git commit hash: 3ad5e7348edc5d6a5fbd0c487210706adb1635e7

Project 2: Lunar Lander

Milad Abedi
Georgia Tech
Department of Computer Science
Atlanta, USA
miladabedi@gatech.edu

Abstract—**In this project we take our learning experience from the previous projects to the next level by designing and implementing a Deep Reinforcement Learning agent that will be trained to play the Lunar Lander game. For the implementation of the agent, we have adopted a Double Q-Learning approach.**

I. Introduction

So far in the progress of the course, we have implemented simple RL agents, using representation of value (or action-value) function that ranged in simplicity from a tabular representation to linear functions. We have also learned how to use variations of famous training algorithms such as simple Bellman update, TD learning, etc. While these approaches worked fine in the simpler environments with discrete state-action spaces, their implementation faces significant challenges in environments with higher dimensional, and continuous representations of state-action spaces. Attempting to solve these problems entails a number of upgrades to the simpler approaches used in the past.

Deep Reinforcement Learning: The first upgrade is the use of functional representations for Q (state-action value). While it is technically possible to map a continuous state-action space to a discrete environment of sufficient resolution, in reality this approach would face insurmountable computational challenges in high dimensional spaces as the size of the Q table would grow exponentially in the number of continuous dimensions. However choosing a functional representation of the (action-)value function is a challenge in and of itself. To begin with, the Q function should be such that fitting would be inexpensive, since we expect a large number of updates to be performed during the agent's training period. The Q function should facilitate incremental updates to avoid unnecessarily performing full regression (until convergence) during the earlier steps of the training phase where the stochasticity introduces enough error to render the extreme accuracy of the Q function fitting-operation moot. Most importantly, the chosen Q function should be complex enough to be able to represent the actual shape of the state-action-value space. This task is difficult since we do not have an accurate idea about what the state-action-value space looks like, and as such we are not generally able to design custom models for the Q function for each problem. As such we need a Q function representation with high complexity that can represent a wide variety of state-action-value function shapes. This combination of needs for inexpensive, incremental training and high complexity has driven our choice of Deep Neural Networks for Q function representation.

Experience Replay: The second upgrade to the Q-learning algorithm used in this effort is the use of the Experience Replay approach, where the agents will be able to re-learn from the previous experiences by remembering those interaction every now and then. This is particularly helpful given the fact that neural networks tend to forget about older data points as they see more and more new data. As such, re-learning the previously seen data helps make more efficient use of the prior experiences.

Double-Q learning: The third upgrade to the Q-learning algorithm is the utilization of the Double-Q learning procedure [1] which allows us to get better estimates of the maximum value of Q*, used within the bellman update thereby leading to a more stable algorithm.

By combining all these tools, I was able to design and implement a Reinforcement Learning agent capable of of taming the complexities of the Lunar Lander environment. Moreover, per instructions given in the assignment, we have studied the effects of 3 hyper parameters on the performance of the agent.

II. Methodology

A. Environment: Lunar Lander

The Lunar lander [2] environment, as the name suggests, aims to emulate a spaceship trying to land on the moon surface. In every episode, the agent begins the game with and initial position and speed, presented in the definition of the state, and is allowed to take actions by using (or not using) one of the three available engines (Left,Up,Right). The goal is to land the spaceship in a landing spot (located at (0,0)), and the reward is designed to be an indicator of the distance from ideal spot as well as the success of landing (as opposed to crashing).

B. Deep Neural Network

Anecdotally inspired by the biological mechanisms of the human brain, Artificial Neural Networks have shown themselves to be capable of learning complex relationships between inputs and output(s) despite the simple nature of the underlying calculations [3]. Despite having been established for the last 50 years, Artificial Neural Networks have risen once more from the ashes to dominate the fields of computer-enabled learning. This is in large part thanks to availability of hardware as well as powerful software tools that makes the training of large ANN's feasible. One such software package is the the Tensorflow [4] library which allows for highly efficient calculation of the gradients of the ANN variables, thereby facilitating the gradient descent operation for the training of the neural network. In this paper we have used the Tensorflow [4] library to build a fully connected ANN. You can see the structure of the neural network in Table I:

TABLE I
STRUCTURE OF THE NEURAL NETWORK USED FOR FUNCTION APPROXIMATION OF Q

Layer	Size	Activation
Input	8	None
Dense	128	Relu
Dense	64	Relu
Dense	32	Relu
Dense	4	None

As seen in Table I, the DQN receives as input the representation of state, and outputs an array of size 4, corresponding to the values of 4 actions.

