RL: Lunar Lander

Andrew Cassidy, *acassidy6@gatech.edu, Georgia Institute of Technology*

Lunar lander is an Atari video game classic where the goal is to land a space ship in between two flags. In this report I will cover learning to play the game using Q-Learning and a neural network for value approximation. I will also cover the various experiments I ran which include varying the ϵ-greedy exploration strategy, the discount factor, and the number of bootstrapping training cycles.

Lunar Lander

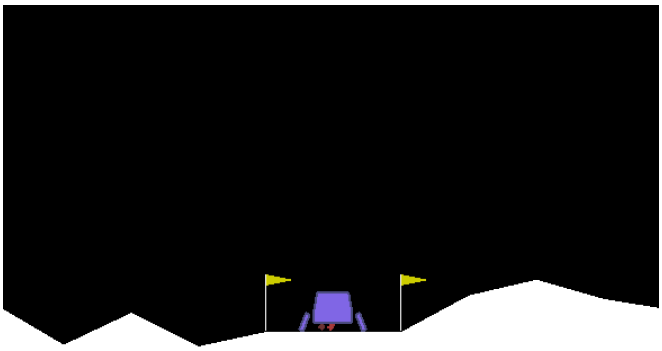

Figure 1: *A successful landing*

In reinforcement learning (RL) the goal is to learn a policy of actions in an environment so as to maximize some notion of cumulative reward. In lunar lander, positive reward is gained for a successful landing and negative reward for crashing or using your engines. An important point is that any reinforcement learning algorithm a-priori does not know that the goal is to explicitly land in between the flags. Instead it only receives the observation vector and the reward.

Observation Vector

The observation vector keeps track of the space ship. It is a vector of length 8 that contains $[x, y, v_x, v_y, angle, v_{angle}]$ as well as a boolean indicator of whether the left and right leg are touching the ground. Ultimately the goal of a RL algorithm is to map any observation vector to an action that in the long term maximizes reward. For the rest of this paper, when we refer to a state we are referring to this observation vector.

Reward

Whenever the space ship takes an action (firing left, right, main, or no engine) it receives a reward from the environment. This action also causes a transition to a new state. Even though lunar lander does not have stochastic transitions, our RL algorithm has no explicit knowledge of the next state or the reward that will be recieved. Instead, we learn from a set of historic observations.

Basis for Learning

The basis for our RL algorithm is a set of environmental observations of the form $[s_t, a_t, r, s_{t+1}]$ where s_t is the state (observation vector), a_t is the action taken in state s_t, r is the reward received from the environment, and state s_{t+1} is the state we transitioned to.

Q-Learning

Q-Learning is an algorithm that makes use of the learning tuple covered in the last section. $Q(s, a)$ is an estimate of the discounted value for taking action a in state s and following an optimal policy after that. Q-Learning learns from experience and continually updates its estimates based on new observations. Typically, Q-Learning, given a new observation $[s_t, a_t, r, s_{t+1}]$, updates its estimates using the formula: $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big(r + \gamma max_a Q(s_{t+1}, a)\Big)$. A huge caveat of this project is that the observation space for LunarLander is continuous. Therefore, there are an infinite number of states. By default, Q-Learning does not work out of the box because for any given incoming observation, with probability 1, we will have never see that state before. To overcome this short coming there are 2 options: 1) discretize the state space and 2) approximate the value of a state action pair using historic observations. Next I will cover value approximation. A detailed discussion of 1) is in the appendix.

Value Approximation

Value approximation is a more robust strategy then discretizing the state space. The idea is, given a set of historical observations, make a model of the value of a state action pair. Then for a new state pick the action that the model estimates will produce the most long term discounted reward. The Deep Q-learning [1] team proposed a similar solution for the general playing Atari video games. In order to make a model, the team kept a historic observation set they called replay memory and sampled mini-batches from relay memory to create a model. This was the approach I used in my implementation.

A similar approach was discussed in lecture and the term bootstrapping was used repeatably. For the first iteration of value approximation, the value of any state is predicted to be 0 (the initial bootstrapped value). For the second bootstrapped value we make use of the stored observation vector: $[s_t, a_t, r, s_{t+1}]$. We use our old model and predict that $Q(s_{t+1}, a) = 0$ for all a. Next we substitute this value into the Q-equation. $Q(s_t, a) = r + \gamma max_a Q(s_{t+1}, a) = r + 0$. Next we train a new model on the tuple $(s_t, a, Q(s_t, a))$ where s_t and a our are independent variables and $Q(s_t, a)$ is our dependent variable.

