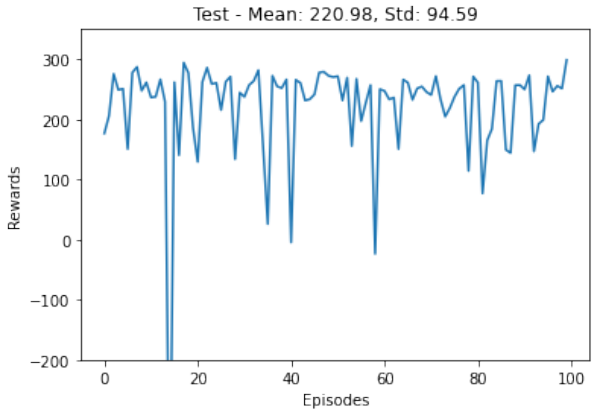

Fig. 2. Rewards achieved by the agent, trained on Table I parameters during testing.

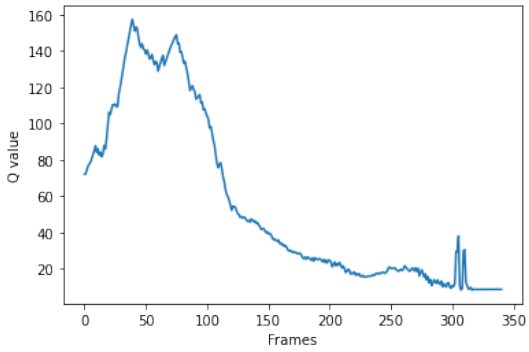

Fig. 3. Q-values predicted by the model, trained on Table I parameters, during testing for an episode.

rewards are reported in the figure. It is clearly visible, higher value of epsilon decay helps the agent to explore more in the earlier stages of training. As a result agent learns more about the environment and later acts accordingly. Epsilon decay of 0.9 worked best and it took least number of episodes (442) to solve the task. But epsilon decay 0.99 worked worst, which is odd. In theory, it should perform similar to it's nearest values. So, I think, it behaved such a way only because of the random initialization during training. If I had used some other random seed, it would have been different.

B. Hyperparameter Tuning: Hidden Size

Figure 5 shows the moving averages of last 100 episodes of accumulated rewards during training. Though, it would have been better to experiment with few more hidden layers and compare the results; because of limited computation power and time available I have used only one hidden layer in the ANN. As shown in the figure, I have experimented with 5 different hidden layer sizes. Apart from the extreme sizes 16 and 256, all other have performed similarly in the training. Hidden sizes 16, 32, 64, 128, 256 achieved test mean of -166, 231, 101, 224, -698 respectively. It clearly shows, hidden size 16 was too small to encode all information needed to generalize, and

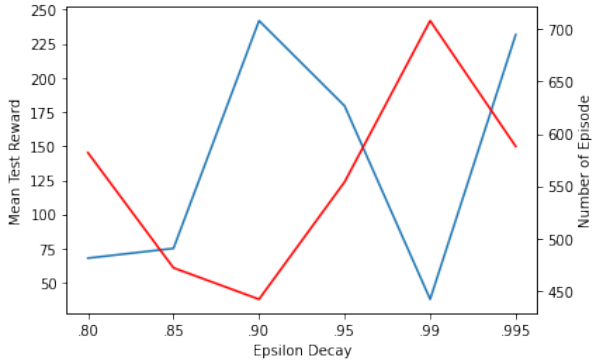

Fig. 4. Mean rewards (Blue Line) for each agent during testing. Each agent trained with different epsilon decay are tested using 10 different random seeds. Mean value of those 10 rewards are reported here. Alternate y-axis (Red line) shows the number of episodes it took to solve the task.

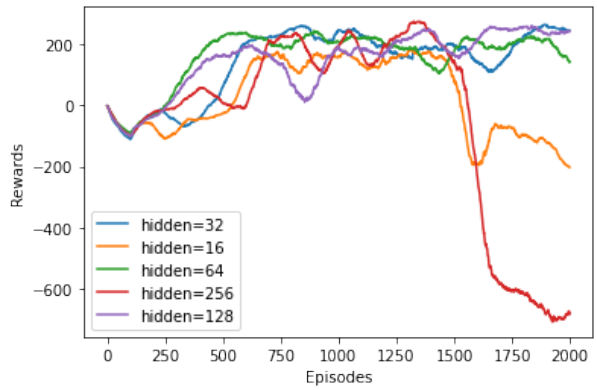

Fig. 5. Moving average using last 100 episodes of accumulated rewards during training using various hidden sizes.

hidden size 256 was too big and became unstable and diverged from the true values.

C. Hyperparameter Tuning: Discount Factor

Figure 6 depicts the performances of agents using different discount factor γ. Discount factor is the most crucial parameter in temporal difference methods. A discount factor of 1 means, the agent values all the future rewards with the same importance. As a result, because of uncertain future rewards, system does not converge and becomes unstable, as is in the case of figure 6. During training, agent with $\gamma = 1$ initially started performing better than others. I think it is because initially the agent had limited experience and as $\gamma = 1$ made the agent more cognizant about it's surroundings, it started doing better. But with the addition of new experiences, future rewards became more and more uncertain, and as the agent is continued to use the same discount factor of 1, the results became more and more unstable. Discount factor of 0.99 performed best as it was both giving importance to the delayed rewards and not waiting for the end of the episode. Other discount factors did not converge at all as they did not give much importance to the delayed reward.

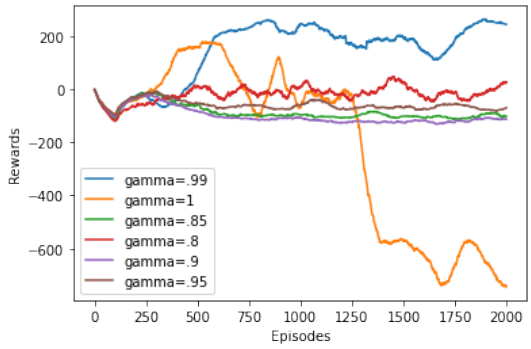

Fig. 6. Moving average using last 100 episodes of accumulated rewards during training using various gamma.

VI. Pitfalls

During the training of the agent, I was a little confused about what should be the stopping criteria of the training. We are told the task is solved if the agent receives average 200 or more rewards in successive 100 episodes. With the hyperparameter tuning it is possible to solve the problem within 350 episodes. But in this way the agent will not understand the approximate true values of many states. As the state space is huge, the agent is bound to encounter few state it has not seen before, so if the agent does not have a good function approximator, the agent will not perform good. To fix this issue, I had to train the agent for 2000 episodes.

VII. Future Work

There are few parameters that I was not able to tune and analyze because of time and compute limitation. For example, frequency of updating the target network using the behaviour network, minimum epsilon value, batch size etc. I would have also liked to experiment with epsilon decay. For example rather than multiplying directly with epsilon decay, I would have liked to use a function like sinusoidal etc. In DQN paper, the author describe the limitation of the memory buffer. New experiences always overwrites the old ones. A better strategy would have been to write in the memory based on the importance of the experience. Last, but not the least, I would like to run the experiments multiple times with different seeds. I tried to do that at the test time, but it would have been better, if I got the chance of doing that at the training time.

References

[1] Mnih, V. et al. "Human-level control through deep reinforcement learning." Nature 518 (2015): 529-533.
[2] Brockman, G. et al. "OpenAI Gym." ArXiv abs/1606.01540 (2016): n. pag.
[3] Sutton and Barto, Reinforcement Learning: An Introduction, 2nd edition http://incompleteideas.net/sutton/book/code/code2nd.html