Full Commit Hash: ca64e18e5d4e0e80f25cc4f1a925a385c6e9ad74

Understanding the effect of hyper-parameters on the performance of Deep Reinforcement Learning

Anh Duc Vu
College of Computing
Georgia Institute of Technology
avu41@gatech.edu

***Abstract*—This paper examines the effect of a range of hyper-parameters on the performance of a deep reinforcement learning process. Specifically, we investigate how a range of values of learning rate, batch size, and target network update frequency affect the performance of Deep Q Network. Our experiments show that different values of batch size and target network update frequency have minimal effects on the algorithm convergence ability. However, the learning rate highly affect the performance of the agent, and we found that said parameter needs to be tuned meticulously for the model to converge at a given time step. Also, we examine the application of cyclical learning rates in complex off-policy Deep Reinforcement Learning. Results indicate that cyclical learning rates in off-policy deep reinforcement learning indeed lead to convergence, and perform similarly to the fixed tuned learning rate.**

***Index Terms*—deep reinforcement learning**

I. Introduction

A. An overview of Q-Learning

Q-learning algorithm is of temporal difference learning method class, because the agent learn by bootstrapping from the current estimate of the action-value function. Q-learning is also an off-policy algorithm, contraries to other on-policy algorithms such as SARSA. The agent learns the optimal action-value function independent of the policy being followed. The update of the action-value function with the Q-learning algorithm is defined by

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

In order to update the action-value function after each step, the agent requires the keeping of a lookup Q-table. Specifically, Q-table is a table with the row as all possible states spaces, and the column being all possible action spaces. The agent uses the table to look-up the optimal action for a given state space. This is the limitation of Q-learning: it only works with discrete and finite action and state spaces. To solve this problem, the Deep Q-Networks algorithm developed by a team of researchers in DeepMind in 2013 swap out the Q-table and replace it with a universal function approximation: The artificial neural network [5].

B. An overview of Deep Q-Network

With the rise of deep learning in recent years, multi-layered neural networks (NN) has been widely used to advance different fields, including but not limited to Natural Language Processing, Computer Vision, and Reinforcement Learning. Because NN is a universal function approximation, a team of researchers in DeepMind used the architecture to approximate the action-value function in replace of the Q-table, introducing the Deep Q-Networks (DQN) algorithm (Mnih et al., 2013, 2015) [4] [5]. There are two main differences between DQN and Q-learning. First, DQN use NN to approximate the values of $Q(S_t, A_t)$ and $\max_a Q(S_{t+1}, a)$ in equation (1). Second, the DQN algorithm store the experience of each step in an experience replay memory list and sample a random mini batch from the list for each step. This mechanism smooths the training distribution over past experience.

Mihn et al. used DQN to teach the the reinforcement learning agent to play 49 different Atari 2600 video games. The agent used a convoluted neural network (CNN) to process the pixel image frame from the video games, and the agent can choose the action which correlated to possible actions of the games. As the algorithm used an experience replay method to de-correlate consecutive experience, Q-learning was chosen as it is a model-free, off-policy algorithm. Using the DQN architecture, Mihn et al. reported impressive results, showing that the DQN learned reached or exceeded human level on 29 of the 46 games. It is important to note the agent learn to play end-to-end, i.e., the agent learn to play directly from the pixel images of the game. In addition, the DQN agent learn without any domain knowledge about each individual game. Due to being an off-policy algorithm, DQN is prone to divergence. In 2015, Minh et al. implemented a DQN with a target network, which is the prediction network with the weights from a fixed previous number of steps in order to stabilize the training process [4].

C. The importance of hyper-parameters tuning

The performance of machine learning models highly depends on the hyperparameter settings. Because certain config-

urations of hyperparamters might help the model learn faster and more accurate than others, it is important for the model to be tuned to perform well. This is true for most machine learning algorithms, including DQN.

