Quarter Progress Report

Cedrick Argueta

December 16, 2018

1 Introduction

This work builds heavily off of [2] 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 [2], 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 [3]. 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

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

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

We use the FEBOL package in [1] for the simulation enviroment. 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 [4] on this task. We chose to do so because it showed promising results in [3], 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 [2].

4 State of the Project

The preliminary approach described earlier has been implemented on my local machine with Keras-rl and FEBOL. The primary difficulty there was learning how to connect all the moving parts of the project: simulation code written in Julia, a deep reinforcement learning package written in Python, and the supporting packages that allowed the two to communicate. Several bugs rose out of the fact that only specific versions of these packages interfaced well with each other, and finding a combination that worked required quite a bit trial and error.

I'm currently in the process of moving the local code to a server with GPUs so that we can begin to run experiments. This, however, brought me back to the stage where I had to find compatible versions of every package. I've been blocked on a considerably nasty bug with the Julia-to-Python pipeline for all of finals week, so that's where my my time on the project over this break will go. I'm also considering a rewrite of several parts of the reinforcement learning infrastructure so that I no longer depend on Keras, hopefully alleviating some of the version woes that have plagued me so far.

In terms of big picture, we have a vastly simplified problem so that we can verify that our infrastructure works. Once this is done, we can begin to expand to the full problem and use a particle filter rather than a discrete filter and allow for the target drone to move. Then I'd be able to run the same experiments as in [2] and compare inference times for MCTS vs. DRL.

References

[1] Louis Dressel. Filter exploration for bearing only localization, 2018.

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

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

[4] 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.