UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

REINFORCEMENT LEARNING FOR COGNITIVE PHASED ARRAY RADAR SURVEILLANCE

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

Shane Flandermeyer
Norman, Oklahoma
2023

REINFORCEMENT LEARNING FOR COGNITIVE PHASED ARRAY RADAR SURVEILLANCE

A THESIS APPROVED FOR THE
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

BY THE COMMITTEE CONSISTING OF

Dr. Justin Metcalf, Chair

Dr. Nathan Goodman

Dr. Dean Hougen

© Copyright by SHANE FLANDERMEYER 2023
All Rights Reserved.

Acknowledgments

I am immensely grateful to Dr. Justin Metcalf for the opportunities and support he has provided during my time at the ARRC. I have learned so much about radar systems, signal processing, and machine learning over the last several years, and his mentorship has made the process fun and rewarding. In addition, I want to thank Dr. Nathan Goodman and Dr. Dean Hougen for agreeing to serve on my committee. They have both provided insights that played a key role in shaping this work. The other students at the ARRC have also been an essential part of this process due to their ability to turn coffee and a game of pool into a valuable discussion.

I would also like to thank my fianceé, Cora DeFrancesco. Since meeting her at the beginning of my undergraduate career, she has inspired me to become the best student, engineer, and person I can be. I am lucky to have found someone so caring and genuine, and this would not have been possible without her constant encouragement. I am also grateful to my parents, Tracey Farrington and Doug Flandermeyer, for teaching me the value of education and the importance of hard work.

Finally, I would like to thank my dog, Toast. She has been with me through every step of my graduate school journey, and her constant howling has forced the rest of the neighborhood to be with me as well.

This work was funded by the NSF Graduate Research Fellowship Program.

Table of Contents

Acknowledgments **iv**

Abstract **xi**

1 Introduction **1**

1.1 Overview 1

1.2 Contributions 3

1.3 Document Outline 3

2 Deep Reinforcement Learning **6**

2.1 Neural Networks and Deep Learning 6

2.2 Reinforcement Learning Fundamentals 8

2.2.1 Actions and Policies 13

2.2.2 Rewards 15

2.2.3 Gymnasium 17

2.2.4 Returns and Value Functions 19

2.3 Policy Gradient Algorithms 22

2.3.1 Policy Gradients 23

2.3.2 Proximal Policy Optimization 28

3 Multifunction Radar **32**

3.1 Radar Background 32

3.2 Phased Array Radar . . . 37
3.3 Target Models . . . 41
3.4 Operating Modes . . . 42
3.4.1 Surveillance . . . 42
3.4.2 Tracking . . . 44

4 Particle Swarm Optimization 45
4.1 Particle Swarm Optimization . . . 45
4.1.1 Global Best PSO . . . 47
4.1.2 Local Best PSO . . . 49
4.1.3 Exploration Strategies . . . 52

5 Cognitive Search Agent 54
5.1 Surveillance PSO . . . 54
5.2 Algorithm Description . . . 55
5.2.1 Dispersion Phase . . . 56
5.2.2 Detection Phase . . . 57
5.2.3 Adaptive Dispersion Inertia . . . 61
5.2.4 Results and Analysis . . . 62
5.3 RL Formulation . . . 71
5.3.1 State Representation . . . 71
5.3.2 Action Space . . . 74
5.3.3 Reward Function . . . 74
5.3.4 Results and Analysis . . . 75

6 Conclusions and Future Work 85

References 87

Appendix A The Policy Gradient Theorem **92**

Appendix B Activation Functions **95**

Appendix C List of Acronyms and Abbreviations **99**

List of Tables

5.1 Parameters for the single-cluster scenario 63

5.2 Radar system parameters 64

5.3 Particle swarm parameters 64

5.4 PPO agent parameters 76

List of Figures

2.1 Artificial neuron architecture . . . 7

2.2 A fully-connected neural network . . . 7

2.3 Agent-Environment Cycle . . . 10

2.4 Single frame of pong in an Atari 2600 emulator . . . 12

2.5 Categorical network architecture for discrete action spaces . . . 14

2.6 Gaussian network architecture for continuous action spaces . . . 15

2.7 Actor-Critic neural network representations . . . 27

3.1 Coherent pulse train with $T_{PRI} = 100\,\mu$s . . . 36

3.2 Uniform linear array . . . 38

3.3 Normalized power pattern for a ULA with $\theta_0 = 0$ and constant element spacing $d = \lambda/2$. The center frequency is $f_0 = 3\,$GHz. . . 40

3.4 Uniform rectangular array topology . . . 40

3.5 Raster scan patter . . . 43

4.1 Star topology . . . 48

4.2 Ring topology . . . 49

5.1 Uniformly initialized swarm . . . 56

5.2 Track initiation ratio for a deterministic agent with $\Delta\theta_{az} = \Delta\theta_{el} = 3\,^\circ$. . . . 65

5.3 Track initiation ratio for a deterministic agent with $\Delta\theta_{az} = \Delta\theta_{el} = 5\,^\circ$. 66

5.4 Track initiation ratio for a deterministic agent with $\Delta\theta_{az} = \Delta\theta_{el} = 10\,^\circ$. 67
5.5 Beam coverage history for various target distributions. 70
5.6 Particle distribution after detecting a cluster of targets at $(az, el) = (20\,^\circ, -30\,^\circ)$ 72
5.7 An example of the state space representation 73
5.9 Agent Network Architecture 77
5.8 Track initiation fraction for the RL-based agent. At the start of each dwell, the agent selects a beamwidth between $3\,^\circ$ and $10\,^\circ$. The other agents use a constant transmit beamwidth of $\Delta\theta_{az} = \Delta\theta_{el} = 3\,^\circ$. 78
5.10 Track initiation ratio for the RL-based agent. At the start of each dwell, the agent selects a beamwidth between $3\,^\circ$ and $10\,^\circ$. The other agents use a constant transmit beamwidth of $\Delta\theta_{az} = \Delta\theta_{el} = 5\,^\circ$. 79
5.11 Track initiation ratio for the RL-based agent. At the start of each dwell, the agent selects a beamwidth between $3\,^\circ$ and $10\,^\circ$. The other agents use a constant transmit beamwidth of $\Delta\theta_{az} = \Delta\theta_{el} = 10\,^\circ$. 80
5.12 Mean episode reward across all parallel environments 81
5.13 Mean episode length across all parallel environments 81
5.14 Policy entropy 82
5.15 Policy (actor) loss over two million time steps 83
5.16 Value (critic) loss over two million time steps 83

B.1 Sigmoid activation function 96
B.2 Tanh activation function 96
B.3 ReLU activation function 97
B.4 Softplus activation function 98

Abstract

The proliferation of phased array radar (PAR) has significantly increased the flexibility of radar systems, making it possible to use a single radar to perform a variety of operational modes such as surveillance and tracking that each traditionally required a dedicated system. To fully take advantage of these capabilities, algorithms must be developed to efficiently distribute the radar's finite time, energy, and processing budget between competing tasks. Although many resource management methods exist for tracking applications, it is common to use a fixed strategy (e.g., a raster scan) for the surveillance task. The resulting allocation of resources is often sub-optimal since fixed approaches do not leverage prior knowledge and thus spend a disproportionate amount of time searching regions that are unlikely to contain new targets.

This thesis presents a novel approach to more effectively utilize the radar timeline in surveillance and track initiation tasks. A variant of particle swarm optimization (PSO) is derived to estimate the density of untracked targets in the search volume, which is then used to inform the parameter selection process for each radar dwell. The resulting method, known as Surveillance PSO (SPSO), is computationally efficient and suitable for real-time implementation on a general-purpose CPU or GPU. SPSO is also highly general, making few assumptions about the properties of the target or the underlying radar system. Finally, the output of the algorithm is a constant-length tensor that can be incorporated into sys-

tems that utilize deep learning and reinforcement learning. Two cognitive agents are developed to demonstrate the utility of the SPSO algorithm. The first is a deterministic agent that directly uses the output of the SPSO algorithm to make decisions on where to steer the radar beam at each dwell. The second is reinforcement learning (RL) agent that uses a slight modification of SPSO to simultaneously steer and spoil the transmitted beam based on the current environment. The performance of each agent is evaluated in several simulated surveillance environments, where both are shown to outperform the standard raster scan approach.

Chapter 1

Introduction

1.1 Overview

Due to their flexibility and increasing commercial viability, phased array radar (PAR) systems have become widely used across several domains including defense, weather, and communications. Unlike traditional radar systems which use mechanical rotation to steer the beam, PARs electronically form beams and steer them in a desired direction [1]. This beam steering can be done almost instantaneously, allowing the radar to perform tasks such as surveillance, tracking, and imaging with a single aperture. More recently, digital beamforming methods have gained traction in modern systems, allowing the radar to simultaneously form beams on both transmit and receive [2]. This beam agility makes it possible for the radar to dynamically adapt its behavior based on current operating conditions, rather than being fixed at design time.

Since radar systems have a limited budget of time, energy, and computational resources to distributed amongst the various functions, robust resource management algorithms must be developed to automatically allocate resources such that an acceptable level of performance is reached for each task. To fully take advantage of the flexibility offered by multifunction PAR systems, resource management

architectures have come to play a crucial role in the radar system design process [3]. These algorithms should be applicable in a wide range of scenarios and should introduce minimal computational overhead so they can be used in a real-time radar processing loop.

Typical radar resource management methods optimize parameter selection (e.g., waveform bandwidth or dwell integration time), task selection, and/or task scheduling. A majority of these methods focus on minimizing the resource load of the tracking task [4]. Comparatively few methods exist for optimizing the search for new untracked targets, and existing methods for adaptive search make restrictive assumptions about the kinematics of the targets [5] or the distribution of targets in the scenario [6]. Many of these algorithms optimize radar parameters, but do not inform *where* the radar should steer its beam for surveillance. Methods that attempt to steer the beam usually do so over a coarse angular grid due to the computational complexity of the underlying optimization methods, which limits their utility. This provides motivation for developing new algorithms that relax the assumptions on the surveillance scenario and are computationally efficient for practical use in real systems.

The field of reinforcement learning (RL) is concerned with learning optimal behavior in sequential decision-making problems through trial and error [7]. In recent years, much of the RL community has focused on deep RL, which uses deep neural networks to model the behavior of the learning agent. Deep RL has been successfully applied to a variety of application domains, including playing board games such as Go at a superhuman level [8] and discovering state-of-the-art algorithms for tensor operations [9]. A major challenge in the practical application of deep RL is designing a suitable representation of the environment state. Most RL algorithms assume the environment state representation is Markovian, such that

the entire history of the environment is encoded in the current state. Moreover, the state representation must be a constant-length tensor in order to be passed as an input to a deep neural network.

This thesis analyzes a state representation of the surveillance task that is suitable for training computational agents with deep RL algorithms. The state representation uses a novel variant of particle swarm optimization (PSO) to form an approximate the density of undetected targets in the region under surveillance. This representation is approximately Markovian and is readily expressed as a tensor. In addition, it is computationally efficient and highly parallelized for real-time operation and makes few assumptions about the scenario or targets.

1.2 Contributions

This thesis presents the following novel contributions:

- A PSO-based approach was developed to provide a Markovian state representation for the radar surveillance task.
- Two computational agents were developed that use the PSO output to inform surveillance behavior. The first agent follows a deterministic rule for determining where to steer the beam. The second agent uses an RL algorithm to select beam steering angles and spoiled transmit beamwidths.
- An open-source simulator was developed in Python for simulating a phased array radar system [10]. This simulator was designed to facilitate rapid prototyping of cognitive agents for resource management tasks.

1.3 Document Outline

This document is organized into the following chapters:

Chapter 2 provides an overview of deep reinforcement learning, starting with a brief discussion of deep learning with fully-connected neural networks. Next, the fundamentals of reinforcement learning are described, including the formulation of sequential decision making problems as Markov decision processes (MDPs). This chapter concludes with a discussion on policy gradient algorithms, particularly focusing on proximal policy optimization (PPO).

Chapter 3 focuses on the phased array radar system model. This includes a mathematical description of waveforms and signal processing techniques for pulse-Doppler radar, along with a discussion on the basics of uniform rectangular antenna arrays and the measurement models assumed for this work.

Chapter 4 gives a brief overview of particle swarm optimization algorithms. The global and local best variants of the algorithm are given as simple examples used to motivate the modifications required for use with the surveillance task. This chapter also discusses exploration methods that adds diversity to the trajectories of particles in the swarm.

Chapter 5 describes the novel surveillance PSO (SPSO) algorithm developed to more efficiently manage resources in the radar surveillance task. First, a deterministic variant of the algorithm is presented to steer the radar beams without reinforcement learning, along with an analysis of its performance in a medium-range surveillance environment. Next, an adaptation of the algorithm is developed to inform a cognitive agent that is trained with reinforcement learning. The chapter concludes with an analysis of this agent's performance in the same medium-range surveillance environment.

In Chapter 6, the thesis is concluded and future directions for this research are given.

Appendix A provides a derivation of the policy gradient theorem, which is a

fundamental result that is leveraged by on-policy algorithms such as PPO.

In Appendix B, several activation functions that are commonly used in deep learning are described.

Chapter 2

Deep Reinforcement Learning

2.1 Neural Networks and Deep Learning

Deep learning is a sub-field of machine learning that uses artificial neural networks to perform a variety of tasks [11]. The fundamental building block of a neural network is the artificial neuron, also known as a perceptron, whose structure is given in Fig. 2.1. Given a length-N tensor $\mathbf{x} = [x_0, x_1, \ldots, x_{N-1}]$ as input, the artificial neuron first computes a weighted sum as

$$f(\mathbf{x}; \mathbf{w}) = \sum_{i=0}^{N-1} (w_i x_i) - w_N \tag{2.1}$$

where $\mathbf{w} = [w_0, w_1, \ldots w_N]$ is a set of weights that are typically updated through a learning process, and w_N is a bias term that does not depend on the input data. In order to learn a more general class of functions, it is necessary to apply a nonlinear activation function ϕ to the output of the weighted sum. The final neuron output is then given by

$$y(\mathbf{x}) = \phi(f(\mathbf{x}; \mathbf{w})) \tag{2.2}$$

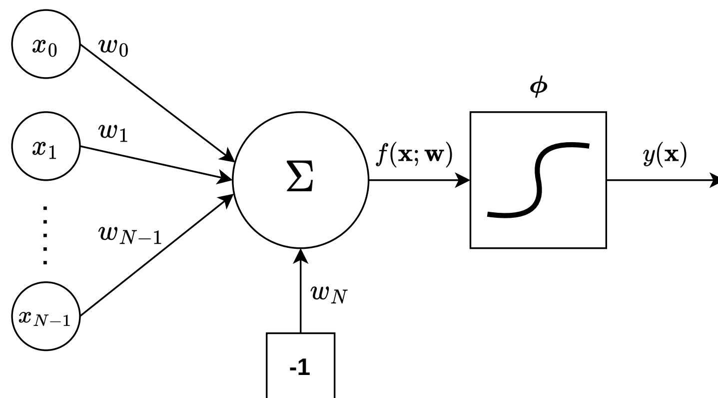


Figure 2.1: Artificial neuron architecture

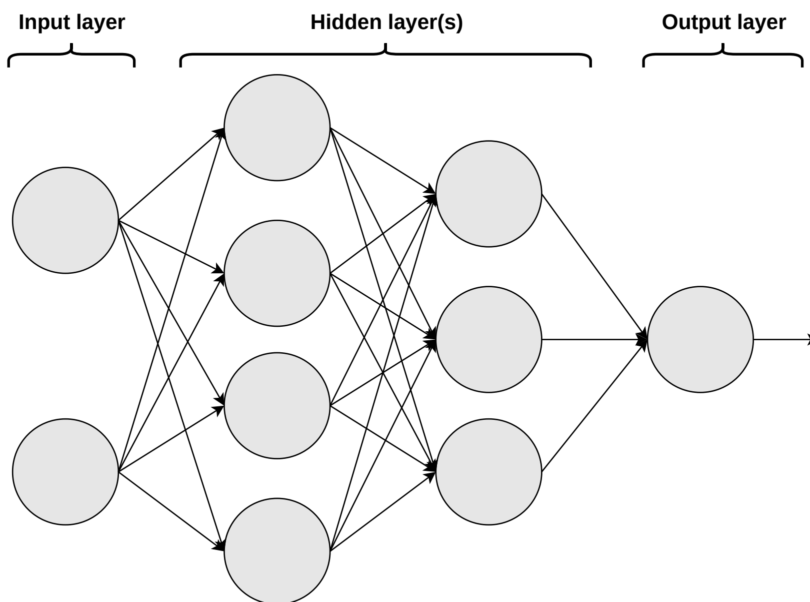


Figure 2.2: A fully-connected neural network

In many applications, the choice of activation function can significantly impact performance. Appendix B discusses some of the most common activation functions in use today.

A single perceptron is of limited use as a function approximator. In practical problems, it quickly becomes necessary to integrate many artificial neurons into a larger structure known as a network. Fig. 2.2 shows an example of a fully-connected neural network. In this architecture, neurons are organized into a series of higher-level structures called layers, and each neuron from a given layer is connected to all neurons from the previous layer to form a directed acyclic graph.

The input layer contains the constant-length tensor $\mathbf{x}$ discussed earlier. Each hidden layer applies a function to the output of the previous layer such that the output of the hidden section is a nested composition of functions applied to the input data. Finally, the output layer performs additional processing to make the output from the hidden layers match the problem at hand. For example, the output layer may convert the hidden output to a set of class probabilities in a classification network. The complexity of functions that can be approximated with this model depends primarily on two factors: the number of layers in the model, known as the model's *depth*, and the number of neurons in the hidden layers, known as the model's *width*.

2.2 Reinforcement Learning Fundamentals

Reinforcement learning (RL) is a sub-field of machine learning that attempts to train computational agents to behave optimally in sequential decision making problems [7]. For these types of problems, the agent can take actions to interact with and influence its environment. In traditional supervised learning, an

external entity provides a dataset of examples to train the agent to generalize to scenarios that it has never seen. Each element in the dataset includes a label, which describes the correct action the agent should take. This allows the agent to improve its performance by directly minimizing some error metric which measures the difference between the agent's action and the optimal action.

In problems where the agent can interact with its environment, it is rarely practical to collect and label a representative dataset to use for training. Moreover, for complex problems the optimal action is not known a priori. Rather than being told the correct action to take at each time step, reinforcement learning agents receive a numerical signal from the environment known as a *reward* that describes how good their actions are, and modify their behavior based on their experiences to maximize the cumulative reward over the time horizon of the problem. The trial-and-error nature of this approach presents a fundamental trade-off between exploration and exploitation: to maximize its reward, an agent should perform actions that worked well in the past. However, the agent must experiment with actions that may be sub-optimal to identify good actions in the first place. Without exploration, the quality of an agent's actions will converge to a local minima. Without exploitation, the agent will do a poor job of performing the required task. This is further complicated in stochastic environments, where state-action pairs may need to be sampled several times to get a good estimate of their value.

The feedback loop between an agent and its environment, known as an agent-environment cycle (AEC), is shown in Fig. 2.3. At discrete time step t, the agent observes o_t, a potentially incomplete representation of the environment's true state s_t. If o_t excludes relevant information about s_t, the environment is *partially observable*. Otherwise, it is *fully observable*. In the general RL formulation, there

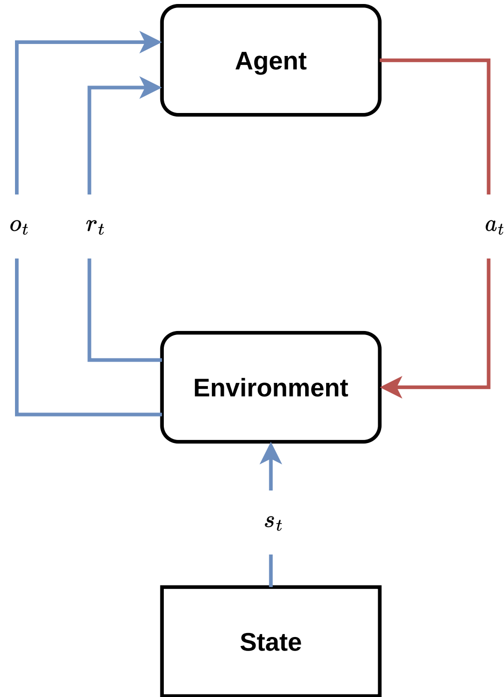


Figure 2.3: Agent-Environment Cycle

is no limitation to how states and observations can be represented. However, most deep RL algorithms require that the input be a constant-size tensor that can be passed as the input to a neural network (e.g., a three-dimensional array representing pixel values in an RGB image or a feature vector that gives the current angles of various joints in a robot).

Using information from o_t, the agent takes an action a_t and receives a reward r_t as a result of its action. Repeated interactions with the environment give rise to a trajectory τ describing the sequence of states, actions, and rewards over time:

$$\tau = (s_0, a_0, r_0), (s_1, a_1, r_1), \ldots \tag{2.3}$$

In an *episodic* task, τ is a finite sequence that continues until the environment reaches a terminal state. In a *continuing* task, there are no clear terminal states and the sequence lasts for the potentially infinite lifetime of the agent.

