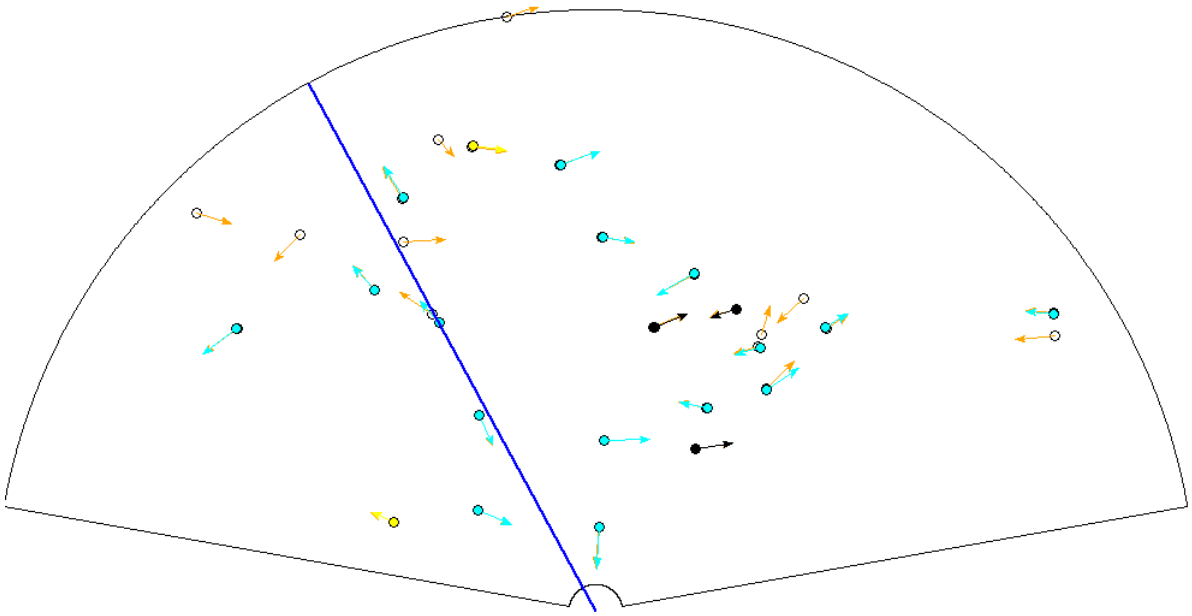


Adaptive Radar Illuminations with Deep Reinforcement Learning

Master's thesis

Albin Ekelund Karlsson
Samuel Sandelius

DEPARTMENT OF MATHEMATICAL SCIENCES

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2023
www.chalmers.se

Master's thesis 2023

Adaptive Radar Illuminations with Deep Reinforcement Learning

Illumination Scheduling for Long Range Surveillance Radar with the use of Proximal Policy Optimization

Albin Ekelund Karlsson
Samuel Sandelius

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Mathematical Science
Chalmers University of Technology
Gothenburg, Sweden 2023

Adaptive Radar Illuminations with Deep Reinforcement Learning
Albin Ekelund Karlsson
Samuel Sandelius

Supervisor: Adam Andersson, Saab AB
Examiner: Peter Helgesson, Department of Mathematical Science

Master's Thesis 2023
Department of Mathematical Sciences
Division of Applied Mathematics and Statistics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2023

Abstract

A modern radar antenna can direct its energy electronically without inertia or the need for mechanically steering. This opens up several degrees of freedom such as transmission direction and illumination time, and thus also the potential to optimise operation in real-time. Long range surveillance radars solve the trade-off between searching for new targets and tracking known targets. This optimisation is often rule-based.

In recent years, Reinforcement Learning (RL) Algorithms have been able to efficiently solve increasingly difficult tasks, such as mastering game strategies or solving complex control tasks. In this thesis we show that reinforcement learning can outperform such rule-based approaches for a simulated radar.

Keywords: Reinforcement Learning RL, Radar Target Tracking, Partially Observed Markov Decision Process POMDP, Active Electronically Scanned Array Antenna, Airborne Surveillance Radar

Acknowledgements

We would like to express our sincere gratitude to the following individuals for the invaluable support and guidance throughout this project.

First and foremost, we would like to extend our deepest appreciation to our supervisor Adam Andersson for his exceptional guidance and mentorship through the project. His guidance has been an essential part in shaping the direction and quality of this thesis, and we are truly grateful for his commitment, patience and dedication to our academic and professional development.

We would also like to thank our examiner Peter Helgesson for his commitment and meticulous evaluation and feedback during this project. His constructive criticism have greatly contributed to the refinement of this thesis. We are grateful for his thorough assessment and valuable suggestions, which have enhanced the quality of our thesis.

Lastly we would also like to extend our gratitude to Saab for this incredible opportunity to conduct a project within such and interesting subject, but also for providing us with the necessary resources, facilities, expertise.

Thank you.

Albin Ekelund Karlsson, Samuel Sandelius, Gothenburg, June 2023

Contents

List of Figures

List of Tables

1
Introduction

The surveillance task of a radar concerns the search of unknown targets and the tracking of known targets within the radar search volume. Sensing with a pulse Doppler radar works by sending pulses of electromagnetic radiation which reflects at distant objects. The reflections are recorded by the radar and through data processing, targets are detected and tracked. A simple illustration in Figure 1.1 shows the propagation and reflection.

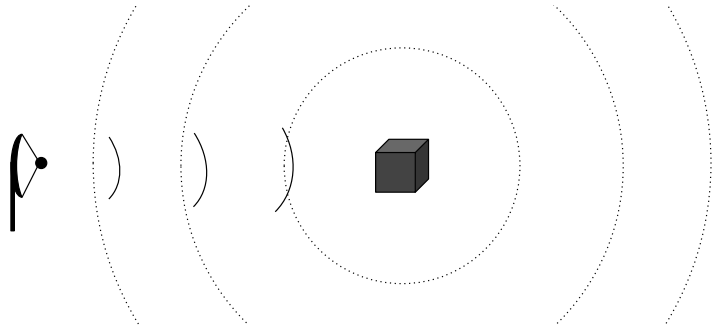


Figure 1.1: Simple illustration of the principle behind radar sensing. The small circle segments originate from the antenna to the left and propagate to the right. The waves are reflected on the object in all directions (dotted circles), eventually returning to the sender.

We consider an Active Electronically Scanned Array (AESA) antenna which consists of multiple smaller antennas that can take on different phase delays, allowing the direction of the pulse to be centered in a specified direction. Older radar systems are typically constructed with a static radiation pattern produced by the geometry of the aperture. Here, the energy is directed by mechanically rotating the entire aperture which is a slow process compared to changing the phase delays in an AESA antenna [3].

Tracking a target means correctly associating multiple detections with known tracks (series of detections originating from a single target) and, in this way, building trajectories for the discovered targets. When performing an illumination, two control inputs mainly affect the illumination outcome; the **direction** (angle of the illumination) and **integration time** (duration of illumination). The trade-off between searching for new targets and tracking known targets needs to be considered in order to maximize the number of tracks held. The primary cost of each action is the time spent as tracks should be frequently updated.

Traditionally, AESA based radar illuminations were scheduled using rule based priority lists with different local optimization of illumination features such as integration time. While such a scheme can result in very good tracking performance, there is good reason to believe that an algorithm that applies a policy based on global optimization may improve the performance. This thesis proposes a method of approaching this problem by Reinforcement Learning (RL) to find a *policy* (probability distribution over actions) that performs better than the baseline (priority lists) in a simulated environment.

1.1 Problem formulation

In our RL approach, an agent (decision maker derived from the policy) is set to perform the radar scheduling in a Python based simulation environment. A training method that is typically used in RL problems is Proximal Policy Optimization (PPO) [14] which has proven especially effective in control tasks. For this project, PPO was mainly used due to its simplicity and generally good performance without requiring much prior parameter tuning. This allowed us to easily adapt the algorithm for solving the illumination scheduling problem.

1.1.1 Evaluation

The agent is trained to schedule radar illuminations to maximize the total number of tracks held while also minimizing the time that new targets remain undiscovered. The performance is compared against the baseline as well as a radar which only searches for new targets by perpetually scanning the search volume, never re-illuminating existing tracks.

1.2 Context

In the present project, the target tracking process is optimized at a high decision-making level, making use of traditional tracking methods in order to discretize the state and action spaces. A first attempt at realizing this approach was made by Nathanson in his master's thesis [8] using a simulated environment and an agent trained using the PPO algorithm. While the simulation environment, tracking algorithm, and reinforcement learning agent were implemented and trained, the deep network agent did not outperform the baseline (priority lists), and the study was not conclusive in whether or not the training method could be successful after a more careful hyperparameter tunning. A strong hypothesis for this was that the simulator had too many layers of realism to it and the task was too hard to start with.

Since Nathanson carried out his work in his thesis [8], Saab's simulator environment has been updated with more options and tunable parameters, allowing, for example, the radar model to be simplified and apply a variable level of realism. As of this,

the problem has been simplified and the focus was initially shifted to optimising the training algorithm on a simpler problem, only later adding some of those realistic limitations and complexities to the environment. In the end we have added all the realism of Nathansson and obtained promising results.

2
Theory

In this chapter we introduce the theoretical foundations of concepts used throughout this project, specifically regarding Markov Decision Processes (MDP), Reinforcement Learning (RL) and an elementary tracking methodology.

Markov decision processes constitute a mathematical formalism used to study sequential desicion-making where state transitions are Markovian (independent of previous history) and controlled by discrete actions. They underpin both dynamic programming and reinforcement learning. In Section 2.1 the concept of MDPs is discussed generally and in the context of this project.

Reinforcement learning is a type of Machine Learning technique which optimizes an agent's decision making through trial and error in an environment. The agent receives feedback based on state transitions, known as a *rewards* which determines how desirable said state transition is. In Section 2.2 we introduce different approaches to solve RL problems such as deep Q-learning, Policy Gradient (PG) methods, Actor-critics, Trust Region Policy Optimization (TRPO), and Proximal Policy Optimization (PPO). In Chapter 4, the implementation used in this project is explained in more detail.

Lastly, we discuss target tracking which is the process of estimating target trajectories (tracks) out of discrete measurements. Traditional tracking algorithms are used in this project in order to create simple, discrete state and action spaces and to evaluate the performance. Section 2.3 covers Bayesian filtering and the Hungarian method which are techniques used for this purpose.

2.1 Markov Decision Processes

In reinforcement learning, problems are typically modelled as Markov decision processes. An MDP is defined as a tuple (S, A, P, R) where S is the set of available environment states s, A is the set of available actions a, $R(s, s')$ is the reward for the state transition from s to s', $P(s_{t+1} = s'|s_t = s, a_t = a)$ is the transition probability from s to s' given an action a between timesteps t and $t + 1$. Lastly, we have the policy $\pi(a|s)$, i.e., the probability distribution of actions a given a state s. While (S, A, P, R) are given by the problem, π is to be optimized. A simple illustration shows the process in Figure 2.1 below. Note how the MDP only depends on the current and previous time step; previous history is not necessary to determine the

future in an MDP.

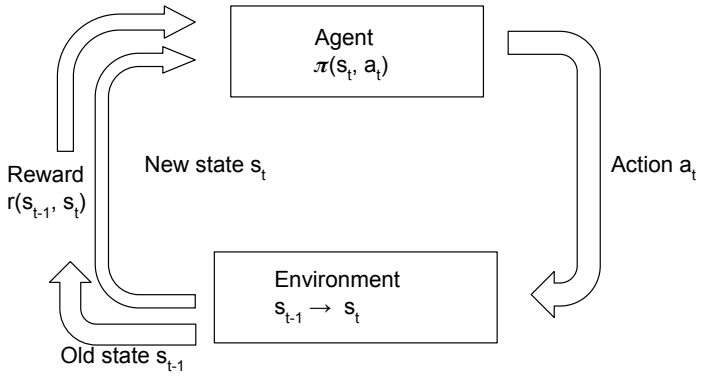


Figure 2.1: The MDP process and interaction between agent and environment.

The policy π is a probability distribution over actions, so we can rewrite the transition probabilities given π as $P_\pi = P_\pi(s_{t+1} = s' | s_t = s)$. The objective is to find the policy π that maximizes some cumulative function of rewards, usually the expected discounted return $\hat{R}$, Q-value $Q_\pi(s_t, a_t)$

$$Q_\pi(s_t, a_t) = E_\pi \left[\hat{R}(s_t | s_t, a_t) \right],$$

where

$$\hat{R}(s_{t_0}) = \sum_{t=t_0}^{\infty} \gamma^t R(s_t, s_{t+1})$$

and $\gamma \in (0, 1)$ is the discount factor. It is used to determine the importance of earlier rewards as opposed to rewards far into the future, as well as improving numerical stability by guaranteeing that the sum converges. A policy that maximizes $Q(s, a)$ is called an optimal policy π^*. Policies can be stochastic or deterministic (greedy). A stochastic policy is a probability distribution over actions and is optimal if it, for all $s \in S$ maximizes $E_{a \sim \pi}[Q_\pi(s, a)] = \sum_{a \in A} \pi(a|s) Q_\pi(s, a)$. An optimal deterministic policy can be written as a function of the state input

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\ Q_\pi(s, a). \tag{2.1}$$

To aid in finding a solution to an MDP or to be able to evaluate the performance of the MDP policy, a *value function* $V_\pi(s) = E_\pi \left[\hat{R}(s) | s \right]$ can be used. The value function predicts the estimated reward from a state s following the policy π. Basically, it disregards the action taken and only considers the expected return from the state itself. The value implicitly depends on π since it affects the transition probabilities $P_\pi(s, s')$. The value can be defined recursively as

$$V_\pi(s) := \sum_{s'} P_\pi(s' | s) \left(R(s, s') + \gamma V(s') \right), \tag{2.2}$$

which allows us to rewrite the Q-value in terms of value as

$$Q_\pi(s,a) = \sum_{s'} P_\pi(s'|a)\left(R(s,s') + \gamma V(s')\right). \tag{2.3}$$

With this formulation, we assume that the state S is known. If this assumption cannot be made, it is typical to instead treat the problem as a Partially Observable Markov Decision Process (POMDP) or the extension of the POMDP, the ρPOMDP, explained in Appendix A [1] [22]. The POMDP is an extension of the MDP framework which allows the agent to take actions based on partial and noisy information while still maximizing the expected return from true states s. It does this by utilizing a belief state b, a distribution over s. The POMDP can be extended further into a ρPOMDP where the agent can be rewarded on the belief state b directly rather than s.

2.2 Reinforcement Learning with Neural Networks

Similar to most modern machine learning, RL commonly uses neural networks. In recent years, deep Neural Networks have been used to solve increasingly difficult control tasks [19] [20] [16]. A neural network is composed of a set of functions referred to as neurons each with their own set of tunable parameters. Multiple neurons are typically arranged in forward-connected layers where a multidimensional input is propagated forward through the network, producing an output, see [5] for further information about neural networks. In short, the input is propagated through the network's *hidden layers* and the output can be parsed off the last layer, although many different, more complex architectures do exist. The hidden layers consist of neurons which are neither used as input nor output, and merely contribute with extra parameters (weights). All neurons otherwise operate in a similar manner and typically process inputs by first weighting the it using the neuron's weight parameter, then summing over all weighted inputs. A bias term is added according to the neuron's bias parameter and is finally passed through a nonlinear activation function, predetermined with no parameter dependence. The result is sent to the next neuron layer or read off output from the network, see Figure 2.2 for an example of a small neural network and its neuron structure.

