https://github.gatech.edu/jliu727/RL_Project2

Agent Training in Lunar Lander Environment

Jiaxin Liu
Department of Computer Science
Georgia Institute of Technology
Georgia, U.S.
jliu727@gatech.edu

***Abstract*—This article looks into utilizing reinforcement learning methods learned from class in openAI Gym Environments. Specifically, we look into training an agent to play lunar lander game and investigate relationship between final performance and hyper-parameters used.**

***Keywords*—torch, DQN, hyper-parameter, generalization.**

I. INTRODUCTION

In previous lectures, we learned to create the agent to learn about the environment and find the optimum solution by creating a Q-table, storing Q values for each state-action. This is the regular Q-learning method. However, it is not applicable to all instances. For example, if we have a continuous state-action space in high dimension, it is not realistic to create a Q-table to solve the problem. Therefore, we need some way to generate Q without having to actually visit all state-action space and here we use generalization. In generalization, we try to approximate the Q value by looking for a 'function' assuming that 'similar' states would share 'similar' Q values. In this way, we do not need to visit all space to learn, instead, we can learn by examples to train the 'function' and generate Q values through it when reaching a state never visited before.

II. LUNAR LANDER ENVIRONMENT

In this project, we would like to use generalization in reinforcement learning to write an agent to solve lunar lander game. We get access to the environment from openAI Gym. The goal of the game is to land the 'lunar lander' successfully on the target pad. The state-space for this problem has 8 elements -- (x, y, vx, vy, θ, vθ, left-leg, right-leg). The first 6 elements are continuous and the last two are discrete. (x,y) represents the lander's position. (vx, vy) represents the linear velocity components. Θ represents the angle of the lander and vθ is the angular velocity. Left-leg and right-leg are 0 when in sky and 1 when touching the ground.

The lander starts from the top of the screen and gets some points by approaching the landing pad while losing some points by deviating from the pad. The lander is deducted a very small reward by firing its main engine. If the lander crashes, it gets -100 and if it lands, it gets 100 and those two states terminates the game. The environment also sends out a done signal when the steps taken by agent exceeds 1000 steps. There are four possible actions, do nothing, fire the left orientation engine, fire the main engine, fire the right orientation engine. This is a very typical problem where state-action space is infinite and we need a 'function' to help find Q of new state.

III. IMPLEMENTATION & ALGORITHM

I implemented the agent using DQN (Deep Q-Learning). It combines neural network from supervised learning with Q-learning where the neural network is treated as the 'function'. By inputting a state into the neural network, we expect it to output the value of each action. The implementation consists of two classes.

First, we have a class for building neural networks NN() to train on. I am using pytorch where we can define the number of hidden layers, number of nodes in the hidden layers and initial weights for the neural network. Here, the input to the network is the 8-element state, therefore the input layer for the network has 8 nodes. I am using only one hidden layer with 50 nodes. The expected output is value for each action and we have four actions, therefore, the network output layer has 4 nodes. Then we use the forward function from torch to produce value from input state.

Then, we also need to create a class object called DQN where all the relevant initialization and training methods are specified in. The parameters related to the DQN process need to be initialized in the __init__ function. There are 13 parameters I care about that will be used in later methods, α, Δα, final α, γ, ε, Δε, final ε, memory cap, iterations to update NN, number of actions, number of states, hidden layer node number, batch size. In the __init__ function, we also create the two neural networks that will be used later. The number of nodes in the hidden layer is set here. For most of my experiments, I set it to 50. I will explain other parameters later in related methods in DQN class.

The first method in the DQN class is select_train_action(self, s) where the state we would like to select the action for is the input. The best action is chosen by feeding state into the train network and pick the action with the highest value from the output. However, we do not always pick the best action. For probability ε, we pick a random action instead of the best action to allow more exploration.

The action is selected and performed and the agent received feedback on reward and next state from the environment. This information will be used later in network training so we need to store it in the memory pool. The information consists of (state, action, reward, next state). By setting the memory cap, we allow no more than the cap pieces of information been stored in the pool. Therefore, the pool is a matrix with number of columns equal to the length of information (18) and number of rows equal to the memory cap. Mostly, I set the memory cap to be 50000, usually storing about 200-500 examples in the pool. When the pool is full, we replace old examples with new examples in order by memory pointer to keep track of where we are in the pool.