Most RL algorithms assume that the environment is a Markov decision process (MDP) and thus satisfies the Markov property, which means that the entire history of interactions between the agent and environment are encoded in the current state. In other words, the system is memoryless and the dynamics of the environment can be modeled with a probability function $p : \mathcal{S} \times \mathcal{A} \times \mathcal{R} \times \mathcal{S} \rightarrow [0, 1]$,

$$p(s', r|s, a) = Pr\left(s_{t+1} = s', r_{t+1} = r|s_t = s, A_t = a\right) \tag{2.4}$$

In many cases, it is simpler to analyze an MDP in terms of its state transition density, which is a function $p : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ that defines the probability of transitioning to state s' after taking action a in state s (independent of the reward received),

$$p(s'|s, a) = Pr\left(s_{t+1} = s'|s_t = s, A_t = a\right) = \sum_{r \in \mathcal{R}} p(s', r|s, a) \tag{2.5}$$

Violation of the Markov property can significantly alter an agent's behavior and often cause training to not converge at all. For example, consider some possible MDP representations for the game of Pong. A naive approach would be to represent the game state as the current frame as shown in Fig. 2.4.

Figure 2.4: Single frame of pong in an Atari 2600 emulator

Although the position of the ball can be determined from this representation, the ball's velocity is ambiguous and so the environment is partially observable. If the game state is instead represented by a stack of frames that includes the current and previous frames (e.g., the previous four frames), then the velocity can be resolved and the environment becomes fully observable. The game of Pong is simple enough that agent's performance is not significantly impacted by frame stacking, but in many cases learning becomes impossible if history is not encoded [12]. A major limitation of the frame stacking approach is that environment dynamics that persist for longer than the stack length appear non-Markovian to the agent. This can be mitigated by employing recurrent layers such as long short-term memory (LSTM) layers in the policy network as in [13].

2.2.1 Actions and Policies

After observing the environment, the agent can influence the next state by taking some action a_t. The set of actions that an agent may take is known as the action space, which depends on the environment and the problem to be solved. If the action space is discrete, the agent selects one or more actions from the set $\mathcal{A} = \{A_1, A_2, \ldots, A_N\}$. If the action space is continuous, each component of the agent's action is a real number. A number of approaches have also been proposed to handle hybrid action spaces ([14], [15]) containing both discrete and continuous components.

The goal of any RL-based agent is to use repeated interaction with the environment to gather experience and learn to select actions that maximize reward in each state. These actions are selected from a function known as a *policy*. If the policy is deterministic (e.g., the output of a neural network), it can be described as a simple function $\pi(s) : \mathcal{S} \mapsto \mathcal{A}$ for some state s. If the policy is stochastic, it is described as a probability distribution over actions conditioned on the current state:

$$\pi(a|s;\theta) = Pr\left(a_t = a|s_t = s, \theta_t = \theta\right) \tag{2.6}$$

where $\theta \in \mathbb{R}^d$ are the parameters of the policy representation. The action at each time step is determined by drawing a sample from the policy distribution. In many practical cases, policies are chosen to be stochastic during training (to encourage exploration) and deterministic during evaluation.

If the action space is discrete, each action can be represented as a category in a categorical distribution. For action indices $\{1, \ldots, N\}$, the probability mass function (pmf) is given by

$$f(x = i) = p_i \tag{2.7}$$

where $i \in \{1, 2, \ldots, N\}$, p_i is the probability of taking action i, and $\sum_{i=0}^{N} p_i = 1$. In deep RL approaches, a neural network is used to compute the category probabilities as shown in Fig. 2.5, where the output logit for each action represents the preference for that action, and a softmax activation function is used to normalize these preferences to create a valid probability distribution. The dotted lines are used to indicate that the structure of the feature abstraction layers are arbitrary. Unlike the value-based methods that will be discussed in Section 2.2.4, this parameterization can converge to a deterministic *or* stochastic policy, depending on which best models the optimal behavior.

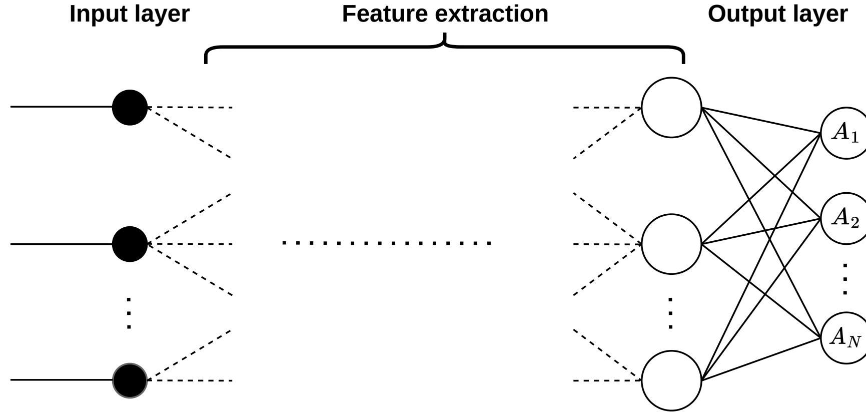


Figure 2.5: Categorical network architecture for discrete action spaces

For continuous action spaces, it is common to parameterize each component of the policy in terms of the mean $\mu(s;\theta)$ and variance $\sigma^2(s;\theta)$ (or standard deviation $\sigma(s;\theta)$) of a normal distribution, where s is the input state. At each time step, actions are sampled according to

$$a_t \sim \mathcal{N}\left(\mu(s;\theta), \sigma^2(s;\theta)\right) \tag{2.8}$$

In the architecture in Fig. 2.6 (which is used in the beam steering agent described in Ch. 5), the mean and variance are estimated with separate network

branches that share a latent feature space, which reduces the number of parameters that the network must learn. Alternatively, the variance of each action can be computed as a network parameter that is independent of the input state. Recent work has also shown that sampling from bounded probability distributions such as a beta function instead of a Gaussian can improve performance in bounded action spaces by reducing the model's bias at the edges of the valid action range [16].

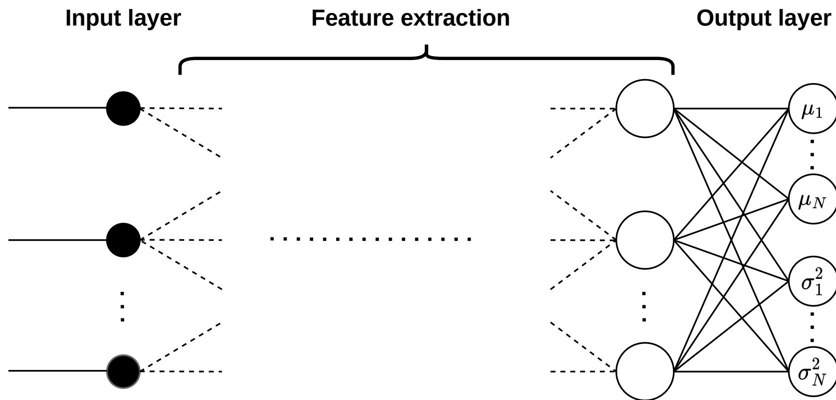


Figure 2.6: Gaussian network architecture for continuous action spaces

2.2.2 Rewards

At each time step, the agent receives a scalar reward signal $r_t \in \mathbb{R}$ that provides feedback on the agent's most recent action. A positive reward indicates that the agent's behavior aligns with its goals, while a negative reward suggests the opposite. The overall goal of the agent is described by a reward function R, which depends on the current state, the action taken, and the resulting next state:

$$r_t = R(s_t, a_t, s_{t+1}) \tag{2.9}$$

The reward function bridges the gap between the agent's learned behavior

and the intentions of the algorithm designer, so it is a critical consideration when creating an environment for RL experiments. If the agent's goal is easy to describe, designing the reward function is straightforward. For example, an agent learning to play arcade games may receive a reward proportional to its game score, while an agent playing a competitive game may receive a positive reward for winning and a negative reward for losing. Reward function design becomes much more difficult in complex problems where the agent must balance multiple tasks or where the optimal outcome cannot be described easily if at all. In these cases, a trial-and-error approach is usually taken to iteratively tweak the reward function until the agent can consistently learn the desired behavior in a reasonable amount of time.

Another challenge in reward function design is the *sparsity* of the reward signal. In sparse reward environments, only a small fraction of state-action pairs produce state transitions that give non-zero reward. This causes the agent to wander aimlessly without making progress towards the goal until it reaches a rewarding state enough times to learn useful patterns. It is possible to mitigate this issue by augmenting the reward signal with supplemental rewards for solving sub-problems that (in principle) move the agent closer to the main goal. Extreme care must be taken to ensure that the agent cannot "cheat" and receive a high reward for solving the supplemental tasks without learning to reach the main goal. As a concrete example, a common reward function for a chess-playing agent is:

$$R = \begin{cases} 1 & \text{if agent wins} \\ 0 & \text{if agent draws} \\ -1 & \text{if agent loses} \end{cases} \tag{2.10}$$

This function ensures that the agent is only rewarded for achieving the goal intended by its designer: to win games. However, it is sparse since the agent only

receives feedback on the last turn of every game. Adding an additional reward component proportional to the value of each captured piece increases the density of rewards over the course of the game, but may cause the agent to focus solely on capturing pieces instead of winning the game. As a general rule, the reward function should encode *what* the agent should achieve rather than *how* it should achieve it. Numerous alternative approaches for handling sparse reward scenarios have been shown to work well in practice, such as reward shaping [17] and more recently curiosity-based intrinsic rewards ([18], [19]).

2.2.3 Gymnasium

Gymnasium (formerly known as OpenAI Gym) is an open-source Python package that provides a standardized interface for formulating reinforcement learning problems as Markov decision processes. Each learning environment inherits from the `Env` class and defines the following methods:

- `step()`: Takes the agent's action as input and propagates the environment to the next time step. This method returns the observation after taking the input action, the reward scalar, the termination signal (a boolean value that is true if the agent has reached a terminal/ending state and false otherwise), the truncation signal (which is true if the environment terminates for reasons that are not part of the MDP formulation, such as a maximum time step limit), and a dictionary of additional information such as performance metrics.

- `reset()`: Ends the current episode and samples a new observation from the environment's initial state distribution, then returns the new initial state and an information dictionary as in `step()`.

- `render()` (optional): Displays the agent's observations in a human-readable format (e.g., animations or text) when the appropriate rendering mode is enabled. This method is primarily used to debug the state space representation and to view the performance of trained agents.

- `close()`: Frees resources that were being utilized by the environment. This is only necessary in environments that use additional software such as game engines, physics simulators, or file system resources.

With these methods, an agent can interact with its environment without exposing any implementation details about the environment or agent. The code snippet in listing 2.1 gives an example of how a single episode of the agent-environment cycle is simulated for the classic Cart Pole environment. If `render_mode` is instead set to `'human'`, a game screen such as the one in Fig. 2.4 will be displayed.

```
import gymnasium as gym
env = gym.make("CartPole-v1", render_mode='rgb_array')
obs, info = env.reset()
done = False
while not done:
    # Agent samples actions from a learned policy
    action = agent.act(obs)
    obs, reward, terminated, truncated, info = env.step(action)
    done = terminated or truncated
env.close()
```

Listing 2.1: Gymnasium representation of the agent-environment cycle

Gymnasium also provides a set of objects for defining the state and action space for environments as described in Section 2.2.1. For example, discrete action/observation spaces are defined by the `Discrete` class, while continuous spaces are defined by the `Box` class. Mixed discrete/continuous spaces are also supported

and can be represented as tuples and dictionaries.

2.2.4 Returns and Value Functions

As stated earlier, the goal of an RL algorithm is to maximize the future rewards received by the agent. This can be generalized so that the agent aims to instead maximize some function of the cumulative reward, known as the return G_t. In the simplest case of a task that lasts for T time steps starting at time t, the return can be defined as the sum of all rewards in the trajectory τ

$$G_t(\tau) = r_t + r_{t+1} + \cdots + r_{t+T} = \sum_{k=t}^{t+T} r_k \tag{2.11}$$

This is a reasonable quantity to maximize for episodic tasks, but may not converge for continuing tasks whose time horizon extends to $T = \infty$. It is therefore more common to maximize the discounted return, which exponentially decays each reward term by a discount factor $\gamma \in [0, 1]$ based on how far each reward is in the future

$$G_t(\tau) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{2.12}$$

Intuitively, a discount factor conveys the idea of the time value of currency, a fundamental principle in economics that states that money today is worth more than the same amount of money in the future, all else equal. A discount factor of 1 indicates that the agent values present and future rewards equally, while a discount factor of 0 indicates that the agent is myopic and only cares about the reward from the current time step.

A formal definition of return makes it possible to quantify the utility of a given state or state-action pair in terms of expected return. The value function $V_\pi(s)$ of a policy π and state s gives the expected return if the agent starts in state s

and acts according to π thereafter

$$
\begin{aligned}
V_\pi(s) &= \mathbb{E}_{\tau\sim\pi}\left[G_t | s_t = s\right] \\
&= \mathbb{E}_{\tau\sim\pi}\left[r_t + \gamma G_{t+1} | s_t = s\right] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s'|s,a)\left[r + \gamma \mathbb{E}_\pi\left[G_{t+1}|s_{t+1} = s'\right]\right] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s'|s,a)\left[r + \gamma V_\pi(s')\right] \\
&= \mathbb{E}_{a\sim\pi, s'\sim p}\left[r + \gamma V_\pi(s')\right]
\end{aligned}
\tag{2.13}
$$

where G_t is assumed to be the infinite-horizon discounted return defined in (2.12) and p is the environment's state transition function defined in (2.5). Similarly, the action-value function $Q_\pi(s,a)$ provides the expected return if the agent starts in state s, takes action a, then acts according to π

$$
\begin{aligned}
Q_\pi(s,a) &= \mathbb{E}_{\tau\sim\pi}\left[G_t | s_t = s, a_t = a\right] \\
&= \mathbb{E}_{s'\sim p}\left[r + \gamma E_{a'\sim\pi}\left[Q_\pi(s', a')\right]\right]
\end{aligned}
\tag{2.14}
$$

where the second expression follows from the same process as in (2.13). The final lines in these equations are known as Bellman equations, which play a key role in many RL algorithms. Intuitively, the Bellman equation states that the expected return for a given state is the reward received for taking action a in that state plus the (discounted) expected return if policy π is taken from state s' onward.

Given an optimal policy $\pi^*(s)$ which maximizes the expected return in all states, the optimal value function is defined as

$$
\begin{aligned}
V^*(s) &= \max_\pi \mathbb{E}_{\tau\sim\pi}\left[G_t | s_t = s\right] \\
&= \mathbb{E}_{s'\sim p, a\sim\pi^*}\left[r + \gamma V^*(s') | s_t = s, a_t = a\right]
\end{aligned}
\tag{2.15}
$$

and the optimal action-value function is

$$\begin{aligned} Q^*(s,a) &= \max_{\pi} \mathbb{E}_{t\sim\pi}\left[G_t | s_t = s, a_t = a\right] \\ &= \mathbb{E}_{s'\sim p}\left[r + \gamma \max_{a'} Q^*(s',a') | s_t = s, a_t = a\right] \end{aligned} \tag{2.16}$$

Therefore, the optimal value and action-value functions also obey the Bellman equations. A class of RL algorithms known as value-based methods use the Bellman equations to iteratively improve an estimate of (2.15) and/or (2.16). For example, the one-step Q-learning algorithm iteratively updates an estimate of $Q^*(s,a)$ as

$$Q_{i+1}(s,a) = \mathbb{E}_{s'\sim p}\left[r + \gamma \max_{a'} Q_i(s',a') | s_t = s, a_t = a\right] \tag{2.17}$$

which can be shown to converge to Q^* as $i \to \infty$, provided the environment is stationary and each state-action pair is sampled infinitely many times [7]. Finding Q^* makes it possible to compute the best action in each state as

$$a^*(s) = \arg\max_{a} Q^*(s,a) \tag{2.18}$$

A popular variant of this algorithm developed in [12] used a deep Q-network (DQN) to learn to play Atari games directly from on-screen pixels[1]. The neural network architecture for a DQN is similar to Fig. 2.5, but the DQN outputs Q-values rather than probabilities for each action. The objective function is designed so that the network minimizes the difference between its output and the Bellman update in (2.17)

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\pi}\left[(y_i - Q(s,a;\theta_i)^2\right] \tag{2.19}$$

[1]The paper also introduced a number of important ideas that made deep RL practical for off-policy algorithms, such as replay buffers and target networks. However, these are beyond the scope of this work and are not discussed further.

where θ_i are the neural network parameters at iteration i and y_i is the Bellman target given by the right-hand side of (2.17).

One major limitation of this approach is that it is not readily extensible to continuous action spaces. For discrete action spaces, the maximization in (2.18) can be performed by comparing all possible actions and selecting the action with the highest action-value. If the actions are continuous, however, a separate optimization step is needed to compute the maximum over actions. In most problems, $Q(s, a)$ is highly complex and the maximization is not feasible. The authors of [20] propose the deep deterministic policy gradient (DDPG) class of algorithms, which extends the DQN approach by combining Q-learning with the actor-critic architecture described in Section 2.3.1 to support continuous action spaces in a manner that is computationally tractable.

In many cases, the true value of a state-action pair is less important than the value of the action relative to other actions in that state. The function describing this relationship is known as an advantage function in the RL literature and can be computed as

$$A(s, a) = Q(s, a) - V(s) \tag{2.20}$$

In other words, the advantage quantifies the extra reward that an agent can expect to receive by taking action a in state s compared to the average reward in that state. A positive advantage indicates that the action is above average. In the next section, it will be shown that the advantage function plays an important role in policy gradient methods.

2.3 Policy Gradient Algorithms

The previous section described several RL algorithms which used estimated value (or action-value functions) to select actions. The policy defining the agent's be-

havior was only optimized indirectly through selecting the action that maximizes the value function. Policy optimization methods provide an alternative approach, attempting to learn the parameters of a function representing the optimal policy directly. In many problems, the policy may be simpler to learn than the value function. Moreover, a policy-based approach can learn arbitrary action distributions (including deterministic strategies), while value-based approaches can generally only learn probabilities proportional to the value of each state-action pair. Although policy gradient methods do not use value functions to select actions, many algorithms use them to improve speed and stability in the learning process. Algorithms that take this approach are known as policy gradient methods, and algorithms that learn a value function in addition to a policy are called actor-critic methods.

2.3.1 Policy Gradients

The goal of RL is to learn a policy π that maximizes a performance metric $J(\pi)$, usually some form of return, over a potentially infinite time horizon (see Section 2.2.4). If π is represented by the parameters θ of a function approximator like a neural network, it is desirable to iteratively update θ to improve performance as the agent gains experience. This can be done by formulating parameter updates as a gradient ascent, which is highly desirable since gradient ascent algorithms provide strong converge guarantees under certain conditions and make it possible to handle optimization of the policy and its function approximator in a unifed manner (e.g., by using the gradient to train a neural network with backpropagation [21]). The parameter update logic for iteration i is then given by

$$\theta_{i+1} = \theta_i + \alpha \widehat{\nabla_{\theta_i} J(\theta_i)} \tag{2.21}$$

where α is the learning rate parameter that controls how quickly the policy parameters change and $\widehat{\nabla_\theta J(\theta_t)}$ is a stochastic estimate of the gradient of performance J with respect to the current policy parameters θ. Thus, (2.21) moves the parameter vector of the policy in the direction that increases performance on each iteration.

To use (2.21) in the learning process, an expression for the policy gradient estimator is needed. Estimating the gradient $\nabla_\theta J(\pi_\theta)$ is a challenge since J depends on both the actions taken by the agent and the distribution of states as the agent takes these actions. While the actions are directly determined by the policy, the state distribution is only indirectly impacted by the policy through the selection of actions. Moreover, the relationship between the policy parameters and the state distribution depends on environment dynamics that are typically considered to be unknown. The policy gradient theorem (derived in Appendix A) provides an analytic expression for the gradient that is independent of the environment dynamics, depending only on the current policy π_θ and on some function $\Psi_t(\tau)$ that depends on the return $G_t(\tau)$

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_{t=0}^{T} \Psi_t(\tau) \nabla_\theta \log \pi_\theta(a_t|s_t)\right] \tag{2.22}$$

where T is the number of time steps in the trajectory. Since (2.22) takes the form of an expectation, it can be estimated with a sample mean to compute $\widehat{\nabla_\theta J(\theta_t)}$. Equation (2.22) has a very intuitive interpretation: the policy parameters should be adjusted so that actions giving above-average returns are more likely to be chosen and actions giving below-average returns are less likely. The amount that the gradient should be modified depends on the log-probability, or how likely it is that the action is selected in the first place.