With this arrangement, the network parameters can be optimised by minimizing a loss function between the input and the output, typically with the help of stochastic gradient descent. With such a large number of parameters and through the use of linear as well as nonlinear functions, even intricate nonlinear dependencies between the input and output can be learned. For Reinforcement Learning, the network outputs are directly fed to the input of some control task, and the loss is determined through some reward after stepping through the environment. The loss function is set up so that minimizing the loss corresponds to maximizing some total reward. The neural network can take the role of the policy $\pi^\theta(s,a)$ with parameters θ and can be used to solve an MDP.

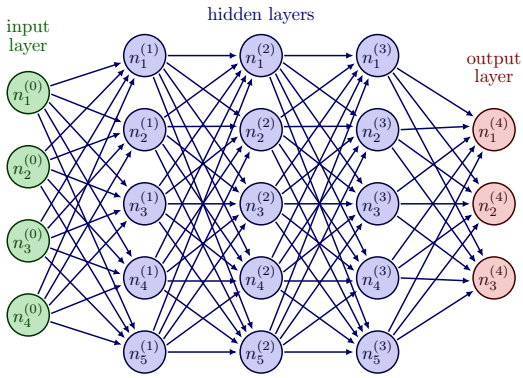


Figure 2.2: Example structure of a neural network. Neurons are labeled n_i^l for neuron i in layer l and forward connected to the next layer as shown by the blue arrows. This network has four input features and produces three outputs through three hidden layers of five neurons each.

2.2.1 Deep Q Learning

Deep Q learning is one of the more straightforward RL algorithms used to train a neural network and there are many references on the topic, for instance, the textbook [18]. "Deep" in this case refers to the use of neural networks as opposed to lookup tables, allowing the agent to interpolate in a large action space where sampling Q values for the entire state space may not be feasible. In deep Q learning, the idea is to have the network approximate the Q-value $Q(s, a)$ and find the optimal (deterministic) policy $\pi^*(s) = \arg\max_a Q(s, a)$. This is done by minimizing the loss

$$\mathcal{L}_t = \left(r_t + \gamma \hat{Q}_{\max_a}(s_{t+1}, a) - \hat{Q}(s_t, a_t)\right)^2.$$

Here, $r_t = R(s_t, s_{t+1})$ and $\hat{Q}(s, a)$ is the network output for action a, given input s. In order to determine the optimal parameters θ, rewards are sampled over many trajectories by *exploration*, typically via the ϵ-greedy policy: at every timestep, with a probability ϵ, pick a random action. Otherwise pick the action with the highest Q-value as in the optimal policy $\pi^*(s)$, referred to as *exploitation*. In order to solve an MDP using Deep Q learning, exploration is necessary in order to sample rewards from states which may still be unknown to the agent. Exploitation is used not only to fine tune the parameter optimization itself, but to also sample the state transition probability $P_\pi(s_{t+1} = s' | s_t = s, a_t = a)$ where π may be close to the optimal policy.

There are a few downsides to Deep Q Learning which makes it less practical compared to other methods in certain situations, some of them are listed below.

- Relatively low sample efficiency as only a single weight update is performed for each trajectory sample. In short, there is no way to know the optimal step size for the gradient descent algorithm, making convergence slow.
- Q-value initialization may have a completely different mean compared to discounted rewards obtained from the environment. While true for any algorithm, Deep Q Learning makes no attempt to normalize the sampled rewards.

- Trade off between exploration and exploitation includes an additional hyperparameter ϵ that requires tuning. On top of that, exploration occurs at random, even if the agent could potentially act with more certainty in some cases compared to others [11].
- Off-policy learning. As we shall get into later, Deep Q Learning is an off policy RL method, meaning the agent does not act according to what it thinks is optimal during training due to the exploration parameter forcing exploration at random. As a result, it is more difficult estimate the state transition probability $P_\pi(s'|s)$ and it may take longer to train.

The problems listed impact most RL algorithms, but there are strategies to mitigate these problems. We get into these types of algorithms in the following sections.

2.2.2 Policy Gradient methods

Policy Gradient (PG) [18], [10] is a term often used in RL. The principle of policy gradient is to learn a parameterized policy for selecting actions directly instead of indirectly by a Q-value estimate. However, value functions can still be used to aid the training procedure. When learning the policy parameters, the gradient of an objective function $J(\theta)$ is used. Here the vector θ denotes the parameter vector of the policy $\pi^\theta(a|s)$. The method wants to maximize the objective function $J(\theta)$, where

$$J(\theta) = E_{\pi_\theta}\left[r(\tau)\right],$$

Where $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_T, a_T, r_T)$ is the trajectory of length T, s_t is the state at time t, a_t is the chosen action at time t and r_t is the reward at time t and $r(\tau)$ is the total return over a trajectory τ.

The goal of the policy gradient methods is to use the method of gradient descent (or ascent) to directly maximize $J(\theta)$ in the direction of $\nabla J(\theta)$, using the update:

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t),$$

where $\nabla J(\theta_t)$ is the gradient of $J(\theta)$ and α the learning rate parameter. Before we introduce how to derive the gradient with the gradient theorem we will first introduce a common trick used in Deep learning refereed to as the "log trick". The log trick is given as:

$$f(x)\nabla_\theta \log f(x) = f(x)\frac{\nabla_\theta f(x)}{f(x)} = \nabla_\theta f(x)$$

using this together with the policy gradient Theorem, the gradient of $J(\theta)$ can be written as:

$$\nabla^\theta J(\theta) = E_{\pi_\theta}[r(\tau)\nabla \log \pi^\theta(\tau)] \tag{2.4}$$

$$= E_{\pi_\theta}[r(\tau)\sum_{t=1}^{T} \nabla \log \pi^\theta(s_t, a_t)]. \tag{2.5}$$

Here $\nabla \log \pi^\theta(s_t, a_t)$ is the score function or log-derivative of the policy using the "log trick" introduced above, which measures the sensitivity of the policy to changes in θ. The full proof of (2.4) can be found here [18, page: 325].

Modifying the policy directly in this way has its advantages. The most obvious is that there is no longer a need for the ϵ-greedy strategy. The entropy (describing the spread of the action distribution) of $\pi(s, a)$ is typically high for an untrained network with randomly initialized parameters which promotes exploration. The entropy decreases over time if the agent manages to differentiate between good and bad actions through $r(\tau)$, so that we can directly sample $P_\pi(s'|s)$ while simultaneously converging to the optimal policy [11].

2.2.3 Actor - Critic

A popular approach of modern RL is to use an actor-critic architecture (see for instance [21]). This combines different features from value-based methods such as Q-learning and policy-based methods. As we saw in the previous section, the gradient of the objective function $\nabla^\theta J(\theta)$ flips its sign together with the total return $r(\tau)$. Since neural networks are typically designed in a way that keeps the policy's output probabilities normalized (using for example a *softmax* output layer [4]), one could potentially consider for instance positive-only rewards, and still converge to an optimal policy. It is however both intuitive and sample efficient to consider positive rewards for "good" actions and negative rewards for "bad" actions, since this is directly reflected in the objective gradient. The goal of the actor - critic architecture is to achieve this effect.

The approach involves the use of both a policy function π^θ referred to as the actor, as well as a value function estimate V^ϕ referred to as the critic. Both of these can be separate neural networks with parameters θ and ϕ respectively. We use essentially the same objective function as for the policy gradient in (2.4), but replace the total return with the *advantage* $A(s, a) = Q_\pi(s, a) - V_\pi(s)$. The resulting objective function is

$$\nabla^\theta J(\theta) = E_{\pi_\theta}[A(s_t, a_t) \sum_{t=1}^{T} \nabla \log \pi^\theta(s_t, a_t)]. \tag{2.6}$$

The advantage is not usually known, so it is typically estimated using $V^\phi(s)$ and the trajectory return $r(\tau)$, for example as $\hat{A}(s, a) = r(\tau) - V^\phi(s)$. This changes the dynamics where previously the sign of the reward directly influenced the sign of the objective gradient. Instead, the sign depends on whether the obtained return is greater or smaller than the estimated value V^ϕ. Remember that the value function is implicitly dependent on the policy π and the state transition probabilities $P_\pi(s_{t+1} = s'|s_t = s)$, so positive advantage comes from actions which were better than what the critic expected based on previous trajectories. The value estimator can be trained directly on the total return $r(\tau)$ by, for example minimizing squared error.

2.2.4 Trust Region Policy Optimization

Trust Region Policy Optimization (TRPO) is a policy gradient method which seeks to further improve the stability of convergence of the policy update and does this by using a parameter to constraint the maximum step length during each update [15]. The change in the policy will not deviate to far from its earlier policy when updating, which allows for decreased risk of instability or convergence to local optima.

The main idea behind TRPO is to optimize the *surrogate function* which is used to approximate the true performance objective while also constraining the policy update to a trust region around the current policy. The surrogate function is the so-called clipped surrogate which is a modification of the previously discussed objective function:

$$L^{\text{CPI}}(\theta) = E_{s,a\sim\pi_{\theta_{\text{old}}}}\left[\frac{\pi^{\theta}(a|s)}{\pi^{\theta_{\text{old}}}(a|s)}A^{\pi^{\theta_{\text{old}}}}(s,a)\right],$$

where θ_{old} are the parameters that were used when sampling the trajectory and $A_t^{\pi_{\theta_{\text{old}}}}(s,a)$ is the advantage function constructed by those policy parameters. The update is then optimised within the trust region defined by a maximally allowed Kullback-Leibler divergence which is stated as a constraint. The idea of keeping a trust region is to make sure that the update does not differ too much from the current policy. However TRPO comes with some disadvantages. Calculating the KL divergence can be computationally expensive and time consuming based on the dimensions of the action space. Moreover, from a practical point of view, the hands-on implementation is also quite complicated.

2.2.5 Proximal Policy Optimization

Proximal policy optimization (PPO) is a Policy gradient method that seeks to capture the spirit of TRPO, but in a way which is both more efficient and simpler. Instead of solving the problem with a complex second-order method, it uses a first-order method and applies some extra tricks not to move too far away from the earlier policy. There are two variants of PPO, but in this thesis, we will focus on the variant known as PPO-clip. Instead of using the KL divergence as a constraint, PPO uses specialized clipping for the objective function to limit the distance one can move from the earlier policy, see [14] for further information. The updating scheme follows

$$\theta_{k+1} = \arg\max_{\theta} E_{s,a\sim\pi_{\theta_k}}[L(s,a,\theta_k,\theta)]$$

where L is given by

$$L(s,a,\theta_k,\theta) = \min\left(\frac{\pi^{\theta}(a|s)}{\pi^{\theta_k}(a|s)}A^{\pi_{\theta_k}}(s,a), g(\epsilon, A^{\pi_{\theta_k}}(s,a))\right)$$

and

$$g(\epsilon, A) = \begin{cases}(1+\epsilon)A & A \geq 0\\ (1-\epsilon)A & A < 0.\end{cases}$$

Here, ϵ is the clipping parameter. With this clipping method, the objective function is updated using the regular policy gradient objective until the policy diverges too far from the original policy. At that point, the objective is clipped, causing the gradient to become zero, effectively halting the learning process until a new trajectory is sampled. This makes the learning process much simpler compared to optimizing the policy completely within a trust region.

2.3 Tracking

A real-world radar observation is a signal composed of informative target data and random noise from the environment. In order to get reasonable estimates of the target states, the noisy parts must be eliminated or at least reduced. One way to achieve noise reduction is by utilizing so-called Bayesian filtering. When estimating target states, this estimation is called a track. However, Bayesian filtering only solves the problem of target estimation, not the problem of multiple target tracking. Solving this problem requires some association to distinguish between tracks, which is essential in radar surveillance. One way to solve this assignment problem is the so-called Hungarian method. This method was chosen because of its simplicity and was already implemented in the simulator. However, in modern implementations, methods like multi-hypothesis tracking [2] or Multi-Object Tracking [6] are instead used.

2.3.1 Bayesian Filtering

Filtering theory is a branch of statistics dealing with the problem of estimating hidden states of a Markov process via noisy measurements. Bayesian filtering theory is simply the Bayesian formulation of this theory [13]. It calculates the marginal posterior distribution (or filtering distribution) $P(x_k|y_{1:k})$ of state x_k at timestep k based on the measurement history up to the current timestep $y_{1:k}$. The posterior distribution $P(x_k|y_{1:k})$ and the predicted distribution $P(x_k|y_{1:k-1})$ at timestep k are updated in a recursive manner. The recursion starts from the prior distribution x_0, thereafter the prediction distribution of state x_k is calculated at time k using the Chapman equation:

$$p(x_k|y_{1:k-1}) = \int p(x_k|x_{k-1})p(x_{k-1}|y_{1:k-1})dx_{k-1} \tag{2.7}$$

Then, using Bayes' rule and given the measurement y_k at timestep k, the posterior distribution of state x_k can be calculated according to

$$p(x_k|y_{1:k}) = \frac{p(y_k|x_k)p(x_k|y_{1:k-1})}{p(y_k|y_{1:k-1})}. \tag{2.8}$$

Here $p(y_k|y_{1:k-1})$ is the normalization constant given as

$$p(y_k|y_{1:k-1}) = \int p(y_k|x_k)p(x_k|y_{1:k-1})dx_k. \tag{2.9}$$

The recursive functions from the Bayesian equation work for linear Gaussian and nonlinear Gaussian state space models. In the case of a linear model with Gaussian noise, the filter distribution is also Gaussian. In the linear case the state and measurement models read:

$$\begin{aligned} x_k &= A_{k-1}x_{k-1} + q_{k-1} \\ y_k &= H_k x_k + r_k \end{aligned} \tag{2.10}$$

Here x_k is the state, y_k is the measurements (observations), $q_{k-1} \sim N(0, Q_k)$ is the process noise and $r_k \sim N(0, R_k)$ is the measurement noise, A_{k-1} is the transition matrix of the model, and H_k the measurement model matrix. The closed-form solution to this system is called the Kalman filter (KF) [13] given from (2.7), (2.8), (2.9) and reads

$$p(x_k|y_{1:k-1}) = N(x_k|m_k^-, P_K^-) \tag{2.11}$$
$$p(x_k|y_{1:k}) = N(x_k|m_k, P_k) \tag{2.12}$$
$$p(y_k|y_{1:k-1}) = N(y_k|H_K m_k^-, S_k) \tag{2.13}$$

where $p(x_k|y_{1:k-1})$ is the predicted distribution, $p(x_k|y_{1:k})$ is the posterior distribution and $p(y_k|y_{1:k-1})$ the predictive posterior distribution. The parameters above can be computed in two steps; prediction and update. In the prediction step we predict the state estimate m_k^- and the error covariance P_k^- according to

$$\begin{aligned} m_k^- &= A_{k-1}m_{k-1} \\ P_k^- &= A_{k-1}P_{k-1}A_{k-1}^T + Q_{k-1}. \end{aligned}$$

In the update step, we compute the measurement residual v_k the covaraince residual S_k, the Kalman gain K_k, update the state estimate m_k and error covariance P_k as follows:

$$\begin{aligned} v_k &= y_k - H_k m_k^-, \\ S_k &= H_k P_k^- H_k^T + R_k, \\ K_k &= P_k^- H_k^T S_k^{-1} \\ m_k &= m_k^- + K_k v_k \\ P_k &= P_k^- - K_k S_k K_k^T \end{aligned}$$

Starting from the prior mean and covariance m_0 and P_0.

