DEEP REINFORCEMENT LEARNING FOR DRONE-BASED RADIO LOCALIZATION

Cedrick Argueta

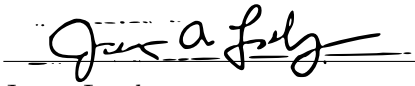
Stanford University
June 2020

An honors thesis submitted to the department of
Computer Science
in partial fulfillment of the requirements for the undergraduate
honors program

Advisor: Mykel Kochenderfer

 Date: 6/1/2020

Mykel Kochenderfer (Thesis Advisor)
Associate Professor
Aeronautics and Astronautics
Computer Science, by courtesy

 Date: 6/1/2020

James Landay
Anand Rajaraman and Venky Harinarayan Professor
Computer Science

Abstract

Unauthorized drones present a danger to airports and disaster areas. Localization and tracking of these unauthorized drones reduces some of this danger. It is possible to use a drone outfitted with with commercial antennas and radios to autonomously localize other drones. Prior work has shown that drones with this equipment may use onboard planners such as Monte Carlo tree search to perform path planning in a localization task. In this work, we demonstrate that the same is possible with deep reinforcement learning, moving computation off the drone computer and into simulation.

Acknowledgments

Thank you to Louis Dressel, who gave me amazing guidance on this project and gave me motivation to attend graduate school. I had no idea that his assignment as my mentor would be such a turning point in my academic life, and I am so grateful for his continued support in spite of all the difficulties I faced. Thank you to Mykel Kochenderfer, who supervised this project and gave invaluable advice. I'm lucky to have met such an interesting and supportive professor. I also thank my coworkers at The Aerospace Corporation, who graciously supported this project and also pushed for me to attend graduate school. Thank you to my friends and family, without whom I wouldn't have had the mental fortitude to finish this project considering the circumstances of Spring 2020.

Contents

Chapter 1

Introduction

The use of drones has become widespread in recent years. In the civil sector, drones can be used for disaster relief or surveillance. In the commercial sector, drones can be used for package delivery, photography or film production, or agriculture. Drones are also often used in the military for reconnaissance and as weapons.

The popularity of drones brings with it several challenges. Drones near airports pose a threat to aircraft operations, and carry the potential for a terrorist attack [1]. This year to date, the UK Airprox Board has recorded 75 "Airprox" events in which drones may have compromised the safety of aircraft [2]. Currently, the FAA prohibits drones from operating near rescue efforts in natural disasters like hurricanes or wildfires [3]. In these situations, it would be beneficial to localize the position of the offending drone. Then it'd be possible to intercept and deal with the drone in a timely and safe manner.

Tracking a drone can be done by monitoring its radio telemetry. Drones usually emit radio signals to their operators, like speed, position, battery life, and other vital data. A seeker drone outfitted with radio receivers and antennas can home in on these radio signals and find the target drone's position. While it is possible to track drones on the ground or with stationary trackers, a drone-based tracker would be as mobile as the target and be able to follow it for longer distances. In essence, this work focuses on localization of a single radio-controlled drone with an autonomous drone.

This work builds heavily off of [4] and [5] in that it has the same goal – improve drone tracking performance over greedy, one-step planners. A greedy solution would execute the best action (in terms of some cost function) at every timestep, without planning ahead to the next. This work develops a planning system that aims to choose the best overall actions, perhaps sometimes making suboptimal myopic decisions in exchange for better hyperopic path planning. We also take on the restrictions of low-cost hardware as seen in [4], so that our experiments are comparable and validation of simulations can occur seamlessly.

This formulation of the problem has two drones: a seeker drone and a target drone. The seeker

drone's objective is to track the moving target drone by capturing emissions from the target drone's radio. Existing solutions using Monte Carlo tree search (MCTS) often require large amounts of memory and computational power that might be lacking on drone avionics boards. Previous work shows how neural networks can be used for guidance of drones to waypoints with much smaller memory footprints than traditional Markov decision process (MDP) solutions in obstacle avoidance and waypoint finding [5]. This work applies the same methodology to the drone localization problem. We show that it is possible to train several deep reinforcement learning algorithms, namely Deep Q-Networks (DQN), Deep Deterministic Policy Gradients (DDPG), and Soft Actor-Critic (SAC), to perform path planning for the seeker drone.

The main contributions of this work are twofold. First, we provide a low-memory approximate solution to continuous control in radio-source localization. Second, we develop a simulation system for localization problems that allows for training of reinforcement learning agents.

