Reinforcement Learning in Continuous State Spaces

Hermann Hans
Department of Computer Science, Georgia Institute of Technology

March 20, 2017

Abstract

This report covers our experiments in reinforcement learning to solve continuous state space environments. The OpenAI Gym[1] Lunar Lander environment was used as the backdrop for our implementation and subsequent experiments. We cover the algorithms that were used to, as well as experiments we ran and results we got from them. A video with a quick overview of the reconstruction of these findings is available at `https://www.youtube.com/watch?v=mvSf2gF3NIE`

1 Introduction

Reinforcement learning systems learn to control agents through trial and error, trading off between exploration and exploitation to optimize their control strategy. In this report we implement one such system and attempt to solve the Lunar Lander environment as made available on the Open AI Gym platform.

2 Environment

Lunar Lander is a simplistic game where the goal is to try to land a spaceship in a given goal area using different thrust vectors. Unlike the environments used in [4] where the agent had to train on the actual images presented as pixels, the lunar lander environment provides an observation vector S of internal state information. The vector S consists of the x and y coordinates, the x and y velocities, angle, angular velocity, and ground contact information of the lander. Six of the eight variables are continuous, with only the last two being discrete (0 or 1). What's important for our agent though is only the size of the vector S meaning we can generally ignore the contents and what each variable represents.

There are four actions available to control the lander at each state: do nothing, left thruster, main thruster, right thruster. The landing pad is always located in the same location (0, 0). Rewards are given for landing within the marked area and having zero speed. Crashing results in a large negative reward (-100) and using the main thruster results in -0.3 reward at each state it's used. Finally, leg ground contact results in a +10 reward.

3 Reinforcement Learning in a Continuous State Space

For our reinforcement learning agent we used Q-learning to learn the control policy for the lunar lander environment. In Q-Learning we try to maximize a scalar value by interacting with the environment. The simple version for discrete state spaced environments of the Q-learning update rule is shown in [1].

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

While [1] works great for discrete MDPs such as gridworlds and the like, it's not feasible to use it for any sort of continuous environment. In order to move from discrete to continuous state spaces,

function approximation methods are used. To do this, our $Q(s, a)$ function becomes $Q(s, a, \theta)$, where θ represents the weights which are used in the neural network. Likewise, instead of updating our table entries we update the weights (θ) in our neural network as part of the learning procedure. Updating θ is then done using a loss function which is beyond the scope of this report, but can easily be read up on in [4].

4 Implementation and Difficulties

The first and biggest difficulty was choosing an implementation that would work. The simplest idea was to attempt to discretize the continuous state space using bins of various sizes. Since Q-learning for discrete state spaces is fairly trivial to implement, this solution seemed like the quickest way forward. Unfortunately this solution is rather impractical. Using bin sizes that are too small results in aliasing, where different raw observations map to the same state. Using a bigger bin size in turn results in a state space so big that the amount of episodes required to visit all states often enough is no longer trivial.

Tile coding as described in [5] was another option that was explored, but eventually dropped in favor of using a Neural Network to do the function approximation. While it seems like our chosen environment should work with Tile Coding (see [3]), the interest in reproducing at least some parts of [4] was large enough to go with the neural network approach.

While Keras [2] simplified the code for setting up a neural network a great deal, the difficulty now became in finding a set of hyperparameters and additional implementations (replay memory, target network, warmup phase) which would allow us to solve lunar lander. The biggest difficulty, once the code was seemingly bug free, was getting stability into the system. A lot of runs our agent seemed to learn well before the network imploded and started to diverge drastically. Some of this may be related to what is called "catastrophic forgetting" ([4]) and to counter it we implemented a replay memory and a target network, which in turn led to more hyperparameters to tune (memory size, update interval of target network). Both of these helped with the stability, but unfortunately we couldn't repeatably solve lunar lander with our final parameters.

Finally, the amount of time required for training and having a deadline didn't allow for as much fine tuning in the end. Most runs took several hours to complete, even after compiling tensorflow from source to take advantage of some CPU flags.

