DQN for Lunar Lander*

*Hash: 61abd4af00a5c569145b95632b7523b06687d2b4

1st Sizhang Zhao (szhao334)
Computer Science Department
Georgia Tech

***Abstract*—Ever since DQN (Deep Q-Learning Network) came into the world, it has become the to-go methodology when it comes to approximation of off-policy Q value functions. In this paper, we applied DQN to an interesting problem, Lunar Lander, from gym and discussed different behavior of the agents under different settings of hyperparameters.**

***Index Terms*—DQN, deep learning, reinforcement learning, lunar lander, gym**

I. Introduction

Q-Learning is a successful off-policy TD control for MDP problems. Traditional Q-Learning will maintain a Q value table and update the corresponding entry whenever the agent took an action and received a reward. However, it will become much less maintainable when the number of states and actions become really large. However, Q value table is nothing but a function which maps a state-value pair to a real value number. With the development of deep learning in recent years, the approximation of any kind of function becomes possible if you have enough training data and strong enough neural network structure. Thus the combination of deep learning and q learning, aka DQN (Deep Q-Learning Network) becomes a active field for research of solving MDP problems. In this paper, we will implement DQN from scratch and apply it to an interesting reinforcement learning problem, Lunar Lander, and research its convergence behaviral with different choices of hyperparameters.

II. Theory

A. Q-Learning

Q-leaning is one of the most famous algorithms when it comes to reinformcement learning fields. It is a off policy methods, which means when it updates its Q value, it use the best Q value of the next states rather than use the Q value from the states which comes from the real action it took. Mathematically, the update of Q value for Q-leaning is,

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

On the contrary to on-policy methods like Sarsa, Q-learning dramatically simplfies the analysis of the algorithm and enabled early convergence proofs. As long as all pairs of states and actions continue to be updated, combined with a variant of the usual stochastic approximation conditions on the sequence of step-size parameters, Q has been shown to converge with probability 1 to $q*$.

Tradeoff between exploration and exploitation is always an interesting problem in reinforcement learning. Usually ϵ-greedy method will be used. So the agent will always maintain a probability of ϵ to take a random action, which leads to a random exploration to other states which might not be the optimal based on the current Q value estimator, this in turn will give agents opportunities to explore different state-value pairs and help the convergence of Q value estimation. Please note, usually a decaying ϵ will be used if the final plan is to gain as much rewards as possible.

B. Neural Network

Neural network has been brought up for many years but it becomes popular in recent years with the improvements of computing power. Neural network is nothing but a composite functions. Traditional artificail neural network consists of different linearl layers with non-linear functions between them. Take Relu activation functions as one example $\sigma(u) = max(u, 0)$, the neural network with L hidden layer can be expressed as:

$$f(x) = W_{L+1}\sigma(W_L\sigma(W_{L-1}...\sigma(W_1x+v_1)...v_{L-1})+v_L) \quad (2)$$

Due to the inclusion of the non-linear component in the network, neural network has the capacity to fit any potential functions form. However, also due to the multiple non-linear function between layers, it's no longer a convex optimziations problem for fitting a neural network. Different techniques should be incorporated to try to get to a not bad local minimum.

C. Stochastic Gradient Descent

For a machine learning problem, if the dataset is small, we usually use gradient descent methods to fit the model. Periodically calculate the gradient of the functions after

the paremeter is updated towards the negative direction of the function to be optimized. However, when the dataset becomes large, it will become very resource consuming to compute the real gradient. But the approximation of the gradient based on small subset of the training data or even the single training data becomes promising. Even the approximation can be very rough, after enough iteration, the descent methods can still help the function to come to a local or global minimum dependent on the model itself.

Usually, for a neural network model, the training data can be extremely large, usually the gradient won't be calculated for the entire datasets. Also to conquer the problem for stochastic gradient that the estimation can be very noisy, mini batch gradient descent will usually be used. To calculate the approximation of gradient, a sub-sample of the training data will be used, this is called a mini batch, and the method is called mini batch gradient descent. How large of the mini batch is usually an important hyperparameter to tune for neural network performance.