While the Kalman filter gives an exact solution to the filtering problem in the linear case, in many applications, models are nonlinear, which makes the Kalman filter not applicable. Fortunately, there are approximate versions of the Kalman filter theory for nonlinear models. One is called the extended Kalman filter (EKF), which approximates the non-gaussian filter distribution using linearisation. The main idea of the EKF is to utilize the first-order Taylor approximation to assume a Gaussian approximation of the filter densities around the last prediction [13]. Using the same

assumptions for the EKF as for the KF, with the differences that f and h are differentiable, the filter density can be approximated as

$$p(x_k|z_{1:k-1}) \simeq N(x_k|m_k^-, P_K^-) \tag{2.14}$$
$$p(x_k|z_{1:k}) \simeq N(x_k|m_k, P_k) \tag{2.15}$$
$$p(z_k|z_{1:k-1}) \simeq N(z_k|H_K m_k^-, S_k). \tag{2.16}$$

From that, the prediction of the state estimate m_k^- and the error covariance P_k^- instead becomes

$$\begin{aligned} m_k^- &= f(m_{k-1}) \\ P_k^- &= F_x(m_{k-1})P_{k-1}F_x^T(m_{k-1}) + Q_{k-1}. \end{aligned}$$

And the updates of the measurement residual v_k, the covariance residual S_k and the Kalman gain K_k instead becomes

$$\begin{aligned} v_k &= y_k - h(m_k^-), \\ S_k &= H_x(m_k^-)P_k^- H_x^T(m_k^-) + R_k, \\ K_k &= P_k^- H_x^T(m_k^-)S_k^{-1}, \\ m_k &= m_k^- + K_k v_k, \\ P_k &= P_k^- - K_k S_k K_k^T. \end{aligned}$$

Here $H_x(m_k^-)$ and $F_x(m_{k-1})$ are the jacobian of f and h of state x evaluated at m_k^-

2.3.2 Hungarian method

The Hungarian method, or the Hungarian algorithm, is an optimization algorithm used to solve assignment problems. The optimization problem involves finding the most optimal assignment, in this case, for a set of detection to a set of tracks, given a cost in the way of a distance between the detections and tracks. The Hungarian method is an algorithm that was developed by Harold Kuhn in 1955. Kuhn named the algorithm Hungarian because of two Hungarian mathematicians, Dénes Kőnig and Jenő Egerváry, since the algorithm was based on their earlier works [7].

Imagine you obtain a set of detections and have a group of already existing tracks, and you want to assign each detection to a track in the most efficient way possible. Since each of the detections has a given distance to each track, the goal is to find the assignment which minimizes the overall distance.

The Hungarian method solves this problem by utilizing a matrix representation of distance. The matrix consists of columns representing the tracks and rows representing the detections. Each cell in the matrix contains the distance associated with assigning a particular detection to a particular track.

The method begins by identifying paths called "augmenting paths" in the matrix representation. Here, an augmenting path is a path that begins with an unassigned

detection and moves to an unassigned track. The goal is to find a set of paths covering all the unassigned cells in the matrix. Once the augmenting paths are found, the next step in the method is to proceed to modify the assignments based on these paths. It starts by selecting the path with the lowest distance. It thereafter updates the assignment along this path, switching the currently assigned detections with the unassigned ones until all the cells in the matrix are assigned. By iteratively finding paths and updating assignments, the Hungarian method eventually finds an optimal solution where the overall distance is minimized.

2.3.2.1 Mahalanobis distance

The algorithm used to generate the cost matrix in the Hungarian algorithm is the Mahalanobis distance. Mahalanobis distance measures the distance between a point x, in our case a detection, and a distribution $D \sim N(\mu, P)$, with the tracks position μ and the tracks covariance matrix P. The distance is then measured in how many standard deviations the detection x is from the track μ given the uncertainty of the track as P:

$$d_M(x, \mu; P) = \sqrt{(x - \mu)^T P^{-1} (x - \mu)} \tag{2.17}$$

3
Radar Simulator

The simulator used for this project is based on the simulator written and used by Nathanson [8]. However, for this project, the codebase has been updated with more variable parameters, allowing us to change the complexity of the simulation environment. This chapter presents how the simulator works, the features relevant to this project, and some fundamental theories necessary to understand the reasoning behind the implementation.

3.1 Radar basics

RAdio (Aim) Detection And Ranging (Radar) is a technique used to detect and gain information about distant targets with the help of radio waves. A radar sends electromagnetic radiation in a direction that reflects off a target. A receiver picks up the reflection to extract information about the target. In particular, its position can be determined by the response time, and a Doppler shift reveals its radial velocity towards the radar. The Doppler effect is manifested through a shift in frequency as a wave bounces off a moving target. The constant velocity of electromagnetic waves makes it possible to accurately measure the Doppler effect with signal processing [9].

Radars can be separated into two categories, those that send waves continuously while simultaneously listening and those that transmit a fraction of the time and listens the rest of the time, repeatedly. Both techniques have advantages and disadvantages, but in the context of airborne radars, the most common radar is the second one, the *pulse Doppler radar*. The primary advantage of the pulse Doppler radar as opposed to continuous wave (CW) radar is the increased range and accurate localization of targets due to their concentrated pulses of large gain [9].

3.1.1 Pulse Radar

Airborne radars usually have one antenna used for both transmitting and receiving. By the time the reflection of the pulse returns to the antenna, the power is greatly diminished and must be noticeably higher than the background noise to detect an object reliably. The signal energy received from a target is given by:

$$S_e = \frac{P_{\text{avg}} A_{\text{e}}^2 t_{\text{ot}} \cdot \text{RCS}}{4\pi R^4 \lambda^2 L}, \tag{3.1}$$

where S_{e} is the signal energy, P_{avg} is the average transmitter power, R is the range, RCS is the radar cross section of the target, t_{ot} is the time-on-target (integration

time), λ is the wavelength and L represents the general losses obtained, e.g., from processing and filtering. We also have the function for noise, which is given by:

$$N_{\mathrm{e}} = kT_{\mathrm{s}}$$

where k is Boltzmann's constant, T_s system noise temperature. Given these two equations, we can calculate the SNR. For further reading, see [9]

$$\mathrm{SNR} = S_{\mathrm{e}}/N_{\mathrm{e}} = \frac{P_{\mathrm{avg}} A_e^2 t_{\mathrm{ot}} \cdot \mathrm{RCS}}{4\pi R^4 \lambda^2 L k T_{\mathrm{s}}}.$$

Since most of these variables are constant, we can write the expression for SNR using an equivalent constant K. We also include the area A_e with its dependence on the azimuth angle from the radar θ extracted.

$$\mathrm{SNR}_{\mathrm{i}} = K \cdot \frac{\mathrm{RCS} \cdot A_{\mathrm{e}}^2 t_{\mathrm{ot}}}{R^4} = K' \cdot \frac{\mathrm{RCS} \cdot \cos(\theta)^2 t_{\mathrm{ot}}}{R^4}. \tag{3.2}$$

3.1.2 Waveform and Integration Time

In radar surveillance, the transmitted signal is often referred to as *waveform*. The waveform has four characteristics, carrier frequency, pulse width, modulation within each pulse or pulse to pulse, and rate at which the pulses are transmitted (pulse repetition frequency, see Section 3.1.3). However, when referring to the waveform, we mean the pulse width regarding the number of pulses sent. Since both the carrier frequency and Modulation are assumed to be constant. In the simulation, the available waveforms contain $w_i \in \{128, 256, 512\}$ pulses and differ only by the number of pulses used.

Integration time is the time it takes for one illumination, i.e., the time for all w_i pulses to be sent, plus a *burn-in* time taken for the first pulse to return from the distant radar horizon. Recorded data during the burn-in time is challenging to process because the data are non-stationary, contaminated with pulses from the illumination before, and not used when processing the signal. As for the total integration time, we denote it $I_{128}, I_{256}, I_{512}$ for the different waveforms, respectively which can be found in Table 3.1.

Number of pulses	128	256	512
Time (s)	0.01197	0.02093	0.03885

Table 3.1: Integration times for different number of pulses used.

3.1.3 Pulse Repetition Frequency and Resolving Measurements

The radar sends multiple pulses at a given frequency to obtain a measurement from a target. The pulse transmission rate is called the *Pulse Repetition Frequency*

(PRF). However, even if one sends multiple pulses using the same frequency, knowing which response corresponds to what pulse is impossible. As a result, using the response time to determine the distance to the target results in ambiguity under the assumption that the response could be coming from any of the transmitted pulses. This issue can be solved by resolving the measurement by varying the PRF in consecutive measurements [9, Chapter 15]. This can resolve the ambiguity by shifting the response time of all pulses but the one corresponding to the actual response, allowing us to single out the accurate distance. After that, by measuring the differences in frequency from the sent signal to the obtained, the target's velocity can be estimated by the size of the frequency shift. The shift is described as follows:

$$f_D = -2\frac{v_{\text{rad}}}{\lambda},$$

where v_{rad} is the radial velocity and λ is the wavelength of the radiowave. The Doppler frequency resolution depends on the number of pulses sent. By properties of the discrete Fourier transform, Doppler frequency, just as range, is measured with an ambiguity that needs to be resolved. The simulator emulates the need to resolve measurements by requiring multiple detections before a non-associated measurement can be sampled. Alternatively, a detection can be matched with an existing track without the need to resolve since it can be correlated with the track's position and velocity instead.

In order to fulfill this requirement, the azimuth angle increments in the search action are reduced to produce overlapping lobes. A parameter m specifies the number of lobes that should all cover a mutual azimuth angle, and the angle increment size is determined to fulfill that requirement (see details in Section 3.5.3). With this, obtaining up to m detections from the same target is possible, all while adjusting the PRF between each detection. A parameter n specifies how many detections of different PRFs are necessary out of the m overlapping pulses to consider a measurement resolved. If less than n matching detections are obtained, no unambiguous measurement is sampled and any unresolved detections are discarded.

3.2 Simulator Overview

The simulator is built to simulate a phased array radar (see e.g [9]) using an AESA antenna. The antenna is computer-controlled, allowing the radar to steer the direction of the radio waves with significant gain without moving the antenna itself, resulting in minimal delay. The radar system can be controlled and used in a simulated environment, providing targets that can be detected based on the radar equation 3.1.

3.3 Initializing the Search Area

All activity happens inside the defined search area, a circle sector of radius $r \in [r_{\min}, r_{\max}] = [20, 450]$ km and angle $\theta \in [-75, 75]^\circ$. Nothing is simulated outside of

this area. If a target moves outside the boundary, it is immediately terminated. A constant $n = 30$ specifies roughly the desired number of targets within the search area (once an equilibrium state has been reached). Initially, $\lfloor 0.6 \cdot n \rfloor$ targets are spawned on the border. The simulation then runs for 500 seconds (simulation time) before the radar is initialized in order for a steady state of evenly distributed targets to be present before tracking begins.

3.4 Targets

The state of a target is described by the position and velocity state vector $v_k = [x_k, \dot{x}_k, y_k, \dot{y}_k]^T$, turn rate $\dot{\theta}$, turn initiation rate expected value ρ and radar cross section RCS. After each time step k of size Δt_k, the state vector v_k is multiplied by the transition matrix X

$$v_{k+1} = X \cdot v_k \tag{3.3}$$

with

$$X = \begin{bmatrix} 1 & \frac{\sin(\dot{\theta}\Delta t)}{\dot{\theta}} & 0 & \frac{-(1-\cos(\dot{\theta}\Delta t))}{\dot{\theta}} \\ 0 & \cos(\dot{\theta}\Delta t) & 0 & -\sin(\dot{\theta}\Delta t) \\ 0 & \frac{1-\cos(\dot{\theta}\Delta t)}{\dot{\theta}} & 1 & \frac{\sin(\dot{\theta}\Delta t)}{\dot{\theta}} \\ 0 & \sin(\dot{\theta}\Delta t) & 0 & \cos(\dot{\theta}\Delta t) \end{bmatrix}. \tag{3.4}$$

In case $\dot{\theta} = 0$, instead use the matrix $\lim_{\dot{\theta}\to 0} X := \left(\lim_{\dot{\theta}\to 0} X(\dot{\theta})_{ij}\right)$.

At the time of spawning, each target is given a permanent turn rate expected value $\rho \sim U(10^{-4}, 10^{-3})$ turn initiations per second, and the time of the most recent turn initiation is recorded as t^{previous}, set to 0 initially. A turn is then initiated at time $t^{\text{initiate}} \sim t^{\text{previous}} + \text{Exp}(\rho)$ where $\text{Exp}(\rho)$ is the exponential distribution with expected value $1/\rho$. A turn duration is sampled at each turn initiation $t^{\text{duration}} \sim \text{Exp}\left(U\left(0.05, 0.1\right)\right)$, so the turn ends at $t^{\text{end}} = t^{\text{initiate}} + t^{\text{duration}}$. A new turn rate is sampled at each turn initiation $\dot{\theta} \sim U(0.001, 0.02)$ rad/s, and the target keeps a constant $\dot{\theta}$ turn rate during the entire maneuver in positive or negative angle direction with equal odds. After the turn is completed, a new initiation time is sampled, and the turn rate is set back to zero.

3.4.1 Spawning

Any new target receives the following permanent properties until it is terminated:

- Velocity $u \sim U(400, 800)$ m/s
- RCS $\sim \mathcal{N}(U(-10, 10), 2)$
- $\rho \sim U(10^{-4}, 10^{-3})$

where $\mathcal{N}(\mu, \sigma)$ is the normal distribution with mean μ and standard deviation σ. RCS is the radar cross section used in (3.1).

The following properties are sampled at the time of spawning but may change during the simulation:

- Position (x, y)
- Heading θ
- Turn rate $\dot{\theta} = 0$

The position is sampled uniformly on the border of the surveillance area, excluding the smaller circle arc, see Figure 3.1 where it is the red part of the border. The heading is sampled as the angle θ from the target position to a uniformly sampled position $\mathbf{z} = (x, y)$ within the surveillance area.

3.4.2 Spawn Rate

During the simulation, a spawn rate $\omega(t) < 1$ sets the expected frequency of new targets being spawned per second. Initially, the spawn rate is set to $\omega(0) = n\frac{600}{450000} = 0.04$ with $n = 30$. The idea is to spawn targets at roughly the replacement rate for n targets with an average lifetime at around $\frac{450000}{600} = 750$ s.

The spawn rate changes at an interval of 750 s with the intended effect of changing the spawn rate after most targets since the last spawn rate update has been terminated. The idea is not to let the target intensity reach an equilibrium for too long before the spawn rate is changed, which is then done according to

$$\omega(t) \sim \text{clip}\left(\mathcal{N}\left(\omega(0), \frac{\omega(0)}{10}\right), 0, 1\right), \tag{3.5}$$

where

$$\text{clip}(x, x_{\min}, x_{\max}) = \begin{cases} x_{\min} & \text{if } x \leq x_{\min} \\ x & \text{if } x_{\min} < x < x_{\max} \\ x_{\max} & \text{if } x > x_{\max} \end{cases}.$$

After each whole second of simulation time, a new target is spawned with probability $\omega(t)$.