There are 2 ideas going on here: 1) that we feed more and more real reward into our system and refine our estimates of $Q(s, a)$ in a similar way to how value iterations for MDPs starts with an arbitrary value estimate and refines it given true rewards 2) that our model will find some structure in s_t and a that is predictive of $Q(s_t, a)$. This "structure" is a surrogate for the discretized state space.

Why is neural-network approximation better? Neural networks are extremely powerful machine learning algorithms that with hidden layers can in theory approximate any function. The idea is that instead of having a discretized state space, the neural network can be thought of as creating a value estimate over the entire 10-dimensional space.

We continually bootstrap a model while performing more actions in our environment. The idea is that our model will provide a basis for making decisions that lead to long term reward. These decisions and their consequences are stored in replay memory and the model can learn whether they were successful or not.

The last two paragraphs described the bootstrapping process, but left out how the model is used to make decisions that lead to long term reward. In the decision making process, when we are presented with s_t, we query our model and get an estimate for the 4 different actions $Q(s_t, left)$, $Q(s_t, right)$, $Q(s_t, main)$, and $Q(s_t, nothing)$ and probabilistically (with 1-ϵ probability) pick the action associated with the maximum Q-estimate.

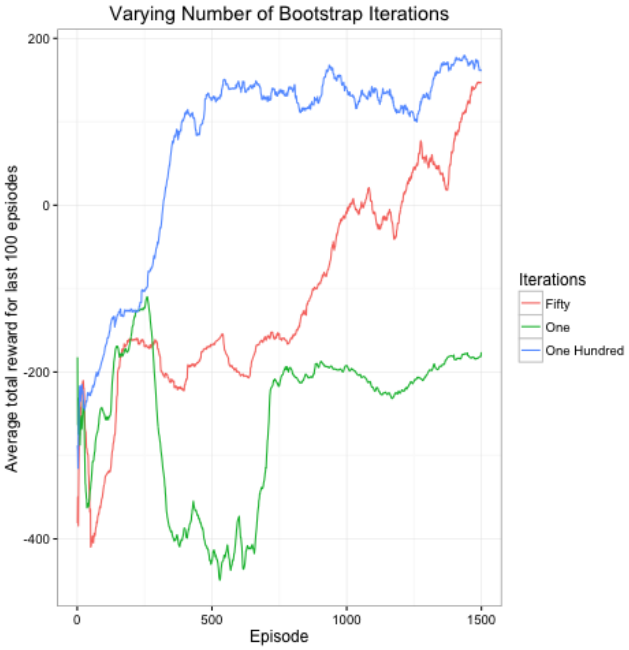

Figure 2: *Varying the number of bootstrap training iterations performed after each episode*

Experiments

For all the experiments I chose 3 different hyper-parameter values. The specific values where chosen to cover as large as range as possible. A more robust exploration of hyper-parameter is ideal, but costly in terms of computational time. By picking a low, medium, and high value for each hyper-parameter I hoped to get a general idea of the effect on the ability to learn.

Varying amount of bootstrapping

The amount of bootstrapping had a huge effect on learning. After each learning episode, I trained a neural network to predict Q-values. I tried training 1, 50, and 100 times after each episode. All other parameters where kept constant. The results in figure 2 show that the number of training iterations performed after each episode had a huge effect on how quickly Q-Learning learned with 100 training iterations greatly increasing the ability to learn. Why are multiple iterations of bootstrapping so important? The experiment in figure 3 shows why.

