COMP 450: Project 5

Raymond Yuan

November 20, 2018

1 Problem

In this project I solved a continuous motion planning problem of manipulating a robot arm with 4 joints. The problem was manipulating Fetch's end effector to reach a randomly chosen goal in 3D as quickly as possible.

2 Methodology and Approach

I approached this problem with a vanilla actor-critic (AC) model attempting to predict the torque applied to each of the four joints of Fetch in order to move the end effector toward the goal. Actor-Critic algorithms represent a policy function independently of the value function. The actor will predict a policy for the given state and the critic will produce a temporal difference error signal given the state and reward. The value network will drive the learning in both the actor and critic models.

I began by implementing a basic actor-critic model to solve the problem of Continuous Mountain Car, an environment where an underpowered car must climb a hill by applying some engine force (continuous space). I did so by creating a linear multi-layer perceptron that predicted policy by sampling from a Gaussian distribution and the value. I also implemented a preprocessing pipeline that utilized Radial Basis Function (RBF) kernels to project the raw observation to a higher dimensional feature space.

Once I had this working I ported the algorithm to the more complex problem of solving FetchReach. I moved from the 1-D case to the 4-D case by having one agent find a multivariate normal distribution of a policy. There are several issues with this approach: this increases the number of values that must be learned to have n continuous outputs from $2n$ (predicting a mean and standard deviation for each action) to $n + n^2$ (n means and an $n \times n$ co-variance matrix. This increases the complexity of the model required to solve this problem. I had trouble getting this model with the vanilla AC learning algorithm to converge. In attempts to solve the problem, I iterated on different model types, altering hyperparameters (depth, number of hidden units, learning rates, etc.), trying to solve the model with a dense reward signal, writing a custom reward function, projecting the raw data using the RBF kernels, and performing gridsearch over

all these permutations. In addition, I tried separating each agent to learn its own 1D mean and standard deviation for each continuous action, as well as creating actor critic agents share low levle features and have them braach of into their own policy/value functions. Despite this, the algorithm performed only slightly better than the random action baseline that I implemented.

I believe that the vanilla AC algorithm failed to converge primarily because without experience replay, there's enormous variance in the value approximation (especially since I was only predicting value for one step in the future). This can cause the error signal to compound the variance in the bad predictions. Experience replay is able to break up the temporal correlations across training episodes. In addition, apparently updating the actor and critic models with the gradient obtained from the temporal difference error signal may cause the algorithm to diverge.

I then moved to a different AC algorithm known as Deep Deterministic Policy Gradients (DDPG). DDPG is a policy gradient algorithm that uses a stochastic behavior policy for exploration, while still estimating a deterministic target policy which makes convergence much simpler. Initially, I could not get DDPG to converge using a naive experience replay buffer, storing the raw state, action, and reward. By processing raw state observation by concatenating the observations and the achieved goals as well as the observations with the desired goal allowed the algorithm to converge faster.

3 Results

Ultimately, I was able to get the algorithm to converge. It solves the problem with a 99% success rate over 100 episodes. In comparison, sampling random actions achieves a success rate of about 20%. The random algorithm has an average solve time of about the number of steps per episode (each step propagates 20 substeps). On the other hand, DDPG is able to solve the problem in about 2 steps of simulation (again, with each step propagating 20 substeps in simulation).