D. DQN

When the state and action space is not large, tabular Q value representation for Q-learning is good enough, however, when the space becomes really large, it's no longer feasiable to maintain the Q value table. As we mentioned earlier, Neural Network is a wonderful method for function approximation. If we applied neural network to Q functions: $Q_\theta : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ approximation, we come to DQN.

Additionally, to improve the performance of DQN, two different tricks are needed, experience replay and target network.

1) Experience Replay: As we mentioned earlier, neural network usually use batch gradient descent methods for model training. However, generally speaking, data points from reinforcement learning are all sequences which are serial correlated. To get a better estimation of the gradient direction, we'd better use "independent" data points for training, that's where experience replay come into being. So at each time t, we store the transion (S_t, A_t, R_t, S_{t+1}) into the replay memory. And then sample a minibatch from the memory and then feed it into neural network with stochastic gradient descent methods during training.

2) Target Network: For a supervised learning problem, we usually have a loss functions with respect to true value and predicted value and purpose of training is to minimize the loss between them. However, under reinforcement learning, we didn't have a true value for Q value, on the contrary, it's what we want to estimate. So how should we formulate the training problem for DQN? Here is when target network come into play. Define target network as Q_{θ^*} with parameter θ^* and define $(s_i, a_i, r_i, s_i')_{i\in[n]}$ are the independent samples chosen from replay memory. The training target is calculated as,

$$Y_i = r_i + \gamma \max_{a\in\mathcal{A}} Q_{\theta^*}(s_i', a) \tag{3}$$

and the loss function will be,

$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n}[Y_i - Q_\theta(s_i, a_i)]^2 \tag{4}$$

parameter for target network is only updated after a given T_{target} steps.

To better understand why target network helps, we firstly note $\mathbb{E}(Y_i|S_i, A_i) = (TQ_\theta)(S_i, A_i)$, where T is the same as we defined in the lectures, the bellman operator. Then let's do variance-bias decomposition on expectation of $L(\theta)$.

$$\mathbb{E}(L(\theta)) = ||Q_\theta - TQ_\theta||_\theta^2 + \mathbb{E}\{[Y - (TQ_\theta)(S, A)]^2\} \tag{5}$$

the first part is called mean-squared Bellman error (MSBE) and the second term is the variance of Y variable. It's easy to notice $L(\theta)$ is the empirical version of MSBE. From equation 5 we can easily tell not only MSBE depends on Q_θ, but also the variance term depends on Q_θ. Minimizing empirical MSBE will be different from minimizing the expectation of $L(\theta)$.

However, if we replace the calculation of conditional expectation of Y from target network, we will have $\mathbb{E}(Y_i|S_i, A_i) = (TQ_{\theta^*})(S_i, A_i)$. Equation 5 will becomes:

$$\mathbb{E}(L(\theta)) = ||Q_\theta - TQ_{\theta^*}||_\theta^2 + \mathbb{E}\{[Y - (TQ_{\theta^*})(S, A)]^2\} \tag{6}$$

And now variance of Y no longer depends on Q_θ, minimizing expectation of $L(\theta)$ will be the same as minimizing:

$$\min_{\theta\in\Theta} ||Q_\theta - TQ_{\theta^*}||_\theta^2 \tag{7}$$

So what DQN effectively does, is it aims to solve the minimization problem 7 with θ^* fixed by minimizing θ and update θ^* after given T_{target} steps.

There are several ways for updating target network, you can directly copy the parameter of policy network to target network after T_{step}, there is also a notion called soft update, meaning you can take weighted sum between privious parameter with the parameter from policy network after given T_{step} (it can be one, meaning you are continuously updating target network). The update weight are usually represented as τ, which is the weight of the current policy network. There is no gurantee which way is a better choice.

III. Experiment

A. Lunar Lander

The objective of Lunar Lander problem is to land the Lander successfully with the feasible position. The states are represented as a 8-dimentional vector $(x, y, vx, vy, \theta, v\theta, leg_L, leg_R)$, where x and y represents the x and y coordinates of the lunar lander's position. vx and vy are the velocity of the lunar lander corresponds to x and y axes. θ is the angle of the lunar lander and $v\theta$ is the angular velocity of the lunar lander. And leg_L and leg_R represents whether its left leg or right leg has touched the ground. There are four different actions, do nothing, fire the left orientation engine, fire the right orientation engine, fire the main engine. The landing pad is always at (0,0), and that's where we are targeted to.