3.5 Radar

In this section, we describe how the different parts of the radar are integrated and used in the simulation.

3.5.1 The Lobe

The lobe width in boresight (zero degrees azimuth) is given by $\Phi = 2°$, and the lobe size changes according to $\omega_\phi = \Phi/\cos(\phi)$ where ϕ is the lobe's angle to the orthogonal direction from the antenna surface (see Figure 3.1). This effect originates from the fact that the lobe width depends on the antenna width, and since phased array antennas do not rotate, the effective width when sending a pulse at an angle is smaller than when sending it straight forward. Detection probability decays at the edges of the lobe, see below.

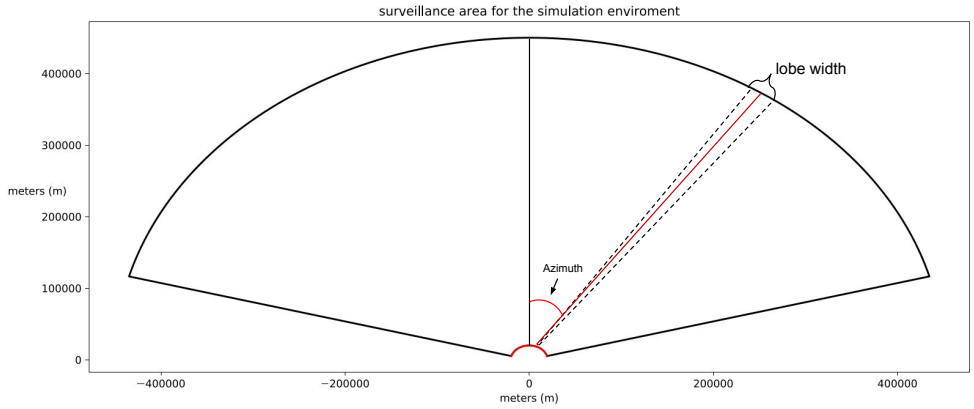


Figure 3.1: The shape of the surveillance area. The smaller circle arc colored red on is the only part of the border where new targets may not spawn. The angle between the black middle and illumination angle (red line) is the azimuth. The dotted line shows the lobe width when illuminating in the direction of the red line.

3.5.2 Detection

When the lobe overlaps with a target, a detection is sampled via a probability distribution. If the detection is successful, a unique target id is returned to the radar used for resolving a measurement. The probability of detection p_D is calculated based on the signal to noise ratio (SNR) using the following formula with ξ as the detection threshold and σ as the duty factor, where the duty factor is the ratio between the pulse width and the *Pulse Repetition Interval* (PRI, inverse PRF):

$$p_D = \begin{cases} (1-\sigma) \cdot C \cdot \exp\left(\frac{-\xi}{1+\text{SNR}}\right) & \text{for SNR} > 1 \\ 0 & \text{otherwise,} \end{cases} \tag{3.6}$$

where C depends on the angle difference between the center of the lobe and the target $\phi_{\text{tr}} = |\phi_t - \phi_r|$, as the detection probability diminishes at the edge of the lobe depending on the lobe decay parameter $\tau = 0.5$:

$$C = \begin{cases} 1 & \text{for } \tau \cdot \omega_\phi/2 - \phi_{\text{tr}} > 0 \\ 1 + \frac{(\tau \cdot \omega_\phi/2 - \phi_{\text{tr}})}{(\omega_\phi(1-\tau)/2)} & \text{otherwise} \end{cases} \tag{3.7}$$

The lobe decay τ specifies the proportion of the lobe angle in the center of the lobe, where $C = 1$. For the remaining part of the lobe width, C decays linearly to 0, see Figure 3.2

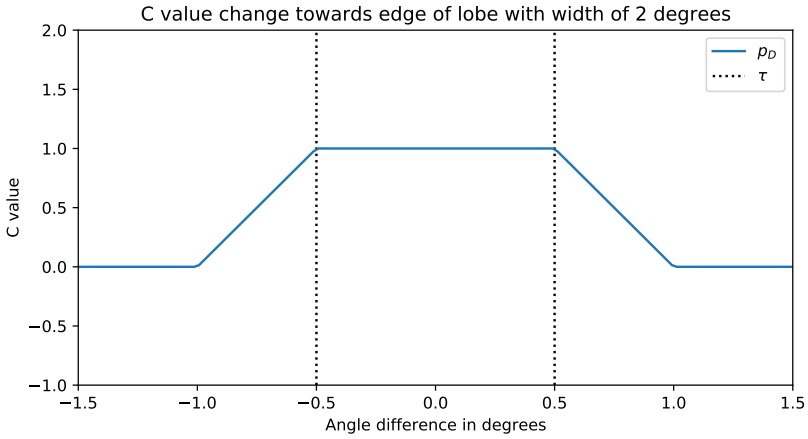


Figure 3.2: Visualization of how the C value function looks when illuminating in azimuth

The signal-to-noise ratio, which were derived from the radar equation (Equation 3.1) is convert into decibels, where decibel is given by $\text{value}_{\text{db}} = 10\log(\text{value})$, and we instead get:

$$\begin{aligned}\text{SNR}_{\text{dB}} = K_{\text{dB}} + 10 \cdot \log(\text{PRI} \cdot \rho) - 40 \cdot \log(r) - r \cdot \kappa \\ + \text{RCS}_{\text{dB}} + 20 \cdot \log(\cos(\phi_t)^\gamma) + 10 \cdot \log(B),\end{aligned} \tag{3.8}$$

where B is the blindness factor between 0 and 1 arising from the blind zone in radial velocity; targets that move too slowly can not be distinguished from the earth's surface and becomes harder to detect if moving slower than r_{v_2}. If the target moves slower than r_{v_1}, it is undetectable, so $B = 0$. Another contribution to the B factor (related to the mod operator) comes from the fact that the same antenna is used for sending and receiving, causing a risk of some pulses being received at the same time as new ones are sent, making them undetectable. The expression for the B factor is

$$B = \min\left(1, \max\left(\epsilon, \frac{\text{mod}\ (|r_v|, \frac{\lambda}{4 \cdot \text{PRI}_R}) - r_{v1}}{r_{v2} - r_{v1}}\right)\right) \tag{3.9}$$

$$K_{\text{dB}} = 225 - 10 \cdot \log(\text{PRI} \cdot 128). \tag{3.10}$$

Here, r_v is the target radial velocity, λ is the wavelength, PRI_R is the uniformly randomly sampled pulse repetition interval, PRI is the mean PRI, $\epsilon = 10^{-6}$, $r_{v1} = 15$ m/s is the minimum detectable radial velocity, $r_{v2} = 40$ m/s is the minimum radial velocity with full detectability ($B = 1$), ρ is the number of pulses, $\gamma = 1.5$ is the decay exponent and $\kappa = 4 \cdot 10^{-6}$ is the atmospheric dampening.

3.5.3 Control Inputs

The control inputs for the radar are to *search* or to *track* (often referred to as re-illuminate, not to be confused with tracks in the tracker).

If the search action is selected, the radar innitially sets the lobe angle φ to equal the minimum azimuth angle in the search area ($\varphi = -75°$). It keeps the radar angle used in the most recent search in an internal memory φ_{old} so that the next time the radar is commanded to search, it sets the radar angle to

$$\varphi = \varphi_{\text{old}} + \frac{\phi}{\cos(\varphi_{\text{old}}) \cdot m},$$

since $\frac{\phi}{\cos(\varphi)}$ is the effective lobe width, the increment in azimuth angle between illuminations is varied accordingly in order to produce m lobes which overlap in azimuth angle. If m is set to one, lobes barely touch but do not overlap with each other.

If the track action is selected, an existing track must be specified. The lobe angle is then set to equal the estimated azimuth angle towards the track. With every action, the waveform $w_i \in \{128, 256, 512\}$ must also be selected.

3.5.4 Measurements

After receiving n detections with the same target id in the last m searches, or after obtaining a detection through re-illumination, a measurement containing information about the target is sampled. From a target with coordinates (x, y) and velocity ($\dot{x}$, $\dot{y}$) (assuming the radar is positioned in the origin), we measure the radial position, the azimuth angle, and velocities in cartesian coordinates $(\hat{r}, \hat{\varphi}, \hat{\dot{x}}, \hat{\dot{y}})$ with noise according to

$$\hat{r} = \mathcal{N}\left(\sqrt{x^2 + y^2}, 100\right), \quad \hat{\varphi} = \mathcal{N}\left(\varphi, \frac{0.12}{\cos(\varphi)}\right)$$
$$\hat{\dot{x}} = \mathcal{N}(\dot{x}, 5), \quad \hat{\dot{y}} = \mathcal{N}(\dot{y}, 5).$$

Additionally, the RCS and SNR are measured.

$$\widehat{\text{RCS}}_{\text{dB}} = \mathcal{N}(\text{RCS}_{\text{dB}}, 2), \quad \widehat{\text{SNR}}_{\text{dB}} = \mathcal{N}(\text{SNR}_{\text{dB}}, 2),$$

The RCS_{dB} for a target is its average RCS_{dB} given at the spawning time. SNR_{dB} for the target is given by (3.2)

Notice that we allow measurement of the full 2-dimensional velocity, while a radar only measures radial velocity. We motivate this as follows. The target tracker is basic and inferior to a real tracker. Helping it with extra velocity information compensates for this to some extent.

3.5.5 False Alarms

In the real world where noise is unavoidable, there is a chance that random noise spikes cause a candidate track to be created despite not correlating to an actual target. In this project, noise is sampled for measurements after enough detections from a target. However, background noise from sources other than targets is not simulated. Instead, false measurements are sampled at random, with a probability

p_{false} of occurring after each action. When a false measurement occurs, the azimuth angle to the track is sampled uniformly around the lobe center, within the lobe width. We want the false measurement to be uniformly distributed within the area covered by the lobe, so we sample the radial distance r according to

$$r \sim \sqrt{U(r_{\text{min}}^2, r_{\text{max}}^2)},$$

since the area of a circle sector covered by the lobe increases in proportion to the radius. The heading angle θ is sampled

$$\theta \sim U(0, 2\pi).$$

The measured velocity, SNR and RCS are sampled with identical distributions as for newly spawned targets. The measurement is then sent to the tracker in the same way as true measurements.

3.6 Target Tracking

Whenever the radar receives a measurement, it is sent to the tracker. The tracker can do one of two things with the measurement; associate it with an existing track and update its state (prediction of target properties) or create a new track with its own prediction. Each track contains estimates of all measured properties of the target and their covariances. The tracker is also able to predict how the tracks change over time. Some of the properties held in a track can be visualized and matched with true targets, shown in Figure 3.3.

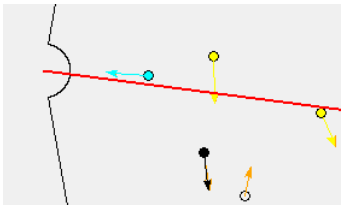


Figure 3.3: A snapshot of the tracker state with true targets drawn as well. Transparent circles represent the location of a true target. Filled circles represent location estimates in tracks with the fill colors yellow, blue, and black representing tracks of confidence levels candidate, tentative, and confirmed, respectively. Additionally, velocity vectors are drawn from the track locations, orange arrows for the true velocity vector, and filled arrows with the corresponding color of confidence level for the track estimates. The red line represents the radar lobe center.

A new track is always initialized with the confidence level "candidate". The confidence level is upgraded to tentative once two additional measurements have been associated with this particular track. Finally, it is upgraded to confirmed if another two measurements have been associated. The confidence level does not affect the track in any way; it is only used as information for the policy as well as the reward function in an MDP.

The tracking of individual targets is done with an EKF described in Section 2.3.1, where each track gets its own EKF and where the process and measurement noise is defined as

$$Q = {\sigma_x}^2 \begin{bmatrix} t^3/4 & t^2/2 & 0 & 0 \\ t^2/2 & t & 0 & 0 \\ 0 & 0 & t^3/4 & t^2/2 \\ 0 & 0 & t^2/2 & t \end{bmatrix} \qquad R = \begin{bmatrix} \sigma_r & 0 & 0 & 0 \\ 0 & \sigma_{\sin\varphi} & 0 & 0 \\ 0 & 0 & \sigma_{\dot{x}} & 0 \\ 0 & 0 & 0 & \sigma_{\dot{x}} \end{bmatrix},$$

with

Parameters	Value
σ_r	100 m
$\sigma_{\dot{x}}$	5 m/s
$\sigma_{\sin\varphi}$	0.12
σ_x	50 m

The jacobian of the measurement model according to the state used for the EKF is:

$$J_h\left((x, y, \dot{x}, \dot{y})\right) = \begin{bmatrix} \frac{x}{r} & 0 & \frac{y}{r} & 0 \\ \frac{-y \cdot x}{r^3} & 0 & \frac{1}{r} - \frac{y^2}{r^3} & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1. \end{bmatrix}$$

3.6.1 Association

The association of measurements to tracks is done in two different ways. The Hungarian algorithm is only used for search mode. For tracking/reillumination mode, we instead use Association boxes. The association boxes are generated around the targets with dimensions three times the standard deviations in the diagonal covariance matrix in radial positioning, radial velocity, and radar angle. If a target is found inside the detection box (detection within all three deviations), that yields a detection. Re-illuminations do not create new instances of tracks, only updating existing ones.

3.6.2 Termination

A track can be terminated in two different ways. Firstly, a track can be removed if the radar misses it a given amount of times based on its confidence level. For candidates, a maximum of 9 subsequent misses since the last detection are allowed. The 10th miss terminates the track. For tentative and confirmed, the numbers are 12 and 15, respectively. A miss is counted for each action taken where the radar lobe is within one lobe width of the target but where it does not grant a detection. Secondly, a track is terminated if too much time has been spent since the last detection. We call this time *coast time*, and the maximum allowed is 30 seconds.

3.7 Simulation Loop

The simulation uses an adaptive time step size based on the integration time of the radar pulse. This allows the simulation to run efficiently with the lowest resolution possible, with hardly any visible artifacts from the radar's perspective. Changes in target trajectories are updated at a much lower frequency than what the radar operates, so any difference in target behavior due to varying time step sizes is minuscule. However, small differences in simulation trajectories build up over time, creating large differences in the long run. Thankfully, since targets are eliminated once they leave the search area, individual target trajectories are replaced by new ones before large differences are given a chance to arise.

Algorithm 1 below describes the steps in which order they are taken in the simulation.

Algorithm 1 Simulation Loop

1. Initialize search area and spawn $\lfloor 0.6n \rfloor$ targets
2. Simulate for 500 seconds while integrating movement and spawning new targets
3. Initialize radar

while $t <$time horizon **do**

a. Select action (sample policy)

b. Integrate target movement and spawn new targets, $t \leftarrow t + \Delta t$

c. Sample detections and save in memory

if n detections from the same target in the last m searches **then**

Sample measurement and associate with existing tracks using the Hungarian algorithm or detection boxes. If no association, create new candidate.

end if

e. Predict track movement.

f. Calculate reward

if time since spawn rate update $>$ 750 seconds **then**

Update Spawn Rate

end if

end while

Calculate average reward over simulation time to obtain a score

3.8 Baseline Policy

Results are compared against a baseline policy, which is a hard-coded heuristic to take logical and intuitive actions providing descent tracking performance. At each time step, the algorithm puts all existing tracks with coast time larger than 15 seconds in a priority list sorted by descending coast times. The list is iterated from the top to check for tracks fulfilling:

- The track is not the most recently reilluminated track
- The angle from the radar to track is not the same as the previous illumination angle

- The estimated radial velocity is considered outside of the zero-doppler blind-zone

If all of the above conditions are true, the track is selected for reillumination and iteration through the priority list is stopped. If no track is selected, a search is performed instead.