The two hyperparamters that involved in the optimization procedures are learning rate and batch size. In the following sections, we will discuss how the model perform under a range of learning rate and batch size's configurations. In addition, we also investigate the performance of cyclical learning rate when being compared with a range of values of learning rate for DQN.

One of the main differences between the two DQN models introduced by Mihn et al in 2013 and 2015 is the target network. The target network help the DQN, which is prone to divergence, stabilize. In the following sections, we also look at how the model perform with different values of target network update frequency.

II. Experiments and Results

A. Deep Q-Network agent and training environment

While there are many different variations of the DQN algorithm, for our agent, we implemented a vanilla DQN algorithm.

In our implementation, instead of using a convoluted neural network similar to that of Mihn et al. used, we used a feed-forward NN with three layers, each has 256, 256, and 84 nodes respectively, and we chose stochastic gradient descent (SGD) for optimization procedure. The algorithm uses experience replay, with the first 1000 steps being randomized to populate the replay list. Initially, ϵ is initiated as 1, and we decayed ϵ the next 50000 steps until $\epsilon = 0.001$. For each learning step, we sample a random mini-batch of size 32. We will explore the effect of different values of batch size in the following sections. Our DQN agent has a target network, which is updated with the weights of the prediction network every 2000 steps. We will explore the agent performance using different target network update frequency in the following sections. Finally, for our algorithm, instead of using a fixed tuned learning rate, we use a cyclical learning rate, which cycle between $\alpha_{max} = 0.001$ to $\alpha_{min} = 0.01$ every 2000 steps. To stabilize the training process, the momentum also cycle with the learning rate in the opposite direction. Specifically, as the learning rate increase and decrease, the momentum decrease and increase between 0.8 and 0.95. The cyclical learning rate method is first introduced to overcome manual learning rate tuning for neural network (Smith, 2015) [1]. In 2020, Gulde et al. implemented cyclical learning rate in deep reinforcement learning method and showed that cyclical learning rate performs similar or better than highly tuned learning rate for on-policy algorithm PPO2 (Gulde et al., 2020) [3]. However, Gulde et al. state that whether this method can be applied to off-policy algorithm is an open question, since off-policy algorithm has weak convergence properties. In our experiment, we show that cyclical learning rate performs similarly to tuned learning rate, and the algorithm do converge. We will further explore how different values of learning rate perform compared to each other and to cyclical learning rate method in the following sections.

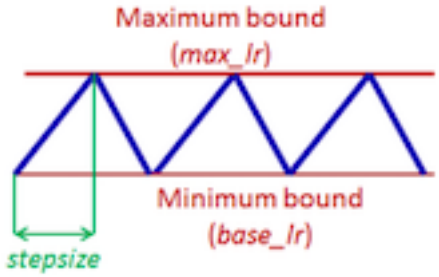

Fig. 1. Learning rate cycle between maximum and minimum bound each step size (Smith, 2015) [1]

We test the DQN agent using the Lunar Lander environment provided by OpenAI's Gym library. The state space is 8-dimension, with six continuous state variables, and two discrete ones. Hence, the first layer of our DQN's NN has 8 nodes, representing the 8 dimensions states. The action space is discrete, with 4 possible discrete actions: do nothing, fire the left or right engine, and fire the main engine. Hence, the last layer of our NN has 4 nodes, representing 4 possible action states. The agent has to learn to land at coordinates (0,0), and the problem is considered solved when the agent achieves an average score of 200 points over 100 consecutive runs.

B. Performance during and after training

We let the agent train using the algorithm described above in 1000 episodes, and plot the episode reward and mean reward for the previous 10 episodes in figure 2. While the absolute value of reward varies a lot between each episode, we observe that the agent performance improved consistently by looking at the trend of the mean reward. The agent reached the 200 points mark after around 400 episodes, and the performance stabilize for the rest of the 1000 episodes.