The choice of $\Psi_t(\tau)$ determines the bias and variance of the estimator. The

REINFORCE algorithm [22], one of the first policy gradient methods, uses the action-value function as a baselines so that $\Psi_t = Q_{\pi_\theta}(s_t, a_t)$. However, it has been shown that using the advantage function in (2.20) instead produces an estimator with the same bias but lower variance, reducing the number of samples required to estimate the gradient. Since the advantage is not known in advance, it must be estimated during training. In [23], a technique known as generalized advantage estimation (GAE) is derived that introduces a parameter $\lambda \in [0, 1]$ to smoothly trade off bias and variance. The GAE estimator is given by

$$\hat{A}_t^{GAE(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V \tag{2.23}$$

where γ is a standard discount factor and $\delta_t^V = \left(r_t + \gamma\hat{V}(s_{t+1})) - \hat{V}(s_t)\right)$ is the Bellman residual for approximate value function V, which gives the difference between the Bellman prediction and the actual value given by V. If $\lambda = 1$,

$$\hat{A}_t = \sum_{l=0}^{\infty} (\gamma)^l \delta_{t+l}^V = \sum_{l=0}^{\infty} \gamma^l r_{t+l} - \hat{V}(s_t)$$

In this case, the bias due to the approximation of $\hat{V}$ is negligible, but the variance is high due to the sum of reward terms (which are random variables with nonzero variance). When $\lambda = 0$,

$$\hat{A}_t = \delta_t^V = (r_t + \gamma\hat{V}(s_{t+1})) - \hat{V}(s_t).$$

Since this estimate is heavily dependent on the value function estimate $\hat{V}$, it induces bias in the advantage estimate unless the value estimate is correct. However, it has much lower variance since it only relies on one stochastic state transition to obtain r_t.

Methods for which $\hat{V}$ are learned alongside the policy are known as actor-critic methods [24]. In these methods, the policy learner is known as the actor and the value estimator is known as the critic. Two common neural network topologies used for actor-critic agents are shown in Fig. 2.7. In Fig. 2.7a, the actor and critic modules learn from independently-trained hidden layers. The policy is updated using the policy gradient from (2.22), and the value function is often updated by minimizing the mean-squared error (MSE) between the value estimate and the true return over the episode. Letting θ_π and θ_V represent the parameters of the actor and critic branch, respectively, the value loss is

$$L^{VF}(\theta) = \hat{\mathbb{E}}\left[(V_{\theta_V}(s_t) - G_t(\tau))^2\right] \tag{2.24}$$

and the policy loss is

$$L^{PG} = \hat{\mathbb{E}}\left[\hat{A}_t \log \pi_{\theta_\pi}(a_t|s_t)\right] \tag{2.25}$$

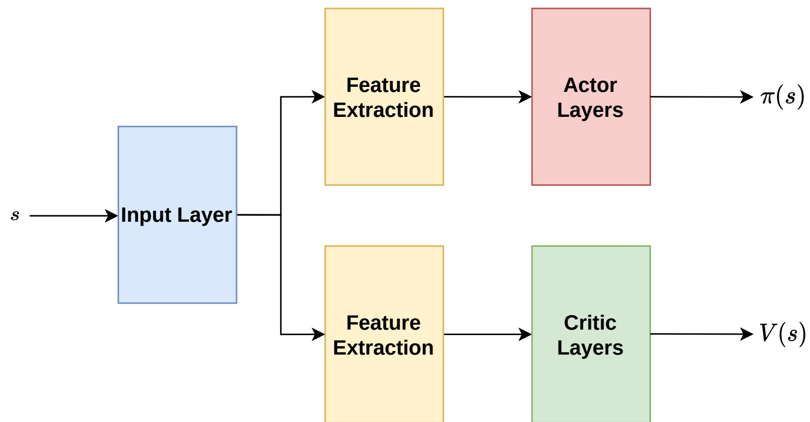
This approach ensures that the value and policy losses are computed and back-propagated independently so that they do not interfere with each other. However, features that are useful for both the actor and critic must be learned separately for both, which may increase the time required to train the agent.

Alternatively, the latent feature space can be shared as shown in Fig. 2.7b. In this case, the actor and critic both optimize the same parameters, so their contributions to the loss are not separable. To account for this, a single loss function is defined to account for both the policy and value objectives. The simplest way to do this is with a weighted sum that is maximized at every iteration

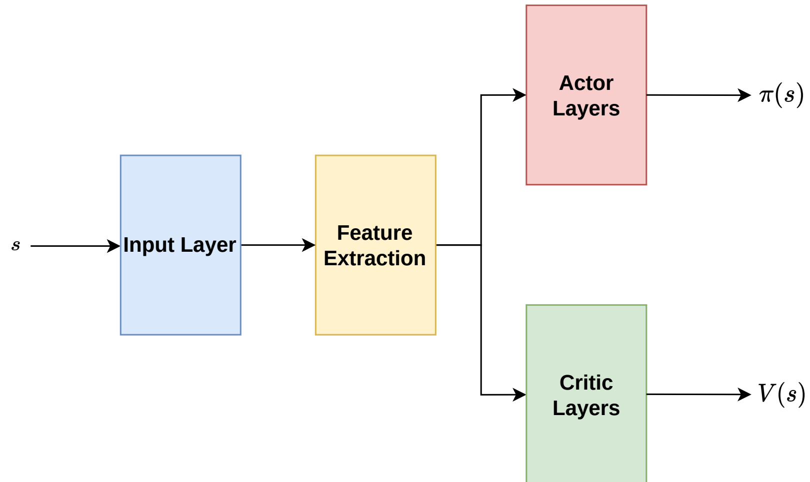
$$L^{PG+VF+H} = \hat{\mathbb{E}}\left[L^{PG} - c_1 L^{VF} + c_2 H[\pi_\theta](s_t)\right] \tag{2.26}$$

where H, the entropy of the current policy, has been added to the objective to

encourage the agent to explore the policy space. The scale factors c_1 and c_2 determine the relative weight of the value loss and entropy relative to the policy gradient loss. c_1 is problem-dependent, and must be set such that the scale of the policy and value loss are similar. This approach helps reduce the number of parameters that the agent must learn and is common in architectures using convolutional neural networks (CNNs) where there is likely to be a large overlap between input features that benefit the actor and critic.



(a) Separate feature spaces



(b) Shared feature space

Figure 2.7: Actor-Critic neural network representations

2.3.2 Proximal Policy Optimization

In RL, the most demanding part of training an agent is often its repeated interactions with the environment. For many problems, complex simulations must run for millions of frames to achieve desired performance levels. In domains like robotics, this process can become even slower if training must be performed on real systems. Therefore, it is highly desirable to make efficient use of data collected from the environment. Since the Bellman equations do not depend on the policy used to collect trajectories, algorithms that optimize a Bellman loss function can store past experiences in a replay buffer for use in future optimization steps. The policy gradient family of algorithms, on the other hand, make the restrictive assumption that the policy being optimized is the same policy that was used to collect the training data. When data is reused in the "vanilla" policy gradient approach, changes in the policy become destructively large and performance suffers.

Proximal policy optimization (PPO) algorithms [25] are an extension of trust region policy optimization (TRPO) methods [23] that relax this assumption and allow data reuse in successive optimization steps as long as the policy does not change too drastically. The most common PPO variant optimizes a clipped objective function L^{CLIP}:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min\left(\hat{A}_t r_t(\theta), \hat{A}_t \text{clip}(r_t(\theta), \epsilon\right)\right] \tag{2.27}$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ is the likelihood ratio between a policy parameterized by θ and the original policy, $\hat{A}_t$ is the estimated advantage for the state-action pair,

and the clip function keeps r_t in the range $[1-\epsilon, 1+\epsilon]$ as

$$\text{clip}(x,\epsilon) = \begin{cases} 1-\epsilon & x \leq 1-\epsilon \\ x & 1-\epsilon < x < 1+\epsilon \\ 1+\epsilon & x \geq 1+\epsilon \end{cases} \tag{2.28}$$

The objective in (2.27) improves stability by removing the incentive for drastic policy changes, and ϵ determines the degree to which the policy can change while still improving the objective. If ϵ is large, larger changes can be made to the policy on each update step. The minimum is used to make the clipped objective performance a *lower* bound on the unclipped objective, which helps to mitigate overestimation bias in the policy updates. With the loss function defined, the PPO algorithm can be implemented as in Algorithm 1.

Algorithm 1 Actor-Critic PPO

```
1:  Instantiate N agents running in N parallel environments
2:  Define a time horizon T for advantage estimation
3:  Define a mini-batch size M ≤ NT for each epoch
4:  while not done do
5:      for each actor i = 1, 2, ..., N do
6:          Run T time steps in the environment using policy π_θold
7:          Estimate advantage Â_1, ..., Â_T for each time step
8:      end for
9:      for each epoch i = 1, 2, ..., K do
10:         for each minibatch do
11:             Compute surrogate policy loss wrt θ_π using (2.27)
12:             Compute value loss wrt θ_V using (2.24)
    // Parameter updates may be shared
13:             Update actor network parameters: θ_π,old ← θ_π
14:             Update critic network parameters: θ_V,old ← θ_V
15:         end for
16:     end for
17: end while
```

The performance of the PPO algorithm is highly sensitive to the selected hyperparameters and subtle implementation details such as the weight initialization scheme in the underlying neural network, so the most relevant PPO hyperparameters are summarized below. A more detailed overview of important design decisions in on-policy algorithms such as PPO is given in [26].

- Number of parallel environments: To increase the diversity of scenarios experienced by the agent, it is common to collect data from N_{env} parallel environments simultaneously. In many learning tasks, the sample efficiency (i.e., the number of environment steps needed for convergence) decreases as the number of parallel environments increases. However, this is usually offset by the increase in speed due to parallelization. Therefore, it is common to choose a number of environments equal to the number of CPU cores available.

- Number of environment steps per rollout: In each parallel environment, N_{step} interactions are performed and the resulting state transitions are collected into a batch that is used to train the agent. This parameter is highly problem-dependent and has a significant impact on the agent's final performance. In general, more environment steps are necessary if the problem is complex or the agent's actions have consequences over extended timescales.

- Number of gradient epochs: Once a batch of experiences has been collected from each environment, K training epochs are performed to update the actor and critic model weights. Common values for this parameter range from 3 to 10. The work in [27] suggests that the value network can tolerate a higher level of sample reuse than the policy network, and introduces the Phasic Policy Gradient (PPG) method to take advantage of this.

- Minibatch size: During each gradient epoch, the batch of experiences is divided into M minibatches. Larger batch sizes are preferred to maximize the throughput for accelerator hardware such as GPUs.

- Discount factor γ: The discount factor determines how heavily future rewards are weighted in the computation of the return and advantage. For most problems, $\gamma = 0.99$ has been found to work well. However, problems that require a large N_{step} may benefit from a larger γ such as 0.999.

- GAE coefficient λ: As discussed in Section 2.3.1, λ is a parameter that can be tuned to smoothly trade off the bias and variance of an advantage estimator. For the learning tasks considered in [23], $\lambda \in [0.92, 0.98]$ was found to perform well, and 0.95 is commonly chosen as an initial guess before performing more in-depth hyperparameter tuning.

- Clip fraction ϵ: This parameter determines how much the policy can change due to a single minibatch of environment transitions. In most environments, $\epsilon \in [0.1, 0.3]$ offers good performance.

Chapter 3

Multifunction Radar

3.1 Radar Background

Radar systems transmit electromagnetic energy from one or more radiating antenna elements, then measure the portion of that energy that is reflected back towards the radar from objects in the environment. The emitted signal is modeled as a real-valued, bandlimited sinusoid of the form

$$x(t) = a(t) \cos \left[2\pi f_0 t + \theta(t)\right] \tag{3.1}$$

where f_0 is the center or carrier frequency of the signal, $a(t)$ is the amplitude modulation function, and $\theta(t)$ is the phase modulation. This signal is generated by modulating a baseband signal, known as a waveform, from DC to the carrier frequency. Waveform design rarely depends on f_0, so it is common to express waveforms in terms of their frequency-agnostic complex envelope $\tilde{x}(t)$, given by

$$\tilde{x}(t) = a(t) \exp \left[\theta(t)\right] \tag{3.2}$$

In radar systems that utilize high-power amplifiers, $a(t)$ must be rectangular to maximize power efficiency by operating in the amplifier's saturation region.

Therefore, $\theta(t)$ is the primary degree of freedom to consider when designing the waveform. Once $x(t)$ is transmitted, it propagates spherically from the radiating antenna element(s) at the speed of light c until it reflects off of objects in the environment, and the radar measures the portion of the reflected energy that is incident on the receive aperture. In practice, the received signal may consist of many components, including clutter, noise, and interference from other emitters near the carrier frequency. Here, it is assumed that the received signal is a scaled and shifted copy of the transmitted waveform corrupted by thermal noise. The received signal is thus modeled as

$$y(t) = \hat{a}(t-\tau_d)\exp\left[j(2\pi f_0(t-\tau_d) + \theta(t-\tau_d) + \phi(t))\right] + n(t) \tag{3.3}$$

where τ_d is the two-way propagation delay between transmission and reception, $\phi(t)$ is an additional phase modulation term from the scatterer, and $n(t)$ is additive Gaussian noise from the receiver. Noting that the signal propagates at a constant speed and assuming the transmitter and receiver are co-located in a monostatic configuration, the propagation delay can be used to determine the radial distance of a scatterer from the radar (i.e., the range) as

$$R = \frac{c\tau_d}{2} \tag{3.4}$$

The accuracy with which the range can be determined is known as the range resolution ΔR, which determines the minimum separation in range required to disambiguate multiple scatterers. The range resolution is inversely proportional to the waveform bandwidth B in Hz

$$\Delta R = \frac{c}{2B} \tag{3.5}$$

When processing radar data, L discrete samples from each transmitted pulse are collected and used to extract range information from the environment. For a pulse that is transmitted at time $t = 0$, samples $l = 0, 1, \ldots, L - 1$ correspond to delays $\tau_d = t_{min} + lT_s$, where T_s is the sampling period of the receiver's analog-to-digital converter (ADC). The t_{min} term is necessary in most monostatic configurations for which sampling does not begin at $t = 0$ because the receiver is isolated while each pulse is being transmitted. These delays are converted to range using (3.4), forming a discrete grid of range values known as a range swath. Each grid point is referred to a range bin or fast-time sample, and each range bin is spaced ΔR meters apart.

In (3.3), $\hat{a}(t)$ is a scaled version of the amplitude modulation function whose value depends on propagation losses, the properties of the radar system, and the reflective properties of the scatterer from which the transmitted signal was reflected. The impact of these physical effects are captured by the radar range equation, which provides a way to compute the received power from models of the system, target, and propagation environment. For a simple target at range R, the power incident on the receiver P_r relates to the transmitted power P_t as

$$P_r = \frac{P_t G_t G_r \lambda^2 \sigma}{(4\pi)^3 R^4} \quad W \tag{3.6}$$

where G_t and G_r are the gain of the transmit and receive aperture, respectively, λ is the operating wavelength of the radar, and σ is the radar cross section (RCS) of the target, which is proportional to the amount of incident power on the target that is reflected back to the radar receiver. RCS is expressed in m^2, but it is not equal to the actual cross-sectional area of the target and depends heavily on the target's shape, orientation, and electrical properties. Several variations of the radar range equation exist, but only the point target version described above is

used for this work.

After the signal in (3.3) is received by the radar, it is corrupted by thermal noise due to analog components in the receive chain. A common approximation for the noise power at the output of the receiver is

$$P_n = k_B \beta F T_s \quad W \tag{3.7}$$

where $k_B = 1.38 \cdot 10^{-23} J/K$ is the Boltzmann constant, β is the receiver bandwidth, F is the noise figure, which is a ratio of the input signal-to-noise ratio (SNR) to the output SNR of the chain of components, and T_s is the temperature of the system in Kelvin. Therefore, the SNR at the receiver output (before signal processing) is

$$SNR = \frac{P_r}{P_n} = \frac{P_t G_t G_r \lambda^2 \sigma}{(4\pi)^3 R^4 k_B \beta F T_s} \tag{3.8}$$

If the transmitted signal reflects off of a scatterer with a nonzero radial velocity with respect to the radar, the received signal will be shifted in frequency due to the Doppler effect. Assuming the object moves with a constant radial velocity v_r and the radar operates at carrier frequency F_0, the Doppler shift is [28]

$$F_D = \frac{2v_r}{c - v_r} F_0 \approx \frac{2v_r}{\lambda} \tag{3.9}$$

where the approximation follows when $v_r << c$. By convention, a positive Doppler shift indicates that the radial velocity is in the direction of the radar.

At most carrier frequencies reserved for radar systems, the Doppler shift cannot be resolved from a single pulse. It is therefore common to transmit a coherent train of M pulses as shown in Fig. 3.1.

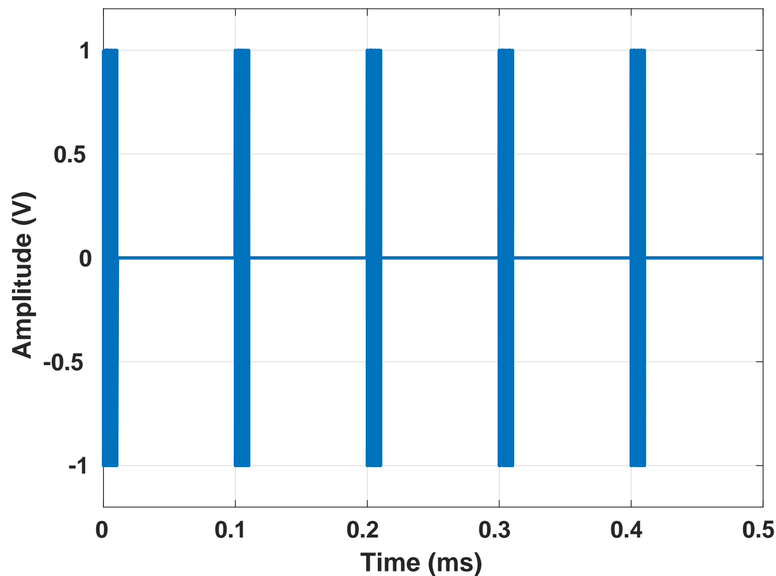


Figure 3.1: Coherent pulse train with $T_{PRI} = 100\,\mu$s

The time delay between the transmission of each pulse is known as the pulse repetition interval (PRI) and has value T_{PRI}. Equivalently, new pulses are transmitted at some pulse repetition frequency (PRF) $f_{PRF} = 1/T_{PRI}$, which is also known as the slow-time sampling rate. The choice of PRI/PRF presents a trade-off between the maximum unambiguous range and velocity. The maximum range that a pulse-Doppler radar system can unambiguously measure is

$$R_{ua} = \frac{cT_{PRI}}{2} = \frac{c}{2f_{PRF}} \tag{3.10}$$

Range measurements from targets beyond R_u will be aliased. Likewise, the maximum unambiguous velocity is

$$v_{ua} = \frac{\lambda}{4} f_{PRF} \tag{3.11}$$

Therefore, increasing the PRF increases v_{ua} but decreases R_{ua} and vice-versa.

This thesis assumes that each pulse is identical, and that the PRI is the same

for all pulses. Therefore, data is processed in coherent processing intervals (CPIs) of duration $T_{CPI} = MT_{PRI}$. Under these assumptions, coherent integration of these pulses improves the SNR given in (3.8) by a factor of M. If the target's radial velocity is constant over the CPI, the M samples from a single range bin are a discrete-time sinusoid with frequency F_D. The resolution with which this frequency can be estimated improves with the integration time, or

$$\Delta F_D = \frac{1}{T_{CPI}} = \frac{1}{MT_{PRI}} \tag{3.12}$$

3.2 Phased Array Radar

Recent advances in multifunction radar have closely followed improvements in phased array radar (PAR) technology. A phased array is made up of a set of antenna elements that are fed coherently, which makes it possible to use a variable phase shift or time delay across elements to form beams that point to particular regions in space [1]. Unlike traditional systems which must be mechanically steered to different angles, phased array antennas steer their beams electronically and nearly instantaneously. In many systems, it is also possible to simultaneously form multiple beams using a variety of beamforming and subarray techniques. This beam agility greatly increases the flexibility of the system, making it possible to perform several functions like surveillance, tracking, and imaging with low latency using a single array.

The simplest phased array topology is a uniform linear array (ULA), which arranges multiple antenna elements in a line such that the spacing between adjacent elements is equal. For example, a ULA with N elements is shown in Fig. 3.2.

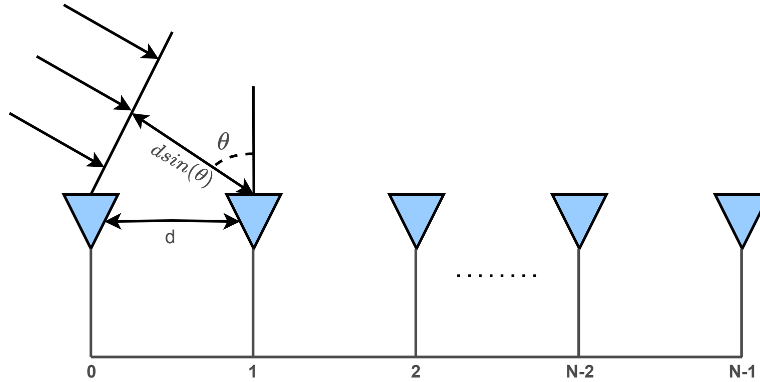


Figure 3.2: Uniform linear array

Each element is spaced d meters apart (such that the total length of the array is $L = (N - 1)d$), and the planar wavefront is incident on the array at an angle θ with respect to the array normal. Defining $t = 0$ as the time at which the plane wave reaches element 0, the time-delay of the signal for each element position k is

$$\tau_k = k\frac{d}{c}\sin(\theta) \tag{3.13}$$

Assuming the received signal is narrowband such that $c/B >> Nd$ for a signal with bandwidth B, the corresponding phase shift is

$$\phi_k = 2\pi f_0 \tau_k = k\frac{2\pi d}{\lambda}\sin(\theta) \tag{3.14}$$

This progressive phase shift across each element forms the basis for many common beamforming strategies [29]. Beamforming can be performed using analog components such as phase shifters or digitally by directly sampling all or a subset of the array elements. Specific beamforming strategies are beyond the scope of this work and are not discussed further.