Chapter 2

Related Work

Low-cost bearing-only localization is explored by Dressel and Kochenderfer in [6]. This work introduces a pseudo-bearing sensor consisting of a directional antenna and an omnidirectional antenna. When used in tandem, the antennas produce pseudo-bearing measurements of the radio target, improving localization time compared to sensors that contain only a single directional antenna. This sensor system, however, imposes the constraint of knowing the gain pattern of the directional antenna. Without this information, accurate pseudo-bearing measurements cannot be taken. Also introduced in this work is the belief MDP framework for solving drone-based localization problems. Because there is uncertainty in measurements provided by the seeker drone's sensors, the target's true position is only partially observable. Including this uncertainty into an MDP results in a partially observable MDP (POMDP) with a much more difficult solution space. To control this additional complexity, the authors use a belief MDP, where the current 'belief,' i.e. some estimate of the true state, is used in the state space. In the drone localization problem, this is a filter that is updated with the measurements provided by the seeker's sensors. The controller used in this work is a greedy solver, selecting actions according to minimum cost at every step. While simple to implement, solutions resulting from the greedy solver are suboptimal and can be improved.

This improvement is realized in a MCTS solution to drone localization, presented by Dressel and Kochenderfer in [4]. MCTS improves performance over the greedy controller in terms of both mean tracking error and near collision rate. In addition to the enhanced solver, this work uses a simpler sensor modality introduced in [7]. The sensor modality used in this work consists of two directional antennas, which are used to compare signal strength in front of or behind the drone. While less informative than the pseudo-bearing sensor developed previously, this sensor has the advantage of being easier to construct. In the low-cost paradigm this work focuses on, we prefer the simpler sensor and thus develop our seeker drone according to this model.

Neural networks can be used to compress the solution to a large MDP [5]. In this work by Julian and Kochenderfer, the lookup table for the discrete value iteration solution is represented by a neural

network. In terms of solution quality, this compression is shown to have negligible differences to a standard solver. However, it produces solutions thousands of times faster and with a much smaller memory footprint, motivating the use of neural network sin path planning for radio localization. Because of the memory requirements imposed by a MCTS solver, compression using a neural network is desirable.

Deep reinforcement learning (deep RL) for unmanned aerial vehicle (UAV) control is applied in [8]. This work demonstrates a deep RL controller for a fixed-wing UAV tasked with wildfire monitoring. This work is notable for its representation of uncertainty in the system with a particle filter, which is also seen in [4, 7]. However, the particle filter representation of the state space is fed directly to a convolutional neural network for processing, rather than used in simulation as in [4, 7]. We combine the concept of convolutional neural networks for particle filter state representations with the drone-based radio localization task, allowing us to decrease onboard computational cost of planning.

There exist a variety of controllers suitable for this problem. In the discrete action space setting, a popular algorithm is Deep Q-Networks (DQN) [9]. While the algorithm was first used on Atari games, simulation environments like those found in [8] and [4] may also be used. It is important to note that DQN necessitates a discrete action space, resulting in a limited set of actions the seeker may take at every step. In the continuous action space setting, the analog to DQN is Deep Deterministic Policy Gradients (DDPG) [10]. Improvements upon the original DDPG algorithm have brought rise to the Soft Actor-Critic (SAC) algorithm, a variation that maximizes entropy in the produced solution. Because drone control is inherently continuous, DDPG and SAC represent more suitable choices.

Chapter 3

Preliminaries

3.1 Drone Hardware

The target drone is a DJI F550 hexcopter with a telemetry radio transmitting using FHSS along the 902-928 MHz band. The seeker drone is a DJI Matrice 100 quadcopter with a simple sensor system: two Moxon antennas mounted on the front and back of the drone. When the signal strength measured is stronger at front, the target is said to be in front of the drone, and vice versa. In modeling the drone localization problem, we choose to represent the world as a 2D plane. In information gathering tasks, changes in altitude with this sensor system do not yield much more information because of almost constant gain over elevation angle to the source [4].

3.2 Drone Dynamics

The drone dynamics are exactly those described in [4]. The seeker drone state at time t is $x_t = [x_t^n, x_t^e, x_t^h]^\intercal$, where x_t^n and x_t^e are the seeker's north and east coordinates and x_t^h is the seeker's heading measured east of north. The state described here does not contain velocity or altitude as simplifying assumptions. The drone follows a first-order motion model, so the state after applying a control input u_t for duration Δt the new state is

$$x_{t+\Delta t} = x_t + u_t \Delta t \tag{3.1}$$

The target drone state at time t is $\theta_t = [\theta_t^n, \theta_t^e]^\intercal$, where θ_t^n and θ_t^e are the target's north and east coordinates. The target drone is assumed to move with a constant velocity $\dot{\theta} = [\dot{\theta_t^n}, \dot{\theta_t^e}]^\intercal$. The drone follows a first-order motion model, so the state after Δt is

$$\theta_{t+\Delta t} = \theta_t + \dot{\theta}_t \Delta t \tag{3.2}$$

3.3 Sensor Model

The bearing from the seeker drone to the target drone is

$$\beta_t = \arctan \frac{\theta_t^e - x_t^e}{\theta_t^n - x_t^n} \tag{3.3}$$

when measured east of north.