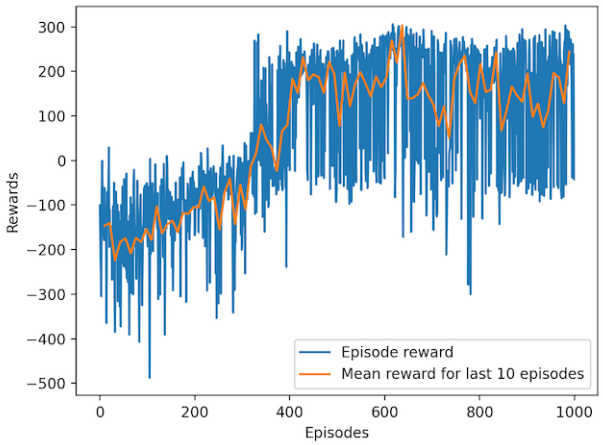

Fig. 2. Reward for each episode and mean reward during training

Using the trained agent after 1000 training episodes, we let it run with strictly greedy policy for 100 episodes, with the rewards for each episode plotted in figure 3. We see that while the agent occasionally get less than 200 points, the mean reward for 100 consecutive episodes is above 200 points, with the maximum points in one episode of up to 300 points. Hence, we conclude that our DQN implementation with cyclical learning rate solved the lunar lander problem.

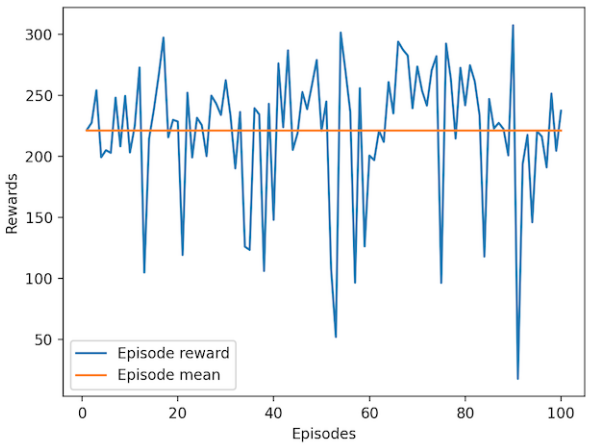

Fig. 3. Reward for each episode and mean reward after training

III. Hyper-parameters analyses

A. Learning rate experiment results and analyses

In the previous section, we see that vanilla DQN agent can solve the Lunar Lander problem using the cyclical learning rate. In this subsection, we take a closer look of the performance of the agent using different fixed learning rate, and compared them with each other, and with the cyclical learning rate's performance. As we attempt to analyze how learning rate affect performance, each run use a range of values of α, which are 0.001, 0.0025, 0.005, 0.0075, and 0.01. With all other hyperparameters consistent with our implementation in the previous section, we let the agent run for 1000 episodes, and plot the mean reward for the previous 10 episodes in figure 4.

Our result shows that the agent with the learning rate of 0.0025 performed best, as it reached the average reward of 200 points after around 400 episodes. The other agents weren't able to converge within 1000 episodes. Looking closer, this phenomenon makes sense. Specifically, the learning rate of 0.001 is too small, thus the optimized step after each episodes were not enough for the agent to learn. The agent with learning rate of 0.001 would need more than 1000 episodes to converge. Indeed, looking at figure 8, where the agent used a fixed learning rate of 0.001 and train for 5000 episodes, the agent reached the 200 points average after approximately 1500 episodes, before completely forgot learned information after around 3500 episodes. We take a closer look at this phenomenon in the next section.

