Project 2 - Lunar Lander

Brian Lee
https://github.gatech.edu/gt-omscs-rldm/7642Spring2019hlee815/tree/47ab454aa0a83f637eac9b205b5af4b9aa78d1e0/project2

Abstract

The lunar lander is a famous Atari game that is used to test various reinforcement learning algorithms on. In this paper, the description of the problem itself as well as a solution to it using DQN and the analysis of the solution will be discussed.

1. The Lunar Lander

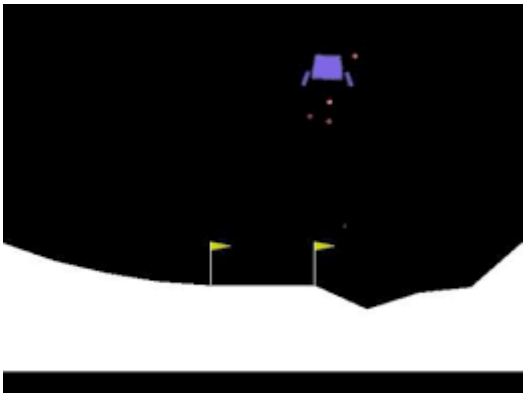

Figure 1. The Open-AI gym Lunar Lander environment

For this project, the OpenAI-Gym environment was utilized to interact with the environment. The lunar lander game's objective is to safely land the spaceship, which start from near the center-top of the screen whose starting orientation is random, in between the two flags (the landing pad) separated laterally by a fixed distance. Obtaining above 200 points over 100 trials is considered solved. The documentation of the lunar-lander environment is a little bit lacking, but the comments on the source code are helpful and more useful. See (Klimov, 2019) for detailed information regarding rewards and states. One important reward scheme not obvious is that if the current frame's situation is better than the previous frame with regard to how close the ship is to the landing pad, and how slow the speed is, etc. the agent could receive small positive or negative rewards per frame. This allows points greater than 260 total points per episode are possible and as can be seen in Section 2.3.

2. Solution

I have first tried solving the Cartpole problem, which is a much simpler problem with smaller state space and action space, to debug and verify that my algorithm is bug-free (see Section 3.1). Then I moved on to trying different parameters (Section 2.4) to train until I reached the solution of over 200 points average in 100 trials(episodes).

2.1. The Algorithm

I first gave the popular DQN algorithm described in the nature paper a try (Algorithm 1 in (Mnih et al., 2015)). The key ingredients to this algorithm are replay memory and the concept of having a separate online learning network and the target network which gets its weights updated with that of the online network every C steps. The target network is what eventually will be the outcome of the learning algorithm whose resultant policy can be used in greedy or very small epsilon-greedy fashion in unseen episodes. A few implementation details important to note in order to implement the algorithm are summarized below, as well as my choices:

- Number of iteration for gradient-descent (backpropagation) of losses for neural network training is unspecified. I chose 1 iteration.
- Preprocessing (of pixel frames to grayscale for example) was not necessary of lunar lander's 8-number states are used.
- Frame-stacking was not necessary.
- For the gradient descent step on the online Q neural network (function approximator) at the end of every time-step, first compute four action values generated by the online neural network, and replace the action value that corresponds to the action index from the stored memory with the label computed y_j with the target neural network $\hat{Q}$.

2.2. Training

In the beginning, without vectorization, training was taking up to 10 hours, so in order to try more variety of hyperparameters quickly, vectorization of keras's model.predict call

was done for the minibatches, which sped up the process by more than 5 times. I also made automated scripts that run a range of different hyperparameters with multi-threading which also collects the results and produces the graphs.

I started with the parameters given near the end of the nature DQN paper 2015. However, since those parameters were not specific to lunar-lander problem, but rather for numerous Atari games, the parameters were more generalized. Nevertheless, the important concepts such as, the replay memory size has to be sufficiently big, adequate batch size is around 32, appropriate discount factor at around 0.99, and the final epsilon value is at around 0.1, all helped guide me to the set of values that generated solutions for me (See Table 1). Maximum number of episodes isn't really a hyperparameter, but I discovered that at around 1200 episodes the "eureka" moment happens and the total rewards spike up to above 200 (See Figure 2). Note that this is approximately equivalent to about 700 episodes of actual training as the first 500 episodes (and therefore approximately 50000 frames) are spent filling up the replay memory without training the neural networks. For decaying epsilon, I linearly decreased the value per step experienced, and set it to a minimum value of 0.1 when it reached the final exploration frame number of 200000, which happens earlier if learning happens fast and each episode takes more steps to complete. If each episode takes longer time, it generally is a good sign as it means the agent is learning to not crash hastily, which is a step towards correct convergence.