The total rewards moving from the top of the screen to the landing pad ranges from 100 - 140 points varying on the lander's position on the pad. If the lander moves away from the landing pad it is penalized the amount of reward that would be gained by moving towards the pad. An episodes finished if the lander crashed or comes to rest, receiving an additional -100 or +100 points respectively. Each leg ground contact is worth +10 points. Firing the main engine incurs a -0.3 point penalty for each occurrence. Fuel is infinate, agent could fly and learn on its first attempt.

B. DQN Implementation

There are several components of DQN for lunar lander, replay memory, neural network, agent learner.

1) Replay Memory: Replay memory is based on list object in python. It will initialize with the given size parameter as the number of records kept in the memory. When samples needed to be created, replay memory will randomly create a mini batch for a given batch size. For performance consideration, the list will be initialized to have the length of the size, and gradually update the unused or old record when new records come in.

2) Neural Network: Single multiple layer perceptron is used for this part. There are three linear layers there, the first two ended with Relu non-linear function and the last layer map it to the Q value for 4 actions. The number of neurons in each layer is a tunable parameter, the network used for the results are,

A) Linear layer map input features to 64 neurons followed by Relu
B) Linear layer map output from last layer to 64 neurons followed by Relu
C) Linear layer map output from last layer to 4 neurons corresponding to 4 actions

3) Agent Learner: The algorithm for agent learning process can be shown in Algorithm 1,

Algorithm 1: Agent Learning Process

```
Result: Q value function approximation
initialization:
Replay Memory with Size 5000
Policy Network
Target Network
for iter = 1 : max_iter do
    reset gym env
    while True do
        ε greedy to find the action a_t
        execute a_t and observe r_t and s_{t+1}
        add sample (s_t, a_t, r_t, s_{t+1}) to replay
        memory
        if episode end then
            if not timeout then
                mark next state value as 0
            else
                mark next state value based on Q
                value
            end
        end
        sample mini batch from replay memory
        use target network to calculate target value
        with Bellman operator
        back propogation to update policy network
        for every T_target step do
            update target network from policy
            network
        end
    end
    if The averge rewards over the last 100
    episodes > 200 then
        break
    end
end
```

C. Results

1) Convergence and Rewards: The default hyperparameter used to train the agent are,

- discount rate, $\gamma = 0.99$
- starting ϵ greedy, $\epsilon_{start} = 1.0$, 0 will be used for testing
- ending ϵ greedy, $\epsilon_{end} = 0.01$, 0 will be used for testing
- ϵ decay, $\epsilon_{decay} = 0.995$, there is no decay for testing
- learning rate. $\alpha = 0.0005$
- target network update step, $T_{target} = 4$
- max iteration = 2000
- batch size = 64
- soft update rate for target network $\tau = 0.001$
- dropout ratio for policy network = 0

The results of reward with respect to training episode are shown in Figure 1.

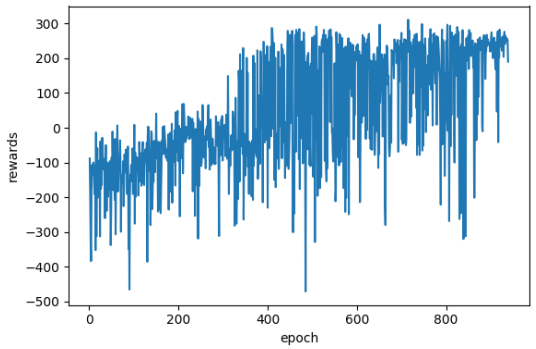

Fig. 1. Reward with respect to number of episode with default parameters.

The agent met the stopping criterion (average rewards over last 100 episodes is greater than 200) at 927 iteration. It can be easily seen that there seems to be a change point for the rewards. Prior to that, agent tends to do more exploration, which leads potentially to low rewards. But after certain steps, when the Q value functions tends to converge more and when ϵ becomes smaller, agent tends to do exploitation more and the rewards started to increase significantly. However, we can find that even when the agent met with the stopping rule, it will still have negative rewards. Two potential reasons are, 1) agent still do exploration because ϵ is not 0. 2) our stopping rule is kind of loose, as we only care for the averge rewards, and we don't care for the downside risks (negative rewards).