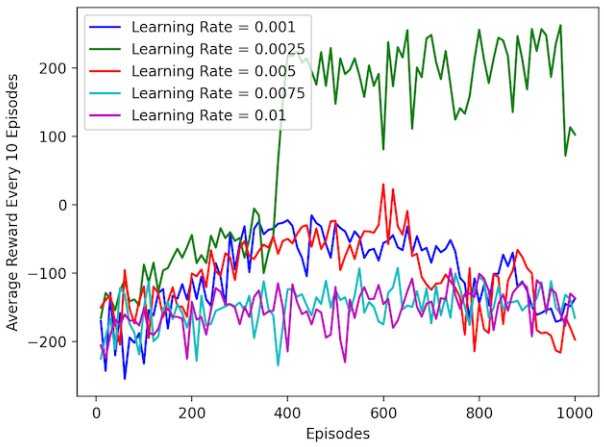

Fig. 4. Average Reward with different learning rate

In the case of the agent with the learning rate of 0.005, 0.0075, and 0.01, the optimized updated after each learning rate are too large, thus also cause divergent behaviors. Hence, from the experiment, we can conclude that the optimal learning rate is approximately 0.0025.

Comparing the performance of the fixed optimal learning rate of 0.025 with that of the cyclical learning rate method in figure 2, we see that both converge and solve the problem after approximately 400 episodes. Hence, we conclude that cyclical learning rate can be successfully applied to vanilla DQN algorithm. Furthermore, experiments show that training with cyclical learning rate can yield performance similar to fixed tuned learning rate. This indicates that using cyclical learning rate method can help reduce the time to manually tune the learning rate hyperparameter to find the optimal value.

B. Batch size experiment results and analyses

In this section, we investigate the performance of the agent with different values of mini batch size. Specifically, we let the agent train with 1000 episodes, with different values of batch size of 16, 32, 64, 128, and 256. We keep the other hyperparameters similar to that of our previous run, and constant during each run. The mean reward for the previous 10 episodes of each agent are plotted in figure 5.

Our result shows that the performance of agents with different batch size are closely similar, with smaller batch size perform slightly better by converging earlier. All agents are able to converge and solve the problem within 1000 training episodes. Note that while the agent with batch size of 128 seems to diverge around episode 750, the agent was able to converge again, and its performance are similar to that with other batch size at the end of episode 1000. Hence, our experiment shows that, while smaller batch size perform slightly better than larger ones, batch size do not have a large impact on whether the agent converge or not.

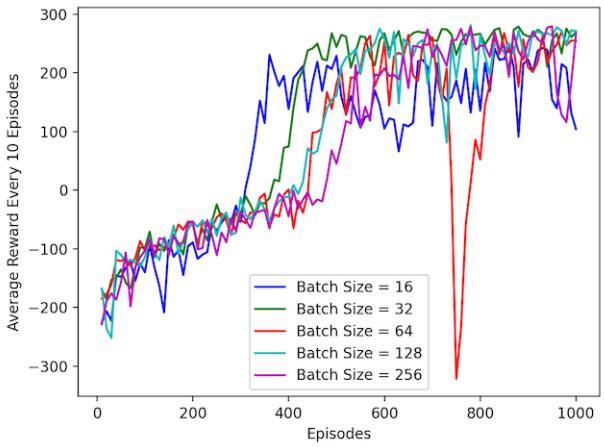

Fig. 5. Average reward with different batch size

C. Target network update frequency results and analyses

Similar to the previous experiment, we implemented the DQN agent with a range of values for the target network update frequency, C. While we keep other hyperparameters values constant, we use a range of C, increasing from 2000 to 10000, with an increasing step of 2000. The mean reward for the previous 10 episodes of each agent are plotted in figure 6.