In this agent, we are using two neural networks, train_nn and final_nn. We mainly use train_nn for choosing action and learning from past examples to improve the network. final_nn is used as a buffer to maintain stability in learning. It is used to calculate the expected value and is only updated once in a while (usually 1000 steps in this particular experiment). By updating it slowly, we avoid getting too much effect from any of specific example and keep learning more stable. The update process is done by function update_param().

The most important part for the agent is to learn from past experience and improve our neural network. In the learn() function, we first randomly select mini-batches of data from the pool. The reason we do this instead of training on the way is to avoid relevance between the sample we feed to the network. For each sample, we need to know the estimated output and the expected output to calculate its loss. Here the estimated output is the output from train_nn neural network. Then we get expected using the following Q-learning function:

$$TargetQ = r + \gamma * \max_{a'} Q(s', a')$$

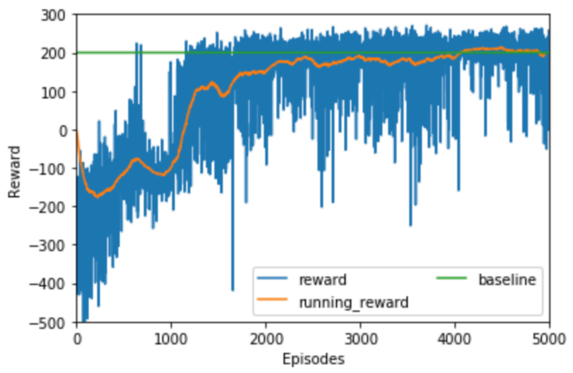
where we get $\max_{a'} Q(s', a')$ from final_nn neural network.

Then, we use Huber-loss function from torch to calculate the loss and use back-propagate the loss to optimize train_nn. The optimizer we use is Adam from torch.

In order to learn well, we would like to reach a healthy balance between exploration and exploitation. We would like to explore a lot in the beginning of the learning process and focus more on exploitation later. Therefore, we need to adjust ε value along the process. I decrease the ε every learning step by Δε. Note also that I do not start learning until the memory pool is full (to avoid learning on meaningless information), we have already collected some amount of information from random exploration before ε decay starts. Here I am also providing the option to decay learning rate α along the process as we know that when approaching the ideal neural network, we need more delicate changes to the weights but we also do not want it to be too small to learn anything along the way. I also tried decaying both ε and α less frequently after an episode instead of after a step. Most of the parameters are free to modify and will have impact to the learning process but due to limited time and long training process, I mainly adjust α, γ, ε and memory cap to see the difference.

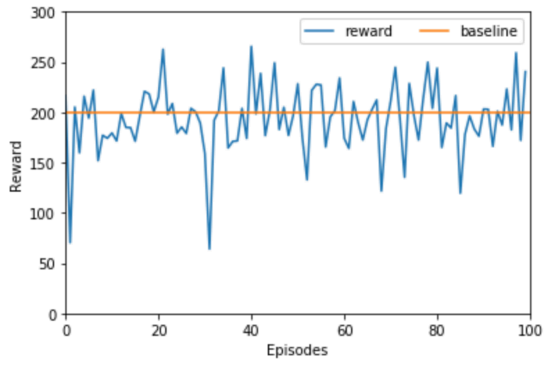
IV. Result Analysis

Episode Reward during Training:



This is the best performing agent found is in setting a large memory pool, memory cap = 500000, parameter update every 10000 steps to maintain stability with a small train batch = 32. We use two neural network each with 1 hidden layer of 100 nodes. Alpha is decayed from 0.001 to 0.00001 in 100 episodes and Epsilon is decayed from 0.95 to 0.05 in 900 episodes with the first 50 episodes maintaining a high epsilon of 0.95. It took around 1100-1200 episodes to increase reward above 0 at which point it learns to land and another 3000 episodes to increase average reward above 200. We trained a total of 5000 episodes.