5 Experiments and Results

The final implementation used a lot of the ideas from [4]. The neural network consisted of 2 hidden layers, an input layer, and an output layer. The layer sizes were set to 64 neurons with uniform initializers and rectifier activation functions, except for the output layer which used linear activation. Mean squared error was used the loss function and RMSprop was the optimizer. We did play around with the network initially, and there's a lot of potential to further tune it. To get to the settings used above, we used a trial and error approach on the Cartpole environment. Once we saw consistent results there, we kept this model only modifying the input and output layer dimensions to adjust for the input state and output actions of lunar lander.

Figure 1 shows the training and trial (100-epsiodes, no learning, full greedy) runs of our final implementation. We always started out with $\epsilon = 1.0$ and decay between .95 and .99. The final implementation used .99 to allow for more exploration initially. After realizing the importance of having the agent see some successful landings during training, a slightly modified implementation was used to run a "warm-up" until a positive total reward was seen. Once this experience was observed, we filled the entire replay memory with it and only then started the learning process. While an even higher $\epsilon - decay$ value would've had a similar effect, the training time would've increased even more.

A partial hyperparamter tuning was done and can be seen in Figure 2. What is labeled as Set 1 used $\gamma = .99, \epsilon - decay = .99$. All the other sets used variations slight variations to play with the

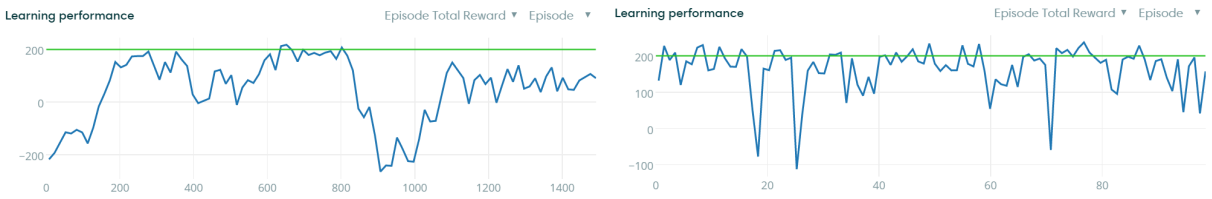

Figure 1: Training [LEFT] and 100 episode trial [RIGHT]

initial exploration rate and discount factor for reward, but like mentioned above, the amount of time required for training limited our parameter tuning a great deal. The most positive effect was seen in Set 1 by tuning the interval of when the target network was copied (2000 steps) which we believe could've been increased even more to great effect. Replay memory size was also changed from 10000 to 100000 experiences, but the effect wasn't tracked properly.

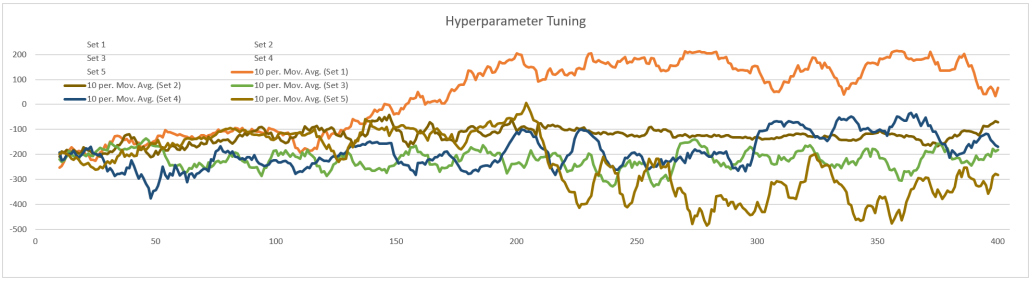

Figure 2: Comparison of training runs with varying epsilon decay and gamma

6 Conclusion

While an initial function approximated Q-learning agent was quickly implemented, we quickly saw that there is a lot of tuning required to get a stable network implemented. Most of the runs and difficulties we observed were due to the extremely long training times and the instability of the network. There's still a lot of potential left in not only parameter tuning, but also in code optimization which could very likely speed up the entire implementation. A few things we didn't have time to explore were the use of masking for the loss function for example, which is likely to have been a cause for error in our implementation. In summary, this project was a great learning experience with an opportunity to use some of the newer ideas in the reinforcement learning setting.

References

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[2] François Chollet. Keras. `https://github.com/fchollet/keras`, 2015.

[3] James MacGlashan. Brown-umbc reinforcement learning and planning.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[5] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.