C. Double DQN with experience replay

Like may of our previous efforts, in this project we will use the bellman equation to update our DQN. The following is the famous Bellman Equation for Q learning:

$$Q(s,a) = R_{t+1} + \gamma \cdot max_a(Q(S_{t+1},a))$$

However, some studies [1] have concluded that using the above mentioned equation for DQNs will result in overestimation of the last term, i.e.: $max_a(Q(S_{t+1},a))$. Intuitively, this is the result of the fact that for every action a, the term $Q(S_{t+1},a)$ contains some error, and when we make a policy of picking the maximum of these terms, in expectation we will take terms with positive error more than those with negative error, thus leading into overestimation of the value in question. One solution to this issue is to use two different Q functions , Q^o,Q^t, called Q-online, Q-target respectively. For the Q-online we will calculate the regression targets (Y^o) as follows:

$$Y^o = R_{t+1} + \gamma \cdot (Q^t(S_{t+1}, argmax_a Q^o))$$

For the Q-target we will use the soft update approach as introduced in [5]. Using this approach, Q-target will be gradually updated towards the Q-online, and the speed of this update will be determined by update rate τ. The weights θ^{target} will be updated as follows:

$$\theta^t = (1-\tau)\cdot\theta^t + \tau\cdot\theta^o$$

As the agent interacts with the environment, it will store its interactions and remember the last

20000 interaction. Then the agent will randomly sample 100 interaction from the past and replay them and train the Q functions.

D. Exploration vs Exploitation

During my experience in developing the agent for this project, I noticed that the Exploration-Exploitation balance is very consequential to the success of the agent, and seemingly slight changes resulted in dramatic change in performance. The agent used in this project utilized the ϵ-greedy algorithm, where the agent acts randomly with probability ϵ, and optimally with probability $1 - \epsilon$. I have adopted the following formula for the value of epsilon in episode number t.

$$\epsilon = 10^{\frac{-t}{h}}$$

where h is defined as the training exploration horizon, defined as the episode where the agent will act randomly only 10% of the time. choice of h indicates how many episodes of significant exploration (10% or more random action) should be allowed. For instance, by choosing h to be 500, I indicate that in the first 500 episodes, I am allowing the agent to explore the environment at least 10% of the time.

III. Results and Discussion

A. Evaluation of the RL Agent

In our first experiment, the agent has been lunched into the environment. The configurations for this experiment are as follows:
$\gamma = 0.99$ (discount factor)
$\tau = 0.005$ (Q-target update rate)
$h = 500$ (training exploration horizon see section II-D)

Per instructions provided for the project, we have displayed the total rewards accumulated by the agent during each episode of the training and testing periods in Figure 1. Note that the training ends at episode 500 (red graph in Figure 1), and for the subsequent 100 episodes (blue graph) the agent has been forced to act optimally.

By inserting the values for h=500 and t=500, we can see that the value of epsilon at the end of the training period had was 10%, meaning that the agent was already acting mostly optimally at that point, which explains why the transition from Training to Testing (red and blue graphs) was a smooth one.

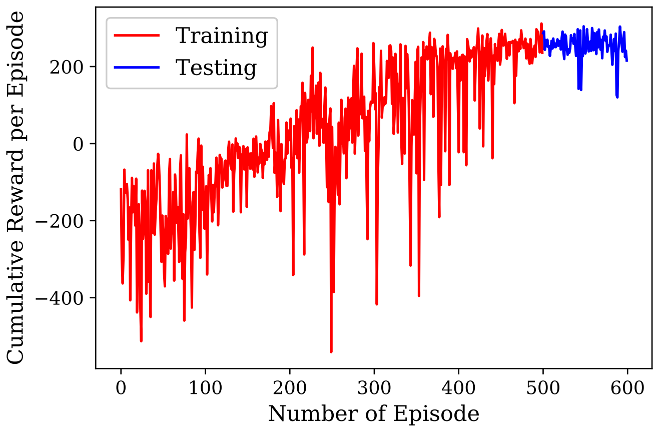

Fig. 1. Total rewards accumulated by the Agent during the training and testing episodes

The instructions for the project explain that the problem is considered solved when achieving a score of 200 points or higher on average over 100 consecutive runs. The average score for the 100 testing episodes at the end of the experiment was 254.7, which indicates that the agent was capable of solving the problem. For further scrutiny, in Figure 1 you can see the histogram of the cumulative rewards for the 100 test episodes following the training.