Episode Reward during Testing:



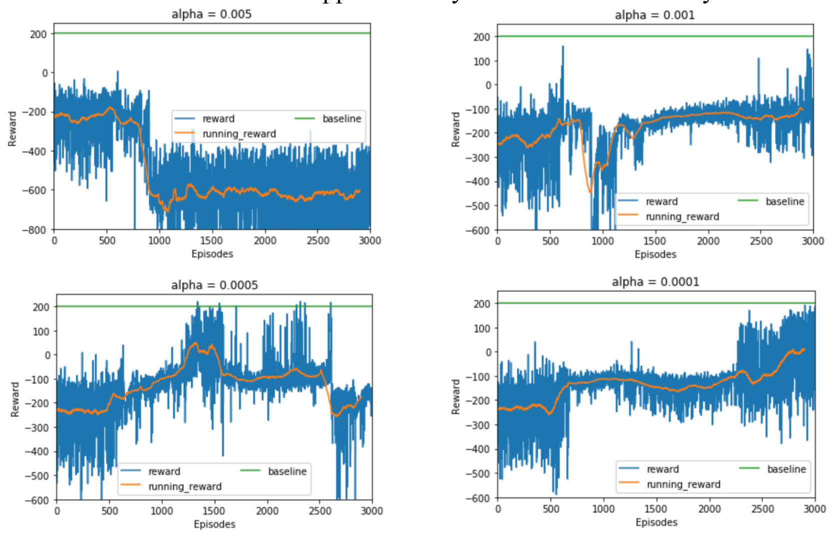
In testing, we use final_nn neural network trained in the training process to determine the optimal action at each state. We can see that most of the reward are around 200 baseline with a couple dropping down to around 100. Looking from the video, it seems that occasionally the agent hovers without complete landing for a long time. The reason could be that for some states, the learning is not thorough yet, resulting in agent trying to avoid crashing rather than proceed to landing. One possible solution to this could be training more episodes to give agent more time to learn. The epsilon value stops decaying at 0.05 so with more time, randomness would run into better action than hovering for the agent to learn.

Effect of Hyper-parameters:

There are many hyper parameters available to adjust as mentioned in the implementation above. We mainly focus on α, ε and memory cap modification. Due to the long runtime for each experiment, we do not use as large a memory space or number of hidden layer nodes as in the optimal agent mentioned above. Instead, we use memory cap = 50000, parameter updates every 1500 steps and a hidden layer with 50 nodes for the neural networks. We are using a large gamma of 0.99 because the large rewards -100 and 100 are at the end of the game and we do not want its effect to be minimized due to too many steps during one episode. Also, we limit number of episodes to 3000 to shorten execution time. Throughout the change for alpha and epsilon, we are using epsilon decay of 0.00005 until it decays to 0.01.

Alpha value change:

In this experiment, epsilon starts from 1 and it take approximately 600-700 for it to decay to its minimum value 0.01.



Here we can see that when memory is small, it becomes relatively hard for the agent to converge compared to when we have a larger memory space. This makes sense as examples of completely random exploration (~500 episodes) at the beginning would be replaced by new non-random examples in less than 200 episodes as the agent starts learning, steps/episodes can increase dramatically. There would not be enough time to learn the free exploration thoroughly.

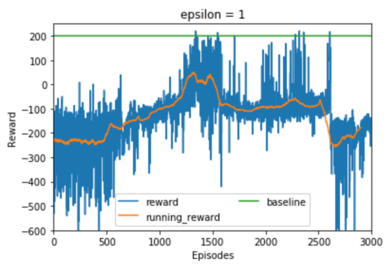
Now looking at the graphs, we chose 4 different alpha values 0.005, 0.001, 0.0005, 0.0001 to compare their difference. We can see that smaller alpha is generally doing better than large alphas. For alphas = 0.005 and 0.001, there is almost no effective learning while for alphas = 0.0005 and 0.0001, we can see the reward curve going up with more episodes. This makes sense as large alphas could result in the trained weight swinging back and forth across the ideal weights without converging as the update per step is not delicate enough.