Assuming the amplitudes of each antenna element are uniformly weighted and neglecting losses due to antenna efficiency, the gain can be computed from the 3

dB beamwidths θ_3 and ϕ_3 (in degrees) as [28]

$$G \approx \frac{26,000}{\theta_3 \phi_3} \tag{3.15}$$

Similarly, the beamwidth for an ideal linear antenna aperture is

$$\theta_3 = 2\sin^{-1}\left(\frac{1.4\lambda}{\pi L}\right) \approx 0.886\frac{\lambda}{L} \text{ radians} \tag{3.16}$$

The total far-field radiation pattern E for a phased array is the product of the pattern of the individual elements E_e and the array factor AF, which is a weighted sum of the contributions from all elements in the array

$$E(\theta) = AF(\theta, \theta_0)E_e(\theta, \theta_0) \tag{3.17}$$

where θ_0 is the array steering angle and θ is the angle at which the pattern is being evaluated. For a ULA, the array factor has a sinc structure given given by

$$AF(\theta, \theta_0) = \frac{\sin\left(\frac{N\pi d}{\lambda}(\sin(\theta) - \sin(\theta_0))\right)}{N\sin\left(\frac{\pi d}{\lambda}(\sin(\theta) - \sin(\theta_0))\right)} \tag{3.18}$$

It is also common to evaluate array performance in terms of the power pattern $P(\theta)$, which is related to (3.17) as

$$P(\theta) = |E(\theta)|^2 \tag{3.19}$$

Fig. 3.3 shows the power pattern for a ULA at center frequency $f_0 = 3\,\text{GHz}$. As the number of elements N increases, the power pattern mainlobe narrows (i.e., the beamwidth decreases) and the integrated sidelobes levels decrease.

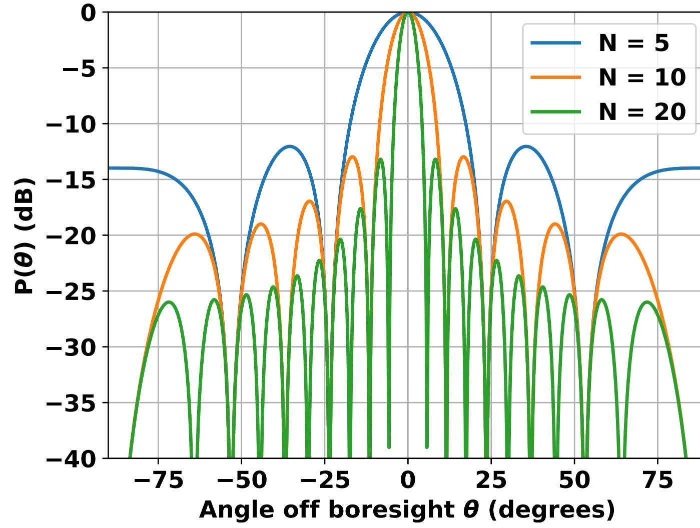


Figure 3.3: Normalized power pattern for a ULA with $\theta_0 = 0$ and constant element spacing $d = \lambda/2$. The center frequency is $f_0 = 3\,\text{GHz}$.

The uniform *rectangular* array (URA) shown in Fig. 3.4 generalizes the theory discussed above to two dimensions. Unlike linear arrays, URAs can steer beams in both azimuth and elevation.

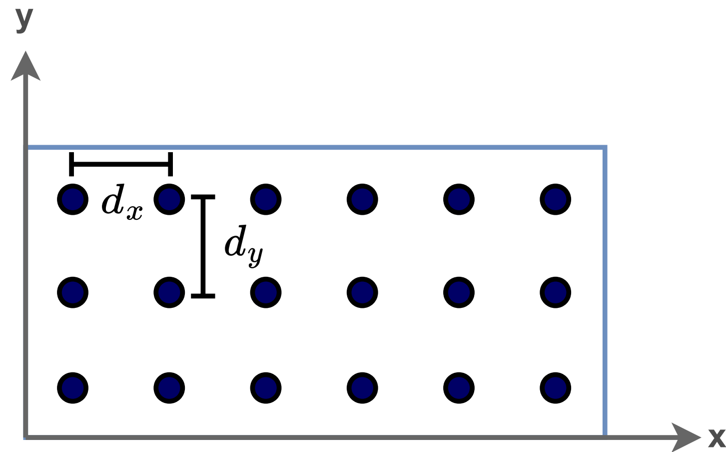


Figure 3.4: Uniform rectangular array topology

Assuming the illumination function of the aperture is separable in the x and

y dimensions, the two-dimensional power pattern $P(\theta, \phi)$ can be expressed as a product of two one-dimensional patterns [28]

$$P(\theta, \phi) = P_\theta(\theta) P_\phi(\phi) \tag{3.20}$$

For an ideal uniformly-weighted aperture, the power pattern is a two-dimensional squared sinc function similar to those shown in Fig. 3.3.

3.3 Target Models

The kinematics of all targets in this work are modeled in three dimensions with a discrete constant-velocity, white noise acceleration (CV-WNA) process. Therefore, each target maintains a six-dimensional state vector $\mathbf{x} = [x, \dot{x}, y, \dot{y}, z, \dot{z}]^T$ describing its position and velocity in Cartesian coordinates. In this model, deviations from the ideal constant-velocity behavior are treated as a zero-mean additive white Gaussian noise process $w(t)$ [30]. At each discrete time step Δt the CV-WNA model updates the target position and velocity separately for each Cartesian coordinate according to

$$x(t + \Delta t) = F(\Delta t) x(t) + w(t), \quad w(t) \sim \mathcal{N}(0, Q(\Delta t)) \tag{3.21}$$

where F is the constant-velocity transition matrix given as

$$F(\Delta t) = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \tag{3.22}$$

and Q is the process noise covariance matrix given by

$$Q(\Delta t) = \begin{bmatrix} \frac{\Delta t^3}{3} & \frac{\Delta t^2}{2} \\ \frac{\Delta t^2}{2} & \Delta t \end{bmatrix} q \tag{3.23}$$

where q is the process noise intensity. Larger values of q correspond to larger deviations from the constant-velocity model, and each value in Q grows with Δt.

The RCS of each target is modeled according to Swerling case 1. This model is suited for targets with many non-dominant scatterers whose RCS is constant across pulses in a dwell. For Swerling 1, fluctuations in the RCS are exponentially distributed with probability density

$$p(\sigma, \bar{\sigma}) = \frac{1}{\bar{\sigma}} \exp\left(\frac{-\sigma}{\bar{\sigma}}\right) \tag{3.24}$$

where $\bar{\sigma}$ is the mean RCS. For a Swerling 1 target, the probability of detection can be computed from the probability of false alarm P_{FA} as [31]

$$P_D = P_{FA}^{1/(1+SNR)} \tag{3.25}$$

3.4 Operating Modes

3.4.1 Surveillance

In the surveillance mode, the radar scans its field of view to discover previously undetected targets in the environment. There are a number of sub-tasks within this mode that each utilize the radar's resources differently. For example, the radar may search the volume to detect targets at short, medium, or long range. In short-range surveillance, targets will appear with higher SNR due to the R^4 dependence on range in (3.6). This enables the radar to scan the entire volume

more rapidly by emitting wider beams and shorter dwells in each region. This is important in self-protect scenarios where close-in targets are considered a high priority. The radar can also use a higher PRF to unambiguously measure doppler if range ambiguities are not a concern. In contrast, long-range search requires narrow beams and long dwell times in order to detect targets with low SNR.

In either case, volume search is typically carried out using a fixed scan pattern such as the raster pattern in Fig. 3.5.

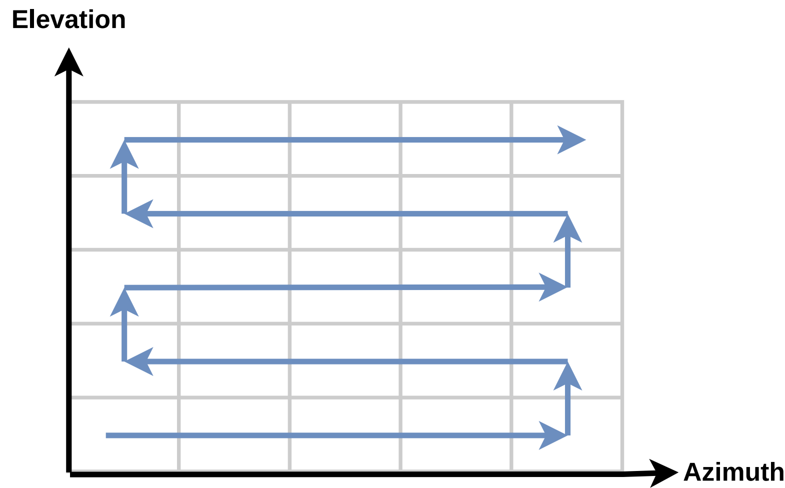


Figure 3.5: Raster scan patter

Here, the radar discretizes the angular search region into a grid and each grid point is searched sequentially. To increase the likelihood of detecting targets that are not centered on the beam, the grid points often overlap by some fraction of the beamwidth. Increasing the spacing between beam positions reduces the amount of time required to search the volume, but increases the amount of the search volume in a low-gain portion of the antenna pattern. The major limitation to this approach is that the radar must sequentially scan each angular bin, regardless of where targets are likely to exist. The main goal of the approach developed in this thesis is to overcome this shortcoming, leveraging prior knowledge of the scenario

to focus the search process on regions with a higher density of undetected targets.

3.4.2 Tracking

In the tracking mode, the radar allocates resources to initiate tracks from new detections, improve tracks on existing targets, or recover lost tracks when a target under track is not in its expected location. To initiate a track on a detected target that is not associated with an existing track, it is common to rapidly collect a set of confirmation measurements of the target in order to provide a good state estimate for the tracking filter.

Many algorithms exist for determining when a track must be updated. In general, targets with complicated maneuvering patterns require more frequent track updates than those with simple motion profiles. The load on radar resources also rapidly grows as the number of targets under track increases, so most existing work in the multifunction radar resource management problem is focused on tracking [3]. Since this work focuses primarily on the surveillance problem, an ideal tracker is used. For this tracker, the predicted state of a target under track is equal to its true state and the tracker correctly associates all detections to the target from which they originated.

Chapter 4

Particle Swarm Optimization

4.1 Particle Swarm Optimization

Particle Swarm Optimization (PSO) methods are a class of algorithms that were originally intended to model the complex but structured choreography of flocks of birds in flight [32]. In PSO, candidate solutions to the optimization problem "fly" through a multidimensional search space to find new solutions. Individual candidate solutions are known as particles, and the set of all particles is known as a swarm. Each particle attempts to find good solutions by incorporating knowledge of the best solutions found by neighboring particles and themselves. Like other bio-inspired algorithms such as genetic algorithms, the quality of the solution found by each particle is characterized by a fitness function, with good solutions exhibiting high fitness. By biasing the trajectory of each particle towards fit individuals, particles gradually cluster around regions of the search space that achieve good results on the optimization problem at hand.

There are a number of advantages to this approach. First, PSO only models the dynamics of each particle and makes no assumptions about the objective function under consideration. Unlike gradient-based methods like stochastic gradient descent (SGD), PSO can be used to optimize any function. Like other evolution-

ary algorithms, PSO is also useful for optimization problems with large search spaces since it does not rely on local information such as gradients. Moreover, the logic dictating the behavior of each particle is computationally inexpensive and easy to perform in parallel. In most cases, evaluating the fitness of each individual is the most time-consuming computation at each iteration.

For the standard PSO algorithm, the motion of the particles is governed by a simple first-order linear model such that at a discrete time-step t, the position of particle i is

$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + \mathbf{v}_i(t+1) \tag{4.1}$$

where $\mathbf{x}(t)$ and $\mathbf{v}(t)$ are the position and velocity at time t, respectively. In a search space with dimensionality N_d, $\mathbf{x}$ and $\mathbf{v}$ are length-N_d vectors for each particle. Once the velocity has been updated for each particle, (4.1) can be expressed as a matrix multiplication and quickly computed on an accelerator such as a GPU.

One major difference between PSO algorithms is the definition of the velocity, which determines the social behavior and priorities of each particle. In general, the velocity consists of three components whose relative weights can be altered to modify the swarm's behavior:

- Inertial: The inertial component is equal to the velocity at the previous time step, scaled by some inertial weight w. As the name suggests, this term determines how quickly the velocity can change at each iteration. If w is too large (relative to the other velocity components), particles will accelerate in their original direction at each time step and the swarm will diverge since the inertia is too large to alter trajectories towards better solutions. If w is too small, particles may decelerate until become stationary. However, a small w allows particles to quickly move toward good regions of the search space. Therefore, it is common to define $w \in [0, 1]$ and scale the cognitive

and social components to ensure convergence.

- Cognitive: The cognitive component biases the velocity towards the position of the best solution that has been found so far, scaled by some factor c_1. This is commonly referred to as a particle's memory or nostalgia, and causes the individual to return to good points in the search space over time. Placing too much weight on this term adversely impacts exploration, causing the particles to focus their search near the area where they were initialized. In dynamic environments where the fitness surface changes over time, this term may also bias the particles towards solutions that are no longer optimal.

- Social: The social component biases the velocity toward the best solution found by other particles in the swarm, scaled by some factor c_2. The best solution may be taken to be the best in the entire swarm or the best in a subset of particles in the swarm, known as a neighborhood. Therefore, the contributions from this component depend not only on its relative weighting, but on the structure of the social network used to diffuse information across the swarm.

The sections that follow describe global best and local best PSO, two classic variants of the algorithm that inspired the approach taken in this thesis.

4.1.1 Global Best PSO

In global best PSO (GBPSO) [33], the social component of the the velocity is computed with respect to the best particle in the entire swarm. Therefore, the velocity update equation is

$$\mathbf{v}_i(t+1) = w\mathbf{v}_i(t) + c_1\mathbf{r}_1(t)\left[\mathbf{y}_i(t) - \mathbf{x}_i(t)\right] + c_2\mathbf{r}_2(t)\left[\hat{\mathbf{y}}(t) - \mathbf{x}_i(t)\right] \tag{4.2}$$

where $\mathbf{r}_1(t), \mathbf{r}_2(t) \sim U(0,1)$ are random modifications to each velocity term that improve exploration. $\mathbf{y}_i(t)$ is the personal best position found by particle i, and $\hat{\mathbf{y}}(t)$ is the best solution found by all particles in the swarm. Since $\hat{\mathbf{y}}(t)$ propagates immediately to all particles in the swarm, the social network for GBPSO has a star topology shown in Fig. 4.1.

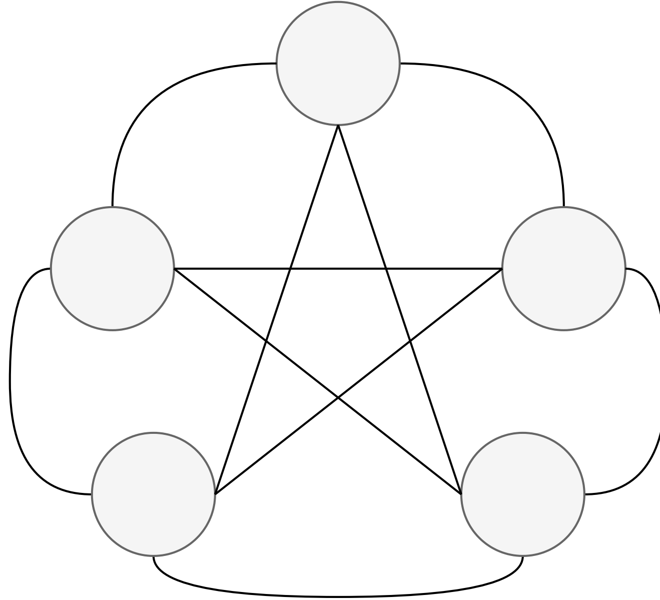


Figure 4.1: Star topology

Here, particles are represented as circles and each line represents direct information flow between particles so that clusters of connected particles form a neighborhood. Since all particles are interconnected, the swarm can be thought of as a single neighborhood in which particles have knowledge of all other particles in the swarm. Since information quickly propagates to all particles, GBPSO tends to converge more quickly than other variants. However, the centralized nature of the star network leads to a lack of diversity in particle trajectories, causing increased susceptibility to local optima. Therefore, GBPSO is better suited for unimodal optimization problems with relatively few local optima. In the context of the cognitive search agent described in Chapter 5, GBPSO is a good representation

of the environment state if new targets are expected to come primarily from one location.

4.1.2 Local Best PSO

In GBPSO, the velocity is biased in the direction of a social "target" position $\hat{\mathbf{y}}$, which was taken to be the position of best fitness found by all particles in the swarm. As the name implies, local best PSO (LBPSO) instead relies on local groupings between particles to compute the social velocity. In LBPSO, the propagation of information between particles is modeled by the ring social network architecture shown in Fig. 4.2.

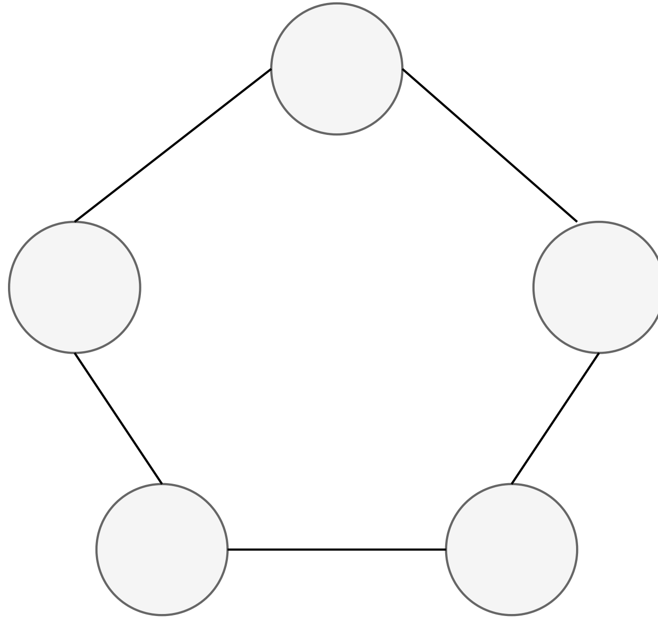


Figure 4.2: Ring topology

Here, particles are grouped into neighborhoods with *subsets* of other particles in the swarm. In each neighborhood $\mathcal{N}_i$, a separate social velocity $\hat{\mathbf{y}}_i$ is computed

with respect to the best velocity as

$$\hat{\mathbf{y}}_i(t+1) = \arg\max_{\mathbf{x}} f\left(\mathbf{x}(t)\right) \;\; \forall \mathbf{x} \in \mathcal{N}_i \tag{4.3}$$

where $f(\mathbf{x})$ is the fitness function being maximized. The velocity of each particle is updated as

$$\mathbf{v}_i(t+1) = w\mathbf{v}_i(t) + c_1\mathbf{r}_1(t)\left[\mathbf{y}_i(t) - \mathbf{x}_i(t)\right] + c_2\mathbf{r}_2(t)\left[\hat{\mathbf{y}}_i(t) - \mathbf{x}_i(t)\right] \tag{4.4}$$

which is equivalent to (4.2) except in the social component, where the global best position $\hat{\mathbf{y}}$ has been replaced by the neighborhood best position $\hat{\mathbf{y}}_i$.

It is important to note that in general, a particle participates in more than one neighborhood. If neighborhoods were disjoint partitions of the swarm, they would be completely isolated from one another and knowledge of good solutions would be unable to flow between them. In an overlapping ring topology, information is exchanged between neighborhoods and the swarm can still converge to a single optimal point. Since information flow is slow compared to the star topology used in GBPSO (Fig. 4.1), LBPSO generally takes longer to converge. However, the distributed nature of the social network ensures that a larger portion of the search space is explored. LBPSO is also more robust to local optima since each swarm operates quasi-independently, and neighborhoods that are stuck in local optima can be "rescued" by other neighborhoods.

A variety of methods have been developed for grouping particles into neighborhoods. In the most basic case, neighborhoods are statically defined by indexing each particle at the beginning of the optimization process, then grouping particles with adjacent indices. For example, consider a swarm of particles defined by the set $\mathcal{P} = \{p_1, p_2, \ldots, p_{N_p}\}$. Particle p_i may be grouped with its k nearest neighbors

to form a neighborhood as

$$\mathcal{N}_i = \{p_{i-k/2}, p_{i-k/2+1}, \ldots, p_{i+k/2-1}, p_{i+k/2}\} \tag{4.5}$$

In the index-based grouping, neighboring particles have no meaningful spatial relationship to each other. This increases intra-neighborhood diversity since each particle is likely to be grouped with a number of distant neighbors. If the initial position of each particle is uniformly random, this is equivalent to a uniform random selection. Since neighborhoods are only computed at the start, this approach introduces minimal computational overhead to the optimization process.