At time t, the seeker drone makes measurement $z_t \in \{0, 1\}$, representing whether the target drone is in front of the seeker drone. To determine z_t, the seeker measures the signal strength of both of its antennas – a strong signal on the front-facing antenna indicates that the target lies in front of the seeker, and vice versa. When the target lies to the sides of the seeker, both measurements are equally likely. Additionally, there is some chance of error resulting from noise in the system [7]. Thus, the probability of measuring $z_t = 1$ is:

$$P(z_t = 1 | x_t, \theta_t) = \begin{cases} 0.9 & \text{if } \beta_t - x_t^h \in [-60°, 60°] \\ 0.1 & \text{if } \beta_t - x_t^h \in [120°, 240°] \\ 0.5 & \text{otherwise.} \end{cases} \tag{3.4}$$

3.4 Particle Filter

The seeker drone maintains a belief of the possible location of the target drone, which is modeled with a particle filter. It is useful to think of a particle filter as a type of hidden Markov model: as evidence from the sensor is received, we can update our belief of the true state of the system. Belief is often strongly non-linear, so particle filters are preferred over standard Kalman filters for this task [4, 8]. With stationary radio sources, a discrete histogram filter may be used instead. Updating discrete filters for moving targets has poorer runtime performance than particle filters in the same task.

Belief at time t is represented by a set of N particles, each representing a hypothesis of the target drone's pose. Each particle has a position and velocity estimate of the true target position and velocity. Updates are made to the particle filter at every timestep to improve the belief's accuracy.

The belief update consists of three steps. The first step is the prediction step, where each particle is propagated according to the dynamics described in equation 3.2. Noise is added to the dynamics to prevent particle deprivation, a situation that arises when all particles converge to a hypothesis that doesn't accurately represent the true state. The second step is the weighting step, where each particle is assigned a weight according to how probable an observation z_t is given the particle's position:

$$b_t(p_i) \propto b_{t-\Delta t}(p_i) P(z_t | x_t, p_i) \tag{3.5}$$

where p_i is a particle and $b_t(p_i)$ is the probability that the true state is modeled by particle i. $P(\cdot)$ is

our observation likelihood function from 3.4 and z_t is the seeker's observation at time t. The third step is resampling, where particles are sampled according to these weights with replacement. In this work, we use stratified resampling to aid in maintaining an accurate estimate of the target while ensuring resiliency to particle deprivation.

3.5 Belief Markov Decision Process

As in [4], we will use the belief MDP framework to solve the drone localization problem. We will first motivate the belief MDP with a discussion of partially observable MDPs (POMDPs).

A POMDP comprises a state space $\mathcal{S}$, an action space $\mathcal{A}$, a reward function R, and a transition function T defining the transition between states. An agent is unable to observe the true state s_t directly and instead makes an observation $\omega \in \Omega$ conditioned on the true state.

Solving a POMDP consists of finding a policy π^* such that

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(\omega_t))\right] \tag{3.6}$$

where γ is a discount factor.

POMDPs have a significant disadvantage when formalizing localization tasks. Rewards that depend on belief of the true state of the system are often difficult to represent in the POMDP framework [4]. For this reason, we instead convert the POMDP to a belief MDP.

Belief MDPs are similar to MDPs where the system state is instead a *belief* of the true system state. We hereafter model the problem as an MDP where each state is a tuple of the fully observable part of the true state and the belief of the partially observable part of the true state.

States

Each state is a tuple $s_t = (b_t, x_t)$ where b_t is the seeker's belief and x_t is the seeker's pose. The seeker's belief is the particle filter mentioned in the previous section.

Actions

The belief MDP framework is general enough to model both discrete and continuous action spaces. In this work, we consider both cases. In the discrete case, the seeker drone is allowed to travel with a constant velocity in 8 directions equally spaced in a radial pattern. In addition to moving in 8 directions, the controller may also elect to change its heading by $+15°$, $0°$, or $-15°$ at each step. This results in 24 total actions available at each step. Null actions can be approximated by taking opposing actions in subsequent timesteps. In the continuous case, the seeker drone travels with a constant velocity in any direction, with the heading change limited to the continuous range $[-15°, +15°]$.

Reward Function

Our reward function for radiolocation captures the desire to maintain an accurate and precise estimate of the target's location *while also* maintaining an acceptable distance from the target. The motivation behind this is the desire to preserve the safety of the seeker drone, but also extends nicely to other applications besides drone localization. For example, if a drone is attempting to localize a wild animal with a radio collar, it is prudent to not fly too close lest the seeker scare the target away.

A precise belief is one that has low uncertainty over the target's position. Minimization of this uncertainty is equivalent to minimization of the entropy of the belief distribution. Particles in the filter are first discretized into M bins. Entropy can then be defined as:

$$H(b_t) = -\sum_{i=1}^{M} \tilde{b}_t[i] \log \tilde{b}_t[i] \tag{3.7}$$

where $\tilde{b}_t$ is the proportion of particles in each bin.