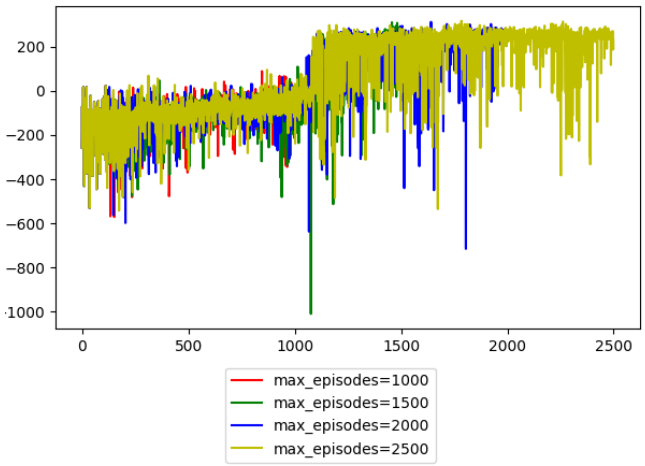

Figure 2. Multiple training runs over various maximum episodes. Total rewards per episode vs episodes. "Eureka" moment around 1200 episodes (approx. 700 actual training episodes)

There are approaches where moving average of 100 episodes can be computed and stopped, but I chose to train them with fixed max number of episodes to see if they exhibit trends of instability. Before settling on approximately 2500 max number of episodes, I've tried 10000 episodes with parameters very much close to the nature's paper 2015, and I saw that at around 7200 episodes, the agent finally achieved above 200 rewards but after a few hundred episodes, it reverts to its bad state where it achieves negative rewards, until after yet another few hundred episodes, it comes back to stable rewards above 200. This proved that the "Deadly Triad" mentioned by Sutton (van Hasselt et al., 2018) does exist for the combination of function approximation (neural network), bootstrapping (TD learning in DQN algorithm), and off-policy learning (C update frequency if greater than 1, and replay memory concept). The concept of random sampling from replay memory can be good in reducing correlations between samples for the supervised learning portion of the DQN algorithm, but one can imagine how after falling into a good optimum solution, if the replay memory still contains sufficiently big samples whose states, rewards, actions can be significantly different from what the online and target networks have currently learnt, and start diverging due to suddenly big losses. This phenomenon is analogous to a soccer player learning how to kick the ball a certain way well, only to realize there are so many other creative ways to kick the ball well from flashback memory, and destroys its own habits and start over again, but does reach the "master" state soon enough from the memory of success he has experienced before.

Another interesting discussion is with regard to the problem of the spaceship hovering over the landing pad and not coming down to stop on the land. This starts happening somewhere between the initial starting point for training and the "eureka" moment at around 1200 episodes (actually around 700 episodes of training) discussed before. By printing out reward for each time-step and analyzing why the agent may do this gave me the insight that explains it. The way the DQN algorithm works is that it samples minibatch of random samples from the replay memory to learn from. If the replay memory doesn't have the successful +100 rewards landing, then without knowing it, and having bountiful stored experiences of crashing, it is obvious the algorithm will not risk the penalty of -100 points to get very close to the land. However, it is able to learn easily that staying in the region between the flags is good because it gives more positive rewards than staying outside of the region. Also, having less velocity is rewarded so it learns to slow down. But when it reaches right near the landing pad, by slightly moving around in that hovering region, it actually is able to accumulate a bit more positive rewards than the negative rewards over time, even with taking into account the -0.3 points per frame for staying in the air. Hence, until the randomness of epsilon greedy policy action forces the ship to not do the "thrust" action favored by the online network and actually land safely and experience the +100 rewards to be stored into the replay memory, the agent tries to milk the most rewards as it can until the episode finishes right above the landing pad. Once the successful landing experiences

Table 1. Set of parameters that solved the problem

PARAMETER	VALUE
BATCH SIZE	32
LAYER SIZE	64
NUMBER OF HIDDEN LAYERS	2
GAMMA	0.995
LEARNING RATE	0.00025
MAX REPLAY MEMORY	100000
FINAL EXPLORATION FRAME	200000
MAX EPISODES	2500
START TRAINING FRAME	50000
C (UPDATE TARGET NETWORK EVERY C STEPS)	10

are selected as part of the mini-batch, the learning from that single +100 reward is so big that the loss backpropagated to the nodes in the neural network that picked an action different from the "no-thrust" action will get updated immensely (i.e. the "eureka moment"). So the next time a similar situation happens, it will start to try not doing the "thrust" action and over some more episodes, and converge faster. The "hovering" phenomenon is actually crucial step that inevitably happens prior to solving the problem for the lunar lander.