Use the learnt Q value function, how will agent perform? The rewards for testing episodes are shown in Figure 2. Average reward is 205.4, but as we mentioned earlier, there are still very low rewards showing up.

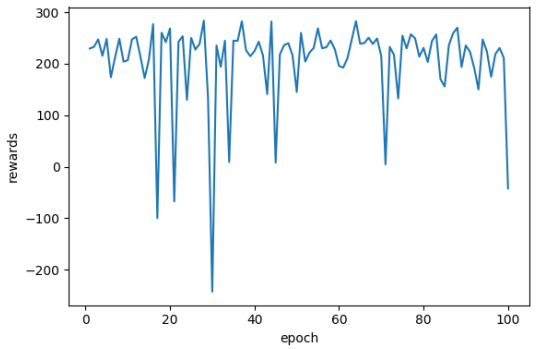

Fig. 2. Reward with respect to number of episode for testing episodes.

2) New Feature Construction: To better train a network, sometimes we need to construct additional features. In this section, we included three additional features and see the influence of them to the agent learning process, they are,

- velocity of agent with respect to origin, $v = \sqrt{v_x^2 + v_y^2}$
- distance of agent to origin, $s = \sqrt{x^2 + y^2}$
- if both of agent's leg touch ground, $leg = leg_L \cdot leg_R$

The rewards for episodes are shown in Figure 3. The agent took 1229 steps to meet with the stopping criterion. This is longer than previous experiments. Leave out the randomness for now, it also kind of makes sense as when the complexity of model goes up, we will need more training data to help the model learn. Also to note, for the testing case, the average return is 190.96. There is definately randomeness here, but adding new features didn't significanly improve or decrease the performance of agent given the same stopping criterion.

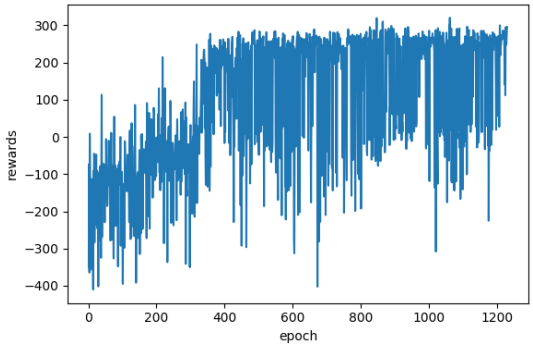

Fig. 3. Reward with respect to number of episode with three additional features.

3) Gamma Influence: Discount rate γ is an important hyperparameter for learning. When γ is large, agent only care for near horizon rewards, and making the learning algorithms easier to converge. On the contrary, a low γ will make agent care for longer historical rewards, it might be promising for real-world applications but it also makde the algorithms harder to converge. Rewards with respect to different choice of γ is shown in Figure 4. It can be easily seen that a really large γ (0.99 or 0.999) don't differ significantly for performance but a lower γ as 0.9 can significantly worsen the performance as we described earlier.

4) Learning Rate Influence: Learning rate α is also an important hyperparameter for learning. A large learning rate can help algorithms converge faster initially but it has the risk to diverge when it comes near to the local minimum. Comparatively, a smaller learning rate will have higher probablity to make sure the model can converge but it might take signicantly longer time. Usually people will use decayed learning rate for learning, but here for simplicity, we all use a fixed learning rate

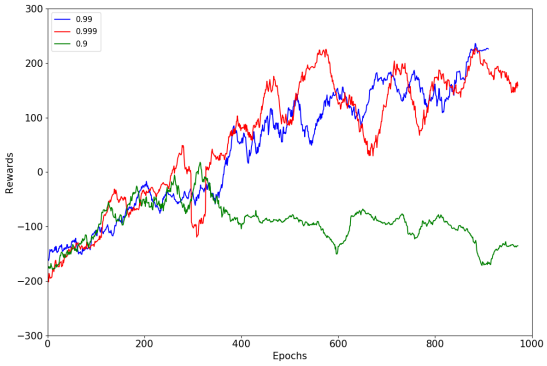

Fig. 4. Reward plots with respect to different choice of γ, to make the plot clearer, a running average with window 30 is used.