The *true* objective then has the form:

$$r(s_t) = H(b_t) + \lambda \mathbb{E}_{b_t} \mathbb{1}(\|x_t - \theta_t\| < d) \tag{3.8}$$

where λ defines the importance of the near collision penalty and d is a distance threshold. The penalty term contains only the belief of the belief of the target's position rather than the true target position. This is to encourage the seeker to maintain a distance from the particles during evaluation. If the belief is representative of the true state, then the seeker will maintain a safe distance. If the belief is not representative of the true state, then the seeker will at least maintain a distance from the belief, which still might contain a noisy or partially accurate model of the target's motion.

This formulation, however, provides little signal to reinforcement learning-based controllers. This is especially prevalent with the stochastic nature of particle filtering for belief estimates. Simply using entropy as a reward presents a problem because it is stochastic, non-Markovian, and the relative difference between a good reward and poor reward is small (often smaller than difference from stochasticity). Conceptually, a good policy for localization with a near collision penalty would be similar to a geologist studying a volcano – get as close to the rim as possible and then hover around the rim. This policy can be easily described with a reward function that is shaped like a volcano, with reward concentrated at the near collision boundary, and a large penalty for being inside. Instead we use a surrogate reward function using proximity to the particle filter centroid R.

$$r(s_t) = \begin{cases} \frac{R}{d} - r_{min} & \text{if } R \leq d \\ r_{scale} \exp\left\{-\frac{1}{r_{scale}}(R-d)\right\} & \text{otherwise} \end{cases} \tag{3.9}$$

where r_{min} is the minimum reward and r_{scale} controls the shape of the exponential. This surrogate

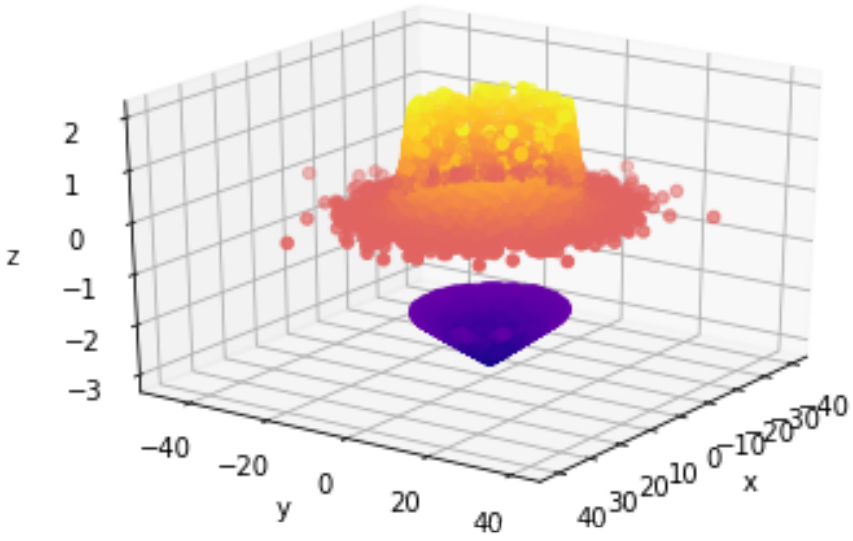


Figure 3.1: Monte Carlo simulation of reward function. Volcano shaped.

reward function is much simpler to optimize compared to the true objective, resulting in faster convergence for RL-based controllers. A visualization of the reward is shown in 3.1.

An issue with this surrogate is that the constants r_{min} and r_{scale} have little interpretability in the context of the original reward function. Choosing values for these parameters to be comparable against a MCTS agent with parameter λ is very difficult. Also, there is no guarantee that this surrogate describes the optimal behavior for our seeker – this surrogate only describes a human intuition for what a decent policy is, as the true objective is very difficult to optimize.

Chapter 4

Methods

4.1 Deep Q-Networks

The Deep Q-network (DQN) algorithm is an off-policy reinforcement learning algorithm originally used to solve the Atari environments [9]. As with other reinforcement learning algorithms, state-action pairs are assigned a value $Q(s, a)$ representing the value of taking that action from that state. This value is defined by the Bellman equation

$$Q(s,a) = R(s) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) \max_{a' \in \mathcal{A}} Q(s', a') \tag{4.1}$$

where $p(s'|s, a)$ is the probability of transitioning to state s' from state s after taking action a. Because computing the true value of Q for every state-action pair is intractable, we use a neural network to approximate Q. We train this neural network by minimizing the Bellman error

$$E_{Bellman} = R + \gamma \max_{a' \in \mathcal{A}} Q(s', a'; \mathbf{w}) - Q(s, a; \mathbf{w}) \tag{4.2}$$

with gradient descent. The optimal policy derived from the Bellman equation is then

$$\pi^*(s) \approx \arg\max_{a \in \mathcal{A}} Q(s, a; \mathbf{w}) \tag{4.3}$$

The exact algorithm is detailed in [1].