Alternatively, neighborhoods can be formed by selecting particles based on their proximity to each other in the search space. At each iteration, a k-nearest neighbor (KNN) algorithm is run to group the particles. A brute-force computation of the distance between N_p particles requires $\mathcal{O}(N_p^2)$ computations, which may exceed the cost of updating the particle positions at each iteration (depending on the complexity of the fitness function). However, this can be reduced to $\mathcal{O}(\log N_p)$ using clustering algorithms such as k-d trees [34]. Spatially grouping particles may also lead to the formations of "islands", which occur when neighborhoods become too isolated to effectively exchange social information. While this slows the rate of convergence, it improves diversity since each neighborhood will primarily focus its search on a specific region of the space.

The general algorithm for GBPSO and LBPSO is given in Algorithm 2. A major limitation of this algorithm is that it is difficult to relate the parameters w, c_1, and c_2 to radar surveillance performance. This motivates the development of the PSO variant described in Chapter 5.

Algorithm 2 Standard PSO

1: Define lower and upper bounds $\mathbf{b}_l$ and $\mathbf{b}_h$ for the search space
2: Define lower and upper bounds $\mathbf{v}_l$ and $\mathbf{v}_h$ for the velocity of each particle
3: Create swarm of N_p particles
   // Prior knowledge can be encoded by initializing particles in specific regions
4: Initialize position of each particle: $\mathbf{x}_i(0) \sim U(\mathbf{b}_l, \mathbf{b}_h)$
5: Initialize velocity of each particle: $\mathbf{v}_i(0) \sim U(\mathbf{v}_l, \mathbf{v}_h)$
6: Initialize personal best position of each particle: $\mathbf{y}_i \leftarrow \mathbf{x}_i(0)$
   // For LBPSO, the social target is the neighborhood best. For GBPSO, it is the global best.
7: **if** $f(\mathbf{x}_i(0)) > f(\hat{\mathbf{y}})$ **then**
8:     Update social target position: $\hat{\mathbf{y}} \leftarrow \mathbf{x}_i(0)$
9: **end if**
10: **while** termination criteria not met **do**
11:     **for** each particle $\mathbf{x}_i, i = 1, 2, \ldots, N_p$ **do**
12:         Compute fitness $f(\mathbf{x}_i(t))$
13:         **if** $f(\mathbf{x}_i(t) > f(\mathbf{y}_i)$ **then**
14:             Update personal best position: $\mathbf{y}_i \leftarrow \mathbf{x}_i(t)$
15:         **end if**
16:         **if** $f(\mathbf{x}_i(t) > f(\hat{\mathbf{y}})$ **then**
17:             Update social target position: $\hat{\mathbf{y}} \leftarrow \mathbf{x}_i(t)$
18:         **end if**
19:         Compute updated velocity $v_i(t+1)$ using (4.2) or (4.4)
20:         Compute updated position $x_i(t+1)$ using (4.1)
21:     **end for**
22: **end while**

4.1.3 Exploration Strategies

In its classic form, PSO is largely a greedy optimization strategy. Particles move in the direction of best performance, and exploration primarily comes from the random variation of parameters $\mathbf{r}_1(t)$ and $\mathbf{r}_2(t)$ and by chance as the particles travel along their trajectories. For complex fitness surfaces, additional mechanisms may be required to facilitate exploration. Many popular exploration mechanisms incorporate strategies from evolutionary computation (EC) since these methods are simple to implement, computationally efficient, and scalable to large swarms.

One simple EC-based method is to apply random mutations to various values at each iteration. For example, in [35] the global best position $\hat{\mathbf{y}}$ is mutated with zero-mean multivariate Gaussian noise with covariance matrix $\Sigma \in \mathbb{R}_+^{N_d \times N_d}$

$$\hat{\mathbf{y}}(t+1) = \hat{\mathbf{y}}(t+1) + w(t),\ w(t) \sim \mathcal{N}(\mathbf{0}, \Sigma) \tag{4.6}$$

where N_d is the search space dimensionality and Σ may be constant or may follow an annealing schedule causing its component variances to decrease over time. Using this method, particles are encouraged to explore a region surrounding the global best position whose size is proportional to σ_j^2 in each dimension j. Another common method is to mutate the position of individual particles at each iteration. That is, with some probability P_m, the position of particle i is mutated (after its velocity update) as in (4.6). The standard deviation in each dimension is set such that the offset of mutated particles is an appreciable fraction of the search space α, or

$$\sigma_j = \alpha(b_{h,j} - b_{l,j}) \tag{4.7}$$

where $b_{l,j}$ and $b_{h,j}$ are the lower and upper bounds of dimension j in the space, respectively. Other EC-based approaches include selection-based PSO [36] and the approach taken in [37], which introduces a reproduction mechanism into the particle swarm. However, only the Gaussian mutation (4.7) is used for the approach in this Chapter 5.

Chapter 5

Cognitive Search Agent

5.1 Surveillance PSO

In this work, a novel variation of the particle swarm optimization (PSO) algorithm described in Chapter 4 has been developed to automate the radar surveillance task. Here, particles in the swarm represent hypotheses for the presence of a target, with a high density of particles corresponding to a high likelihood that a target is present in that region. The algorithm presented in the following sections can be used on its own in the track initiation logic of a radar system, but this formulation of the problem also provides a representation of the surveillance environment that can be re-used in future reinforcement learning (RL) approaches for radar resource management. As discussed in Chapter 2, one of the major challenges in RL is formulating complex tasks as Markov decision processes (MDPs). This is especially challenging for the surveillance problem, which is heavily non-Markovian without the modifications presented in this section. For instance, very little of the environment's history is encoded in a range-doppler or angle-doppler map. This is in contrast to tracking tasks, where each track encodes a history that mixes the object's predicted and measured position. In fact, traditional tracking algorithms such as the Kalman filter are almost entirely Markovian and only require the most

recent state for the prediction and track update step. In much of the cognitive multi-function radar literature, most of the focus is placed on methods for track control rather than surveillance, which is typically handled using a simple raster scan or a rule-based approach such as alert-confirm. These methods have some advantages; since a raster scan is equivalent to a periodic sampling of the search space without replacement, every azimuth and elevation bin is guaranteed to be searched at a fixed revisit interval. However, this method is entirely deterministic and does not take advantage of prior knowledge where targets are more likely to appear. Alert-confirm methods partially remedy this by scheduling confirmation dwells when a detection is made that cannot be associated with an existing track. However, this method only considers the immediate region where the unconfirmed detection was made, without accounting for context in other parts of the environment. It also becomes easily overloaded in environments with many targets or high false alarm rates. This chapter presents an alternative approach that represents the detection history as an MDP using PSO. This approach is computationally efficient enough to be performed in real time, scales to longer timescales and larger input dimensions, and can be processed as a constant-length tensor by a neural network.

5.2 Algorithm Description

At the beginning of the surveillance process, particles are randomly initialized throughout the search space. The bounds of the search space are defined by the field of view of the radar in azimuth and elevation. Fig. 5.1 shows an example of the initial distribution of a 5,000-particle swarm for a radar that can steer its beam from $-45°$ to $45°$ in azimuth and elevation. In this case, the particles are uniformly initialized with a low velocity throughout the search space. Prior

knowledge about the location of each target can be incorporated by initializing the particles in a particular angular region or biasing their velocities so that each particle is more likely to travel to a given point.

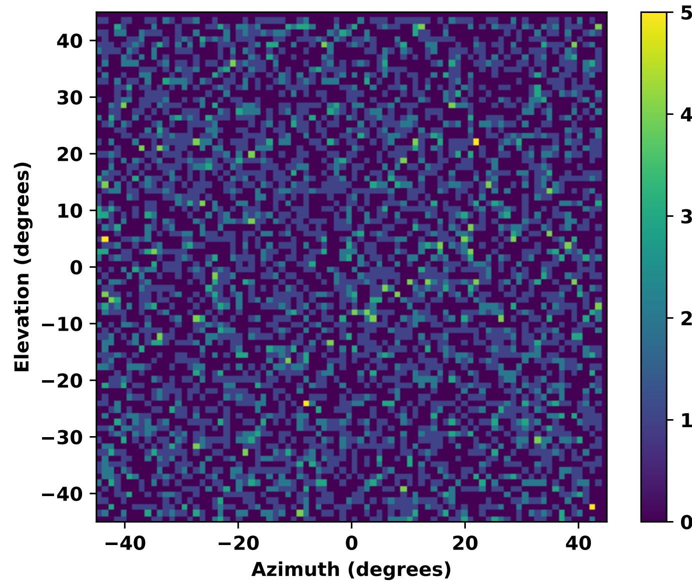


Figure 5.1: Uniformly initialized swarm

Once the particles have been initialized, the algorithm proceeds in two phases for the duration of the surveillance process. The dispersion phase described in Section 5.2.1 drives the exploration of the agent by removing particles from regions where new targets are unlikely to exist. The detection phase discussed in Section 5.2.2 does the opposite, encouraging exploitation by pulling particles towards regions with a high density of recent detections.

5.2.1 Dispersion Phase

Intuitively, the uncertainty that a target may be present in a particular location of the search space is reduced whenever the radar beam illuminates that region.

Since the density of particles in a given location represents the likelihood that a target is present there, the particles should disperse after being illuminated. When the radar beam is steered in a given direction, particles in the beam are given a velocity radially away from the beam center. If the beam is steered in the direction $\boldsymbol{\theta} = [\theta_{az}, \theta_{el}]$ with beamwidths $\boldsymbol{\Delta\theta} = [\Delta\theta_{az}, \Delta\theta_{el}]$, the velocity of each particle i within the beam is updated as

$$\mathbf{v}_i(t+1) = \mathbf{r}_{disp}(t)\frac{\mathbf{x}_i(t) - \boldsymbol{\theta}}{\|\mathbf{x}_i(t) - \boldsymbol{\theta}\|} \tag{5.1}$$

where $\mathbf{x}_i(t) = [x_{az}(t), x_{el}(t)]$ is the position vector of particle i in azimuth and elevation at time t, and $\mathbf{r}_{disp}(t) \sim U(\mathbf{0}, \boldsymbol{\Delta\theta})$ is sampled independently for each particle. The $\mathbf{r}_{disp}(t)$ ensures that, on average, particles in the beam are scattered by half the beamwidth in angle. As in the original PSO formulation, the random nature of $\mathbf{r}_{disp}(t)$ improves exploration by adding diversity to the trajectories of the scattered particles. Next, the standard particle swarm update is performed on all particles as given by Eq. (4.1). Following the position update, all particle velocities are multiplied by an inertia parameter $w_{disp} \in [0, 1]$. This parameter controls how far each particle is able to travel after being illuminated, with greater inertia causing scattered particles to travel farther from their initial positions. Therefore, the cognitive search agent becomes more exploratory as w_{disp} approaches unity.

5.2.2 Detection Phase

Following the dispersion phase, a second optimization step is performed to incorporate knowledge gained from new detections into the swarm. To improve robustness to false alarms, the detection phase is not performed on all detections. Instead, it is only carried out for detections that are associated with an unconfirmed track. Therefore, the impact of false alarms depends heavily on the data

association algorithm used by the tracker rather than on the SPSO algorithm itself.

For a detection d with angle vector $\boldsymbol{\theta}_d$, the direction vector pointing from each particle i to the detection is computed as

$$\mathbf{x}_{i,d}(t) = \boldsymbol{\theta}_d - \mathbf{x}_i(t) \tag{5.2}$$

Next, the probability that a PSO update is performed on a given particle is inversely and exponentially proportional to its distance from the detection

$$p_{move,i} = \exp\left(-\beta_g \|\mathbf{x}_{i,d}(t)\|\right) \tag{5.3}$$

where $\beta_g \in \mathbb{R}$ is the gravity parameter, which determines how strongly particles are "pulled in" by nearby detections. In other words, a high β_g causes the position update probability to decrease more rapidly with distance, confining particles to their local region of the search space. It has been experimentally determined that $\beta_g \in [0.05, 0.10]$ provides reasonable performance for both unimodal and multimodal distributions of targets.

If a particle is selected for a movement update, its cognitive velocity is set radially towards the detection as

$$\mathbf{v}_{i,d}(t) = \frac{\boldsymbol{\theta}_d - \mathbf{x}_i}{\|\boldsymbol{\theta}_d - \mathbf{x}_i\|} \tag{5.4}$$

so that its new velocity becomes

$$\mathbf{v}_i(t+1) = w_{det}\mathbf{v}_i(t) + c_{det}\mathbf{r}_{det}(t)\mathbf{v}_{i,d}(t) \tag{5.5}$$

where $\mathbf{r}_{det}(t) \sim U(0,1)$ serves the same purpose as $\mathbf{r}_{disp}(t)$ in Eq. (5.1). Note

that the social component of the velocity is zero, so this method is similar to the cognition-only model in traditional PSO [38]. This helps ensure that particles attend largely to their initial location, giving the swarm the ability to simultaneously focus on multiple clusters of targets in different regions of the search space. After all detections have been processed, a Gaussian mutation is applied to a small fraction of the particles according to Eq. (4.7). For the remainder of this work, this variant of PSO is referred to as Surveillance PSO (SPSO). The process described above is summarized in Algorithm 3. Although this algorithm was developed with RL in mind, it can also be used in isolation. In the simplest case, the beam could be steered to the location with the most particles. Alternatively, a sampling approach could be taken where the probability of selecting an azimuth/elevation is proportional to the number of particles in that bin.

Algorithm 3 Surveillance PSO

Parameters: $\beta_g, w_{disp}, w_{det}, c_{det}$
// The position bounds are set by the azimuth and elevation scan limits of the radar
1: Define lower and upper bounds $\mathbf{b}_l$ and $\mathbf{b}_h$ for the search space
2: Define lower and upper bounds $\mathbf{v}_l$ and $\mathbf{v}_h$ for the velocity of each particle
3: Create a swarm of N_p particles
// Each position/velocity is a 2-vector in azimuth and elevation
4: Initialize position of each particle: $\mathbf{x}_i(0) \sim U(\mathbf{b}_l, \mathbf{b}_h)$
5: Initialize velocity of each particle: $\mathbf{v}_i(0) \sim U(\mathbf{v}_l, \mathbf{v}_h)$
6: **while** surveillance task is not done **do**
// Dispersion phase
7: Steer a radar beam with beamwidths $\Delta\boldsymbol{\theta} = [\Delta\theta_{az}, \Delta\theta_{el}]$ to look angles $\boldsymbol{\theta} = [\theta_{az}, \theta_{el}]$ in azimuth and elevation.
8: **for** each particle $i = 1, 2, \ldots, N_p$ **do**
9: **if** particle is in the main beam **then**
10: Recompute velocity using Eq. (5.1)
11: **end if**
12: $\mathbf{x}_i(t+1) \leftarrow \mathbf{x}_i(t) + \mathbf{v}_i(t)$
13: $\mathbf{v}_i(t+1) \leftarrow w_{disp}\mathbf{v}_i(t)$
14: **end for**
// Detection phase
15: Perform detection processing using methods from Chapter 3
16: Remove detections associated with existing tracks to obtain N_d unassociated detections
17: **for** each detection $d = 1, 2, \ldots, N_d$ **do**
18: **for** each particle $i = 1, 2, \ldots, N_p$ **do**
19: Compute relative position $\mathbf{x}_{i,d}$ using Eq. (5.2)
20: Compute movement probability using Eq. (5.3)
21: **if** move particle **then**
22: Compute velocity $\mathbf{v}_{i,d}$ towards the detection using Eq. (5.4)
23: $\mathbf{v}_i(t+1) \leftarrow w_{det}\mathbf{v}_i(t) + c_{det}\mathbf{r}_1(t)\mathbf{v}_{i,d}(t)$
24: $\mathbf{x}_i(t+1) \leftarrow \mathbf{x}_i(t) + \mathbf{v}_i(t)$
25: **end if**
26: **end for**
27: **end for**
28: (Optional) Perform the Gaussian mutation from Eq. (4.7) on each particle with probability P_m
29: **end while**

5.2.3 Adaptive Dispersion Inertia

Of the parameters discussed in Section 5.2, surveillance performance is most heavily dependent on the dispersion inertia w_{disp} and the gravity parameter β_g. If w_{disp} is too small, the agent will spend too much time searching a region even after initiating tracks on all targets. In contrast, if w_{disp} is too large, particles may quickly disperse from regions with a high density of targets if the agent spends too much time searching in other regions. In a sense, w_{disp} determines how quickly the agent "forgets" about targets in an area. It is thus desirable for the agent to have a low inertia in regions where targets are likely to be present (to encourage the agent to thoroughly search that area), and high inertia elsewhere to promote exploration. This trade-off is not possible in the default formulation in Algorithm 3, which assumes w_{disp} is a constant. Therefore, algorithm 4 presents a method for modifying w_{disp} during the surveillance based on this simple heuristic.

Algorithm 4 Adaptive Dispersion Inertia

```
    Parameters: w_{disp,min}, w_{disp,max}, c_{disp}
 1: for each dwell do
 2:     if particle i moved towards a detection then
 3:         w_{disp,i}(t+1) ← w_{disp,min}
 4:     else
 5:         w_{disp,i}(t+1) ← min(c_{disp} w_{disp,i}(t), w_{disp,max})
 6:     end if
 7:     Update the swarm according to Algorithm 3
 8: end for
```

Here, $w_{disp,min}$ and $w_{disp,max}$ are the minimum and maximum possible inertia for each particle, respectively, and c_{disp} is an adaptation factor that determines how quickly particles transition from exploitative behavior (low inertia) to exploratory (high inertia). Particles become more exploitative if they have recently been associated to a detection, and the longer a particle goes without a detection, the more exploratory it becomes (up to the limit $w_{disp,max}$). During experimen-

tation, it was found that a $w_{disp,min} = 0.25$, $w_{disp,max} = 0.95$, and $c_{disp} = 1.5$ performed well in all simulation environments.

5.2.4 Results and Analysis

In this section, the performance of an SPSO-based beam steering agent is evaluated in a number of multi-target surveillance scenarios. This agent uses SPSO with adaptive dispersion inertia to form images as in Fig. 5.1 then steers its beam to the azimuth and elevation angles with the most particles on each dwell. Interpreting the density of particles as the probability density of untracked targets in a region, this agent greedily focuses only on regions that are highly likely to contain targets. This is one of the simplest and most computationally efficient method for using SPSO to schedule dwells, since the SPSO output is used directly without additional processing. The beam steering agents under consideration are compared by the fraction of targets they each initiate over the course of the scenario.

In the first simulation environment, targets are clustered around a single point in the search volume. At the start of each new scenario, the simulation samples a center point from a uniform distribution that spans the radar's field of view. In this case, the field of view is $-45°$ to $45°$ in azimuth and elevation. The simulator then samples the span of the cluster from $U(0°, 40°)$ in each angular dimension in order to add diversity to the scenarios encountered by the agent. The scenario is reset when all targets are detected or after 2500 dwells have been processed.

In each new scenario, the simulator generates fifty targets prior to the first dwell. New targets spawn at the start of each dwell according to a Poisson birth process with some birth rate λ. When a new target is generated, its initial range is sampled from $U(5\,\text{km}, 50\,\text{km})$, and the x, y, and z components of its initial velocity

are independently sampled from $U(-100\,\mathrm{m/s}, 100\,\mathrm{m/s})$. The RCS of each target fluctuates according to a Swerling 1 model with a mean RCS of $10\,\mathrm{m}^2$. All targets move according to the constant-velocity white noise acceleration model described in 3 with an acceleration process noise of $q = 10$. Table 5.1 summarizes the environment parameters for the first experiment. Table 5.2 gives the parameters of the radar system, and Table 5.3 shows the parameters parameters for the SPSO algorithm described in Section 5.2.

Parameter	Value
Mean target azimuth	$\sim U(-45°, 45°)$
Mean target elevation	$\sim U(-45°, 45°)$
Target azimuth span	$\sim U(0°, 40°)$
Target elevation span	$\sim U(0°, 40°)$
Initial target range	$\sim U(5\,\mathrm{km}, 50\,\mathrm{km})$
Initial target velocity (x, y, z)	$\sim U(-100\,\mathrm{m/s}, 100\,\mathrm{m/s})$
Acceleration Process Noise (q)	10
Initial Num. Targets	50
Mean target RCS (σ^2)	10
Target birth rate λ	0.05
Num. confirmation detections	3
Time limit	2500 dwells

Table 5.1: Parameters for the single-cluster scenario

Parameter	Value
Array geometry	Rectangular
Num. horizontal elements	32
Num. vertical elements	32
Center frequency	3 GHz
Element spacing	$\lambda/2$
Tx element power	10 W
Element gain	3 dB
Beam pattern	Sinc
Noise figure	4 dB
Position (x, y, z)	$(0, 0, 0)$ m
Rotation offset $(yaw, pitch, roll)$	(0°, 0°, 0°)

Table 5.2: Radar system parameters

Parameter	Value
Num. particles	$10e3$
β_g	0.10
$w_{disp,min}$	0.25
$w_{disp,max}$	0.95
c_{disp}	1.5
w_{det}	0.25
c_{det}	$\Delta\theta/2$
Mutation rate	0

Table 5.3: Particle swarm parameters

Fig. 5.2 shows the fraction of targets initiated over time for three different agents for a beamwidth of 3° in azimuth and elevation, averaged over twenty trials. The blue curve is the traditional raster scan agent that scans according to the pattern in Fig. 3.5. Each position in the raster grid is separated by 0.75 beamwidths. This behavior is equivalent to sampling the angle space without replacement. The orange curve, on the other hand, represents an agent that uniformly selects a random beam position at each time step, which is equivalent to sampling with replacement.