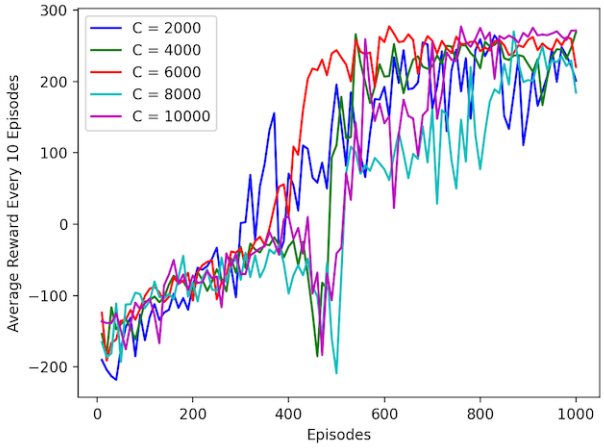

Fig. 6. Average reward with different target network update frequency

While we know that agent with $C = 0$ can still converge, since the first implementation of DQN by Mihn et al. in 2013 didn't have a target network, it is interesting to investigate how different target network update frequency affect the agent performance. Our result shows that while all values of C lead to converge, and all agents solve the problem by the end of the episode 1000, different values of C does affect how fast the agent learn. Specifically, agent with $C = 1000$ learn the slowest, reaching the average score of 200 points after approximately 850 episodes. The best performing agent used an intermediate value of C, $C = 6000$. Also, we see that large values of C such as 8000 and 10000 cause the agent performance to diverge before converging again. Hence, we conclude while target network update frequency affect the rate of learning of the agent, it would also not affect whether the agent converge or not if given enough episodes of training, similar to the effect of different batch size.

IV. Problems and Pitfalls

In this section, we discuss the problems and pitfalls encountered when implementing the DQN agent to solve the Lunar Lander problem. The first pitfall I encountered is not setting the action value for the terminating step as the reward of that step. Specifically, the action value for the sample random mini batch are calculated using equation (2) below. During my implementation, I have found that if we calculate $y = r_i + \gamma \max_a Q(S_{i+1}, a)$ for all step, ignoring terminating step, the agent won't be able to learn no matter how long we let it train. Only when the algorithm is updated to reflect exactly the equation (2) did it learn within 1000 episodes. Intuitively, this make sense. By letting the agent know that the action value for the terminating step is only the reward received from that action, we signal that the agent won't receive additional rewards if executing that specific action at that specific state. Hence, the agent can learn to optimize the reward, knowing that it won't receive additional rewards following said step. Implementing equation (2) is imperative for the DQN agent to learn and solve the Lunar Lander problem.

$$y = \begin{cases} r_i & \text{if episode terminate at step i + 1} \\ r_i + \gamma \max_a Q(S_{i+1}, a) & \text{otherwise} \end{cases} \tag{2}$$

The second problem I found is that the DQN agent learn how to solve the problem, and then abruptly and completely forget how to solve it. Specifically, letting the agent with a cyclical learning rate in the second section run for 5000 episodes, we observe that while the agent converged and solved the problem within 1000 episodes, it slowly diverged after around 1500 episodes, and completely forgot what it learned by approximately 1700 episodes, as shown in figure 7.

Initially, I theorized that cyclical learning rate make the DQN algorithm training process unstable, which lead to divergenct behavior. However, an experiment with the same agent using a fixed learning rate of 0.001 and all other hyperparameters stay the same show that this is not the case. In figure 8, we see that an agent with a low learning rate required more episodes to converge, and stayed converge for a longer time than an agent with a cyclical learning rate. However, after approximately 3500 training episodes, the agent suffered the same forgetting problem, and completely forgot how to solve the problem by the end of 5000 training episodes.

Agents forgetting learned information can't happen in tabular Q-learning, but can happen when we use function approximation such as the neural network in DQN. This is because

