Quarter Progress Report

Cedrick Argueta

March 16, 2019

1 Introduction

This work builds heavily off of [1] in that we have the same goal – improve drone tracking performance over greedy, one-step planners. We have two drones, a seeker drone and a target drone, and wish to track the moving target drone with the seeker drone by capturing emissions by the target drone's radio. However, the MCTS solution presented is often requires large amounts of memory and computational power that might be lacking on drone avionics boards. We propose using deep reinforcement learning to combat this problem, since the amount of memory required for inference is relatively small compared to traditional POMDP planners.

2 Background

As in [1], we use a belief-MDP to model the problem. The state of the system at time t is $s_t = (b_t, x_t)$, where b_t is the belief of the distribution of possible target states, and x_t is the seeker's state. b_t is used to represent the information that we've gathered from the target drone's radio. b_t can be approximated well with a discrete filter for stationary targets, or a particle filter for moving targets. The action taken at time t is u_t, and is one of a discrete set of commands given to the seeker drone. These actions are constant velocity actions in a radial pattern about the seeker drone. Because we've modeled the position of the target drone with a probability distribution, we'd like to minimize the uncertainty in our target estimate. This happens when we have a low amount of entropy in the particle filter. To find entropy, we must first discretize the particle filter into M bins. We can then define this entropy as:

$$H(b_t) = -\sum_{i=1}^{M} \tilde{b}_t[i] \log \tilde{b}_t[i] \tag{1}$$

where $\tilde{b}_t$ is the proportion of particles in each bin i. We'd also like to penalize near-collisions, so our cost function at time t is:

$$J(s_t) = H(b_t) + \lambda \mathop{\mathbb{E}}_{b_t} \mathbb{1}(\|x_t - \theta_t\| < d) \tag{2}$$

where θ_t is the target drone's actual position, and $\mathbb{1}$ is an indicator function.

To solve this belief-MDP, we make use of deep reinforcement learning. Much of the basis for deep reinforcement learning in this task comes from [2]. We define the Q-value of a state-action (s, a) pair as:

$$Q(s,a) = r(s) + \gamma \sum_{s' \in S} T(s,a,s') \max_{a' \in A} Q(s',a') \tag{3}$$

where $r(s)$ is the reward received for being in state s, γ is the discount factor, $T(s,a,s')$ is the transition probability of going from s to s' with action a. We use function approximation to mitigate the issues that arise from having such a large state space. Then our goal is to minimize the Bellman error, defined as

$$E = r + \gamma \max_{a' inA} Q(s',a';\mathbf{w}) - Q(s,a;\mathbf{w}) \tag{4}$$

where we have a state, action, reward, next state tuple (s,a,r,s') and $Q(\ldots;\mathbf{w})$ indicates that the Q-value function is parameterized by some weights $\mathbf{w}$. Then the optimal policy π_{opt} may be computed as:

$$\pi_{opt}(s,a) = \arg\max_{a \in A} Q(s,a;\mathbf{w}) \tag{5}$$

which tells us the best action to take from a given state-action pair.

3 Approach

Contrary to the previous approach, we instead are writing our own simulation environment so that the codebase may be simplified. For a preliminary implementation, we simplify the problem in two ways: we assume a stationary target, and use a discrete filter rather than a particle filter to model the target belief.

Preliminary results will come from using the DQN algorithm presented in [3] on this task. We chose to do so because it showed promising results in [2], and is relatively simple to implement compared to state-of-the-art algorithms. Since we have two types of inputs, a belief distribution represented as a matrix and the seeker drone's state, we need begin with two separate networks. The seeker state network is a simple fully connected network that takes an input in $\mathbb{R}^3$, representing the seeker's x position, y position, and bearing. The input is fed into five consecutive hidden layers of 100 units each, each with rectified linear unit activations. The belief network is a convolutional neural network that takes the filter matrix as input. This input is fed into three convolutional layers, each with 64 units and a filter size of (3×3). Each of these convolutional layers is followed by a max pooling layer with a filter size of (2×2). After the final max pooling layer, the outputs are fed into a dense layer with a size of 500 units. Finally, the outputs of both these networks are concatenated and fed into two consecutive dense layers, each with a size of 200 units and followed by rectified linear unit activations. The output layer has a dimensionality that equals the number of actions that the seeker can take, and each represents the Q-value for that action and the current state.

Learning the network parameters can be done with gradient descent or any of its derivatives. We use AdaMax in our implementation.

With the simulation environment and testing infrastructure in place, we will then begin to compare inference times and performance to the MCTS approach described in [1].

4 State of the Project

Progress made this quarter can be divided into two parts: attempting to wrangle the segfault on GPU and then attempting to rewrite the FEBOL package.

When running training sessions on `astoria`, a segfault would arise a few thousand training itertions in. The number of training iterations needed to cause the segfault wasn't consistent at all, and would happen anywhere from a few hundred to a few hundred thousand iterations. At the advice of people in your lab, I looked to see if this was a memory error – but `astoria` has more than enough memory to run a neural network of our size. I looked into seeing if one of the hyperparameters could give any insight into the problem, but no matter the settings the issue would happen sporadically on every training session that was longer than a trivial amount of time.

After fiddling with hyperparameters for a while, I decided it wasn't worth it to try every combination and dove into a debugger to try to figure out the issue. The debugger, though, was little to no help: the core dump told me little more than that there was an unauthorized memory access at a specific location. Running through the control flow of the program was nearly impossible for a few reasons. One was the nature of the package that we used to combine the Julia simulation code and the Python reinforcement learning system. The package would open up an interpreter and run the Julia commands inside the interpreter, giving us little access to the objects that Julia made use of besides the fact that they were Julia objects that required the interpreter for evaluation. Breakpoints couldn't be set at specific points in the Julia code that was being executed, they could only be set withing the Python codebase and even then weren't of much use when it came to seeing what happened within the separate Julia interpreter. The second was that we executed the Julia code dynamically, i.e., a string was constructed at runtime in Python and fed to a Julia interpreter for evaluation. This was necessary for the reinforcement learning system to give commands to the Julia simulation - an observation would be given to the RL system, the RL system would produce a command, and the command would be given to the drone in the Julia simulation. This, however, made it almost impossible to follow control flow for the Julia code without tracing through the entire Python code that lef up to the segfault, something that was infeasible when a segfault would occur hours into training. Because of the obfuscation of these two problems combined, Louis and I decided that it'd be better to do a minimal rewrite of the FEBOL simulation package in Python. A minimal rewrite would provide us with the most important features of the FEBOL package, such as a different types of filters and a simple representation of a drone, along with the removal of the Julia to Python library that might have been the cause of the segfault. An interesting point is that the training code works locally; only when the training code is moved to a system with a GPU does the segfault arise. Hopefully rewriting the FEBOL package will give us either a better insight into what's going wrong when switching to GPU, or remove the issue altogether. At the very least, it will be much easier to use a debugger and find the problem if it persists.

References

[1] Louis Dressel and Mykel J. Kochenderfer. Hunting drones with other drones: Tracking a moving radio target.

[2] Kyle D. Julian and Mykel J. Kochenderfer. Distributed wildfire surveillance with autonomous aircraft using deep reinforcement learning. *CoRR*, abs/1810.04244, 2018.

[3] 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.