In practice, there are several steps we must take to ensure that our environment is compatible with DQN. As mentioned earlier, DQN relies on a discrete action space. The preprocessing function ϕ ensures that the state space can be fed to the neural network approximating the Q function. Since particle filters are necessarily continuous in their estimates of the true state, they must be discretized for input to the convolutional neural network. This also involves an important feature of

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ **do**
        With probability ϵ select a random action a_t
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action a_t in emulator and observe reward r_t and observation x_{t+1}
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on the loss $(y_j - Q(\phi_j, a_j; \theta))^2$
    end for
end for

[8] – ensuring that observations are relative to the seeker. This assists convergence in that the agent doesn't need to learn a different policy for each starting position of the seeker in the domain. The agent can use prior experience more easily, as observations in episodes are no longer as sensitive to initial conditions as before. An image illustrating this is shown in 4.1, compared to a non-relative observation shown in 4.2.

4.2 Network Architecture

A neural network is used to approximate the Q value function. In a style similar to [8], a two-stream architecture (Figure 4.3)is used. This is to accommodate the tuple that represents our state: a convolutional neural network is used for the filter, while a standard neural network is used for the seeker state variables. The particle filter representing belief is first discretized to a 2d histogram. Convolutional layers are then used to extract spatial information from this downsampled belief. Only particle positions are used in this downsampling – the mean of the velocity of all particles is concatenated to x_t. The downsampling allows us to take advantage of the spatial relationships that particles have. As a form of feature engineering, we include the particle filter centroid with the seeker state variables. ReLU activations are used for all fully-connected layers, while instance normalization is used for the convolutional layers.

Additionally, several improvements over the original DQN algorithm are used. Namely, we use a dueling architecture introduced in [11] and use the double DQN update rule described in [12].

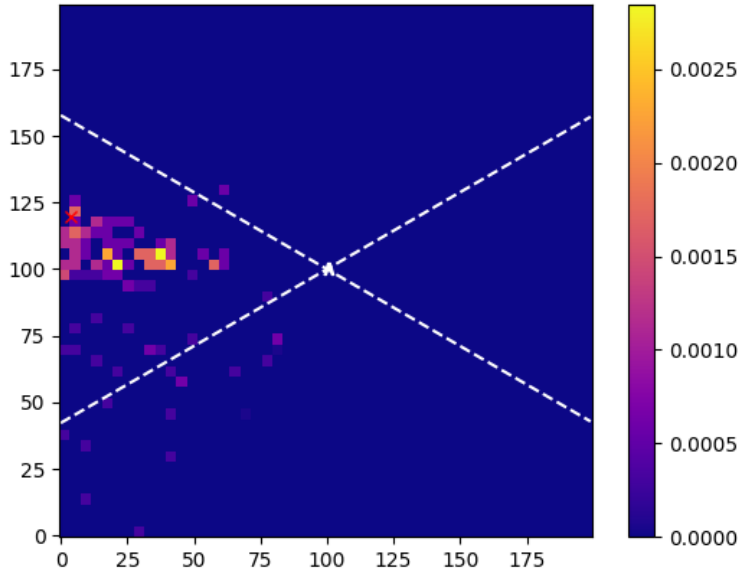


Figure 4.1: Downsampled and histogrammed particle filter, rotated relative to ownship. Dotted lines indicate extent of sensor accuracy.

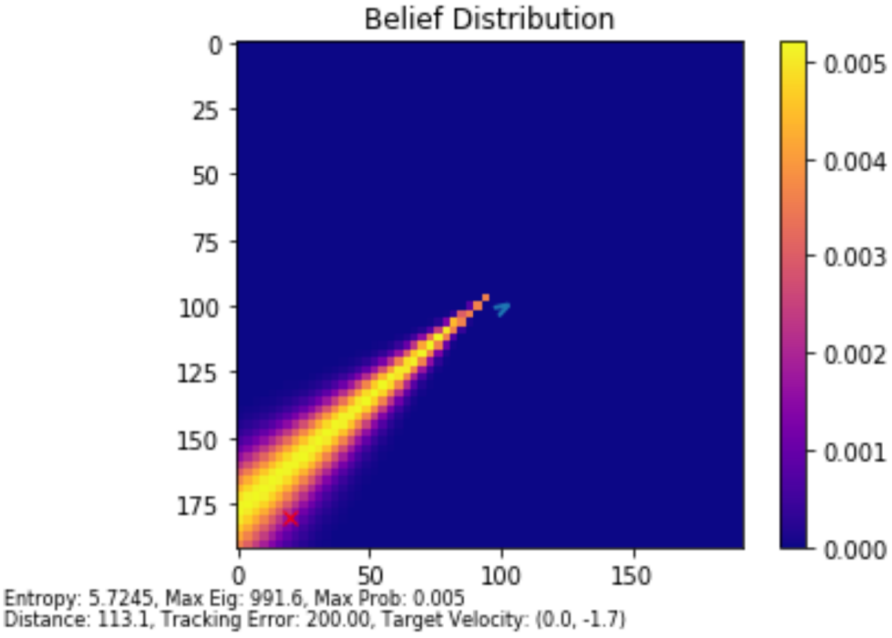


Figure 4.2: A non-rotated belief. This scene is not the same as that shown in4.1, but showcases how the rotation of the seeker is not taken into account when constructing the observation.