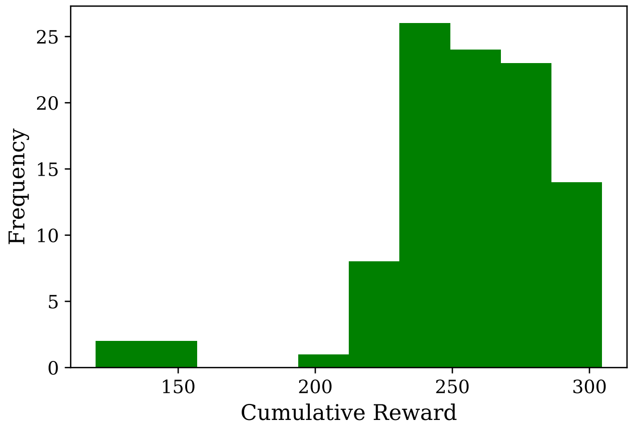

Fig. 2. Histogram of the scores for 100 test episodes with an agent trained for 500 training episodes ($\gamma = 0.99$, $\tau = 0.005$, $h = 500$)

As seen in Figure 2, not only does the agent solve the game by achieving an average score of 200 or more, it does so quite reliably, since in 96 out of the 100 episodes the score was strictly above 200.

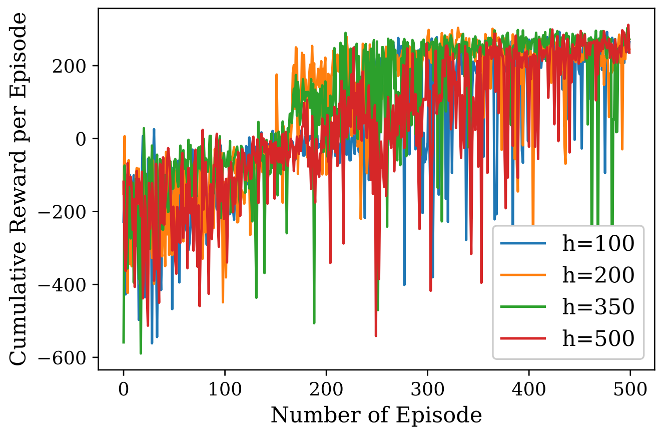

Fig. 3. Effect of exploration horizon (h) parameter, inversely related to epsilon decay on the training ($\gamma = 0.99,\ \tau = 0.005$)

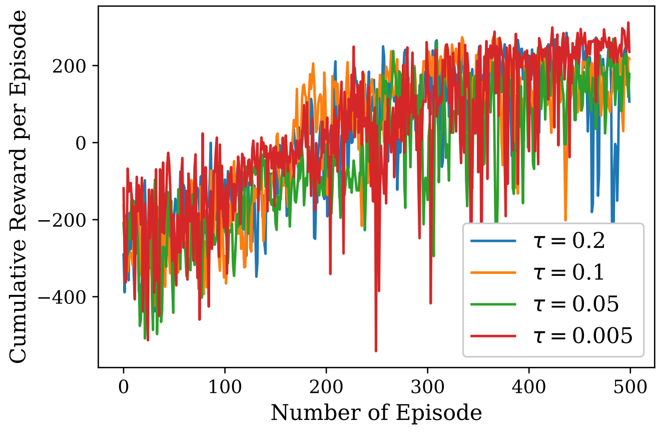

Fig. 4. Effect of Q-target update rate (τ) on the training ($\gamma = 0.99,\ h = 500$)