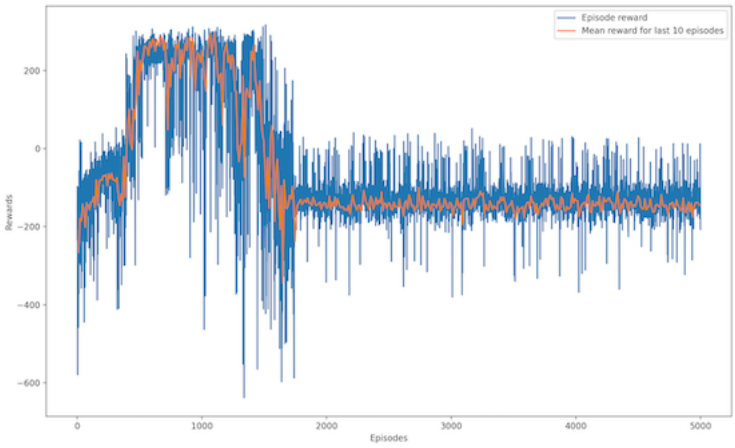

Fig. 7. DQN agent learn and forget with cyclical learning rate

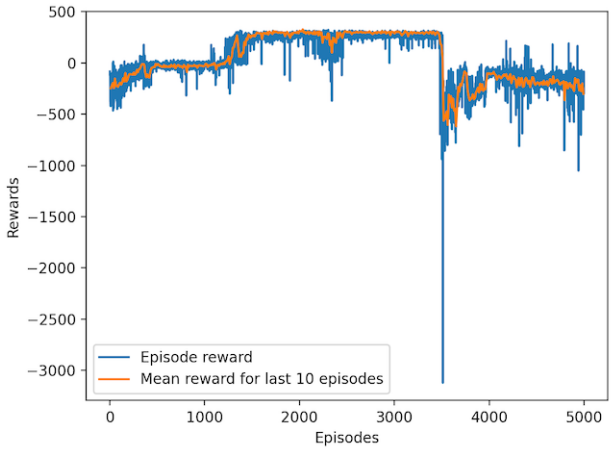

Fig. 8. DQN agent learn and forget with fixed learning rate of 0.001

the issue is not related to Q-learning specifically, but related to neural network, and is known as catastrophic interference, or catastrophic forgetting. The tendency for neural network to forget learned information is first introduced by McCloskey and Cohen in 1989 [2].

As the DQN agent are learning to solve the problem, the experience replay memory are populated with both good and bad experience. By randomly sampling mini batch from this mixed memory list, the neural network optimize to produce more of good experience, and thus the DQN agent learn to perform better after each step. However, once the agent learned how to solve the problem, after a certain number of episodes, the experience replay memory are only populated with successful experiences. As a result, the neural network forgot bad experience, and approximate all states values higher than it should. This would not happen in tabular Q-learning, since we would keep track of the exact action-value of each states using a Q-table, rather than approximating it. To overcome the issue, I stopped the training process after training for 1000 episodes, which is enough for the agent to learn and before the agent forget how to solve the problem.

V. Conclusions and Future Works

We show that a vanilla DQN agent with cyclical learning rate can solve the Lunar Lander problem. In addition, our results show that different values of batch size and target network update frequency have minimal effects on the algorithm's convergence ability. The learning rate, however, highly affect the performance of the agent. Specifically, we found that the learning rate needs to be tuned if we want the agent to converge at the optimal time step. Also, we examine the application of cyclical learning rates in DQN agent. Our experiments show that cyclical learning rate in off-policy method such as DQN lead to convergence, and perform similarly to the highly fixed tuned learning rate.

For future works, it would be interesting to explore how other off-policy deep reinforcement learning method with cyclical learning rate perform in other environments. In addition, it would be beneficial to take a closer look into how to prevent the catastrophic forgetting phenomenon we observed in our experiments.

References

[1] L. N. Smith, "Cyclical Learning Rates for Training Neural Networks," arXiv.org, 04-Apr-2017. [Online]. Available: https://arxiv.org/abs/1506.01186.

[2] M. McCloskey and N. J. Cohen, "Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem," University of Illinois Urbana-Champaign, 20-Jun-2015. [Online]. Available: https://experts.illinois.edu/en/publications/catastrophic-interference-in-connectionist-networks-the-sequentia.

[3] R. Gulde, M. Tuscher, A. Csiszar, O. Riedel, and A. Verl, "Deep Reinforcement Learning using Cyclical Learning Rates," arXiv.org, 31-Jul-2020. [Online]. Available: https://arxiv.org/abs/2008.01171.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," Nature News, 25-Feb-2015. [Online]. Available: https://www.nature.com/articles/nature14236.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," arXiv.org, 19-Dec-2013. [Online]. Available: https://arxiv.org/abs/1312.5602.