dgiron3

Project 2: Lunar Lander

The Lunar Lander problem is a very simple game. The task, in short, is to successfully land a spacecraft with various random initial state variable in between a targeted zone without crashing. The state variables consist of eight variables (x position, y position, x velocity, y velocity, angle, angular velocity, and two Boolean variables). Using reinforcement learning, the test is set up to receive various rewards (positive/negative) depending on the environment’s state-action path.

What makes this problem difficult compared to other openai gym problems is that the state space is continuous, meaning that there are exponentially more states a Lunar Lander could be in as opposed to the Taxi problem (only 500 possible states) we worked on in homework 4. Because of this, implementing a simple Q learning algorithm is not possible. While one could hypothetically try to implement a Q learning algorithm with a discrete state space, it makes more sense intuitively to find other possible strategies.

After reading the Atari DeepMind paper which focused on a Q-learning algorithm variant called “Deep Q-Learning with Experience Replay”, it made sense to try and successfully build such an algorithm to help solve the problem and pass the Lunar Lander test. While Q-learning algorithm acts as a table lookup function to find and update the Q values structured to pick the action that maximizes reward, the Deep Q-Learning Algorithm utilizes a Machine (Supervised) learning neural networks. The idea is that the NN works as a function approximator. After keeping track of its most recent experiences and given new experiences, it can use a trained neural network that takes the state space as an input and outputs the action that is expected to give the highest reward. Then after each iteration, it can store the new experiences into the dataset, and update the neural network model simultaneously. So in a sense, Q-learning works very similar, except the function approximator is just a table lookup, and after each action, the Q values in the table are being updated according to the Bellman equation. Deep Q-learning (with Experience Replay) uses an ever-training neural network as the function approximator. This allows the agent to learn even in a continuous/infinite state space.

As one might guess, I especially found this task of finding an agent that will solve the problem much more time consuming than the taxi problem. So in terms of saving time with my model and learner algorithm, I opted to attempt to solve a smaller (yet similar) problem called the CartPole problem (also in openai gym). Instead of 8 state space inputs, it has 4 and instead of 4 actions it only has 2. This made the already lengthy trial and error process much shorter. Fortunately I was able to successfully solve the CartPole problem after 162 episodes using the experience replay algorithm (<https://gym.openai.com/evaluations/eval_UD02m2RRkuc3hIlJ5WDugs)>. I was also able to to save the weights and have the agent solve the problem in a much faster timeframe (<https://gym.openai.com/evaluations/eval_33Oi50v5Sfmd7DRA81D8BA)>. Encouraged by this, I tried to switch these results over to the Lunar Lander problem with the aid of proper hyperparameter adjustments. This is where I encountered major pitfalls.

My performance with the Lunar Lander was not ideal. Using a single-layered neural network with 83 neurons, I found that not only did the training take much longer for each episode, I also found that this problem in particular required much more exploration than the CartPole problem. Before discovering this, I was unable to get hardly any positive total rewards after an episode. While my training method is not ideal, I decided to incrementally train my DQN agent while saving the weights after every run. (Also, I was having trouble with the video recorder when uploading the agent to openai. This caused many of my runs to finish shorter than desired. For this reason, I had to upload multiple times, and on the final run, I saved the rewards per episode and created my own plot). The training results are below:

<https://gym.openai.com/evaluations/eval_DHnja3TTvub63Gvcz1Dsw>

<https://gym.openai.com/evaluations/eval_UoGi8S4BRTeh8taq5O76fQ>

<https://gym.openai.com/evaluations/eval_sYX3jywfRziMtQmeXUAzdQ>

<https://gym.openai.com/evaluations/eval_gm3hlEZRl234obzgfYTGg>

<https://gym.openai.com/evaluations/eval_DHnja3TTvub63Gvcz1Dsw>

<https://gym.openai.com/evaluations/eval_Pv2XvpwOTZSZ94VTefLjhA>

<https://gym.openai.com/evaluations/eval_Ex0KWAhlTcyEldhRtt5yVA>

<https://gym.openai.com/evaluations/eval_PHVpKSVxRoeO4ZY9a5flYg>

First off, I wanted the lander to only explore so for over 1000 episodes, I had the lander randomly act and record the data. The next 7 or 8 runs, I used an epsilon decay function:

episolon = epsMin + (epsMax - epsMin) ^ (-epsDecay \* totalNumberOfSteps)

What this allowed me to do was specify a starting epsilon value and an ending epsilon value, and slowly decay the exploration of my agent from max to min based on the decay rate. Typically I kept this value really low (0.0001 or even 0.00001) to make sure it had plenty of episodes to explore. As my rewards started to increase over time as my weights updated, I tended to start with less exploration (0.8, 0.5, 0.3, etc) and/or I had the decay rate increased (0.001 or 0.01). Also, some runs I put the decay rate at 0 just to focus on a desired exploration parameter. Once I felt my agent had been trained enough. I kept the saved weights, and turned off the video recorder. I set the epsilon max to start as 1, the decay rate to 0.0001, and the epsilon min to 0.01. Up until this point I was unable to get any results that I considered consistent, or desirable. To my surprise, my brute force training methodology was able to output somewhat desirable results (shown in previous page). As you can see, even after 1000s of iterations held before, the weights and learner still needed an adjustment period to achieve results close to the solution. After 500 or 600 episodes, the Lunar Lander was able to almost consistently achieve rewards over 200 with the exception of a small amount of noisy outputs.

Unfortunately, I was not able to reproduce exciting results after setting epsilon to 0, even with my trained agent with weights carried over from the previous runs. After looking at fellow classmates runs in openai, and seeing that they were able to show how well their models learned, it made me wish I had more time to explore more options. Time was the deciding factor for me unable to accomplish this goal. Combining Q-learning and neural networks, made for an extremely wide variety of parameters to tune. The past two semesters, I have learned from ML and RL is that much of “learning” tools is just parameter tuning and understanding what various parameters do. For example, epsilon was my favorite parameter to tune. It was the difference between exploration and exploitation. They are both very important to a learning and performance and thus, there is fine balance that must be found when training with a Q learning algorithm. Another crucial determinism to find was the NN setup (how many layers, what activation function, how many neurons, etc). I simply was unable to dive deep into this parameters, but I am sure that there could be some adjustments to my NN setup. I assumed that since my CartPole setup worked, and that this project wasn’t to explore NNs, that I should just adjust the Density of the NN and move onto other parameter adjustments. I did find, that for these particular problems, my best results with the loss function of the NN were found when I set the learning rate to a very low value (0.00025). Any higher and my results were not as consistent(see below). In conclusion, I cannot say that this case has been closed. There is much more to explore and many more angles to take. On the bright side, this is the beauty of this subject. Machine Learning and Reinforcement learning are so powerful, and yet still very young in terms of discovery. I look forward to studying more efficient techniques (already discovered/yet to be discovered) so that I can learn how to build a more efficient agent with more parameter optimization knowledge.