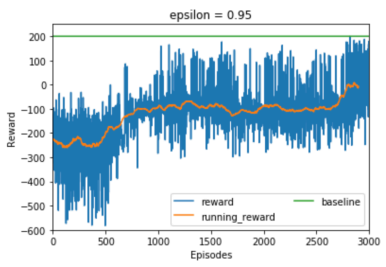
Now if we compare alpha = 0.0005 and alpha = 0.0001, we can see that agent with alpha = 0.0005 improves much faster than agent with alpha = 0.0001 at the beginning. This makes sense as smaller alphas indicates more steps to converge to the ideal weight. However, we can see that the performance for alpha = 0.0005 drops after around 1500 episodes. While for alpha = 0.0001, although we are limiting episodes for time concern, we can still see that it increases almost monotonically with the increase in episodes. This could be because with more training, the neural network would require more and more delicate change to approximate the current weights to ideal weights and alpha = 0.0005 no longer satisfies the small change required so it does not improve any more.

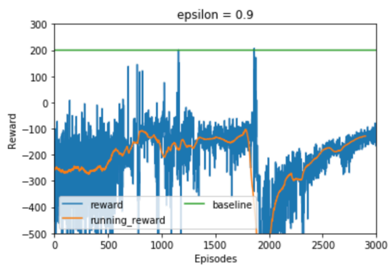
The best way to control alpha from numerous experiments seems to be decaying alpha along the way. Reasonably large alpha at the beginning to allow fast learning and small alphas towards the end to allow small tuning.

Epsilon value change

In this experiment, we are using alpha = 0.0005 without decay.



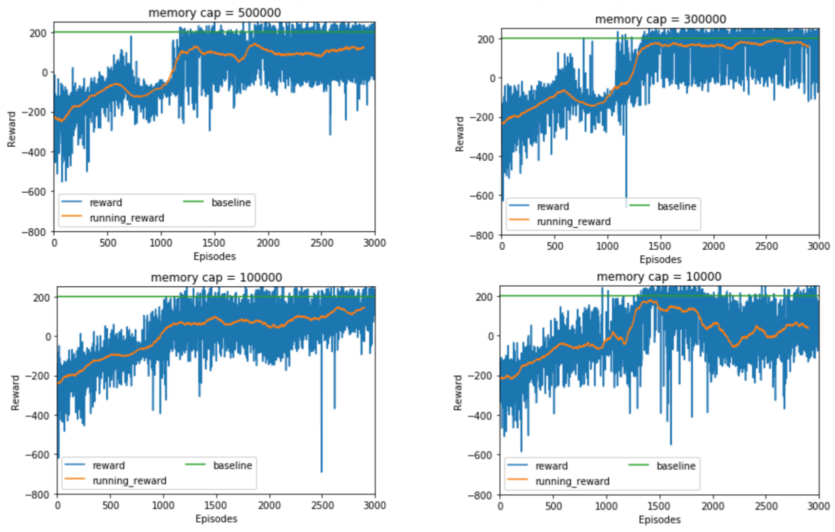




Here, we are seeing the same problem of convergence due to small memory space as above. Now looking at the graphs, we choose 3 different epsilon values 1, 0.95 and 0.9. The epsilon value determines the degree of exploration at the beginning of the game. In theory, the more we explore, the better we can be at generalization. We can see that the experiment result resonates with our expectation. For epsilon = 1 and 0.95, we can see trend of improvement and learning. When we drop epsilon to 0.9, the learning barely happens. Note here we are still decaying epsilon at rate of 0.00005 so epsilon does not always stay as high as the initial value. Although we do not have experiment result shown here but decaying epsilon too slow also cause bad learning. It could be because the randomness is too much that the agent is not using what it has learned to find the best solution. Therefore, I think the best epsilon setting would be starting from a very high epsilon value and decays epsilon in a reasonably fast speed so we can gather enough random samples without interfering later learning and convergence too much. Also, I think it is helpful to not decay epsilon to 0 entirely because a lot of experiments show that the agent could easily get stuck in long hovering states when getting close to the ground, it is helpful to have a little bit of randomness coming in to let it rest and figure out that landing is better than hovering in these states.