3.9 Selecting Waveform

Selecting the waveform is essential to successful tracking. However, to simplify the problem initially, it is not considered part of the decision making but analyzed separately in Section 4.5. Instead, an algorithm selects the waveform "automatically" instead of being considered a control input. Instead, it can be considered part of the state transition probabilities while also affecting the size of the time step. The algorithm below describes how the waveform is selected automatically, and is used for most of the project, except explicitly stated otherwise. The process differs whether the selected action is search or track.

3.9.1 Search

When the search action is performed, the number of new tracks that were found is stored in a history. A history of the last 30 seconds of search actions is kept and the average number of tracks found α is calculated over the history. This value is used to determine the number of pulses j to use for the next search according to

$$j = \begin{cases} 512 \text{ if } \alpha < 1/30 \\ 256 \text{ if } 1/30 < \alpha < 2/30 \\ 512 \text{ if } 2/30 < \alpha \end{cases} .$$

The idea behind this method is to maximize the rate of finding new tracks regardless of their detectability. If previous searches provided many new tracks, a higher integration time is considered unnecessary. On the other hand, if previous searches did not yield many new tracks, it is assumed that the integration time is too low to find any remaining targets with low detectability.

3.9.2 Track

When reilluminating an already existing track, it is easier to argue for a specific waveform choice to be optimal. The goal here is to minimize the expected integration time necessary to get a detection. Since the SNR directly influences the detection probability (ignoring Doppler and PRI blindness), we use the previously measured SNR to determine the waveform choice of the following illumination. A collection of properties that primarily affect the SNR of detections are

- Radar cross section
- Distance to target
- Azimuth Angle (affects power gain / intensity of the lobe)
- Waveform

All of the above except the selected waveform is assumed to be near constant during the time span from one detection to the next. We then assume that a subsequent illumination yields a similar SNR if the same waveform is used. Additionally, we can use the fact that doubling the integration time leads to a 3 dB increase in SNRdB (doubling the linear SNR value). With this information, we normalized the estimated SNR to be the equivalent of using a waveform of 128 pulses in decibels and denote it as $\mathrm{SNR}_{\mathrm{dB}}^{128}$. Based on the estimated signal to noise ratio $\mathrm{SNR}_{\mathrm{dB}}$ and number of pulses used j for that detection, we calculate the 128 pulse-normalized SNR

$$\mathrm{SNR}_{\mathrm{dB}}^{128} = \mathrm{SNR}_{\mathrm{dB}} - 3\log_2\left(\frac{j}{128}\right) \tag{3.11}$$

With the estimated $\mathrm{SNR}_{\mathrm{dB}}^{128}$, we can calculate which waveform should give us the lowest expected integration time necessary for a detection. Choosing the waveform which minimizes the necessary integration time should be optimal most of the time, at least from a local optimization standpoint. We look at (3.6) and assume $C = 1$ to give us a detection probability. Given a detection probability

$$\mathrm{Prob}\left(\mathrm{SNR} = \mathrm{SNR}_{\mathrm{dB}}^{128} + 3 \cdot \log_2\left(\frac{j}{128}\right)\right), \tag{3.12}$$

for $j \in \{128, 256, 512\}$ the estimated number of attempts needed for one detection is $1/p$. Multiplying this by the integration time $I(j)$ gives us the expected integration time necessary for a detection

$$I_{\mathrm{detect}}(j) = I(j)/p(j) \tag{3.13}$$

so we can select the optimal number of pulses $\arg\min_j I_{\mathrm{detect}}(j)$.

4
Methods

This chapter describes all the necessary details regarding the training and performance evaluation of different approaches. The overall flow of the training algorithm is illustrated in Figure 4.1 below. Further details regarding each step are described throughout this chapter.

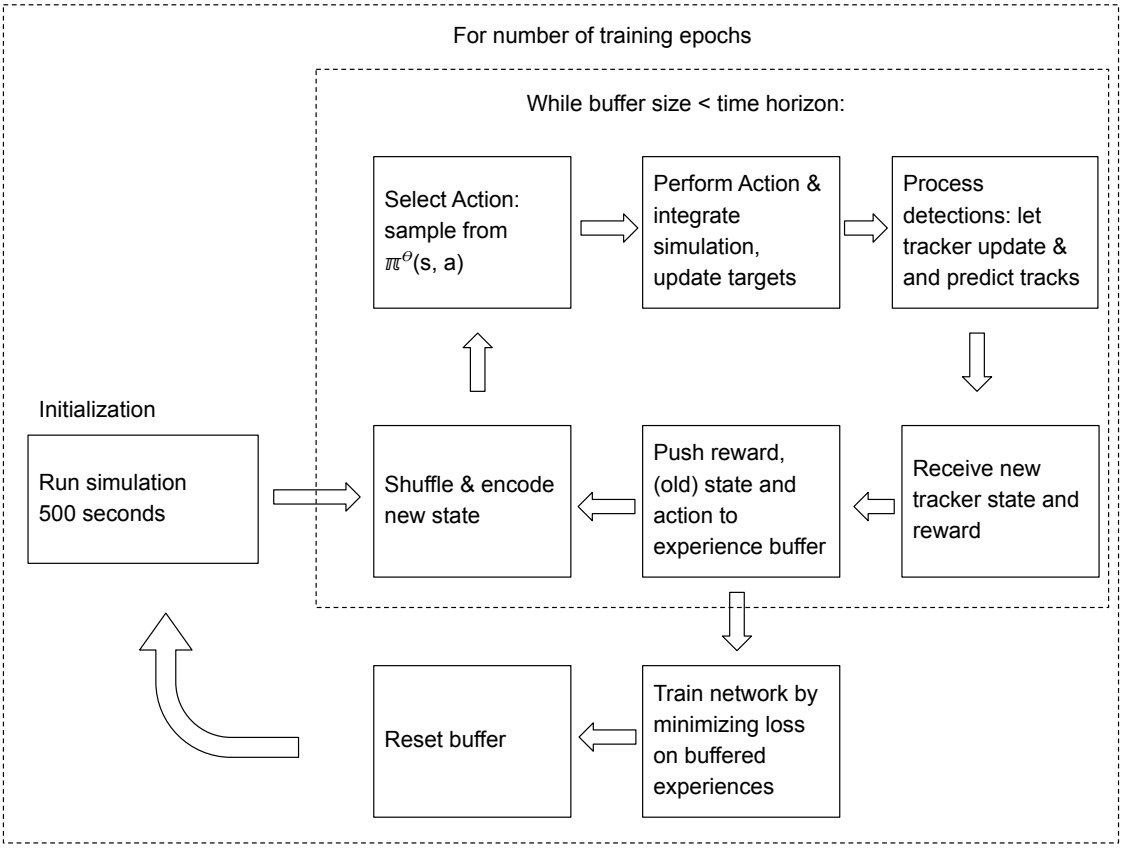


Figure 4.1: Diagram describing the flow of the training loop used for this project.

4.1 MDP

Section 2.1 described the MDP framework where a policy π can be trained in a Markovian environment in order to maximize the expected return. Figure 4.1 describes the training procedure used in this thesis, however it may not be immediately clear how the MDP framework is used here. We look closer at the simulation loop

regarding how the policy interacts with the environment in Figure 4.2. Here, the state s_{t+1} (representing the true target distribution) is sampled without dependence on the policy from a probability distribution $P(s_t)$. The selected action a_{t+1} as well as s_{t+1} influences the obtained observation O_{t+1}. Using all previous actions and observations $a_{1:t+1}$ and $O_{1:t+1}$, we can predict the environment state with a belief state B_{t+1}. Here, the belief state is a distribution over S describing the probability of being in the true state s. However, for predicting the belief state, we use the tracker (Section 2.3) built into the simulator denoted TT which uses the implemented tracking methods to output a good belief state $\tilde{B}_{t+1}$ which corresponds to a single likely state s.

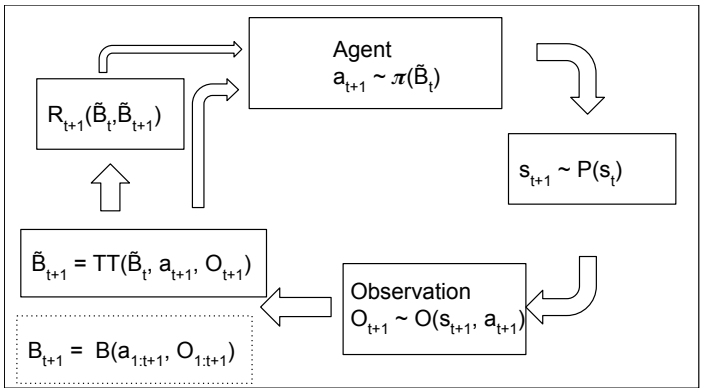


Figure 4.2: The simulation loop describing the policy's interaction with the environment.

From the perspective of the agent, the belief state (tracker state) $\tilde{B}_t$ is the only relevant input to the policy, so the sampling of observations and true state transitions can be considered part of the transition probability when "sampling" the belief state $\tilde{B}$ after using an action. In doing this, we obtain something like in Figure 4.3 where we consider the belief state to be sampled from a probability distribution $\tilde{P}$. We can compare this to the typical MDP in Figure 2.1 and see that these are clearly equivalent, so we can describe the environment transitions using the standard MDP framework.

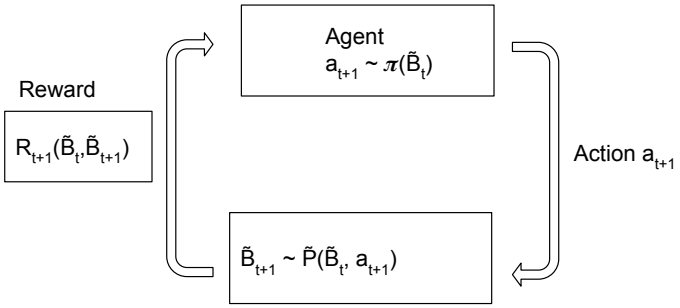


Figure 4.3: State transitions and observations and belief state updates are all considered in the belief state transition probabilities $\tilde{P}$.

4.2 Neural Network

The implementation is based on the PPO-clip training procedure with an actor-critic architecture. We use two completely separate neural networks for the actor and critic, respectively. The network architectures are the same between the actor and critic except for the number of output nodes. They are separated into a *feature stack* and a *track stack* of neuron layers. The feature stack is initially applied to all features of each track individually, in parallel, giving a set of outputs each for every track and works like an encoder for the track features, decreasing the dimensionality of the input before it is passed on to the fully connected track stack. The outputs from the feature stacks are concatenated before they are used as the input to the track stack. Since we want to use a finite number of neurons, we limit the total number of allowed tracks in the tracker. All the features used and how they are represented can be found in Section 4.3

For $n = 31$ features per track, $m = 50$ maximum number of allowed tracks, o outputs (equal to the number of actions (51) for the actor, single output for the critic), we describe the network architecture using Linear(i, j) layers with i inputs and j outputs fully connected. The ReLU layer is the *rectified linear unit*, a nonlinear activation function. For more details regarding the neuron layers, see the official Pytorch documentation [12].

Feature stack:
- Linear(n, 64)
- ReLU
- Linear(64, 64)
- ReLU
- Linear(64, 3)

Track stack:
- Linear($3 \cdot m$, 1024)
- ReLU
- Linear(1024, 1024)
- ReLU
- Linear(1024, 512)
- ReLU
- Linear(512, o)

Figure 4.4 illustrates the idea behind the network architecture. The important detail regarding the feature stack is that the weights (labeled w_{ij} in Figure 4.4) are shared across all tracks, since there is only a single feature stack which is applied to all tracks in sequence.

4.3 State Encoding

The agent receives information about the environment from the tracker directly, so in the MDP formulation, the tracker is part of the environment and its transition

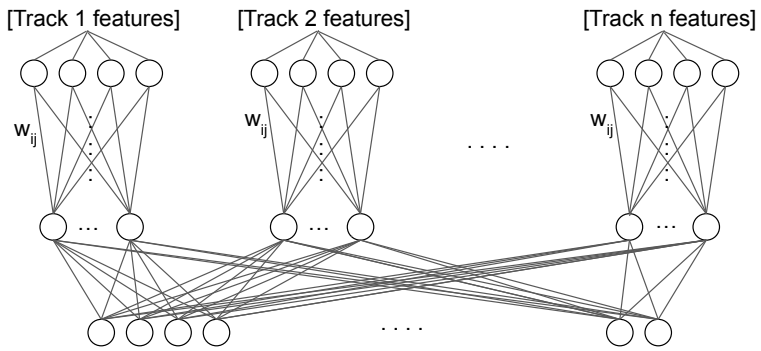


Figure 4.4: Illustration of the idea behind the network architecture. Weights labeled w_{ij} are shared across nodes connected to different track features.

probabilities. Each track contains an array of information about the corresponding target, encoded into primarily binary encoding with $+1$ or -1 to make the information easy for the agent to interpret. All features are listed below, where some features are encoded using multiple bits (one bit for each neuron), meaning most features are represented by multiple neurons. In total, 31 neurons/bits encode all features for a single track. The first feature *Track existence* is set to 1 if the feature vector corresponds to a track. Otherwise, it is set to -1. When the feature vector does not correspond to a track (has not yet been found), the input is zero-padded (all remaining features set to 0). We list the features of each track as well as details regarding the number of bits and range used for each feature in Table 4.1. Suppose a feature's value is outside the encoded range. In that case, it is clipped to the maximum or minimum value in the range, whichever is closest. For the features where the provided number of bits is insufficient to represent all whole numbers in the range or if the values are continuous, the range is rescaled and rounded to fit the range of the bit representation. Where each bit represent an interval of integer values.

Feature	No. Bits	Range
Track existence	1	$[-1, 1]$
Certainty (candidate, tentative, confirmed)	3	One hot
Coast time (time since last measurement)	5	$[0, 30]$ [s]
Number of subsequent misses	4	$[0, 15]$
Estimated SNR_{dB}	4	$[-3, 12]$ (dB)
Angle to track	3	$[-75, 75]$ (deg)
Radial distance	4	$[0, 450000]$ [m]
Zero doppler detectability	2	[None, Reduced, Full]
Detections since last upgrade	2	$[0, 3]$
Search angle	3	$[-75, 75]$ (deg)

Table 4.1: Encoding of input features

The number of outputs corresponds to the number of available actions for the actor, in this case, 51, one action for the reillumination of each track and a single output

for searching. Ouptuts corresponding to tracks which do not exist are set to $-\infty$, afterwards a *softmax* layer normalizes all outputs with total sum equal to one, so that they can be interpreted as probabilities. The value network has a single output corresponding to the value estimate.

4.3.1 Input Shuffle

In order to prevent the agent from learning (nonexistent) patterns in track permutation, the ordering of the input order is shuffled randomly between the feature stack and the track stack. Effectively this means that the concatenation of outputs from the feature stack is done in random order. We then make sure to match the output with the ordering of the input.

