Midterm Progress Report

Ben Bush, Steven Hartman, Eric Tang

**Project Status**

The project is well underway and on pace with the work plan that was outlined in the Short Review Essay. Ben has undertaken the development of the A3C algorithm, Eric the DQN algorithm, and Steven the experimental data. All team members have devoted time to understand the challenge, observation space, action space, and reward function and are now fully engaged in their respective tasks.

The challenge provides a “boiler-plate” repository, with sample PPO and DDPG implementations, and instructions for how to create new agents. We have cloned this helper repository and used it as a starting point for our own work. The challenge provides the simulation environment as a Python 3 module, so all code is written in Python 3 to interface with the simulation environment. All neural network controllers are implemented using Keras and Tensorflow. Keras has a sister library, Keras-RL, that serves as an API for many common deep reinforcement learning algorithms. It allows users to program against DQN, PPO, and DDPG agents. Users can supply their standard Keras neural network architectures and any hyperparameters to the API, which then takes care of constructing the training and prediction functions that would otherwise need to be implemented. Furthermore, Keras-RL natively integrates with OpenAI Gym, which the provided simulation environment is based on. Thus, Keras-RL is a natural fit for our project and allows us to focus on some more interesting parts of the challenge, such as the neural network controller architectures and use of experimental data.

Steven is using the experimental walking kinetics data of Schwartz et al. *Journal of Biomechanics* 41 (2008) to initialize a model intended to mimic human walking. While a reinforcement learning agent is, in principle, able to find the best policy purely by experience, the optimization problem can be difficult due to a large search space and the presence of many local minima which are not globally optimal. For example, if the musculoskeletal model is initialized with default settings, it simply falls over backward. It accumulates negative reward for each timestep that it moves backward, so the locally optimal strategy may be to fall over backward as quickly as possible to reduce the negative reward. Steven has mapped the experimental data to specific observations and actions within Opensim (the simulation environment) and has completed a vanilla linear regression model using Sci-Kit Learn. This model predicts muscle activations based on the angle of several important joints, although it model has not yet been interfaced with the simulation and agents. A linear regression was chosen because the initial standing start position is outside the range of experimental data, which is from continuous walking, so a low-variance model may give a lower extrapolation error. The linear regression model has an .

Steven has also trained the provided Keras DDPG and Tensorforce PPO agents 500,000 timesteps (approximately 5,000 episodes) each to serve as a baseline for comparing any custom-made agents. The default PPO agent never achieved forward motion and was still falling backwards at the end of this training with a reward ranging from -300 to -450. The default DDPG agent began with better results, although it was still falling backward, but during the training it learned to fall backwards much faster, with much more negative cumulative reward.

Eric has completed the continuous DQN implementation. Due to the fact the traditional DQN algorithm is only available for discrete actions space and our action space is continuous, we needed a way to implement DQN with continuous actions. One such method is to call normalized advantage functions (NAF). NAF represent the Q-function in such a way that the maximum can be determined easily and analytically during the Q-learning update step. In its implementation, NAF constructs a neural network that separately outputs three function terms: a value function term, an advantage term, and a policy term that parametrize the Q-learning update step. Keras-RL provides an implementation of the NAF agent, which we are using in our code. In theory, the DQN agent should outperform the DDPG agent. After training the DQN agent first 1000 epochs, the reward of the first episode is -57.352. The first 1000 epochs training reward for the default DDPG is -460.387 in the first episode test. Both agents have the same number of layers and units and use the same stochastic process function Ornstein–Uhlenbeck process, with . Both the DQN and DDPG agents are still falling backward. The next challenge for DQN is how to optimize the neural networks within different stochastic process functions and different structures or types of neural networks. The random process and its parameters determine the level of exploration vs. exploitation our agents will pursue during training, so we must consider both the neural network architectures and random processes while fine-tuning our agents. We have encountered another paper, *Deep Reinforcement Learning for Robotic Manipulation with Asynchronous Off-Policy Updates*, that draws on ideas from NAF and A3C. This paper outlines and a method with asynchronous NAF executing on N collect threads and 1 trainer thread. Given the potential speed up we may see with this new method, it may be in our interest to attempt its implementation. Unlike A3C, Asynchronous NAF is an off-policy method and each collect thread will not provide any gradient information. Given this large disparity, we must evaluate if it is worthwhile attempting its implementation before dedicating much more time to it.

Ben has nearly completed his implementation of the A3C algorithm. Unfortunately, Keras-RL, or any other deep reinforcement learning library, does not currently provide an A3C agent due to the complexity of the algorithm. As a result, we have had to develop the algorithm from scratch. Online blogs have been very helpful in understanding some of the specific implementation details. One major challenge presented by the A3C algorithm is that it the algorithm is asynchronous: it uses multithreading to create many rollouts in the simulation environment, and then once the rollouts have completed, each thread contributes what it has “learned” to the global gradients. Each thread is then updated with the new global gradients, and new rollouts are executed. Multithreading has many benefits: it speeds up the training process and encourages diversity in the training environment, which should make the model more robust. However, a multi-threaded agent was difficult to develop. We implemented multithreading using the threading module in Python. This module allows us each thread to execute a thread training function, which use performs the rollouts and updates the model parameters. We have validated that the agent is able to interact with and learn from the simulation environment and are in the final steps of proofreading the code for correctness. Functionality of the algorithm has taken precedence over the architecture of the actor and critic networks. The actor and critic networks are currently implemented as fully connected networks with all non-output layers shared, as was suggested in the original A3C paper. In the future, we intend to experiment with convolutional and recurrent neural networks for the controller. Under the current network architecture and training for only 1000 episodes, the A3C algorithm achieves a score of around 50.

**Work Plan for Remaining Tasks**

Given that we are on pace with our original plan, we do not see the remaining tasks deviating too much from the schedule we outlined in the Short Review Essay. The next immediate task is to bootstrap the neural network controllers using the supervised model created with the experimental data. The remaining tasks involve tweaking network architecture and exploring new methodologies, such as frame skipping, to speed up training time. A time schedule for remaining tasks is outline below. Hours spent waiting for neural network controllers to finish training are not included in estimates.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Task** | **Resources Needed** | **Hours Spent** | **Hours Needed to Complete** | **Target Completion Date** |
| Midterm Progress Report | Text editor | 3 | 3 | 11/15 |
| Integrating Supervised Model | Supervised model, access to completed agents | 1 | 5 | 11/18 |
| Tweaking Controllers | GPUs | 0 | 18 | 11/28 |
| Exploring methodologies | GPUs | 0 | 6 | 12/2 |
| Presentation | PowerPoint, functioning agents | 0 | 8 | 11/29 or 12/2 |
| Final Report | Text editor, all other tasks completed | 0 | 8 | 12/6 |

**Issues**

As we noted in our Short Review Essay, it may be difficult to communicate or work on some of the hands-on tasks during the break for Thanksgiving. We have planned for this disturbance in our original schedule and plan to work on more “administrative” tasks, such as code documentation and an initial draft of the report and presentation during this time.

One major issue we foresee is availability of resources. Most of our algorithms are extremely compute-intensive. We have seen many posts on the challenge page describing some breakthrough strategies as brute force and using many GPUs to train a given agent for hundreds of hours. For this reason, we are currently investing more time and energy into asynchronous that may provide faster training times such as A3C and asynchronous NAF. Although we have 3 GPUs across our team and a handful of CPU cores, we no longer have weeks to train our agents and may simply be constrained by compute time.