Solving Sawyer Lift with RL

Abinavraj Ganesh Sudhakar

January 11, 2021

0.0.1 Problem Description

To solve a problem such as Sawyer Lift, it is important to consider the type of action space and observation space with which we are working with. In the case in which we are given numerical values by a simulated environment describing robot joint and object position. The robosuite environment actually gives a total of 32 input features to describe these physical states. Each of these 32 values were not constrained to a domain of values and had infinite potential values making the observation space a continuous observation space. As for the action space, Sawyer Lift has 8 degrees of freedom constrained by values from -1 to 1 with infinite potential values. Thus, the action space of this problem is also continuous. Additionally, I decided not to use HER since this was not a goal-based environment in which we were given a goal state in the beginning. Additionally, this provided stability for both on policy and off policy algorithms that desire reward-shaping.

0.0.2 Considerations

Given that the output data are all continuous state spaces we can no longer perform discrete action space algorithms such as Deep Q Networks or Q Tables. Additionally, the output of our policy networks must be either deterministic or stochastic with distribution parameters as the output (such as the mean and standard deviation of a Gaussian distribution.) With the limited time and resources that I had, I could not pursue imitation learning. Imitation learning techniques such as Generative Adversarial Imitation Learning (semi-supervised) or even supervised learning often require expert data to learn from. Robosuite allows for keyboard input, but a human input would take too much time and effort to train the network. As a result, the following algorithms could be applied to the following problem: DDPG, PPO, TD3, TRPO, A2C, and D4PG.

0.0.3 Testing Environment

Mujoco-py and robosuite is not compatible with Windows, so as a result I had to install Ubuntu Linux 20.04 LTS to test the simulations and run the programs. This system comes with Python 3.8.5 which is not supported by several of the libraries used, so pyenv was required to test various python versions. Additionally, D4PG uses the Google Deepmind Acme library which uses Tensorflow 2 while stable-baselines for other algorithms uses Tensorflow-1 which may affect the program runtime.

0.0.4 DDPG

DDPG, or Deep Deterministic Policy Gradient was my first approach. I coded DDPG using Tensorflow and Keras which uses an actor-critic model with deterministic actions. The goal of the actor is to maximize the q-value of the action while the goal of the critic is to provide a proper q-value. Exploration of the environment is controlled by OU Noise added to the action during training. This was tested with the OpenAI Pendulum environment before testing on the Mujoco environment. I chose to code this algorithm without an RL library to gain an inner understanding to the algorithm and its fine details that could be missed when using a library. The remaining algorithms use libraries for the sake of time and debugging and avoiding to reinvent the wheel.

On my Linux machine, due to lack of experience with OpenMPI for multiprocessing and time, I trained DDPG and many other algorithms with 1 actor. With 1000 episodes, I found that using 1 actor did not perform as well and took a longer time to train in comparison to the other algorithms. The average reward at the end was around 0.4 without any reward scaling.

0.0.5 PPO, TRPO and A2C

PPO was released a little while after TRPO making with a few advancements made in the algorithm. TRPO is described as a monotonously increasing in its returns. However it didn't perform as well or as stable as PPO at around 1000 episodes with 1 actor, so I stopped testing and running TRPO. PPO would reach a maximum of around a moving average of 140 as the return. I ran A2C with OpenMPI multiprocessing with 4 actors which lead to an average of 120 after 1000 simulations. A2C was the fastest algorithm, especially since it could run more episodes at a time, but it was not the best performing algorithm. Overall, I found that while PPO took a bit longer to run, it was definitely one of my strongest choices. After running PPO for a while with 5000 simulations, it was not able to pick up the cube.

I couldn't find many resources on how to proceed from here, but I found some benchmarking reports for Mujoco-Py where D4PG has the best performances for multi degrees of freedom robotic tasks. However, I later found the benchmarking report using SURREAL where results with DDPG and PPO were compared. It seems the best performances occured after 10 hours of training with multiple actors which I could not afford. Also, some of the better results occured when pixel values were inputs which I decided to try after testing D4PG.

0.0.6 D4PG

D4PG is different from DDPG in that it actually has a policy actor network that is distributed. Also, D4PG makes use of N-step returns rather than a 1 step return. Finally, it makes use of a prioritized experienced replay. Overall D4PG was able to perform faster and more efficiently than PPO. One of the leading factors that makes this off policy method efficient is the N step return along with the distributional updates of D4PG. D4PG was able to perform a lot faster than DDPG with reaching heights of rewards like 580 after 5000 episodes.

0.0.7 Modifications and Problems

After looking at the benchmarking SURREAL paper, I was interested in using pixel information via a CNN (Convolutional Neural Net) for feature extraction as the state space of the ML program with robosuite's current support. After implementing a CNN, I realized that I didn't have the proper resources to run this program. Ubuntu 20.04 LTS is not supported by Robosuite and Mujoco Py for off-screen rendering (which is required for camera input feed). Robosuite is working on alternative rendering options according to GitHub issues. As a result, this could not be tested.

0.0.8 Conclusion

In conclusion, out of the approaches tried, D4PG was one of the best options. Normally experiments are supposed to be done with repetition for more accuracy, but with the given time and different environments for testing standardization was not possible. I learned that pixel inputs are supposedly more reliable for training effectively. Also, with more actors as mentioned in the SURREAL article, we should see a significant improvement in training. Overall, this project was difficult in the learning process, but it was very fun when I got to see how it turned out in the simulation. I hope to make modifications to these algorithms such as adding HER support with robosuite and D4PG to see if it is as efficient in sparse reward environments. I also hope to test and try modifications to reinforcment learning algorithms to be able to explain why various hyperparameters and design structures affect an agent.

1 Resources

1) Spinning Up [Link]
2) SURREAL Paper [Link]
3) Robosuite [Link]
4) Stable Baselines [Link]
5) Acme - Deepmind [Link]