Memory cap change:

The above experiments on alphas and epsilons also indirectly shows how memory cap can influence the training process. Now, we want to take a look at the effect of memory pool size while keeping other parameters unchanged



https://github.gatech.edu/jliu727/RL_Project2

We chose memory cap = 500000, 300000, 100000 and 10000. We can see that high memory caps are apparently converging better compared to low memory cap. 500000, 300000, 100000 are converging while 10000 fluctuates a lot. We can also see that when memory cap is high enough, there will be no additional benefit in increasing it, for example, there is no apparent difference in using 500000 and 300000 from the graph. Now if we compare memory cap = 100000 and cap = 5000000, 3000000, we can see that learning for 100000 happened more quickly in the beginning. It is because we do not start the learning process until the memory space is partially full, therefore, memory cap = 100000 starts learning faster than the other two. Therefore, in tuning parameters, we should find a memory cap that is large enough for agent to be able to retrieve old examples for better learning while not too large to decrease learning speed.

V. PITFALLS & PROBLEMS

This project enables us to combine theoretical methods with actual problem solving. The implementation for the theory was rather straightforward. However, the process of training the agent is hard and time-consuming. The main problem I encounter is the degree of exploration and exploitation.

In the beginning, I had the wrong impression that if the method is correct, it should be easy for the agent to find optimal solution with exploration (ε) and exploitation (α) set in a reasonable range. However, when starting to train the agent, I realized how rarely the lander can achieve a successful landing (+100 reward) at the beginning. Without large exploration, we would only have very few successful records or even no successful records in memory to learn and this leads fatal consequence to the learning. The agent would be unaware of possible reward of 100 and with more episodes, it would reinforce its own idea of optimal solution without successful landing.

I tried three ways to tackle this problem. First, I do not start epsilon decay until after some time has passed (either until learning starts or until after 50 episodes), in this way, I gain a lot of random state action samples at the beginning. Secondly, whenever I see an instance of landing successfully (+100), I copy this instance multiple times into the memory space, increasing the chance of it being fetched later in the learning process. This method increased the probability of successful landing. However, during the training, I have seen instances of lander favoring one side to the landing pad. I think it could be because adding the same state-action example into the memory pool multiple times has reinforced the lander to land on this particular position even though it loses some points by deviating from the pad. Therefore, I did not include this modification. Also, I tried modifying the reward saved in memory space. For example, I increased the reward of landing successfully and decreased the reward of crashing to encourage landers to try more landing. However, it did not seem to have a large effect which makes sense to me as the key is still not enough successful landing records. Lastly, I tried to increase the memory pool space so successful information would be kept longer and more likely to be retrieved. In the end, I kept delaying epsilon decay and increasing memory pool to allow better learning.

VI. CONCLUSION

In this project, we had a better understanding of generalization by actually implementing it and training it in a complex environment. We can see how useful generalization is in case of continuous state space. The algorithm of combining the ability of relating similar states from neural network and the ability to provide large amount of samples and reasonable loss function by Q-learning can also be applied to play other games (successful in playing Cartpole).

The disadvantage of this algorithm would be that a lot of times, it may not converge. It requires very careful parameter tuning to train a successful agent. Each parameter does not work by its own. A change in one parameter can sometimes dramatically affect other parameters. A simple example would be changing the alpha value without changing alpha decay value could cause the time spent to get to minimum value, eventually affecting the training. There is no single good value but only good combination of values.

REFERENCES

[1] S. Richard, “Learning to Predict by the Methods of Temporal Differences,” Machine Learning 3: 9-44, 1988

[2] S. Richard, G. Andrew, “Reinforcement Learning: An Introduction”, Second Adition: Chapter 9, 2017

[3] M. Volodymyr, K. Koray, S. David, G. Alex, A. Ioannis, D. Antonoglou, W. Daan and R. Martin, “Playing Atari with Deep Reinforcement Learning”, DeepMind, 2013

[4] OpenAI Gym Documentation: https://gym.openai.com/docs