In this scenario, none of the agents initiate tracks on all targets in the 2500-

dwell time limit. However, the SPSO agent significantly outperforms the other agents, tracking nearly 95% of targets by the end of the simulation. The random agent slightly outperforms the raster agent, with both initiating tracks on less than 70% of the targets. Throughout the simulation, the SPSO agent also quickly initiates tracks on a majority of the targets, reaching a track fraction of 75% in under 1000 dwells. For narrow beamwidths (which are necessary for long-range surveillance), methods like raster scanning and random search experience difficulty due to the number of angular bins that must be searched. Since there is a large amount of time between successive scans of a given bin, missed detections must wait for the entire grid to be searched (on average) before being illuminated again. The SPSO agent, in contrast, maintains a history of where targets have been detected in the past and prioritizes those regions until it initiates tracks on many targets.

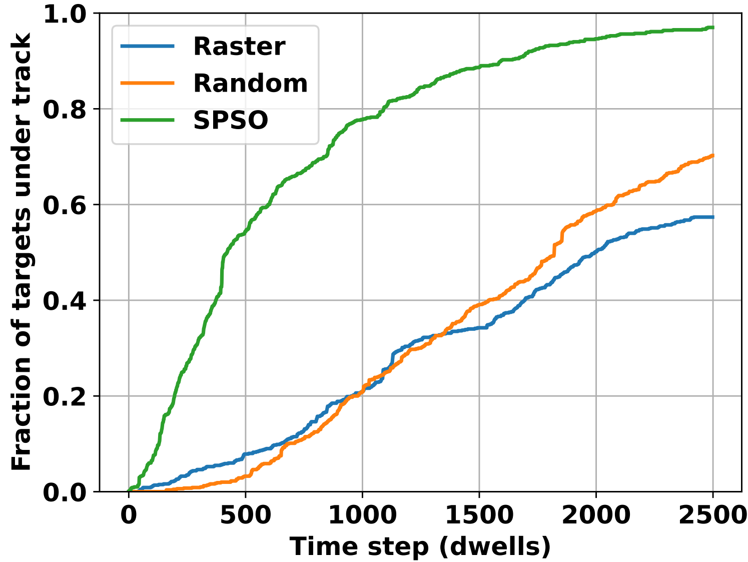


Figure 5.2: Track initiation ratio for a deterministic agent with $\Delta\theta_{az} = \Delta\theta_{el} = 3°$.

In Fig. 5.3, the scenario described above is repeated, but each agent now

steers a 5° beam. Like before, the uniform and random agents perform similarly throughout the scenario and both initiate tracks on most or all of the targets, on average. The SPSO agent's performance also improves significantly. Although it takes the SPSO agent nearly as long as the other agents to detect all of the targets, it visibly outperforms the other two agents throughout the course of the simulation, tracking 80% of targets in 500 dwells. When the beamwidth is increased to 10° (Fig. 5.4), the SPSO agent tracks the targets slightly more quickly than the other agents. However, the difference between each agent's performance is much smaller than before.

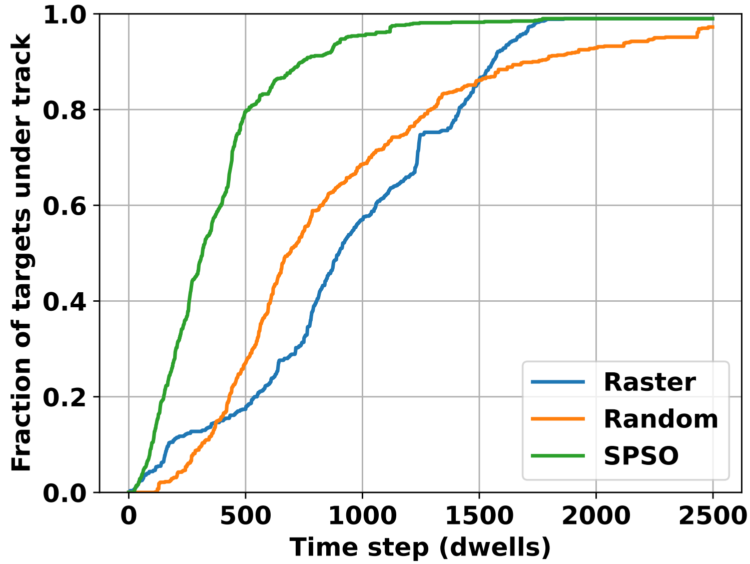


Figure 5.3: Track initiation ratio for a deterministic agent with $\Delta\theta_{az} = \Delta\theta_{el} = 5°$.

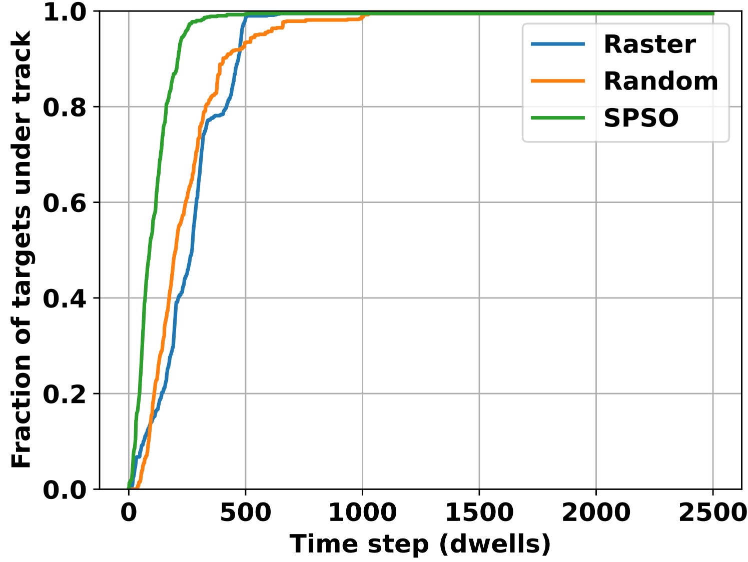


Figure 5.4: Track initiation ratio for a deterministic agent with $\Delta\theta_{az} = \Delta\theta_{el} = 10°$.

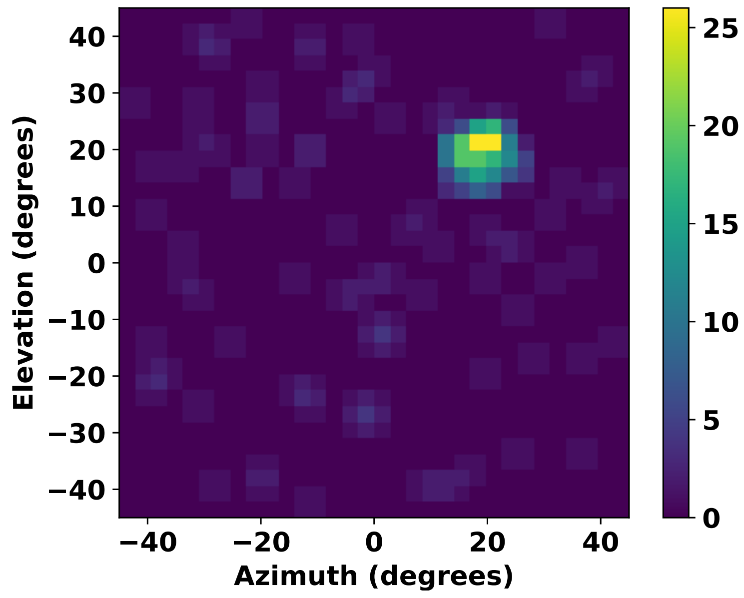
Fig. 5.5 shows the beam coverage history for four different distributions of targets. In these images, the value of each pixel corresponds to the number of times the radar beam was pointed to the corresponding angular region. In Fig. 5.5a, targets are initialized in a cluster around $(az, el) = (20°, 20°)$ with a width of $\pm 5°$, and the radar has a beamwidth of 3° in each dimension. In this case, the agent's beam coverage corresponds very closely to the true target distribution. The agent explores the entire search space until it locates the target cluster, then quickly initiates tracks on all targets and ends the episode.

In Fig. 5.5b, the cluster is centered around the same region, but the width of the cluster is $\pm 20°$ in azimuth and elevation. The SPSO agent automatically adapts to this change, broadening its search to a wider angular region that once again corresponds to the true distribution of the targets. In the extreme case where targets are uniformly initialized throughout the space, the agent's behavior approaches a random search (Fig. 5.5c).

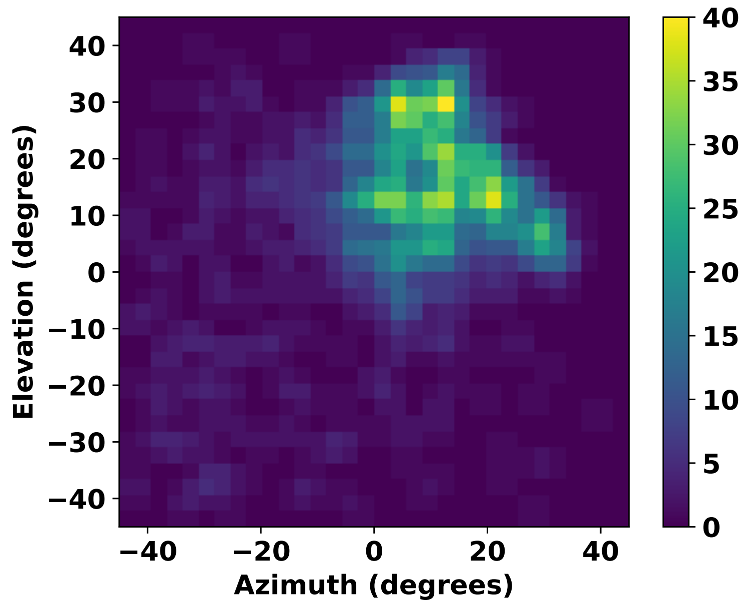
Since SPSO makes no assumptions about the distribution of targets or their motion profile, it also generalizes to more complex target distributions without requiring changes to the swarm parameters. For example, Fig. 5.5d shows the beam coverage history for a scenario with a multimodal target distribution. Targets are divided roughly evenly into two clusters that are 10° wide in each dimension, and the agent once again prioritizes the regions of the search space where untracked targets are most likely to be present.

These results indicate that the SPSO algorithm provides a useful method for informing where a radar system should steer its beams to improve performance on the surveillance task. Since SPSO does not require information about the distribution of targets, its performance generalizes to a range of scenarios. It is also computationally efficient, requiring only one swarm update per detection associated with an unconfirmed track (in addition to an update at the start of each dwell). The method examined in this section has two limitations: first, it can only be applied to compute the steering angle rather than other dwell parameters such as the beamwidth. Second, it provides no inherent mechanism to balance resources among other tasks.

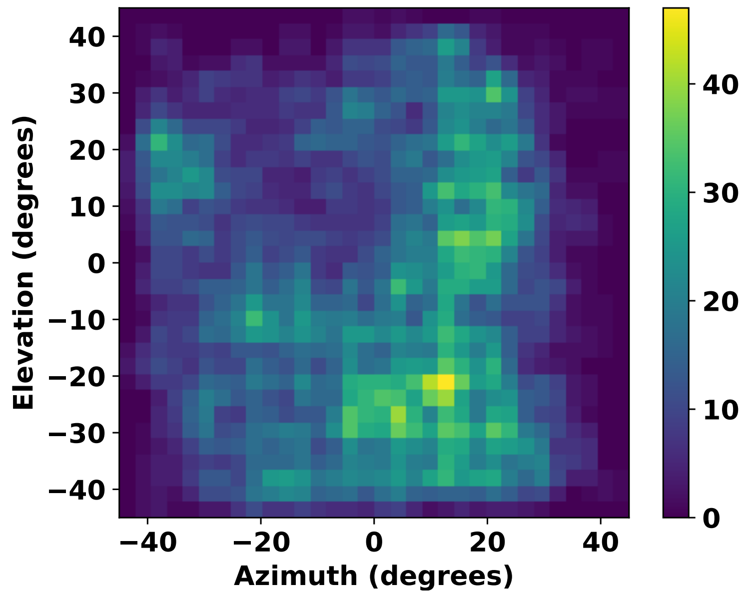
The reinforcement learning (RL) extension to the algorithm presented in Section 5.3 is intended as a first step to mitigating these problems. Novel multi-agent RL approaches can be developed in the future to treat each radar task as a separate computational agent in order to balance competing tasks. Additional parameters can easily be learned by the agent as simple extensions to the action space of the environment. For example, the RL agent described Section 5.3 outputs a beamwidth at each dwell, in addition to the steering angle.



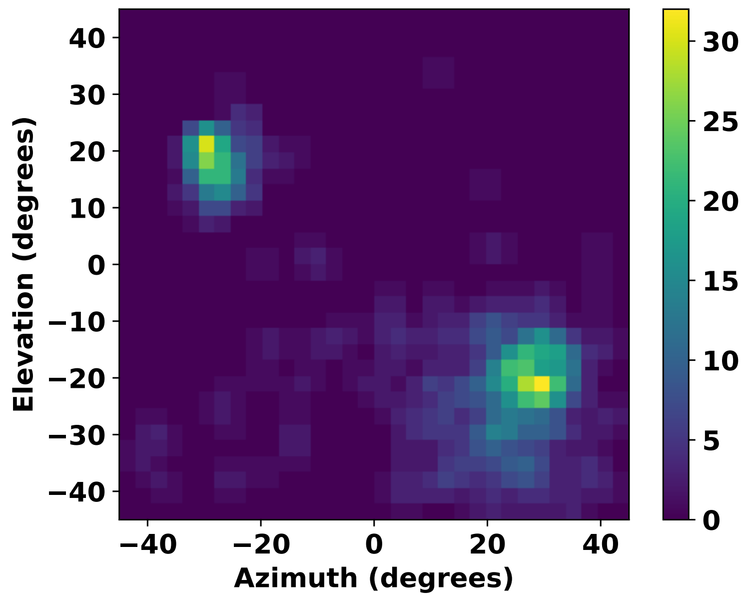
(a) Targets initialized ±5° from the center of the cluster



(b) Targets initialized ±20° from the center of the cluster



(c) Targets uniformly initialized throughout the space



(d) Two separate target clusters. The number of targets in each cluster is approximately equal, and targets are initialized $\pm 5°$ from the center of each cluster.

Figure 5.5: Beam coverage history for various target distributions.

5.3 RL Formulation

The SPSO algorithm developed in the previous section is used to express the surveillance problem as an MDP that can be integrated into an RL system. The various components needed to fully describe the MDP as defined in Chapter 2 are given below.

5.3.1 State Representation

The simplest way to represent the state of the environment is to discretize the search space into a grid where the value of each bin is equal to the number of particles it contains, forming a sequence of images similar to Fig. 5.1 that can be processed using a convolutional network architecture. During testing, it was found that the images produced by this method were often sparse, and its features were not distinct enough to be effectively learned by a CNN. For instance, consider the particle distribution after a cluster of targets has been detected at a particular angle (Fig. 5.6). Although it is clearly visible that particles have clustered near the location of the targets, a majority of the space contains few (if any) particles.

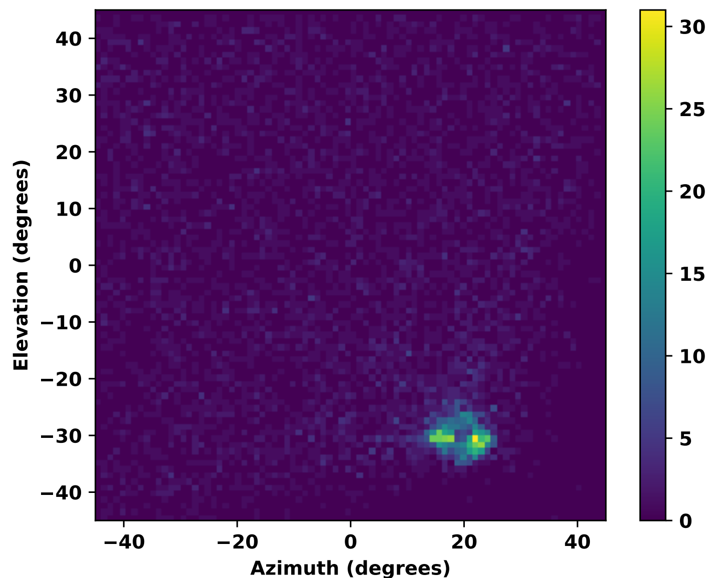


Figure 5.6: Particle distribution after detecting a cluster of targets at $(az, el) = (20°, -30°)$

To mitigate this issue, additional processing must be performed on the swarm output to create a more succinct state space representation (Fig. 5.7). First, images are generated as described above. Rather than directly passing raw pixel values to the network, each bin is ranked according to the number of particles it contains. A new $N \times 3$ image is formed from the N bins containing the most particles, where the first and second columns of this image contain the mean azimuth and elevation of every particle in each of the N bins, respectively.

If the agent is only required to select the steering angles for a beam with constant beamwidth, the first two columns encode all of the required information. In this work, the cognitive agent selects the azimuth and elevation beamwidth in addition to the steering angle. To effectively manage the trade-off between beamwidth and SNR, the state must also encode information about the range of potential targets. Therefore, the third column of the image provides the mean

"range" of the particles in each bin, where the of each particle is the range of the most recent track update that pulled the particle in during the detection phase (Section 5.2.2). With these values computed, the resulting image is flattened into a length-$3N$ tensor. Since neural network training is generally more stable on smaller inputs, the values in the first two columns are scaled to the interval $[-1, 1]$, and the values in the range column are divided by $10e3$.

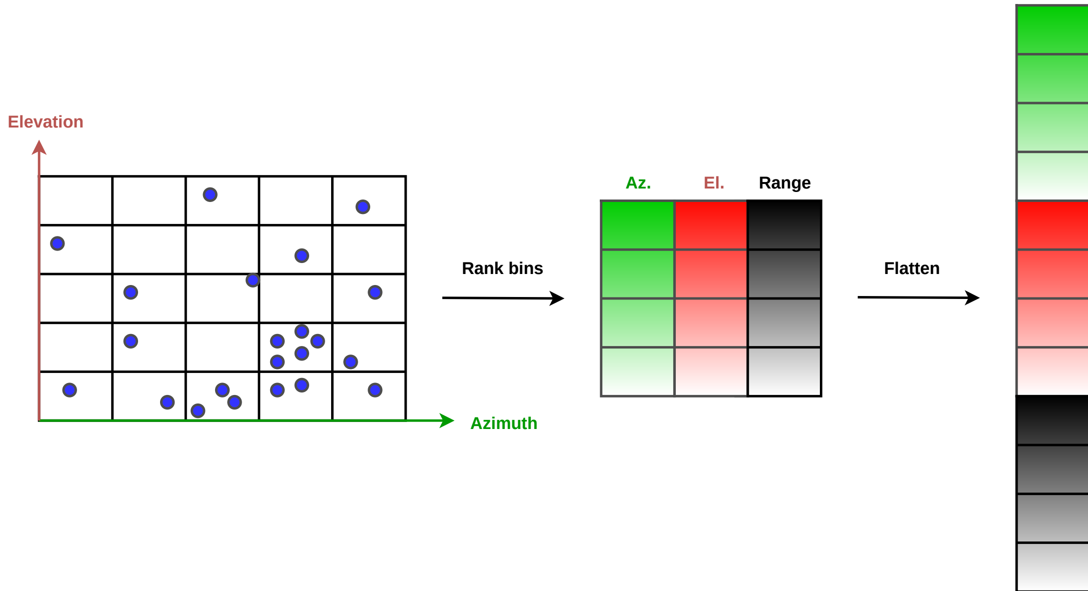


Figure 5.7: An example of the state space representation

This state representation has a number of advantages. Unlike the raw pixel state representation, the state space in Fig. 5.7 only encodes information about the N bins that are most likely to contain untracked targets. Since the feature vector has already been processed, it can be passed directly into a simple fully-connected policy network to select the dwell parameters. This network requires far fewer parameters than if raw pixels were used since the state vector is only length-$3N$, which greatly reduces the computation required for action selection and makes real-time implementation feasible. Finally, this approach is invariant to the specific details of the particle swarm, so the swarm parameters (such as

the number of particles or inertia values) can be changed without repeating the training process.

5.3.2 Action Space

At the start of each dwell, the cognitive surveillance agent outputs the angle to which the beam should be steered and the transmit beamwidth in both azimuth and elevation. The beam is only spoiled on transmit such that the full aperture is used for computing the receive gain and angular resolutions. The process of selecting actions is formulated as a continuous control task, where the input to the policy network is the feature vector described in Section 5.3.1 and the output is a mean and variance for each parameter of the four parameters to be selected. The agent then samples actions from a multivariate Gaussian distribution by concatenating the mean values into a vector and the variances into the diagonal of a covariance matrix. If a sampled action is invalid for a particular radar configuration (e.g., the steering angle exceeds the radar's field of view), it is clipped into the valid range. Although only beamwidths and steering angles are considered in this work, the Gymnasium environment (see Section 2.2.3) also supports dynamic selection of the bandwidth, pulse width, PRF, and the number of pulses per dwell in its action space.

5.3.3 Reward Function

The goal of the surveillance agent is to select beam steering angles such that the number of tracks initiated is maximized. To reflect this goal, the agent receives a reward of +1 for each track initiated at the end of each dwell and a reward of 0 otherwise.