2.3. Results

The resulting graph for the final 100 greedy-policy runs with the trained network (using the parameters in Table 1 is shown in Figure 3. As you can see, it achieves over 250 average points, and solve the lunar lander problem (above 200 is considered solved). It can be seen that no episode experiences negative reward, but it does fall below 200 points from time to time, as the neural network cannot generalize over every single combination of the 8-states (since it is a linear regression of 4 Q-action values from continuous state space), which is infinite. However, I believe that if the agent was trained for longer episodes with a bit more exploration, it may be able to achieve a state where it doesn't fall below 200 points at all for some 100 episode runs. But there is no guarantee that over infinite episodes the total rewards per episode will always stay above 200, due to nature of neural network function "approximator" (cannot be perfect). Also, every possible parts of the code that can be given a seed has been given a fixed value for all runs for reproducibility.

2.4. Hyperparameters

There were quite a lot of hyperparameters for the problem. Other than the ones as noted in Table 1, there are also: number of backpropagation iteration per frame, soft-update rate if soft-update from online network to target network is used, type of activation functions for the neural network, loss function such as huber-loss and its parameters, optimization

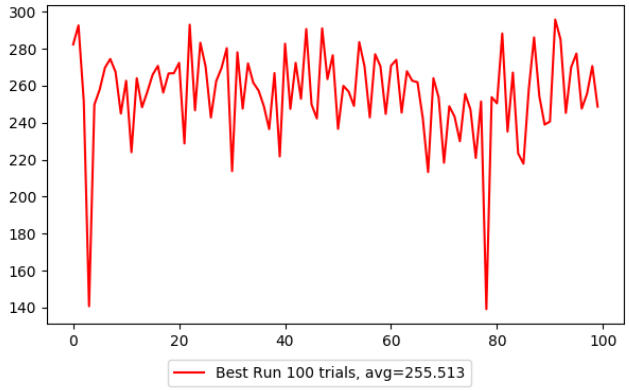

Figure 3. One of the best run results

parameters such as momentum, and so on. Since there are so many hyperparameters, a select few will be discussed for their impacts in this section.

2.4.1. BATCH SIZE

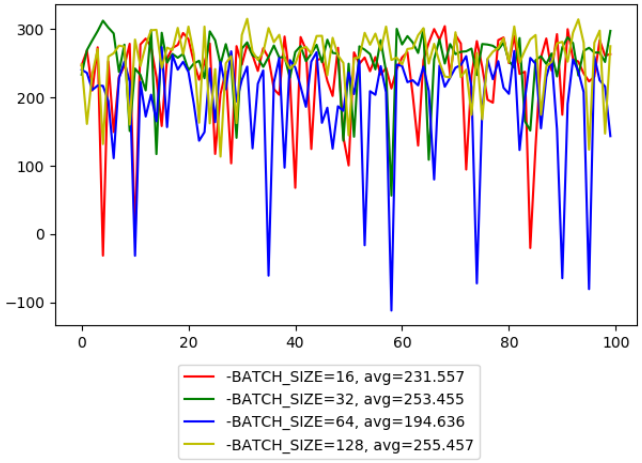

Figure 4. Showing Impacts of Different Batch Sizes

Batch size has an impact of averaging the losses over multiple random experiences, and thus has somewhat similar impact as decreased learning rate in that the updates per frame are softer. However, it is not the same as lower learning-rate. The weight updates will be generally softer, as the NN learning tries to generalize over all samples in the batch, but with lower learning-rate, the updates, while softer, are from smaller sample sizes and the generalization characteristic covered by batch size is not there. As can be seen in Figure 4, there seems to be no definite trend for which batch size is most ideal. Coupled with various impacts of other hyperparameters, it probably is hard to generalize if higher or lower value would be better for lunar lander. Hence, I chose 32 as was chosen in the nature paper 2015.