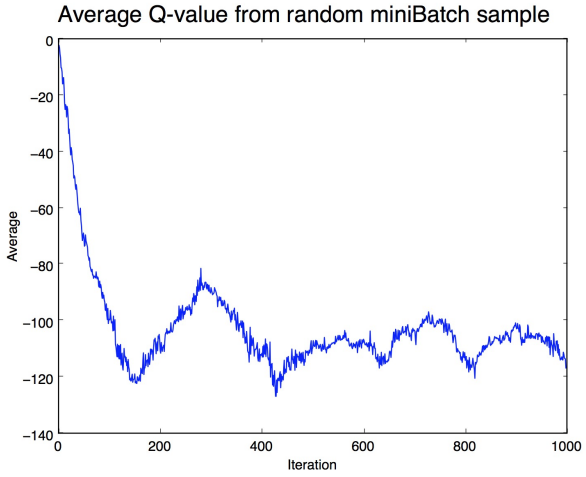

Figure 3: *The average max Q-value from 100 episodes of random play*

I also ran a small experiment to analyze convergence of bootstrapping. I ran 100 episodes of LunarLander where an agent played completely randomly and stored the tuples: $[s_t, a_t, r, s_{t+1}]$. Next I trained a regression algorithm on random mini-batches from the stored tuples. After each training iteration, I calculated the mean Q-estimate for each state using the optimal action. Figure 2 displays my results. It can be seen that in random play, the best estimates of reward are about -100 which co-insides with simply crashing the spaceship and doing nothing. Of interests, however, is that it takes approximately 100 iterations before the average value estimates converge and cycle around -100. This provides further support for the results showing LunarLander learning best with 100 training iterations for each episode of play.

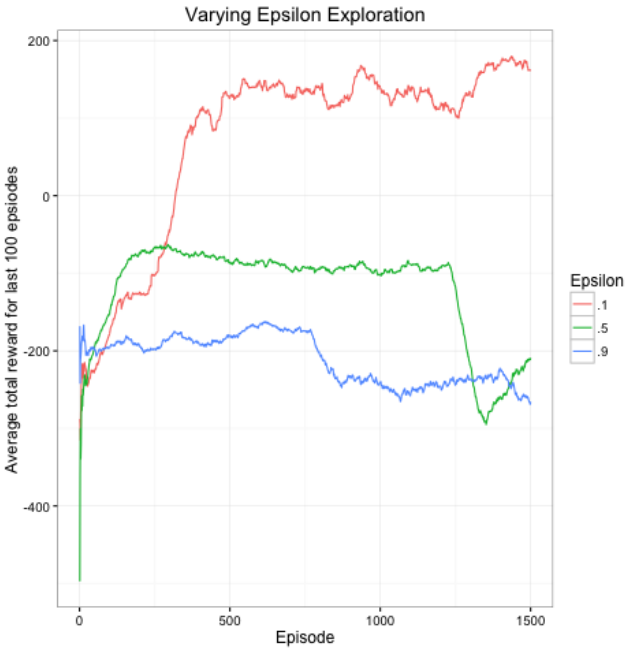

Figure 4: *Varying the epsilon for exploration*

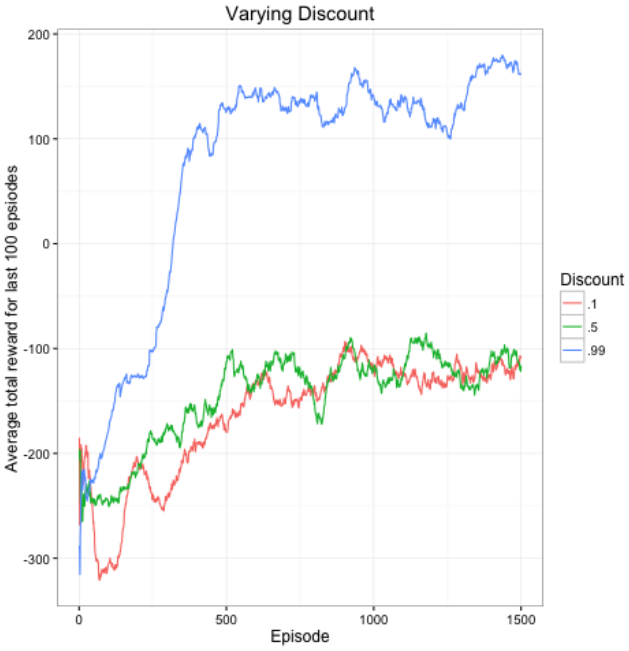

Figure 5: *Varying the discount of rewards*

Varying exploration (ϵ)

Figure 4 shows a simple experiment varying the ϵ used for the ϵ greedy exploration scheme. It can be seen that too much exploration greatly reduces learning. Also of important note is that $\epsilon = .1$ never converges to an average reward of 200. My hypothesis is that if I decayed ϵ instead of keeping it constant we might see Q-Learning converge to a reward of 200+. Additionally, I may want to freeze the training of the neural network at later learning episodes because enough bootstrapping and exploration have occurred to get a robust model.