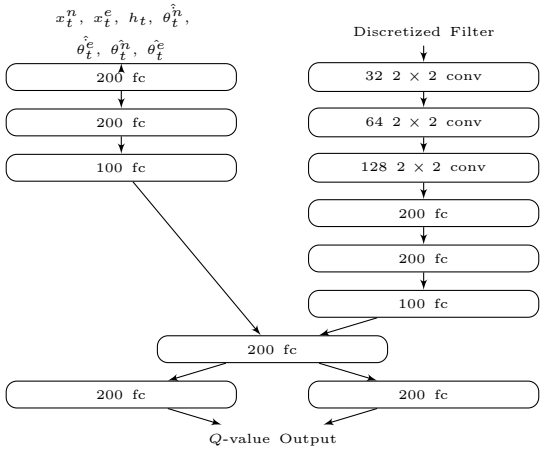


Figure 4.3: A two-stream dueling architecture is used to approximate the Q value function. `fc` denotes a fully-connected layer while `conv` denotes a convolutional layer.

4.3 Transition to Continuous Actions

Action discretization is inherently suboptimal when compared to continuous action spaces. However, it is necessary when solving with DQN. Performing action discretization while conforming to the experiment structure defined in [4] requires 109 actions, no small feat for a DQN-based controller. We mitigate this somewhat by limiting our action space to just 24 separate actions, but even this presents a challenge. Many actions are very similar to each other – it would be useful to leverage these similar actions when performing updates on our value functions, but the standard DQN algorithm does not permit this.

Instead, we implement continuous control algorithms: DDPG [10] and one of its successors SAC [13, 14]. Instead of choosing one of 24 actions, the agent now had two continuous outputs – a relative bearing for navigation and a heading for sensor movement.

4.4 Deep Deterministic Policy Gradients

DDPG is an off-policy reinforcement learning algorithm for use with continuous action spaces [10]. The algorithm works by maintaining an actor function and critic function, both approximated by neural networks. The actor function receives as input the state of the environment, and outputs an action in the continuous action space. The critic function receives as input the state of the environment and the action taken at that time, and outputs the expected value attained from taking that transition. By minimizing the Bellman error as in DQN, we approximate the Q value function. The actor function is optimized directly to output actions that maximize expected return.

A variety of modifications were needed to make DDPG a viable solution method in the context of

deep reinforcement learning. First, an experience buffer is needed to minimize correlations between samples. Second, target networks are required to stabilize learning. Both of these modifications were introduced in [9]. A detailed exposition of DDPG can be found in [2].

Algorithm 2 Deep Deterministic Policy Gradient

Initialize replay memory $\mathcal{D}$
Initialize actor network μ and critic network Q
Initialize target networks μ' and Q'
for episode $= 1, M$ **do**
    Initialize sequence s_1
    Initialize random process $\mathcal{N}$ for exploration
    for $t = 1, T$ **do**
        Select action $a_t = \mu(s_t) + \mathcal{N}_t$
        Execute action a_t and observe reward r_t and observe new state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
        Sample random minibatch of N transitions (s_i, a_i, r_i, s_{i+1})
        Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}))$
        Update critic by minimizing loss equal to MSE of $y_i = Q(s_i, a_i)$
        Update actor with policy gradient $\nabla J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a) \nabla \mu(s)$
        Update target networks
    end for
end for

The preprocessing on filter and state space remains the same as in the discrete action case. The neural network architectures are as similar as possible to DQN while maintaining correct input and output layers, i.e., only the input and output layers are modified and the intermediate layers are the same.

4.5 Soft Actor-Critic

The SAC algorithm represents a modification of the DDPG algorithm [13, 14]. The crux of SAC is that actions are taken to maximize the entropy of the policy, i.e. solving the task as randomly as possible. A full treatment of the algorithm may be found in [14]. However, we do note here that the stochastic policy produced by the algorithm offers a major advantage over DDPG: the stochastic nature of the environment is better handled with a stochastic actor than a deterministic one. Like DDPG, we apply the same preprocessing we performed in DQN and ensure that our neural network architectures are as similar as possible.

4.6 Simulation

Training the planner on the physical system is infeasible because of time constraints – the training environment must be reset every episode, human intervention is required to replace batteries, and weight updates are limited by the frequency of actions. Thus, each algorithm is trained on a simulator that captures the essential aspects of the system described in section 3. The simulator code may be found at [15].