2.4.2. Layer Size

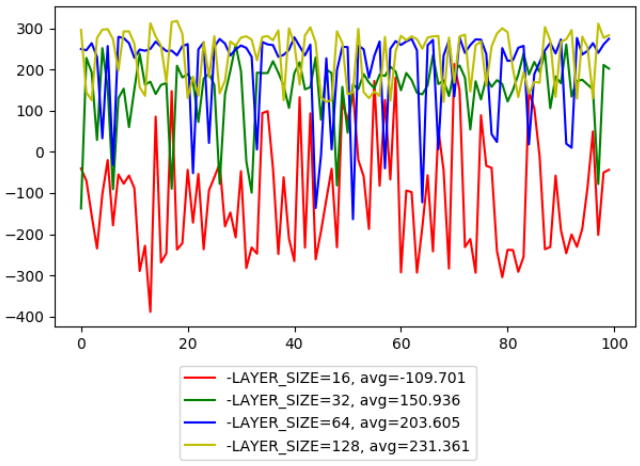

Figure 5. Showing Impacts of Different Layer Sizes

The layer size seemed to have a minor impact, as long as there were enough nodes. One could think higher layer size with non-linearity adders like Relus can help the agent reason more complex situations better, but as known in classification networks, too deep or big networks do perform worse. I believe similar phenomenon can be expected in the reinforcement learning case as well. Either 64 or 128 solved the problem (see Figure 5 for me so I went with 64.

2.4.3. Gamma

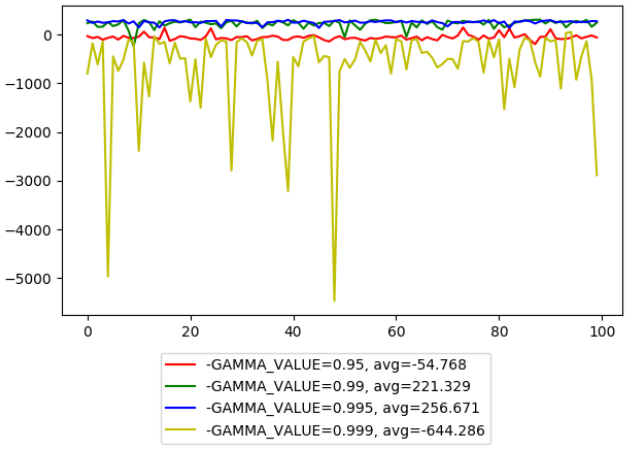

Figure 6. Showing Impacts of Different GAMMA Values

Higher gamma (closer to 1) has the effect of "valuing further future rewards more" due to the structure of TD updates. The higher the value, the more next time-steps' reward matters more recursively. However, looking too far ahead may make the training fall into a sub-optimal solution of hovering in the air forever as it sees too far into the future and the possibility of crashing penalty will overwhelm risking moves that will bring it towards proper convergence. This can be somewhat visualized in Figure 6 where the high value of 0.999 fails to converge within the fixed 2500 maximum episodes. I chose 0.995 as it gave the best performance overall.

2.4.4. Learning Rate

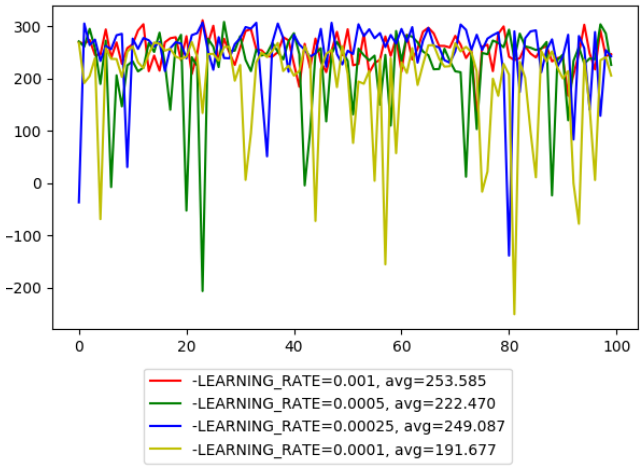

Figure 7. Showing Impacts of Different Learning Rates

Lower learning rate would be a safe approach as it will prevent the network weights to explode and reduce possibility of divergence, but will take longer episodes for convergence. Also, having higher learning-rate can as a side-effect encourage exploration since they could produce more changing Q network weights and thus the action values, but as long as it is not too high (around 0.001 0.0001), it can converge to a solution. As can bee seen in Figure 7, for a fixed max number of episodes of 2500, it is insufficient for 0.0001 learning rate to converge fully. But since all learning rates in the right range provide solutions, I chose the value suggested in 2015.