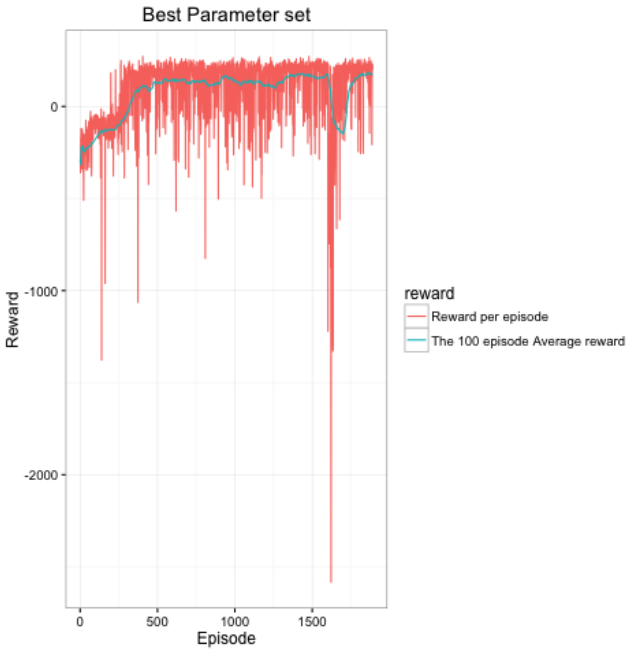

Figure 6: *The 100 episode average reward and the reward per episode for my best parameter set*

Varying discount (γ)

Figure 5 shows a simple experiment varying the discount (γ) used for the reward. Any discount besides 0.99 had a very poor learning ability. This is likely because there are big rewards only at the end an episode. Thus, with such a large discount on future rewards, the agent is never willing to perform actions (use engines for example) that will move the agent closer to successfully landing the space ship. This is highlighted in figure 5 because discounts of 0.5 and 0.1 both seem to converge around an average reward of about -100, meaning they learned to do nothing "now" and just get -100 at the end of an episode instead of losing reward "now" and getting +100 at the end of an episode.

Best Parameter Set Results

The results in figure 6 show my reward per episode and my 100 episode average using the best parameters determined in the previous experiments. As I mentioned before, my average reward does not converge to 200+ and there is a large amount of variation in later learning episodes (with a huge deviation around episode 1600). I will discuss attempts to correct this behavior by lowering the neural network learning rate, lowering the ϵ exploration, decreasing the neural network bootstrapping, and increasing the size of the miniBatch with which the neural network is trained on in my presentation.

Conclusion

This project was very challenging and a lot of fun. I always wondered how the deep mind team learned to play alphaGo and Atari games and now I do. The largest point I want to make is that by using value approximation we can take Reinforcement Learning into problem spaces where states are represented as continuous values. Furthermore, the most important tuning point for me was to perform a lot of bootstrapping iterations, not explore too much, and not discount the future too much. If I could further work on this project, I would implement a decaying ϵ exploration strategy. Also of interest would be to feed Q-Learning games played by humans in a similar fashion to how alphaGo learned from humans actually playing games. Q-learning could quickly learn how humans play, but also explore new strategies.

References

[1] 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.

Appendix

Discretize the State Space

One option for this project is to snap the state to a particular bin. For example, say we have an ob-

servation: $[-0.0053834, ...]$). For the x-value (first value in the vector), we may chose a bin size of 0.001. So, -0.0053834 falls between -0.005 and -0.006 so it would fall into that bin. There would be a bin for each dimension in the 10-tuple observation. The idea is to group similar points and come up with an action that works best for all those points. The Q-value estimates for this bin action pair would be updated whenever an state action pair snaps to the bin.

One of the potential pitfalls of this approach, however, is that bin size is a user decision. Thus, we might choose a bin size that is not appropriate for grouping the different states. For example, say we had too large of a bin near the flags and for a large range of v_y. This area is a spot where very particular actions can have a great effect on whether we crash or not. For example, in some of these cases we want the lunar to slow down (fire main engine)and in some we want it to do nothing, but since we chose too large of a bin for v_y, we are unable to distinguish when we are going to fast and when we are going the right speed.