The seeker and target drones are modeled in a 200 $m\times$ 200 m area, where the seeker begins each episode at the center of the area and the target begins at a randomly selected corner and travels to an adjacent corner at 1.7 m/s. In the discrete case, the seeker drone can move at 5 m/s in 8 equally-spaced directions. In the continuous case, the seeker drone can move at the same speed in any direction. The particle filter has 2000 particles, uniformly distributed at initialization and pruned using stratified resampling. In all experiments, the distance threshold is taken to be 15 meters.

Chapter 5

Results and Discussion

5.1 Metrics

Agents are evaluated quantitatively on two criteria: near collision rate and tracking error. Near collision rate is the proportion of timesteps that a collision occurred. Tracking error is the distance between the centroid of the belief and the true target position per timestep. These two criteria are adversarial – a tradeoff exists between minimizing tracking error by obtaining measurements closer to the target and maintaining a safe distance.

5.2 Baseline

The RL-based planners are compared against two baselines found in [4], a UCT-based MCTS planner and a greedy planner. Both these methods are online and require no training. To this effect, we have already accomplished one of our goals by simply using reinforcement learning – an RL-based planner will have computation done off the drone, prior to any flights. Any calculations done for in-flight planning are merely the forward passes through the neural network, rather than simulations done in-flight at every step. We observe that the baseline agents match the performance seen in [4], effectively validating our simulation environment as correct.

5.3 Reinforcement Learning-based Controllers

5.3.1 DQN

We find that discretization at the scale required to make DQN viable does not produce an effective solution. Estimation of Q-values for even just 24 actions, compared to 109 used in the MCTS and greedy baselines, results in greatly degraded performance. Solutions produced often degenerated to

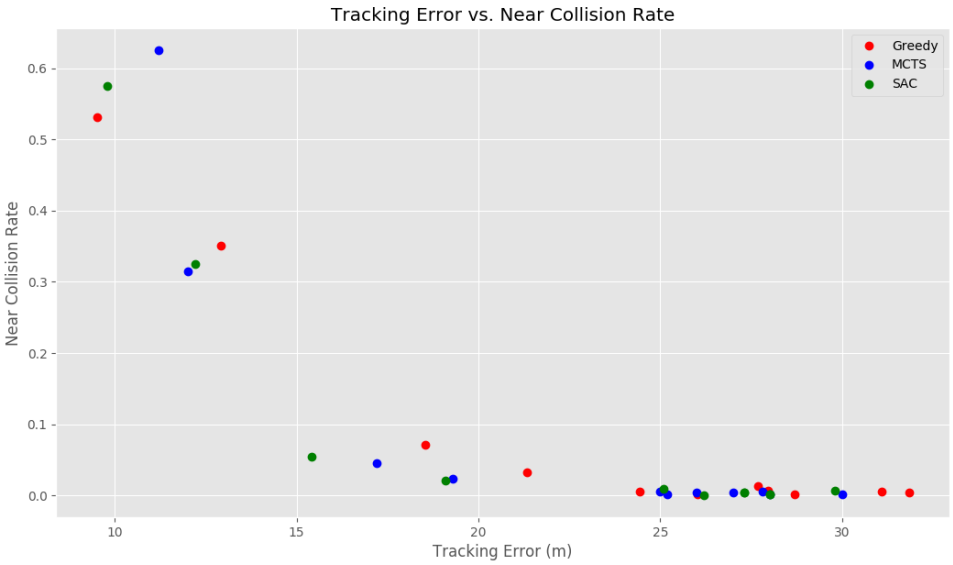


Figure 5.1: MCTS and SAC outperform the greedy controller, but there is no discernible difference between MCTS and SAC themselves. For both axes, lower is better.

constant policies, where a single action is chosen for all states. It's possible that this degradation is because higher fidelity action spaces allow for greater control when maneuvering the drone, and the extremely small action space that discretization necessitates is not enough for good localization. It is also possible that the action space is too large for DQN to effectively optimize over – the arg max term when choosing the best performing action must be done over all 24 actions, and optimizing over this large a search space is difficult.

5.3.2 DDPG and SAC

As both of these algorithms are very similar, we evaluate controller performance with SAC. Comparisons to the MCTS and greedy controllers can be found in 5.1. We find that the continuous RL-based controller trained with SAC performs about as effectively as MCTS. Just as with MCTS, discretization of the particle filter results in low performance in situations with highly penalized near collisions. In situations where near collisions are not penalized nearly as much, all agents behave similarly.