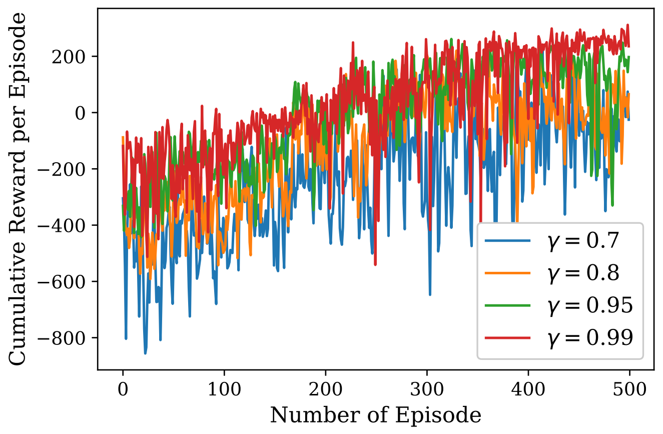

Fig. 5. Effect of discount factor γ on the training ($\tau = 0.005,\ h = 500$)

B. *Evaluation of the effect of Hyper Parameters*

In this subsection we will investigate the effect of three hyper parameters on the performance of the agent. These hyper parameters are: Training exploration horizon (h), Q-target Update Rate (τ), Discount Factor (γ).

As can be seen in Figures 3, 4, and 5, the choice of hyper-parameters can effect the performance of the agent. In all three Figures, the red line represents the experiment that was performed in the previous subsection with ($h = 500,\ \tau = 0.005,\ \gamma = 0.99$), which can serve as a visual benchmark to compare performances across Figures. As you can see in Figure 3, the reduction in the training exploration horizon, negatively impacts the performance of the agent, where after 400 episodes the agent (which is mostly acting greedily) still takes actions which result in very low rewards indicating a crash. This could be explained by the fact that a problem as complex as this one in it's nature, with a Q function representation that has a high number of parameters, relies heavily on adequate exploration of the environment to be able to meaningfully update all the model parameters.

Upon inspecting Figure 4, you can see that choice of tau is also affecting the ability of the agent to learn effectively. Smaller values of τ seem to work better. This make a lot of sense, since the entire reason we chose to use a soft update scheme for updating Q-target was to make sure it retains a certain amount of temporal distance from Q-online. Choosing large values for τ brings Q-target and Q-online closer together, thereby resulting in overestimation of Q(S,A)

Similar to the previous two cases, Figure 5 indicates that γ can affect the ability of the agent to learn. Smaller values for γ (around 0.99) seem to do much better than larger values. This is particularly true since we are measuring the success of our algorithm by summing up all the rewards (as if the γ was equal to 1), thus it is not surprising to see that choice of smaller discount factors shows a decrease in effectiveness with the chosen measure of success. More intuitively, choosing small values of γ, tells the algorithm that the taking a bunch of steps to successfully land and receive a bonus is not much different from just hovering for over 1000 steps and terminating the game, since the promised reward is too far in the future to care.

IV. Lessons Learned

A. Algorithms used

In my first try, I used a Gaussian Regression Process as my Q function, but the computational load happened to be unmanageable, mostly due to the non-parametric nature of the model. The size of the underlying kernel was growing in n^2 (n: number of data points) and iterations of the Bellman equation were growing more and more expensive.

Then I turned to DQN, without using double networks and I was not successful in reaching total rewards beyond 60-70.

The Double Q-Learning was the only method among my trials that actually worked, and was able to reach total rewards beyond 200.

B. Pitfalls

To be able to implement the algorithm more effectively, it would help if you had some experience in choosing good hyper parameters during the development phase. I had to do a lot of computationally expensive try and error to arrive at the right answer. I started by using a simple copy operation to update the Q-target but the problem with that approach was that every time the weights got updates, the Q-target and Q-online would be very similar for a great length of time, especially if the learning rate of the underlying Tensorflow optimizer was small. Soft updating the weights helped me avoid this problem all together.

One of the biggest problems I had related to the logistics of the implementation, where the newer version of the Tensorflow library was very slow (issues with eager implementation). I found this much later in the process and as a result I was not able to do as much as I wanted to.

C. Plans for the future

One thing that I really wanted to do is to make the exploration more efficient. I tried to do it by introducing a rarity factor that would have explored less seen state-action pairs more frequently than those seen before. I tried to do this by fitting a Multivariate Gaussian Distribution over the State-Action space, but my efforts were not successful and I did not have enough time to fix the problems.

References

[1] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *CoRR*, vol. abs/1509.06461, 2015. [Online]. Available: http://arxiv.org/abs/1509.06461

[2] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[3] M. H. Hassoun *et al.*, *Fundamentals of artificial neural networks*. MIT press, 1995.

[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[5] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.