4.4 Implementation Specific Details

In this section, we go over some details regarding the implementation which are not standard in the PPO-clip algorithm and are specific to this project or should be clarified in more detail. This includes details regarding the specific scoring and reward function used, managing varying time step sizes and corresponding γ discount, and other optimizations to enhance the training procedure.

4.4.1 Varying Time Step Size

The environment is sampled with multiple independent trajectories τ_i simultaneously before they are used for training. All trajectories are stored in a single buffer. We now look at the PPO training algorithm for a single trajectory τ_i, and call it τ. The trajectory is divided into a partition of time intervals with $t_1, ..., t_n$ for n fixed, denoting the start of each interval. $\Delta t_k = t_{k+1} - t_k$ is not constant and may vary between different k. An action has a specific integration time $w \in [w_1, w_2, w_3]$ based on the selected waveform which determines the size of the current time step and gives us $\Delta t \in [\Delta t_1, \Delta t_2, \Delta t_3]$ where $\Delta t_1 < \Delta t_2 < \Delta t_3$. The trajectory consists of n actions, each having a corresponding waveform that determines the size of the time step. The total simulation time, therefore, depends on the actions chosen.

4.4.2 Performance Score and Reward

Since we have varying time steps we have to pay some extra care to the reward and discount. We do this by introducing a performance score and use it as a basis for the reward. Discount needs to be applied with care to avoid undesirable biases in the step length choice.

4.4.2.1 Performance Score

The performance score Ψ_k over time step k is calculated as

$$\Psi_k = \int_{t_k}^{t_k+\Delta t_k} \dot{\Psi}(t)\mathrm{dt} \tag{4.1}$$

where $\dot{\Psi}(t)$ is the amount of reward given per second based on the number of tracks in the tracker and their confidence level. It is determined according by $\dot{\Psi}(t) = \sum_i C_i(t)$ for all tracks i where

$$C_i(t) = \begin{cases} 0 & \text{if track } i \text{ does not exist at time } t \\ 10/3 & \text{if track } i \text{ has confidence level candidate at time } t \\ 20/3 & \text{if track } i \text{ has confidence level tentative at time } t \\ 30/3 & \text{if track } i \text{ has confidence level confirmed at time } t. \end{cases} \tag{4.2}$$

Since tracks can terminate during the time Δt_k it is important to not only consider the tracker state at the beginning of the time step, since that might cause unfairness in the choice of integration time. $\dot{\Psi}(t)$ is therefore considered a continuous variable, and is unlike Ψ_k not limited by the resolution of the simulation time steps. The total score over the trajectory τ is

$$\Psi_\tau = \sum_{k \in \tau} \Psi_k. \tag{4.3}$$

We see that the score received over a time step k is proportional to the number of tracks (and their confidence level) held during the time step and the duration of the time step. In order to obtain a large reward, the agent should confirm tracks quickly and keep the confirmed tracks for the rest of the trajectory.

4.4.2.2 Reward and Discount

The non-discounted immediate reward rate of change $\dot{r}_k$ at time step k is

$$\dot{r}_k = \frac{\Psi_k}{\Delta t_k} - \frac{\Psi_{k-1}}{\Delta t_{k-1}} := \bar{\dot{\Psi}}_k - \bar{\dot{\Psi}}_{k-1}, \tag{4.4}$$

which can be interpreted as the increase in the average score rate of change from the previous time step to the next. Taking the average score rate of change $\frac{\Psi_k}{\Delta t_k}$ is done in order to get rid of the time dependence in $\dot{r}$ so that it can be identified by a single index k.

As for the total reward, one could integrate over $\dot{r}_k$ over the trajectory, but we are more interested in the discounted reward (starting from a specific time step k). The discounted reward $\hat{r}_k$ is

$$\hat{r}_k = \sum_{j=k}^{\infty} \dot{r}_j \cdot \int_{t_j}^{t_j + \Delta t_j} e^{-(t-t_k)\gamma} \mathrm{dt}, \tag{4.5}$$

with discount factor $\gamma > 0$.

4.4.3 Discounted Reward Fast Algorithm

For the implementation of (4.5), the entire integral over $[t_k, \infty]$ is not calculated individually for each time step k. The reason is that although the integral over exponential decay can be precalculated for three different integration times, each time

step would require calculating a sum over all future time steps for the rest of the simulation. Doing this would require an algorithm of complexity $O(n^2)$ for n time steps, but it can be improved to linear time using Algorithm 2 below. Note also that the discounted reward depends on the reward over an infinite time horizon. It is not possible to sample an infinite amount of time steps, and therefore we assume that any reward past the time horizon $\dot{r}_{k>n}$ stays equal to zero for the rest of time so that $\int_{t_k}^{\infty} \dot{r}_{k>n} \cdot e^{-(t-t_k)\gamma} \mathrm{dt} = 0$.

Algorithm 2 Discounted Reward

$\hat{r}_n = \dot{r}_n \cdot \int_{t_n}^{t_n+\Delta t_n} e^{-(t-t_n)\gamma} \mathrm{dt}$
for $k = n-1, n-2, ..., 2, 1$ **do**
    $\hat{r}_k = \dot{r}_k \int_{t_k}^{t_k+\Delta t_k} e^{-(t-t_k)\gamma} \mathrm{dt} + \hat{r}_{k+1} e^{-(t_{k+1}-t_k)\gamma}$.
end for

This algorithm yields the same result as (4.5) for all r_k since

$$\begin{aligned}\hat{r}_k &= \sum_{j=k}^{\infty} \dot{r}_j \int_{t_j}^{t_j+\Delta t_j} e^{-(t-t_k)\gamma} \mathrm{dt} = \\ &= \dot{r}_k \int_{t_k}^{t_k+\Delta t_k} e^{-(t-t_k)\gamma} \mathrm{dt} + \sum_{j=k+1}^{\infty} \dot{r}_j \int_{t_j}^{t_j+\Delta t_j} e^{-(t-t_k)\gamma} \mathrm{dt}.\end{aligned}$$

Looking closer at the last summation, we see that

$$\sum_{j=k+1}^{\infty} \dot{r}_j \int_{t_j}^{t_j+\Delta t_j} e^{-(t-t_k)\gamma} \mathrm{dt} = \sum_{j=k+1}^{\infty} \dot{r}_j \int_{t_j}^{t_j+\Delta t_j} e^{-(t-t_{k+1})\gamma} e^{-(t_{k+1}-t_k)\gamma} \mathrm{dt}.$$

However, the factor $e^{-(t_{k+1}-t_k)\gamma}$ does not depend on t and can therefore be moved outside the integral. It also does not depend on j, so we can move it outside the sum. Doing this gives

$$\begin{aligned}\sum_{j=k+1}^{\infty} \dot{r}_j \int_{t_j}^{t_j+\Delta t_j} e^{-(t-t_k)\gamma} \mathrm{dt} &= e^{-(t_{k+1}-t_k)\gamma} \sum_{j=k+1}^{\infty} \dot{r}_j \int_{t_j}^{t_j+\Delta t_j} e^{-(t-t_{k+1})\gamma} \\ &= e^{-(t_{k+1}-t_k)\gamma} \hat{r}_{k+1},\end{aligned}$$

so we get

$$\hat{r}_k = \dot{r}_k \int_{t_k}^{t_k+\Delta t_k} e^{-(t-t_k)\gamma} \mathrm{dt} + e^{-(t_{k+1}-t_k)\gamma} \hat{r}_{k+1}.$$

Note that $(t_{k+1} - t_k) = \Delta t_k$, so the algorithm can be performed quickly by precalculating the exponential decay for all Δt_k.

The reward discount algorithm was verified not to contain any bias by running a training session where a constant reward per second was given regardless of action. Together with a small waveform entropy loss, the waveform selection remained evenly distributed, showing that the discounted reward is not biased towards any action length using this reward discount.

4.4.4 Advantage Estimation

The advantage is $A(s_k, a_k) = Q(s_k, a_k) - V(s_k)$ where $Q(s_k, a_k)$ is the expected discounted reward $\mathbb{E}(\hat{R}_k)$ for taking action a_k in state s_k and $V(s_k)$ is the expected discounted reward based on the current policy $\pi^\theta(s, a)$. In PPO, we want to use the advantage to train the agent. We use a neural network to predict $V(s_k)$ by training on discounted reward, but predicting $Q(s_k, a_k)$ is not so straight forward. One way to estimate the advantage at time step k is

$$A_k = \hat{r}_k - V(s_k), \tag{4.6}$$

$\hat{r}_k$ is the discounted reward over the rest of the trajectory, which is not a great approximation of $Q(s_k, a_k)$. This advantage estimator may work well if a deterministic policy is used and if the environment is deterministic in its transition probabilities as well as reward. In addition to randomness in the policy π^θ, the discounted reward $\hat{r}_k$ does not consider the variation in state transition probabilities but instead samples a particular outcome.

In our implementation using a radar simulation, neither the policy nor the environment (belief state) is deterministic, and estimating the advantage this way is expected to suffer from significant variance due to the randomness in reward. Alternatively, the advantage can be estimated with less variance using

$$A_k = \dot{r}_k \int_{t_k}^{t_k+\Delta t_k} e^{-(t-t_k)\gamma} \mathrm{dt} + V(s_{k+1}) e^{-\Delta t_k \gamma} - V(s_k). \tag{4.7}$$

This way, the value estimator estimates the future reward, which considers the environment transition probabilities. One downside of this advantage estimate regards delayed rewards since such rewards would only be considered if they strongly correlate to a specific state transition. The discounted future reward term $V(s_{k+1})$ depends only on the state itself and previously obtained rewards from similar states according to the policy and state transition probabilities. If a, for the current policy, atypical action was taken which resulted in a large but delayed reward, it would not be reflected whatsoever in this advantage calculation since it relies on immediate rewards and accurate value estimations to function. But the value estimator will always lag behind the policy since it requires training from several trajectories before it can accurately predict the value.

Another option is to use the *generalized advantage estimate*. What we present here is based on the work of Schulman et al [14] but is modified using the time continuous discount factor

$$\hat{A}_k = \delta_k + e^{-(t_{k+1}-t_k)\gamma} \lambda \delta_{k+1} + ... + e^{-(t_n-t_k)\gamma} \lambda^{n-k} \delta_n,$$

where $\lambda \in [0, 1]$ and

$$\delta_k = \int_{t_k}^{t_k+\Delta t_k} \dot{r}_k e^{-(t-t_k)\gamma} \mathrm{dt} + V(s_{k+1}) e^{-\Delta t_k \gamma} - V(s_k).$$

δ_k is essentially the low variance advantage estimate in (4.7). Since we modified the advantage expression to utilize reward discount with varying time step sizes, the expression looks much more complicated. But functionally, it works the same as the original expression in Schulman et al [14], so we suggest taking a look at their original expression to understand the idea behind the generalized advantage estimate. By recursion, we calculate advantages in linear time using

$$\hat{A}_k = \delta_k + e^{-(\delta t_k)\gamma}\lambda\hat{A}_{k+1}. \tag{4.8}$$

The parameter λ serves as a tunable trade-off between near-future rewards from the present trajectory as in (4.6) versus the more general, low variance value estimate as in (4.7).

4.4.5 Training Algorithm

In PPO, we use the advantage estimate to calculate the loss and update the network weights to a policy $\pi^{\theta_{l+1}}$ with network parameters θ_{l+1} that maximizes the expected discounted reward. After sampling a trajectory τ and calculating discounted rewards $\hat{r}_k$, value estimates V_k and advantage estimates A_k for $k \in \tau$, we update the network weights according to

$$\theta_{l+1} = \text{argmax}_\theta \frac{1}{n}\sum_{k=1}^{n} \min\left(\frac{\pi^\theta(a_k|s_k)}{\pi^{\theta_l}(a_k|s_k)}A(s_k,a_k), g(\epsilon, A(s_k,a_k))\right), \tag{4.9}$$

where $g(\epsilon, f)$ clips f so that

$$g(\epsilon, f) = \begin{cases} f & \text{if } 1-\epsilon < f < 1+\epsilon \\ 1-\epsilon & \text{if } f < 1-\epsilon \\ 1+\epsilon & \text{if } 1+\epsilon < f. \end{cases}$$

Here, the clipping parameter ϵ is set to equal 0.2. $\pi^{\theta_l}(a_k|s_k)$ is the probability of choosing action a_k given state s_k for the policy with weights θ_l. This training algorithm strives to increase the probability of selecting actions with a large advantage and decrease the probability of actions with a negative advantage within the clipping ratio ϵ.

The value estimator network is trained by minimizing the mean squared error between the value estimate and discounted reward according to

$$\phi_{l+1} = \text{argmin}_\phi \frac{1}{n}\sum_{k=1}^{n}(V_{\phi_l}(s_k) - \hat{R}_k)^2.$$

Here $V_{\phi_l}(s_k)$ is the value estimate of state s_k for the value network with weights ϕ_l, and $\hat{R_K}$ is the discounted reward.

4.4.6 Entropy Loss

In order to prolong exploration, entropy loss is used in the training loop and subtracted from the loss function. The general expression of entropy loss for a policy with outputs x_i where $i = 1, ..., n$ is

$$H(x) = -\sum_{i=1}^{n} P(x_i) \log_e(P(x_i)). \tag{4.10}$$

The entropy $H(x)$ for a single output x_1 ($n = 1$) is shown in Figure 4.5. We see that the entropy contribution from a single output is low (near 0) if its probability is near 0 or 1, high if the probability is near half. What is more important, though, is that the derivative of the expression within the sum is

$$\frac{\mathrm{d}}{\mathrm{d}x} P(x) \log_e(P(x)) = -\log_e(x) - 1, \tag{4.11}$$

so that when an output's probability approaches 0, the derivative of the entropy contribution from that output approaches positive infinity. Since the agent learns based on the gradient of the loss function, including the entropy in the loss function incentivizes the agent to refrain from completely ignoring specific outputs without randomly forcing it to explore as opposed to the ϵ-greedy strategy typically used in Q-learning.

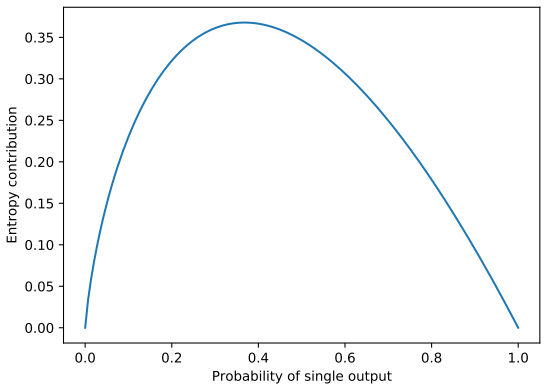


Figure 4.5: Entropy contribution as a function of the output probability of a single output

We have also used a waveform-specific entropy loss that allows the agent to select a waveform. For the waveform-specific entropy loss, all actions with the same waveform are merged into the same category by summing their probabilities. The three resulting categories are then re-normalized, and the entropy loss is calculated.

Unfortunately, in neither case did the entropy loss improve performance, so it was removed in the final implementation. However, it could still be helpful in junction with other hyperparameter settings or network architectures. And as mentioned, this entropy loss was used to verify that the reward discount implementation did not induce biases in waveform selection.