2.4.5. Weights Copy Frequency (C)

The rate at which online network's weights are copied to the target network has an interesting impact. The idea behind having a C value higher than 1 is that the target network, which produces the labels for the next time-step's states from the replay memory against which the online network's learning will happen, can produce the labels (via bootstrapping) in a more stable manner. The learning part for the online network is a mini-supervised neural network where the network tries to learn to properly map the states to actions, the system that generates the labels shouldn't be moving too much, as it is harder to hit a moving target. However, for lunar lander, the problem is simple enough that like the Cartpole problem (see Section 3.1) which didn't need C

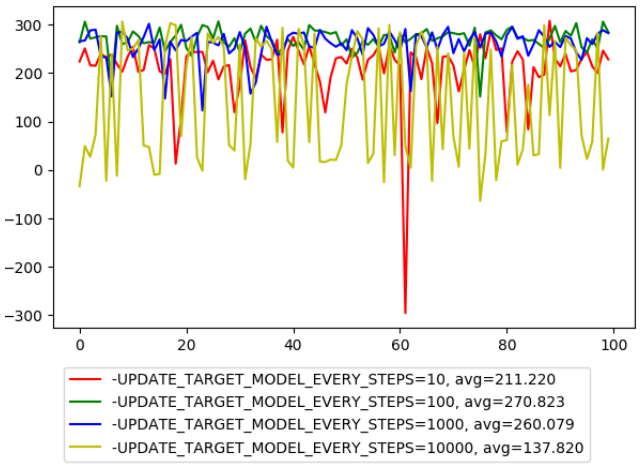

Figure 8. Showing Impacts of Update Frequency C Value

value greater than 1, actually converged faster with lower C values. See Figure 8 for this phenomenon.

3. Experiments

3.1. The Cart-Pole Problem

In order to debug the code and gain further understanding of what exactly are the roles of the target and online networks are and how exactly rewards and losses are determined and back-propagated to update the weights, I tried my algorithm on a simpler problem - The Cartpole problem. The goal is to keep the pole from falling due to gravity, where the only two controls are moving left or right, and the reward is +1 for each frame alive (including the termination step). If the pole's angle is over certain threshold away from the 90degrees, then the game terminates. I learned one crucial lesson from solving this problem, as in the beginning I thought it is weird that the reward is given even at the termination step and considered the problem not solvable without special reward manipulation. However, I came to realization that the Q learning as well as DQN learning learn to output the Q values and takes the argmax of them to decide which action (greedy-policy) to take after training is done. This means that even if there is a 0.001 difference between one action and the other, the one that is 0.001 greater will be favored by the greedy action. And even though the termination step also gives reward +1, there is no reward beyond the termination step (which is 0). This makes the TD learning part of the DQN learn to output values that generates that extra 1-step's +1 reward more over time, and I managed to solve this problem within 50 episodes successfully (average 200 points or above for 100 trials).

3.2. DDQN

I also explored the double DQN described in (Hasselt, 2010) as it is a minor change to the existing DQN algorithm to see its effect. However, it turned out it did not have much of an impact for me when I ran it for 10000 episodes, as it converged around at the same time as DQN. So I did not explore it further.

4. Future Works

If I had more time, I'd like to try the PPO (Schulman et al., 2017) algorithm and other various policy gradients. They are known to have more variance and harder to train, but being able to converge with less episodes, I would like to see how effective the learned solution is. Also, as I am participating in a flying drone racing challenge, I'd like to explore it further, solve the lunar lander problem with PPO, and apply it to the drone racing problem as well.

5. Citations and References

References

Hasselt, H. V. Double q-learning. In Lafferty, J. D., Williams, C. K. I., Shawe-Taylor, J., Zemel, R. S., and Culotta, A. (eds.), *Advances in Neural Information Processing Systems 23*, pp. 2613–2621. Curran Associates, Inc., 2010. URL http://papers.nips.cc/paper/3964-double-q-learning.pdf.

Klimov, O. OpenAI-Lunar Lander, 2019. URL: https://github.com/openai/gym/blob/master/gym/envs/box2d/lunar_lander.py. Last visited on 2019/03/17.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M. A., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. Human-level control through deep reinforcement learning. *Nature*, 518 (7540):529–533, 2015. doi: 10.1038/nature14236. URL https://doi.org/10.1038/nature14236.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347.

Sutton, R. S. and Barto, A. G. *Reinforcement learning: An introduction*. MIT press, 2018.

van Hasselt, H., Doron, Y., Strub, F., Hessel, M., Sonnerat, N., and Modayil, J. Deep reinforcement learning and the deadly triad. *CoRR*, abs/1812.02648, 2018. URL http://arxiv.org/abs/1812.02648.