5.3.4 Results and Analysis

The multi-target surveillance scenario described by Tables 5.1-5.3 is repeated to evaluate the performance of the RL beam-steering agent. In order to more thoroughly evaluate the beam spoiling behavior of the RL agent, the initial range of each target is uniformly distributed from 5 km to 150 km. This requires the RL agent to consider both short, medium, and long-range scenarios when learning to sacrificing SNR for beamwidth.

To form the input state, the agent discretizes the distribution of particles into an 84×84 pixel image. The 100 pixels containing the most particles are used to form the input state vector as described in Section 5.3.1, producing a length-300 tensor. In order to capture particle velocities in the state representation, the previous three state vectors are stacked with the current state in the channel dimension, producing a 1D image with shape 4×300. The agent uses separate but identical hidden layers to compute the policy and value function. The first layer of each network consists of a `Conv1d` layer with kernel size 1×1, stride 1, and a single filter. This is nearly identical to a fully-connected linear layer, but processes all input channels at once to take advantage of the state history encoded by the frame stacking operation. The output of the convolutional layer is flattened and passed through a tanh activation function, followed by an additional fully-connected hidden layer with 64 units and another tanh activation. In the final processing stage, the actor network outputs the mean and standard deviation for each of the four actions (steering angles and transmit beamwidths in azimuth/elevation), and the critic network outputs a scalar estimate of the value of the current state. The network architecture is summarized in Fig. 5.9. The network is then trained using the PPO algorithm described in Section 2.3.2.

Parameter	Value
Horizon (T)	512
Num. Epochs	10
Batch Size	2048
Discount Factor (γ)	0.99
GAE Parameter (λ)	0.95
Value coefficient	N/A
Entropy coefficient	0
Policy Clip Fraction (ϵ)	0.2
Num. Parallel workers	16

Table 5.4: PPO agent parameters

Table 5.4 summarizes the PPO hyperparameters used during training. Since the actor and critic do not share parameters, value function scaling does not need to be performed (see Section 2.3.1).

Fig. 5.8 shows the fraction of targets initiated over time for the three deterministic agents described in Section 5.2.4 and the RL-based agent. As before, the raster and random agents are only able to initiate tracks on $60 - 70\%$ of targets over the duration of the scenario, and the deterministic SPSO agent tracks approximately 90% of targets. The RL agent, on the other hand, outperforms the other methods for the duration of the scenario and detects all targets in the 2500 dwells.

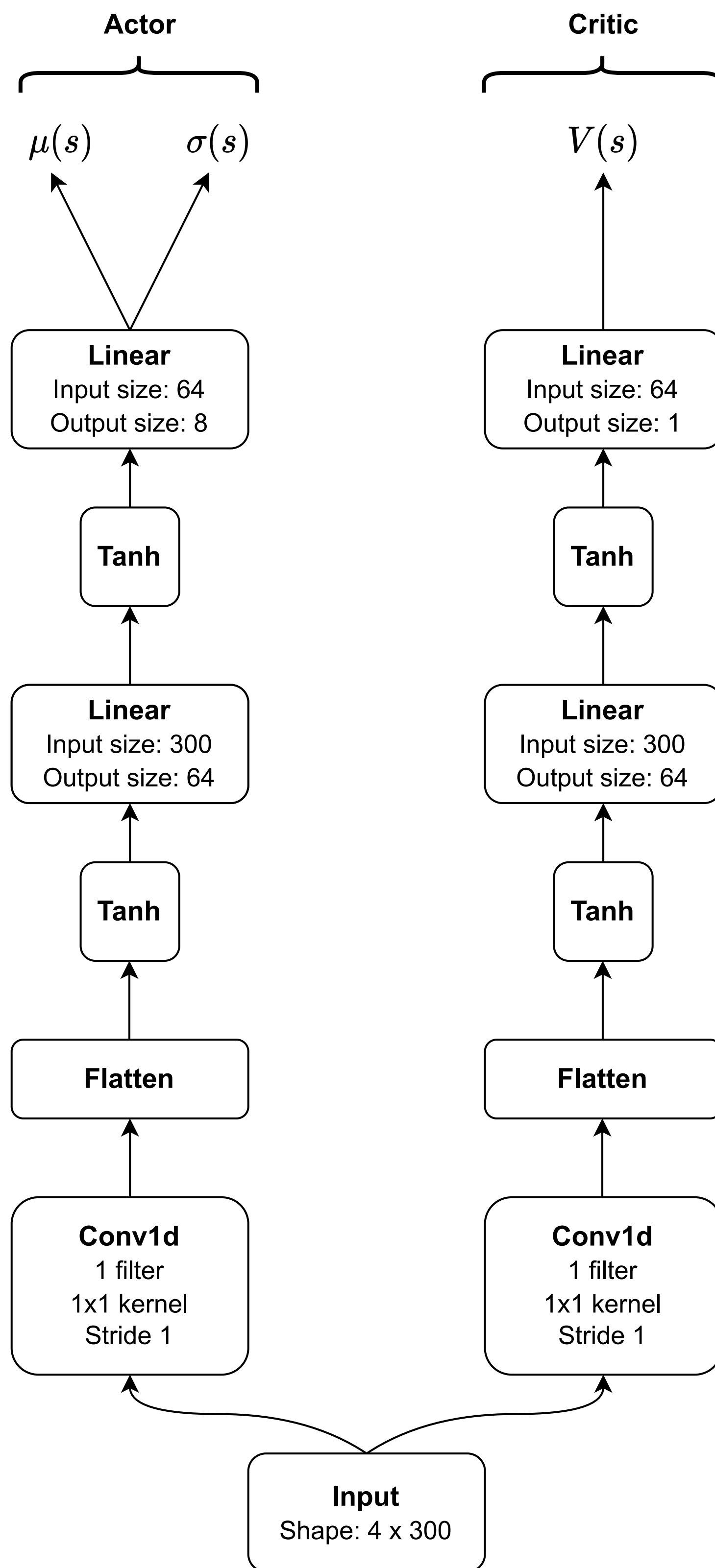


Figure 5.9: Agent Network Architecture

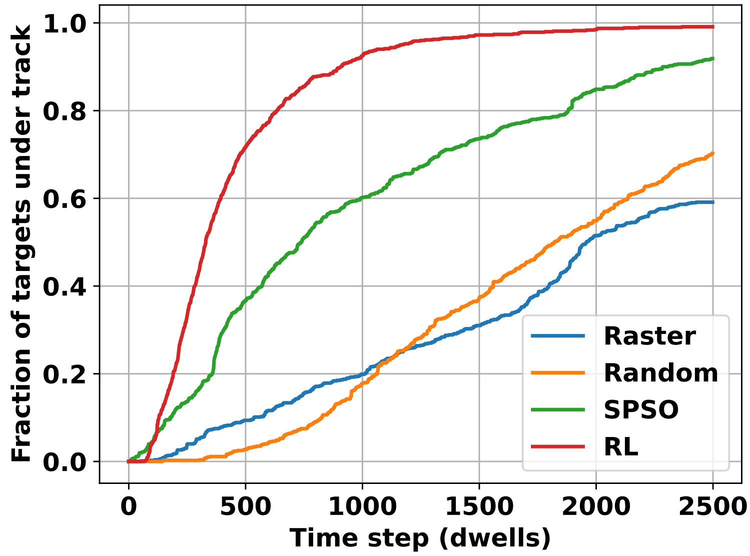


Figure 5.8: Track initiation fraction for the RL-based agent. At the start of each dwell, the agent selects a beamwidth between 3° and 10°. The other agents use a constant transmit beamwidth of $\Delta\theta_{az} = \Delta\theta_{el} = 3°$.

Fig. 5.10 shows the track initiation performance for all agents in the same scenario as before, except the random, raster, and SPSO agents now steer 5° beams on transmit. Compared to Fig. 5.10, the performance of the deterministic agents is considerably improved, with each agent tracking approximately all of the targets. In this scenario, the RL and SPSO agents perform nearly identically, and both outperform the raster and random agents.

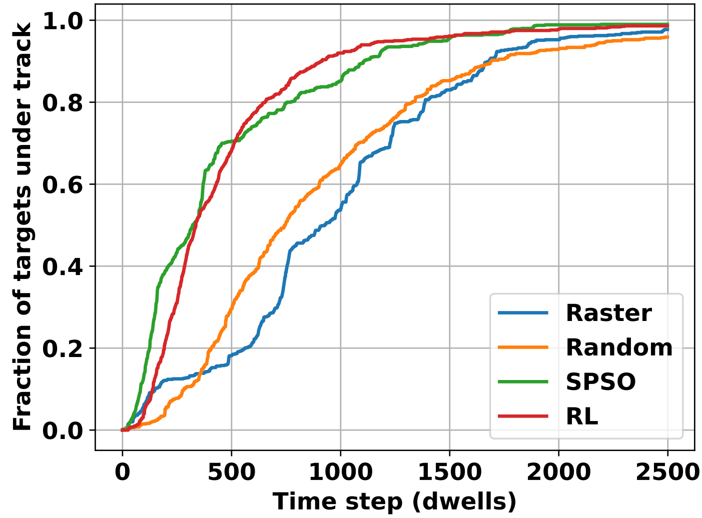


Figure 5.10: Track initiation ratio for the RL-based agent. At the start of each dwell, the agent selects a beamwidth between 3° and 10°. The other agents use a constant transmit beamwidth of $\Delta\theta_{az} = \Delta\theta_{el} = 5°$.

Similar results are observed when the SPSO, raster, and random agents steer 10° beams on transmit (Fig. 5.11). In this scenario, however, the RL agent performs worse than the SPSO agent for the first 500 dwells, then outperforms all other methods for the duration of the scenario. With a fixed 10° beam, the deterministic agents are unable to detect all targets in the scenario due to poor SNR for long-range targets. This demonstrates the utility of the RL beam-steering agent, which incorporates knowledge of the range of each target when selecting its beamwidth.

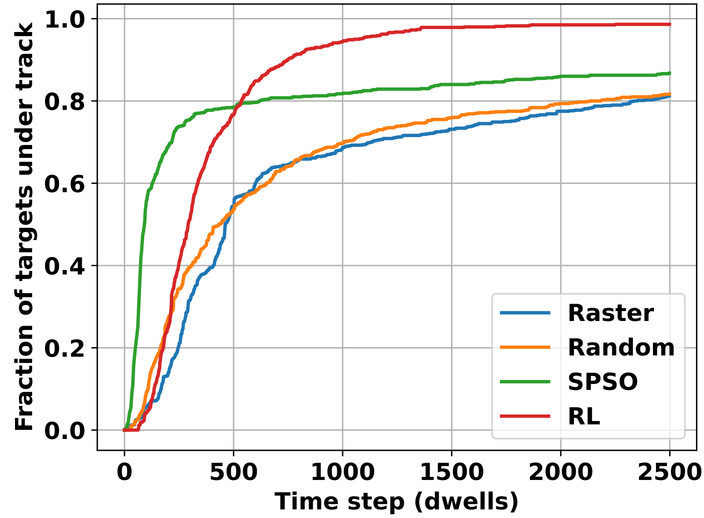


Figure 5.11: Track initiation ratio for the RL-based agent. At the start of each dwell, the agent selects a beamwidth between 3° and 10°. The other agents use a constant transmit beamwidth of $\Delta\theta_{az} = \Delta\theta_{el} = 10°$.

Next, the convergence behavior of the RL agent is analyzed. The following results were obtained by training the agent on the same scenario as above for two million time steps over five random seeds. Fig. 5.12 shows the mean episode reward across the 16 parallel environments at each time step. Initially, fewer than half the targets are tracked per episode. However, the agent quickly learns the task and consistently achieves the maximum reward after 0.25 million environment steps. After this point, it learns to track targets more quickly in order to obtain more reward. This behavior is illustrated in Fig. 5.13, where the average length of each episode steadily decreases from 2000 to 500 over the training interval.

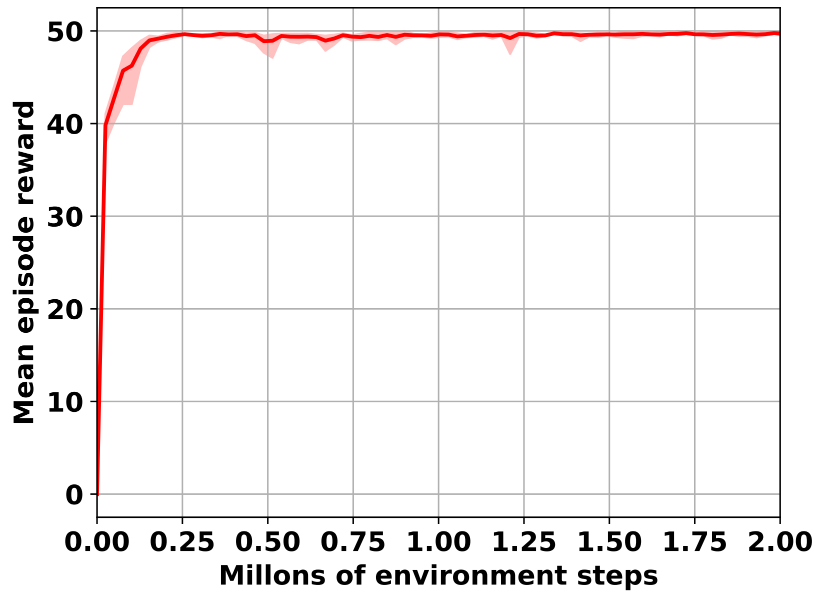


Figure 5.12: Mean episode reward across all parallel environments

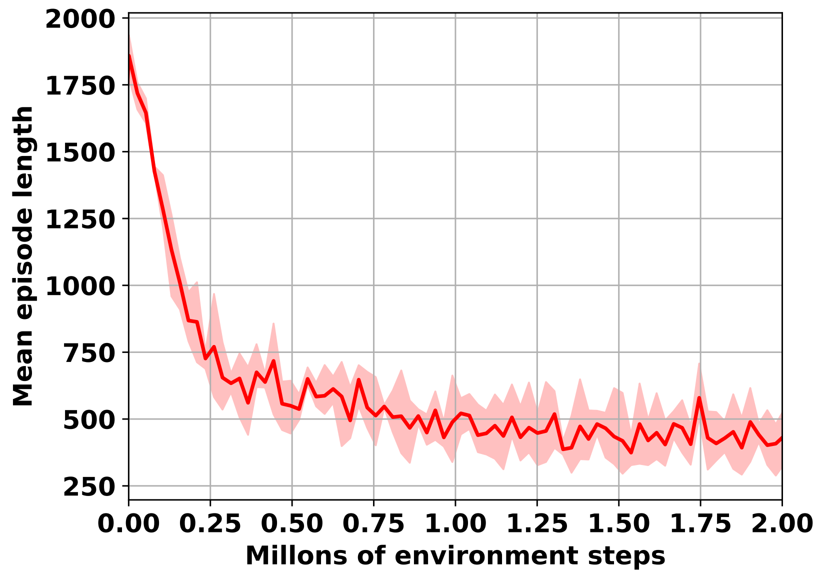


Figure 5.13: Mean episode length across all parallel environments

Fig. 5.14 shows the entropy of the learned policy as training progresses. Initially, the agent takes random actions in order to explore the environment. As the agent gains experience, the policy improves and becomes more exploitative,

causing the entropy to monotonically decrease before converging to around 0.8.

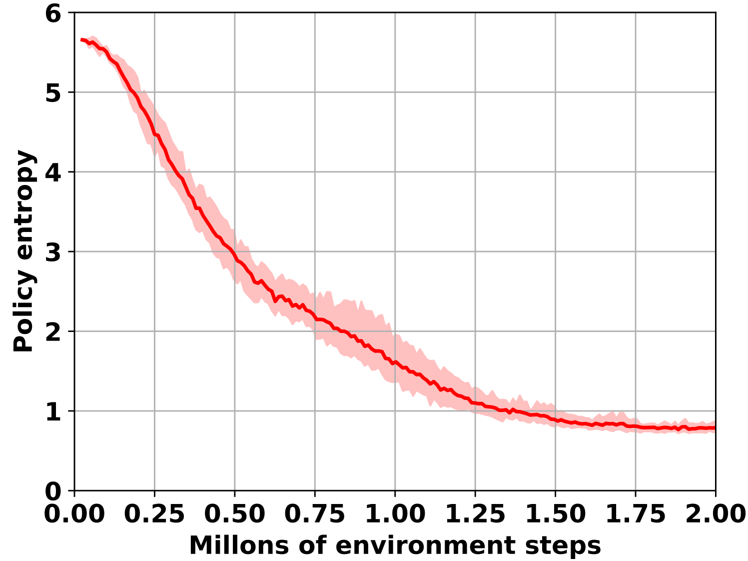


Figure 5.14: Policy entropy

Fig. 5.15 shows the policy network loss (computed using Eq. (2.25) for each environment step. Although the policy loss is highly oscillatory, it decreases slightly over the course of training before converging to a mean of -0.02. The value network loss from Eq. 2.24 is shown in Fig. 5.16. As the agent obtains more reward, the value loss increases before converging to a stable value. Unlike in supervised learning, a low loss value does not necessarily correspond to improved performance, and these figures are prototypical loss curves that show that the learning process is proceeding as expected.

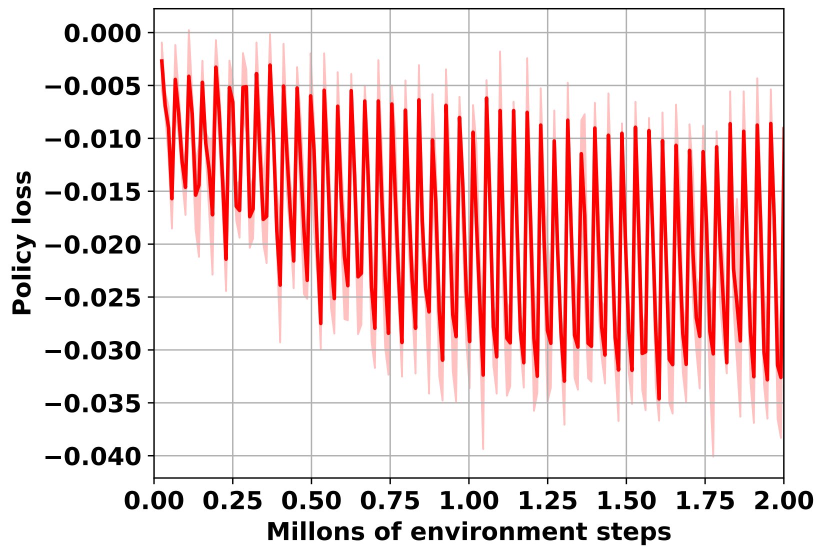


Figure 5.15: Policy (actor) loss over two million time steps

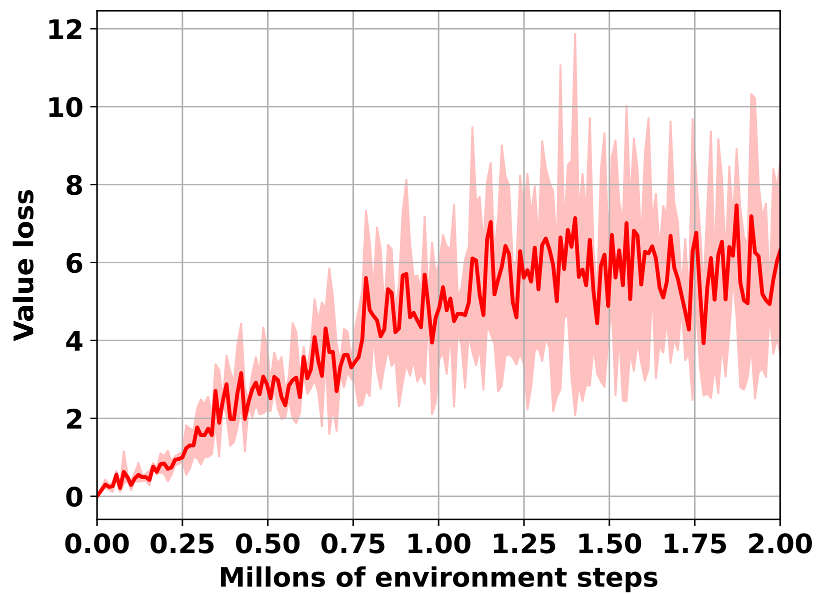


Figure 5.16: Value (critic) loss over two million time steps

The results in that have been discussed in this section indicate that the SPSO algorithm provides a Markovian representation of the surveillance task that can be used to train cognitive agents with RL. By keeping the state representation

compact, the agent successfully learned the task using the simple fully-connected network in Fig. 5.9. This network architecture does not require many parameters and is computationally efficient enough to implement in a real time system. These experiments also demonstrated the flexibility of the RL approach: with only a small modification to the network architecture, the agent learned to effectively steer and spoil the beam to improve its performance on the surveillance task.

Chapter 6

Conclusions and Future Work

In this thesis, a novel variant of the classic PSO algorithm was developed to encode prior knowledge of the location of undetected targets. This method, known as SPSO, is intended to inform decision making during the surveillance process. By focusing the search on regions that are more likely to contain new targets, more of the radar timeline can be allocated towards resource-intensive tasks such as track maintenance with no loss in performance. The resulting algorithm is intuitive and computationally efficient enough for real time operation, does not depend on the underlying motion model of the targets being detected or the number of targets, and outputs a constant-length tensor that can be used as the input to a neural network.