4.5 Selecting Waveform

An initial goal of the thesis project was to allow the agent to select the waveform; the number of pulses to use for each illumination. This was done by multiplying the output size by 3 (one for each available waveform) but keeping everything else otherwise the same. Thus, the neural network is given 153 output neurons where the first 150 outputs correspond to the reillumination of a track with one out of three possible waveforms. The last three outputs correspond to searching with one of three waveforms.

The best performing network architecture that was found did not allow for effective waveform selection despite managing to perform well in other regards. Ultimately, the best performing agent uses the feature stack network architecture described in Section 4.2 and is not responsible for waveform selection. Instead, we use a coded algorithm to optimize the waveform selection locally after the illumination direction has been selected by the agent. If waveform selection is enabled by increasing the output size on this architecture, then the agent converges to using only a single waveform, despite the use of waveform specific entropy, while still performing reasonably well in other regards.

The fully connected network architecture was also implemented to bypass the feature stack. It allows the agent to learn to select waveform in a more convincing manner, although performing much worse in other regards. This prompts interest in further investigation in the future, as there is good reason to believe that other network architectures may allow both, but for now we decided to continue optimizing the training algorithm without waveform selection being decided by the agent. The impact of the different architectures is discussed in Section 4.8.

Regardless, in this section we go over what we did find in regards to training an agent with waveform selection, before putting the matter aside for the rest of the project. However, first, in order to gain an intuition of the kind of situations where a particular waveform is desirable (although locally optimized), let us think about the integration time needed with a specific waveform on a target in order to receive a detection. As discussed in Section 3.9.2, we can calculate the expected integration time necessary for a detection using Equation (3.13) with the integration times implemented in the simulation $I(j) = \{0.13, 0.22, 0.38\}$. See Figure 4.6 where the expected integration times are plotted for different waveforms against a normalized $\mathrm{SNR}_{\mathrm{dB}}^{128}$ (as described in Equation (3.11)) on the x-axis. Here we see that it is optimal to use a waveform of 512 pulses against targets where $\mathrm{SNR}_{\mathrm{dB}}^{128} < 1.6$, 256 pulses where $1.6 < \mathrm{SNR}_{\mathrm{dB}}^{128} < 5.7$, and 128 pulses where $5.7 < \mathrm{SNR}_{\mathrm{dB}}^{128}$ as it results in the lowest expected integration time necessary for a detection.

4.5.1 Feature Stack Architecture

Despite performing well otherwise, the agent does not efficiently select waveform using the feature stack architecture seen in Figure 4.4. Many variations of the ar-

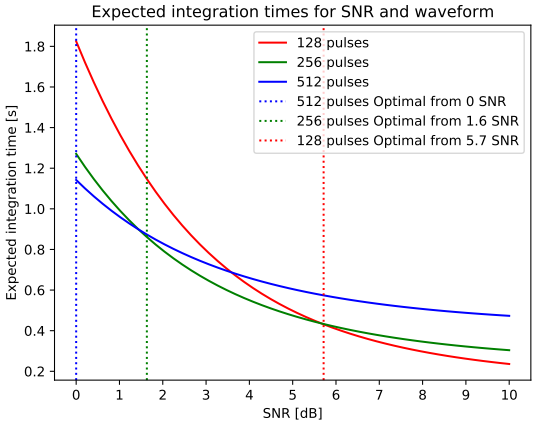


Figure 4.6: Expected integration time necessary until detection in terms of SNR. Since SNR depends on the waveform selected, the SNR on the x-axis shows normalized SNR equivalent to using 128 pulses. 256 and 512 pulses are assumed to increase the SNR by 3 and 6, respectively.

chitecture was tested, such as different numbers of bottleneck nodes (feature stack outputs) as well as the general depth and size of the network, however many of these variations caused the agent to perform considerably worse in general while not helping in the slightest when it comes to waveform selection. Only a weak waveform dependence on measured SNR can be seen with this architecture. See Figure 4.7 where the proportion of selected waveforms are shown out of all reillumination actions taken. The estimated SNR is adjusted to be equivalent to 128 pulses of integration.

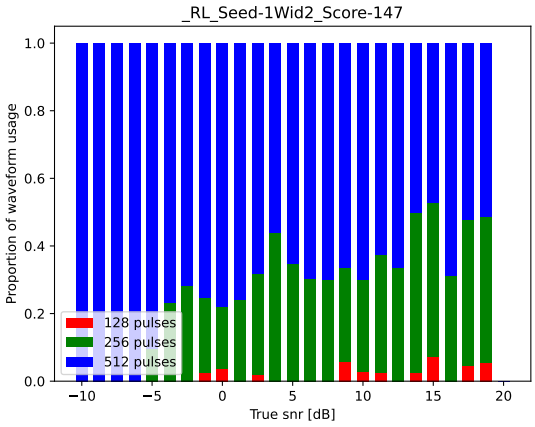


Figure 4.7: Number of pulses used when tracking targets with SNR shown on the x-axis. The SNR values are estimated SNR for 128 pulses of integration.

4.5.2 Fully Connected Architecture

We also evaluate an alternate network architecture that proved to perform better at selecting waveforms. The *fully connected* network architecture works by effectively

skipping the feature stacks and instead connecting the input features of each track directly to the fully connected network. See the illustration in Figure 4.8 below.

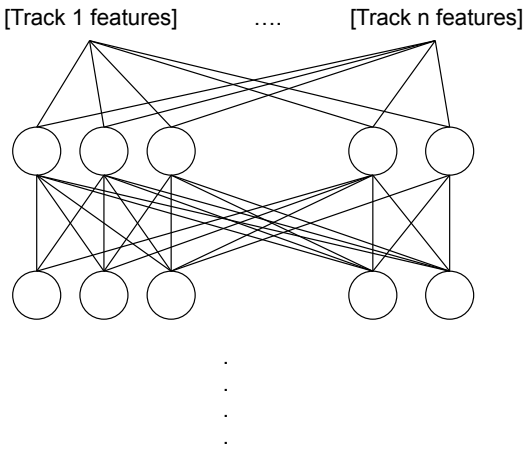


Figure 4.8: Illustration of the idea behind the fully connected network architecture.

With the fully connected network architecture, the agent does not perform as well in general but manages to select waveforms more similar to the theoretical SNR thresholds, as shown in Figure 4.6. An example of a trained agent's waveform selection is shown in Figure 4.9 below.

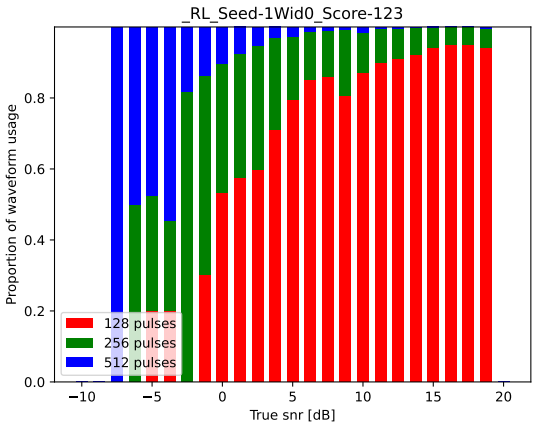


Figure 4.9: Number of pulses used when tracking targets with SNR shown on the x-axis. The SNR values are estimated SNR for 128 pulses of integration.

While the agent seems to use longer integration times against targets with lower measured SNR, the result is quite different from the theoretical algorithm whose results are shown in Figure 4.10. Since the agent performs bad in general, it is the more difficult to confirm whether the waveform selection is any good or not.

4.5.3 SNR-Based Algorithm

The SNR-based waveform selection algorithm described in section 3.9 chooses waveform based on previously measured SNR. The resulting behavior is presented in Figure 4.10.

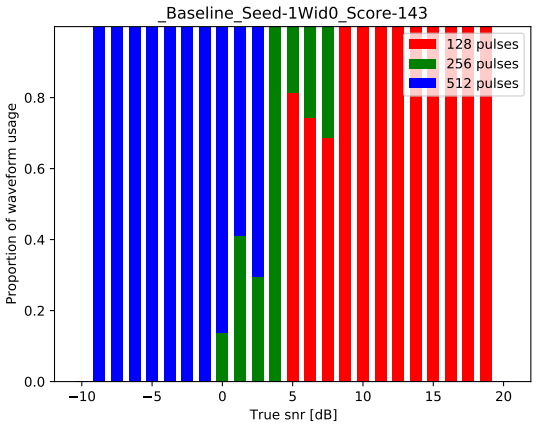


Figure 4.10: Number of pulses used when tracking targets with SNR shown on the x-axis. The SNR values are estimated SNR for 128 pulses of integration.

Since the agent could not learn to select waveforms efficiently, this SNR-based waveform selection algorithm is used for the rest of this project, for both baseline and the agent.

4.6 Hyperparameters

A search was made to find good learning rates for the actor and value networks. The 5 learning rates tested were $[10^{-6}, 5.6 \cdot 10^{-6}, 3.2 \cdot 10^{-5}, 1.8 \cdot 10^{-4}, 10^{-3}]$ for the policy and value network respectively totalling 25 different learning rate combinations in total. The best learning rates were $1.8 \cdot 10^{-4}$ for the actor and 10^{-3} for the critic. These values were used throughout this project. The remaining hyperparameters were found manually and are all listed below in Table 4.2.

Parameter	Value
γ	0.083 (a factor 0.92 per second)
λ	0.8
Actor Updates per Episode	10
Value Updates per Episode	1

Table 4.2: Hyperparameters used for training. Learning rates were found by grid search

4.7 Input State Optimization

A good input state was selected through forward selection, adding features one by one to see which ones positively impact the agent's ability to make good decisions. Independently, it was verified that binary encoding of features was the most effective as opposed to continuous inputs.

4.7.1 No Input

Initially, an agent was trained with no input state except whether a track exists. While one might expect this agent to learn nothing, decent improvement can be made exclusively through obtaining a well-proportioned distribution over actions, regardless of the exact state.

What is most interesting about this agent's performance, besides the fact that its performance is much better than an untrained network, is that its action distribution is very similar to the baseline implementation and other well performing agents. In terms of integration time, 49% of time is spent searching versus 51% spent on track. This is consistent over all obtained well-performing agents and significantly contributes to high performance.

4.7.2 Input features

There were many features considered for the input space. Because of the long training times, some grouping was done. The grouping of features can be found in Table 4.3. In the Table, we can see that some feature groups performed better than the no-input implementation. This tells us that those features provide the model with some essential information. However, the others will still be considered since they might give partial information.

4.7.3 Radial Distance, RCS, Radial Velocity

One of our first intuitions was that having the target's distance and radial velocity together with the estimated RCS would be a good state. However, the model performed worse than the model with zero input. After closer reconsideration it is not surprising since the model does not have any direct information regarding the termination criteria or the model's earlier actions. The only thing that the model could learn is zero Doppler.

4.7.4 Search Angle and Track Angle

The idea of using track and search angles was to allow the model to avoid illuminating tracks it is about to search over. The model performed better than the no-input model, which was unsurprising since we got some relevant information regarding the latest action.

Feature used
No input
Radial Distance
RCS
SNR
Radial Velocity
Search Angle
Track Angle
Confidence Level
Coast time
Waveforms
Number subsequent misses
Detections since last upgrade
Zero Doppler Detectability
Track Volume Derivative

Table 4.3: Table of all features used in the forward selection

4.7.5 Observation History

Observation memory contains the miss/detection history, the time between those actions, and the waveform used to obtain the observation. This single most well-performing track feature may not be surprising since it also takes up most nodes. The observation history contained the last 15 illuminations, whether they were successful or not. The time since the previous observation attempt, where only the last attempt gets updated, and lastly the waveform used for each attempt. This is more specific information regarding how the track has been illuminated and how often we found it and not, but also the time between illuminations. Since the information is so much more than the earlier features, there is no surprise that it performs well. However, since this input state is large, some optimization of the feature representation was done when including it in the forward selection where both detections, misses, waveform and times were all considered by themselves.

4.7.6 Confidence Level

The confidence level indicates how certain we are that it is a real track. It consists of three levels, candidate, tentative, or confirmed, represented as a one-hot encoding. This feature in itself was the worst-performing feature. This is not surprising since it gives no direct information about a track more than if we found it multiple times. It could be useful with other features since the confidence level impacts the certainty and the termination criteria.

4.7.7 Zero Doppler Detectability

The zero Doppler detectability is binary encoded, indicating if a track is fully detectable, has reduced detectability, or is entirely undetectable according to (3.9).

The idea was to allow the model to avoid illuminating tracks in zero Doppler blindness, which it cannot find, and indicate if a track is near the blind zone.

4.7.8 Track Volume Derivative

The Track volume derivative estimates the change in the number of tracks (to represent whether a steady state has been reached). The reasoning behind this feature is that if we found a steady state, we could use more of the time searching in case new targets appear.

4.7.9 Selected Features

The features selected for the final model are the ones stated earlier in Table 4.1. These features were found by adding the best performing feature as long as it improved the model until no more improvement could be achieved. However, RCS estimate was completely replaced by SNR estimate since it contains more relevant information.

4.8 Network Structure Optimization

When optimizing the model's architecture, we considered changing the feature stack and the track stack, removing the feature stack, and merging them.

4.8.1 Feature stack

When optimizing the size and shape of the Feature stack part of our network, we considered four different architectures. The different architectures which were evaluated can be found in Table 4.4. During the optimization we used the Track stack introduced in Section 4.2.

Feature stack			
Stack 1	Stack 2	Stack 3	Stack 4
Linear(features, 128) ReLU Linear(128, 64) ReLU Linear(64, $1 \cdot 3$)	Linear(features, 128) ReLU Linear(128,128) ReLU Linear(128, $1 \cdot 3$)	Linear(features, 128) ReLU Linear(128, 64) ReLU Linear(64, 32) ReLU Linear(32, $1 \cdot 3$)	Linear(features, 64) ReLU Linear(64, 64) ReLU Linear(64, $1 \cdot 3$)

Table 4.4: The 4 different network architectures used for optimizing the feature stack of the neural network

4.8.1.1 Conclusion

The best performing feature stack was stack 4. However, all the other feature stacks were within 2% of one another when evaluated over 100 runs of 1500 seconds each. Stack 4 was selected, considering it achieved the best performance. However, if non-convergence problems appear, one of the larger stacks could be considered to solve that problem.

4.8.2 Track Stack

The architectures which were considered when optimizing the Track Stack can be found in Table 4.5. For this evaluation we used feature stack 4 from Table 4.4, since it was the best performing.

Track stack		
Stack 1	Stack 2	Stack 3
Linear($3 \cdot m$, 512)	Linear($3 \cdot m$, 256)	Linear($3 \cdot m$, 1024)
ReLU	Relu	ReLU
Linear(512, 512)	Linear(256, 512)	Linear(1024, 1024)
ReLU	Relu	Relu
Linear(512, 256)	Linear(512, 256)	Linear(1024, 512)
Relu	Relu	Relu
Linear(256, o)	Linear(256, o)	Linear(512, o)

Table 4.5: 3 different network architectures used for network structure optimization of the track stack

4.8.2.1 Conclusion

All the track stacks considered reached approximately the same score when evaluated over 100 runs of 1500 seconds each. The difference in score was around 1% between the worst and the best model. However, the best performing stack was stack 3. Therefore it was selected as part of the architecture.