Qualitatively, we find that the reward function induces the expected behavior. The seeker will first travel towards the target, then oscillate its sensors while orbiting from a safe distance. An intersting feature of this oscillation is that the agent prefers to keep the target in the periphery of its

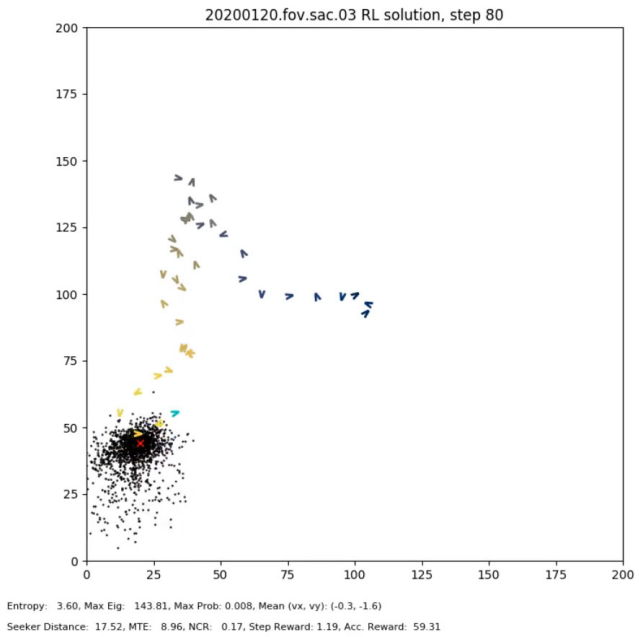


Figure 5.2: One episode of drone localization with SAC.

sensors, as this provides more information than simply keeping the target in front of the seeker. This can be seen in 4.1, where the periphery is a smaller angle than the accurate sensor angle. Oscillating pins the target to a 60° area rather than a 120° area. A video of one episode depicting this behavior can be found here, while a picture depicting the episode with seeker history can be found in 5.2.

Failure cases include instances where the seeker veers too close to the target, resulting in a low tracking error at the expense of a high near collision rate. A rare but interesting failure mode was concentration of particles in a location completely off the target, resulting in particle deprivation. This is possible when particle resampling is not performed often enough, resulting in stagnation of the belief and degeneration to just a few hypotheses. To combat this, more noise was injected into the particle filter update dynamics.

This result is a testament to the difficulty of reinforcement learning. Each trial necessitated the training of a new agent with different hyperparameters controlling the penalization of near collisions. A grid search over the hyperparameter space of the reward function produced a curve that shows similar performance to the MCTS solution. While we did accomplish the goal of moving the bulk of the computation to an offboard computer, it resulted in having to retrain an agent to convergence when slightly different environment details (target speed, seeker speed, starting positions of the seeker or target, etc.) were changed. Additionally, the surrogate reward function provides little interpretability compared to the original reward function. Despite this, the SAC agent still produces

results comparable in performance to the MCTS solver.

Chapter 6

Conclusion

In conclusion, we demonstrate the ability of a reinforcement learning-based controller in the drone localization setting. In doing so, we addressed and solved several major challenges. The sensor system [7] provided little information useful towards localization, so we construct a belief with a particle filter to integrate information over time. Coarse discretization of the action space proved difficult for discrete RL algorithms to handle, so we use the continuous control algorithm SAC instead. The true objective in the localization task is difficult to optimize, so we introduce a surrogate reward function that is more well suited for reinforcement learning. Finally, sample efficiency and logistic issues prevent training on a real drone, so performing training with a simulation environment makes this problem tractable [15]. The result is a controller that performs as well as previous controllers, but with the added benefit of decreased runtime computation at the cost of offline training. Ultimately, we find that moving the planning computation offline results in even greater computational cost when modifying environment parameters. This tradeoff may be acceptable, however, if the seeker platform is severely restricted in computational power and requires as compressed a controller as possible.

Future work will explore the transfer of these policies to actual hardware. Future work may also explore the information-sharing relationship between multiple drones in the multiagent localization scenario, or apply these information-gathering techniques to other tasks. Model-based methods for control such as [16] may also alleviate some of the issues with sample inefficiency and training speed.

Bibliography

[1] Associated Press. Worried about drones flying too close to airports? here are answers to your questions, Jan 2019.

[2] UK Airprox Board. Monthly meeting april 2020, Apr 2020.

[3] Federal Aviation Administration. Airspace restrictions, Dec 2018.

[4] L. Dressel and M. J. Kochenderfer. Hunting drones with other drones: Tracking a moving radio target. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 1905–1912, May 2019.

[5] Kyle D. Julian and Mykel J. Kochenderfer. *Neural Network Guidance for UAVs*. 2017.

[6] L. Dressel and M. J. Kochenderfer. Pseudo-bearing measurements for improved localization of radio sources with multirotor uavs. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6560–6565, May 2018.

[7] Louis Dressel and Mykel J. Kochenderfer. Efficient and low-cost localization of radio signals with a multirotor UAV. *CoRR*, abs/1808.04438, 2018.

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

[9] 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.

[10] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[11] Ziyu Wang, Nando de Freitas, and Marc Lanctot. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015.

[12] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.

[13] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018.

[14] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Soft actor-critic algorithms and applications, 2018.

[15] Cedrick Argueta. Python filter exploration for bearing only localization, 2020. Available at https://github.com/cdrckrgt/PyFEBOL.

[16] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Gabriel Dulac-Arnold, Ian Osband, John Agapiou, Joel Z. Leibo, and Audrunas Gruslys. Deep q-learning from demonstrations, 2017.