for experiments. Rewards with different choices of α is shown in Figure 5. Similar to what we dicussed earlier,

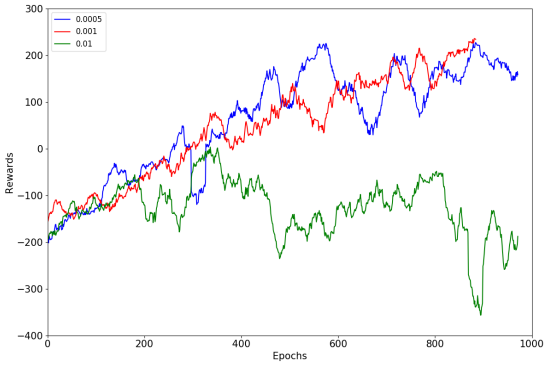

Fig. 5. Reward plots with respect to different choice of γ, to make the plot clearer, a running average with window 30 is used.

a comparatively small α will help the convergence but a larger one (0.1 in this case) will make the learning diverge.

5) Epsilon decay Influence: There is always the dilemma between exploration and exploitation in RL field. ϵ is the key parameter to control the tradeoff between the two. A larger ϵ will make the model explore more, so it will have a better estimation of Q value function. On the contrary, a smaller ϵ will lure the agent to exploit more, so more rewards will be collected if the agent already has a good estimation of Q value function. Usually a decay of ϵ will be used so agent tends to explore more at the beginning and exploit more when it's more confident about Q value function it estimated. Rewards with different choices of ϵ decay rate is shown in Figure 6. We can find a smaller ϵ decay rate will help the agent meet with stopping criterion earlier if the agent explored enough earlier. And a larger ϵ decay rate will make the convergence more robust but also making the agent later to meet with the stopping criterion because it

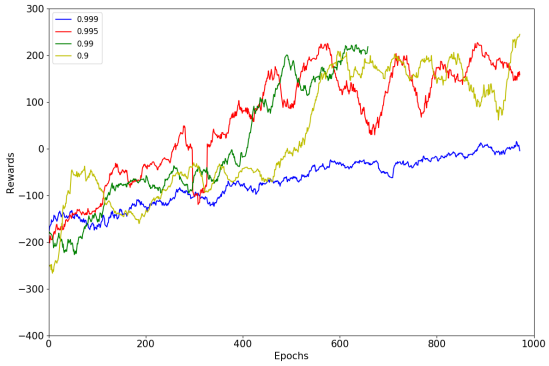

Fig. 6. Reward plots with respect to different choice of ϵ decay, to make the plot clearer, a running average with window 30 is used.

still maintain a fair number of explorations, which might lead to low rewards.

6) Target Network Update Rule Influence: As we mentioned earlier, when and how to update target network will also make a difference for agent learning. Different choice of update rule is shown in Figure 7. I won't discuss too much here as I am running out of

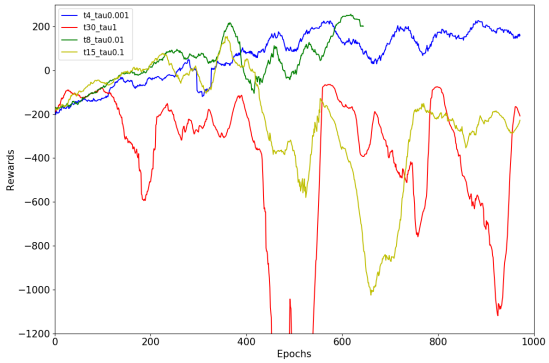

Fig. 7. Reward plots with respect to different choice of ϵ decay, to make the plot clearer, a running average with window 30 is used.

space. But a more aggresive update (lower τ, smaller T_{target}) seems to perform worse. Probably because a really negative rewards will have too much influence to both networks, which leads the model to a very bad local optimal. A larger replay memory might help.

IV. Conclusion

We researched the application of DQN to lunar lander and its convergence bahavior with choices of different hyperparameters here. DQN can perform reasonably well with careful choice of parameters. More experiments will be done if I have more spaces.

References

[1] Yang, Z., Xie, Y., & Wang, Z. (2019). A theoretical analysis of deep Q-learning. arXiv preprint arXiv:1901.00137.