Two use cases for the SPSO algorithm was presented. In the first case, the raw swarm output for the algorithm was used to directly select the steering angle for the beam from a phased array radar system. In the second case, the particle swarm output was used to compose a state representation that was used to train a cognitive agent to both steer and spoil (on transmit) a beam using reinforcement learning. Both use cases significantly outperformed a traditional raster scan strategy in all environments considered.

All simulations in this work considered a specific set of scenarios, where tar-

gets were initialized in at least one cluster, each centered around some point in the angular search region. Although these scenarios effectively demonstrated the performance of the algorithm, future work could be performed to consider more interesting and realistic target formations. This work considered agents whose only objective was surveillance. A more holistic view of the performance of the algorithm could be gained by integrating it into a system that must also balance resources with tracking tasks.

There are numerous extensions to this work that could be pursued in future work. The most obvious extension is to augment the action space of the RL agent so that it also selects resources such as the integration time, PRF, and false alarm rate of each dwell alongside the beam parameters. To more effectively balance resources between search and tracking functions, an MDP representation of the tracking task could be developed. This tracking state representation could be encoded with the SPSO representation described in Chapter 5, or a multi-agent RL approach could be taken, treating the each task as an independent agent [39].

References

[1] R. Mailloux, *Phased Array Antenna Handbook* (Artech House antennas and electromagnetics analysis library). Artech House, 2018, ISBN: 9781630810290. [Online]. Available: https://books.google.com/books?id=vMW_tAEACAAJ.

[2] C. Fulton, M. Yeary, D. Thompson, J. Lake, and A. Mitchell, "Digital phased arrays: Challenges and opportunities," *Proceedings of the IEEE*, vol. 104, no. 3, pp. 487–503, 2016. DOI: 10.1109/JPROC.2015.2501804.

[3] P. Moo and Z. Ding, *Adaptive Radar Resource Management*. Elsevier Science, 2015, ISBN: 9780128042106. [Online]. Available: https://books.google.com/books?id=dKDDCQAAQBAJ.

[4] S. Blackman and R. Popoli, *Design and Analysis of Modern Tracking Systems* (Artech House radar library). Artech House, 1999, ISBN: 9781580530064. [Online]. Available: https://books.google.com/books?id=lTIfAQAAIAAJ.

[5] F. Katsilieris, A. Charlish, and Y. Boers, "Towards an online, adaptive algorithm for radar surveillance control," in *2012 Workshop on Sensor Data Fusion: Trends, Solutions, Applications (SDF)*, 2012, pp. 66–71. DOI: 10.1109/SDF.2012.6327910.

[6] J. L. Williams, "Search theory approaches to radar resource allocation," *Proc. Defense Appl. Signal Process.*, 2011.

[7] R. Sutton and A. Barto, *Reinforcement Learning, second edition: An Introduction* (Adaptive Computation and Machine Learning series). MIT Press, 2018, ISBN: 9780262039246. [Online]. Available: https://books.google.com/books?id=5s-MEAAAQBAJ.

[8] D. Silver, T. Hubert, J. Schrittwieser, *et al.*, *Mastering chess and shogi by self-play with a general reinforcement learning algorithm*, 2017. DOI: 10.48550/ARXIV.1712.01815. [Online]. Available: https://arxiv.org/abs/1712.01815.

[9] A. Fawzi, M. Balog, A. Huang, *et al.*, “Discovering faster matrix multiplication algorithms with reinforcement learning,” *Nature*, vol. 610, no. 7930, pp. 47–53, 2022.

[10] S. Flandermeyer, *Multi-function phased array radar simulator*, `https://github.com/ShaneFlandermeyer/mpar-sim`, 2023.

[11] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, `http://www.deeplearningbook.org`.

[12] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, *Playing atari with deep reinforcement learning*, 2013. DOI: `10.48550/ARXIV.1312.5602`. [Online]. Available: `https://arxiv.org/abs/1312.5602`.

[13] M. Hausknecht and P. Stone, *Deep recurrent q-learning for partially observable mdps*, 2015. DOI: `10.48550/ARXIV.1507.06527`. [Online]. Available: `https://arxiv.org/abs/1507.06527`.

[14] J. Xiong, Q. Wang, Z. Yang, *et al.*, *Parametrized deep q-networks learning: Reinforcement learning with discrete-continuous hybrid action space*, 2018. DOI: `10.48550/ARXIV.1810.06394`. [Online]. Available: `https://arxiv.org/abs/1810.06394`.

[15] M. Neunert, A. Abdolmaleki, M. Wulfmeier, *et al.*, *Continuous-discrete reinforcement learning for hybrid control in robotics*, 2020. DOI: `10.48550/ARXIV.2001.00449`. [Online]. Available: `https://arxiv.org/abs/2001.00449`.

[16] I. G. B. Petrazzini and E. A. Antonelo, *Proximal policy optimization with continuous bounded action space via the beta distribution*, 2021. DOI: `10.48550/ARXIV.2111.02202`. [Online]. Available: `https://arxiv.org/abs/2111.02202`.

[17] A. Y. Ng, D. Harada, and S. J. Russell, “Policy invariance under reward transformations: Theory and application to reward shaping,” in *Proceedings of the Sixteenth International Conference on Machine Learning*, ser. ICML ’99, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, 278–287, ISBN: 1558606122.

[18] M. G. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos, *Unifying count-based exploration and intrinsic motivation*, 2016.

DOI: 10.48550/ARXIV.1606.01868. [Online]. Available: https://arxiv.org/abs/1606.01868.

[19] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, *Curiosity-driven exploration by self-supervised prediction*, 2017. DOI: 10.48550/ARXIV.1705.05363. [Online]. Available: https://arxiv.org/abs/1705.05363.

[20] T. P. Lillicrap, J. J. Hunt, A. Pritzel, *et al.*, *Continuous control with deep reinforcement learning*, 2015. DOI: 10.48550/ARXIV.1509.02971. [Online]. Available: https://arxiv.org/abs/1509.02971.

[21] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[22] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Mach. Learn.*, vol. 8, no. 3–4, 229–256, 1992, ISSN: 0885-6125. DOI: 10.1007/BF00992696. [Online]. Available: https://doi.org/10.1007/BF00992696.

[23] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, *High-dimensional continuous control using generalized advantage estimation*, 2015. DOI: 10.48550/ARXIV.1506.02438. [Online]. Available: https://arxiv.org/abs/1506.02438.

[24] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," in *Advances in Neural Information Processing Systems*, S. Solla, T. Leen, and K. Müller, Eds., vol. 12, MIT Press, 1999. [Online]. Available: https://proceedings.neurips.cc/paper/1999/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf.

[25] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, *Proximal policy optimization algorithms*, 2017. DOI: 10.48550/ARXIV.1707.06347. [Online]. Available: https://arxiv.org/abs/1707.06347.

[26] M. Andrychowicz, A. Raichuk, P. Stańczyk, *et al.*, *What matters in on-policy reinforcement learning? a large-scale empirical study*, 2020. DOI: 10.48550/ARXIV.2006.05990. [Online]. Available: https://arxiv.org/abs/2006.05990.

[27] K. Cobbe, J. Hilton, O. Klimov, and J. Schulman, *Phasic policy gradient*, 2020. DOI: 10.48550/ARXIV.2009.04416. [Online]. Available: https://arxiv.org/abs/2009.04416.

[28] M. Richards, *Fundamentals of Radar Signal Processing, Second Edition*. McGraw-Hill Education, 2014, ISBN: 9780071798327. [Online]. Available: https://books.google.com/books?id=qGZCAQAACAAJ.

[29] H. Van Trees, *Optimum Array Processing: Part IV of Detection, Estimation, and Modulation Theory* (Detection, Estimation, and Modulation Theory). Wiley, 2002, ISBN: 9780471093909. [Online]. Available: https://books.google.com/books?id=J5TZDwAAQBAJ.

[30] T. K. Yaakov Bar-Shalom X.-Rong Li, "Estimation for kinematic models," in *Estimation with Applications to Tracking and Navigation*. John Wiley and Sons, Ltd, 2001, ch. 6, pp. 267–299, ISBN: 9780471221272. DOI: https://doi.org/10.1002/0471221279.ch6. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/0471221279.ch6. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/0471221279.ch6.

[31] D. Barton, *Radar Equations for Modern Radar* (Artech House radar library). Artech House, 2013, ISBN: 9781608075218. [Online]. Available: https://books.google.com/books?id=n4FsPITxhK4C.

[32] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95 - International Conference on Neural Networks*, vol. 4, 1995, 1942–1948 vol.4. DOI: 10.1109/ICNN.1995.488968.

[33] A. P. Engelbrecht, *Computational Intelligence: An Introduction*, 2nd. Wiley Publishing, 2007, ISBN: 0470035617.

[34] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, 509–517, 1975, ISSN: 0001-0782. DOI: 10.1145/361002.361007. [Online]. Available: https://doi.org/10.1145/361002.361007.

[35] V. Miranda and N. Fonseca, "Epso - best-of-two-worlds meta-heuristic applied to power system problems," in *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*, vol. 2, 2002, 1080–1085 vol.2. DOI: 10.1109/CEC.2002.1004393.

[36] P. Angeline, "Using selection to improve particle swarm optimization," in *1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98TH8360)*, 1998, pp. 84–89. DOI: 10.1109/ICEC.1998.699327.

[37] C. A. Koay and D. Srinivasan, "Particle swarm optimization-based approach for generator maintenance scheduling," in *Proceedings of the 2003 IEEE Swarm Intelligence Symposium. SIS'03 (Cat. No.03EX706)*, 2003, pp. 167–173. DOI: 10.1109/SIS.2003.1202263.

[38] J. Kennedy, "The particle swarm: Social adaptation of knowledge," in *Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC '97)*, 1997, pp. 303–308. DOI: 10.1109/ICEC.1997.592326.

[39] K. Zhang, Z. Yang, and T. Başar, *Multi-agent reinforcement learning: A selective overview of theories and algorithms*, 2019. DOI: 10.48550/ARXIV.1911.10635. [Online]. Available: https://arxiv.org/abs/1911.10635.

[40] J. Achiam, "Spinning Up in Deep Reinforcement Learning," 2018.

Appendix A

The Policy Gradient Theorem

To update the parameters of the policy function π_θ using stochastic gradient ascent as discussed in Ch. 2, it is necessary to derive an analytical expression for the gradient of agent performance J with respect to the policy parameters θ, known as the policy gradient. The derivation below follows largely from [40]. The goal is to maximize the return $G_t(\tau)$ for a trajectory τ with T steps, or

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[G_t(\tau)\right] \quad t = 0, \ldots, T-1 \tag{A.1}$$

The gradient is then given by

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}\left[G_t(\tau)\right] \tag{A.2}$$

Expanding the expectation and bringing the gradient into the integral gives

$$\nabla_\theta J(\theta) = \int_{\tau\sim\pi_\theta} \nabla_\theta p(\tau|\theta) G_t(\tau) \tag{A.3}$$

where $p(\tau|\theta)$ is the probability of trajectory τ occurring under policy π_θ. In a Markov decision process, state transitions are assumed to be independent and

$p(\tau|\theta)$ can be expressed as a chain of multiplications

$$\begin{aligned} p(\tau|\theta) &= p(s_0, a_0, \ldots, s_T, a_T|\theta) \\ &= \mu(s_0) \prod_{t=0}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t) \end{aligned} \tag{A.4}$$

where $\mu(s)$ is the initial state distribution that gives the probability that an episode begins in state s and $p(s_{t+1}|s_t, a_t)$ is the environment's state transition function from (2.5). If T is large, this expression may become numerically unstable since all numbers in the multiplication are less than one. It is therefore more convenient to work with the log-probability of the trajectory

$$\log p(\tau|\theta) = \log \mu(s_0) + \sum_{t=0}^{T} \log \pi_\theta(a_t|s_t) + \sum_{t=0}^{T} \log p(s_{t+1}|s_t, a_t) \tag{A.5}$$

To substitute this into (A.3), the $\nabla_\theta p(\tau|\theta)$ term must be expressed as a log-probability. From the chain rule of calculus, the gradient of the log-probability is

$$\nabla_\theta \log p(\tau|\theta) = \frac{1}{p(\tau|\theta)} \nabla_\theta p(\tau|\theta) \tag{A.6}$$

which can be substituted into (A.3) to get

$$\nabla_\theta J(\theta) = \int_{\tau \sim \pi_\theta} p(\tau|\theta) \nabla_\theta \log p(\tau|\theta) G_t(\tau) \tag{A.7}$$

Only the second term in the log-probability (A.5) is a function of θ, so the policy gradient becomes

$$\begin{aligned} \nabla_\theta J(\theta) &= \int_{\tau \sim \pi_\theta} p(\tau|\theta) \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t(\tau) \\ &= \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t(\tau) \right] \end{aligned} \tag{A.8}$$

which is the key result of the policy gradient theorem. Since $\nabla_\theta J(\theta)$ takes the form of an expectation, it can be approximated using a sample mean with samples collected from multiple trajectories. For example, the REINFORCE algorithm [22] performs a stochastic gradient ascent after each trajectory to update the policy weights as

$$\theta_{t+1} = \theta_t + \alpha G_t \nabla_\theta \log \pi_\theta(a_t|s_t) \tag{A.9}$$

where α is the learning rate parameter that determines how quickly the policy parameters change at each time step.

Appendix B

Activation Functions

The output from the classic perceptron in Fig. 2.1 is an affine transformation of its inputs, followed by a nonlinear activation function. The choice of activation function is an important hyperparameter that can have a significant impact on the final performance of the model. An early choice was the logistic sigmoid function shown in Fig. B.1, which is defined as

$$\sigma(x) = \frac{1}{1 + \exp(-x)} \tag{B.1}$$

The sigmoid function is bounded in the range $[0, 1]$ and varies smoothly for small values of x before saturating. The unsaturated region provides gradient information that can be used to update the network parameters using a variant of the backpropagation algorithm. The hyperbolic tangent function (Fig. B.2) has similar properties, but is bounded in the range $[-1, 1]$. Although these activation functions work well for small networks, they are rarely used in large architectures. This is because for very large and very small inputs, these functions saturate and have nearly zero gradient. When backpropagation is used to update the network weights, the gradient gradually diminishes to zero during the backwards pass of the algorithm (known as the vanishing gradient problem). However, these activation functions are still commonly used in reinforcement learning to scale the

output of policy networks into a valid range for bounded continuous action spaces (see Section 2.2.1).

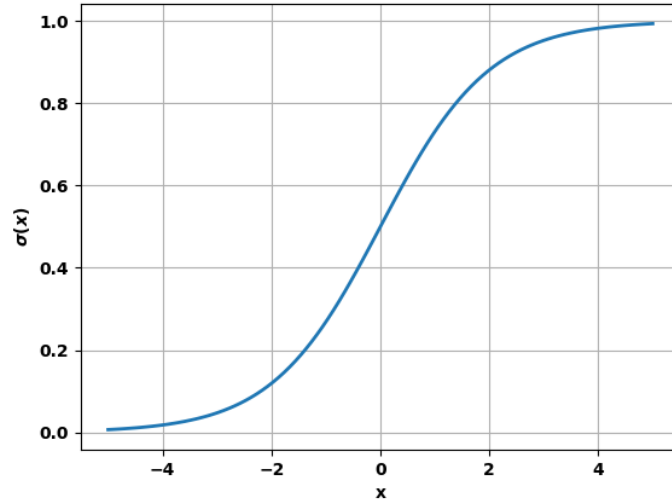


Figure B.1: Sigmoid activation function

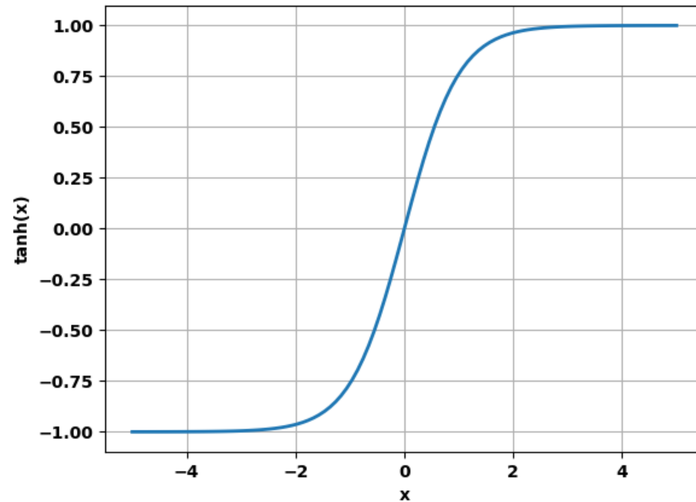


Figure B.2: Tanh activation function

In deep neural network architectures, it is more common to use activation functions that do not saturate in order to avoid the exploding/vanishing gradient

problem. One such activation function is the rectified linear unit (ReLU) function, which is characterized by the following expression

$$ReLU(x) = \max(0, x) \tag{B.2}$$

In other words, the ReLU function is a linear with a slope of 1 when $x \geq 0$ and zero when $x < 0$ (Fig. B.3). The ReLU function is used frequently in deep learning applications because it is easy to compute, nonlinear, and does not saturate for positive x.

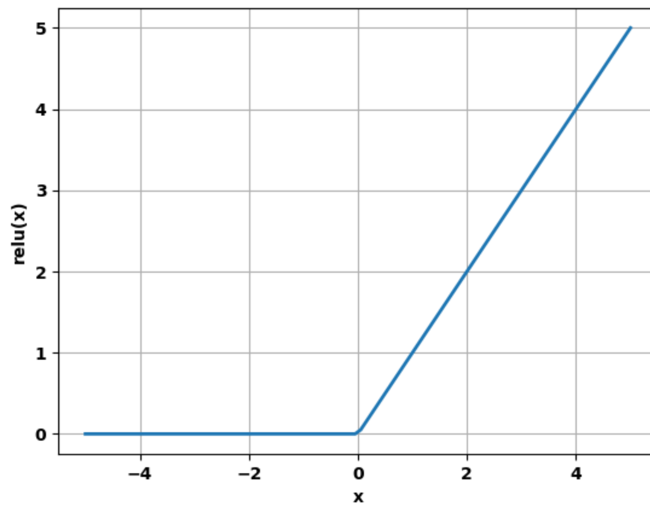


Figure B.3: ReLU activation function

Due to its general popularity, many variation of the ReLU activation have been developed for specific use cases. For example, the Softplus function (Fig. B.4) is a continuous approximation to the ReLU function and is defined as

$$Softplus(x) = \frac{1}{\beta} \log(1 + \exp(\beta x)) \tag{B.3}$$

The β parameter determines the smoothness of the transition, approaching ReLU

as $\beta \to \infty$. In this work, a Softplus activation is used for the beam steering agent in Chapter 5 to ensure that the output of the network that computes the action variance is always positive.

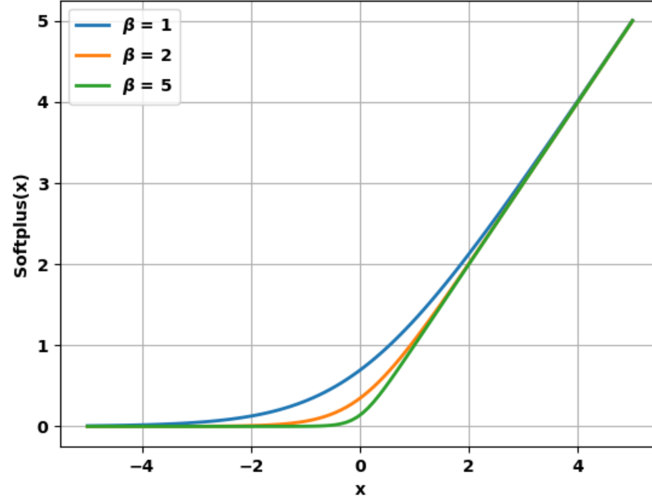


Figure B.4: Softplus activation function

In some situations, it is desirable to transform the output of a neural network into a probability distribution. For example, a classification network may output a set of values corresponding to the probability that the input belongs to each class. This is traditionally done with a softmax activation function. For inputs $x_1, x_2, \ldots, x_N$, the softmax activation function is computed as

$$Softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} \tag{B.4}$$

which ensures that the elements of the output lie in the range $[0, 1]$ and that the entire output sums to unity.

Appendix C

List of Acronyms and Abbreviations

ADC	Analog-to-digital converter
AEC	Agent-environment cycle
CNN	Convolutional neural network
CPU	Central processing unit
$CRLB$	Cramér-Rao lower bound
$CV-WNA$	Constant-velocity white noise Acceleration
EC	Evolutionary computation
GAE	Generalized advantage estimation
$GBPSO$	Global best particle swarm optimization
GPU	Graphical processing unit
KL	Kullback-Leibler
KNN	K-nearest neighbor
$LBPSO$	Local best particle swarm optimization
$LSTM$	Long short-term memory
MDP	Markov decision process
ML	Machine learning
MLE	Maximum likelihood estimator
PAR	Phased array radar
$POMDP$	Partially observable Markov decision process

PPG	Phasic policygradient
PPO	Proximal policy optimization
PSO	Particle swarm optimization
RCS	Radar cross section
RL	Reinforcement learning
RRM	Radar resource management
SGD	Stochastic gradient descent
SNR	Signal-to-noise ratio
$SPSO$	Surveillance particle swarm optimization
TWS	Track-while-scan
ULA	Uniform linear array
URA	Uniform rectangular array