5
Results

In this chapter, we analyze the agent's performance in depth. Since including waveform selection in the output space yields inferior results as described in Chapter 4, for this chapter, the agent can only choose between the search action or the reillumination action, including which track to reilluminate. This means the neural network is given 51 output neurons mapped to reillumination of all 50 tracks and one output for search. Table B.1 shows the specific simulator parameters used. The simulation environment is simplified with no false alarms (noise induced candidate tracks) and immediately resolving measurements. To compensate for the need to resolve, we increase the time for each action to $m \cdot I_{\#\text{Pulses}}$. Furthermore, the tracker delay is set to 0. This is the environment where the training algorithm has been optimized. Other simulator configurations are considered later in this chapter.

A typical training session ran for around 600 episodes. An episode contains 15000 time steps from three different trajectories of 5000 time steps each. The average score of all three trajectories is plotted as one data point for each episode. An average over the last 50 episodes is plotted as well. Figure 5.1 shows how the episode scores evolve throughout training.

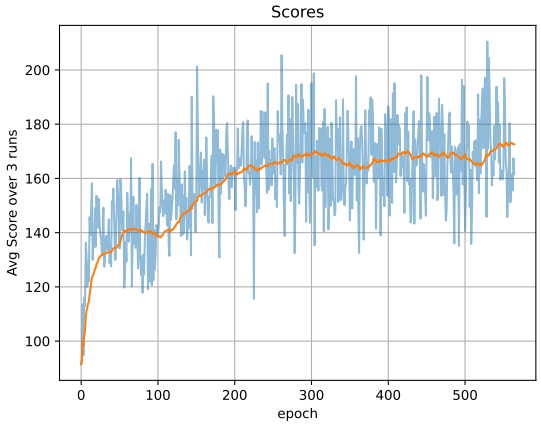


Figure 5.1: Trajectory scores over a training session. Blue line shows the average over three trajectories in one episode. Orange line shows the average score over the last 50 episodes (average score over the last 150 trajectories).

The outcome of this particular training session is used for the evaluation in the following subsections.

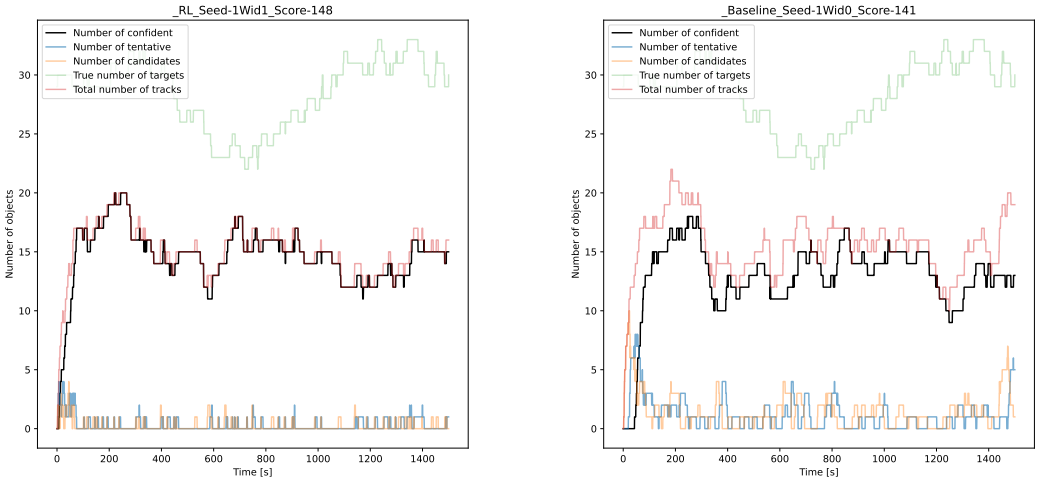
5.1 Agent Performance

The agent is compared to the baseline implementation explained in section 3.8, as well as a policy that only searches, never reilluminating an existing track, referred to as the *search-only* policy. Their performance is evaluated according to the score function in Section 4.4.2 over ten trajectories, lasting 1500 seconds each. Their scores are summarized in Table 5.1. The environment for each run is seeded so that the policies are compared over the same environments.

Table 5.1: Performance Scores, average score over 10 trajectories of 1500 seconds each

Agent	Baseline	Search Only
156.6	145.9	84.9

The RL-based agent consistently outperforms the baseline implementation regarding the scoring. The main difference in behavior between the two is that the agent behaves greedily due to γ-discount. In contrast, the baseline implementation primarily focuses on keeping tracks while searching for candidates. It will not greedily upgrades candidates or tentative tracks. We show in Figure 5.2 the number of tracks of each confidence level held by their respective trackers over the first run. The specific performance scores obtained in this one are 148.2 for the RL agent and 141.8 for the baseline.



(a) (b)

Figure 5.2: Number of tracks over the course of the trajectory for the RL Agent (a) and the baseline implementation (b)

Here we see that, while the total number of tracks stays similar, the RL Agent is much quicker at confirming tracks than the baseline implementation. An additional 500 trajectories were obtained from the agent and baseline with their corresponding

median confirmed track volumes and 25th and 75th percentiles plotted in Figure 5.3.

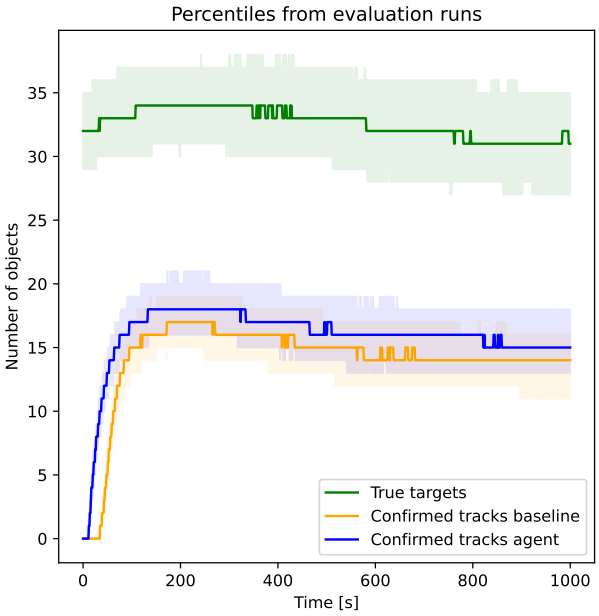


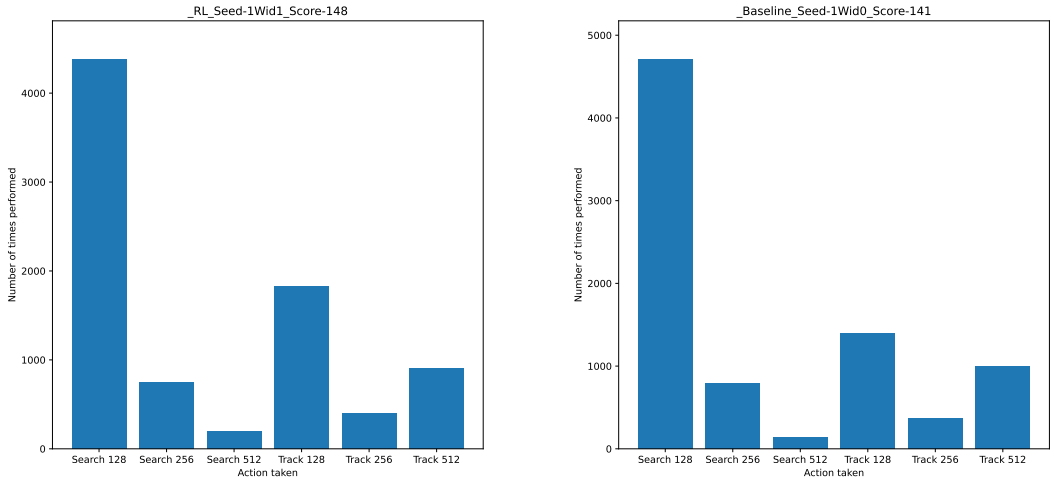
Figure 5.3: 500 trajectories each with their median confirmed track volume as well as 25th to 75th percentile ranges.

5.2 Agent Behavior

In this section, we perform a detailed analysis of the behavior of the trained agent in an attempt to interpret what it learns and what it is trying to do. In addition, we compare it to how the baseline implementation would behave in similar situations. From the plots showing the tracker history in the previous section, we saw that the agent tends to confirm tracks much more quicker than the baseline implementation. We should identify the reason behind this.

5.2.1 Action Distribution

First, we look at the distribution of selected actions for each policy in Figure 5.4. While the graphs are very similar, we see that the agent uses reilluminations slightly more often, particularly those with low integration times. Remember that the waveform selection $w \in \{128, 256, 512\}$ pulses is not selected by the agent but is determined by the algorithm described in Section 3.9. However, this would indicate that more reilluminations are performed towards targets with a high SNR, perhaps to upgrade their confidence levels for higher scores quickly.



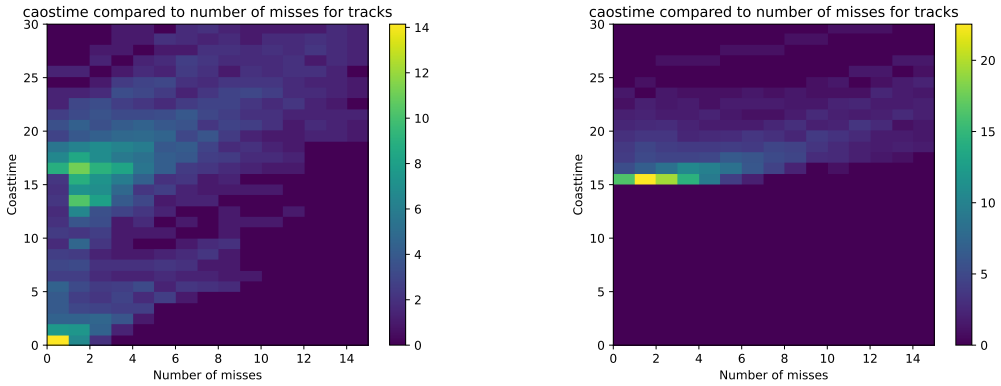
(a) (b)

Figure 5.4: Actions taken over the course of the trajectory for the RL Agent (a) and the baseline implementation (b). Bins on the x-axis are labeled "action #pulses", so that "search 128" represents search actions using 128 pulses.

5.2.2 Reilluminations

Let us also look at specifically the re-illuminations and when they are performed. In Figure 5.5, we show a heatmap of the square root of the number of re-illuminations performed towards tracks as a function of previous consecutive misses and coast time (time since last detection). The first exciting thing is that the baseline implementation uses a set threshold of 15 seconds of coast time before considering re-illuminations. This was selected through testing and is seen clearly in the heat map. The RL Agent re-illuminates after roughly the same 15 seconds of coast time. This is coincidental since the 15 second threshold for the baseline implementation was set by manual testing before the agent was trained.

A likely source of the score advantage in the RL agent is from the large number of re-illuminations being done against tracks that have very low coast time or consecutive misses. These are likely the greedy confidence upgrades performed to obtain scores earlier. Another observation is that the baseline in Figure 5.5(b) has very few illuminations at high coast time, as opposed to the RL implementation, which spreads its illuminations out more and lets tracks coast to above 25 seconds even though the track is removed after 30 seconds, which would result in a significant score penalty. Assuming that the tracks are not lost due to exceeding the coast time limit, delaying re-illuminations of confident tracks in this manner could free up time that could be spent on searching instead.



(a) (b)

Figure 5.5: Square root of the number of reilluminations over the course of the trajectory for the RL Agent (a) and the baseline implementation (b)

5.2.3 Time Before Confidence Upgrade

As we have seen, the agent upgrades the confidence level of tracks much earlier than the baseline implementation. While this is not a surprising result considering the baseline implementation's neglect of this aspect of the problem, and the agent's implicit incentive to do so through γ-discount, let's look closer at this behavior in Figure 5.6. Here, we see the time taken for confidence upgrades to occur with the agent and baseline respectively, and it is clear that the agent is quick to do so. As for the baseline implementation, we see spikes around 15 and 30 seconds, corresponding to the threshold before re-illuminations occur, with 30 seconds being a reasonable number due to the tracker requiring two associations for one upgrade. Something interesting to note however is the fact that the agent seems to be quicker in upgrading candidates to tentatives, compared to upgrading tentative to confirmed. This effect is not due to any aspects of the simulation, as confidence level does not explicitly affect the difficulty of association. As such, this effect is not seen in the baseline implementation.

It is not obvious where this behavior comes from since the additional reward given for each upgrade is the same regardless. However, this means that the relative increase in score given per track is larger for lower confidence levels, but it is not clear why the agent would see value in this. It is more likely that the agent sees the potential future reward in upgrading a candidate. If such an upgrade is successful, it is likely that a second upgrade is possible, yielding even more reward. However, an upgrade from tentative to confirmed means the reward from that particular track would be exhausted and no more can be obtained in the future.

5.2.4 Lost Tracks

Finally, we analyze the tracks which were lost. In Figure 5.7, we plot a histogram over the number of removed tracks with different bins corresponding to the reason

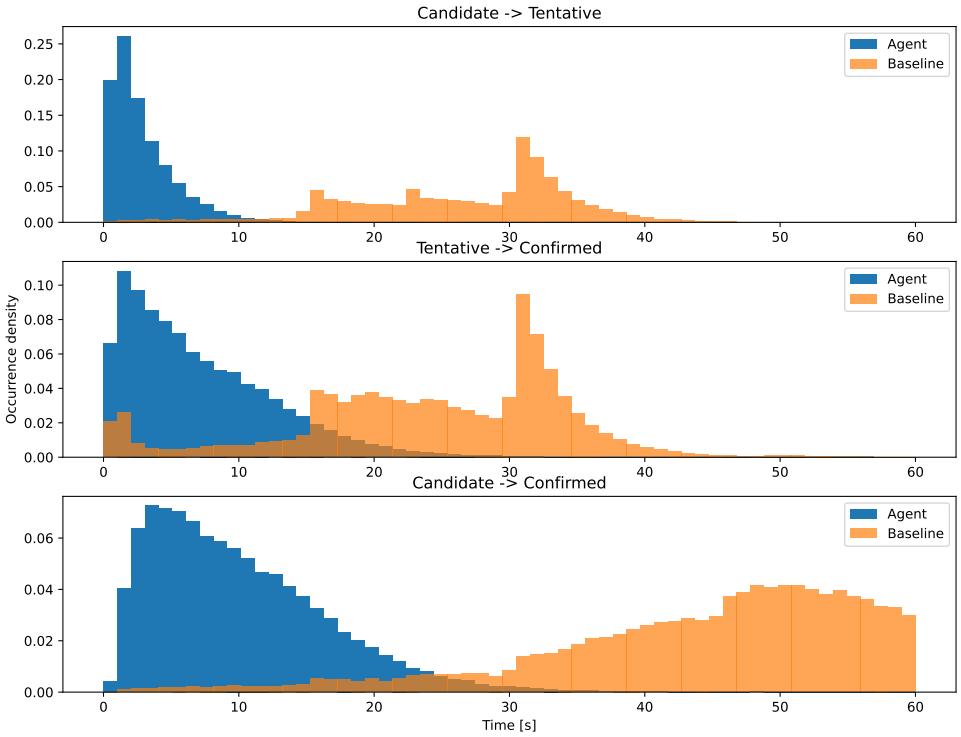
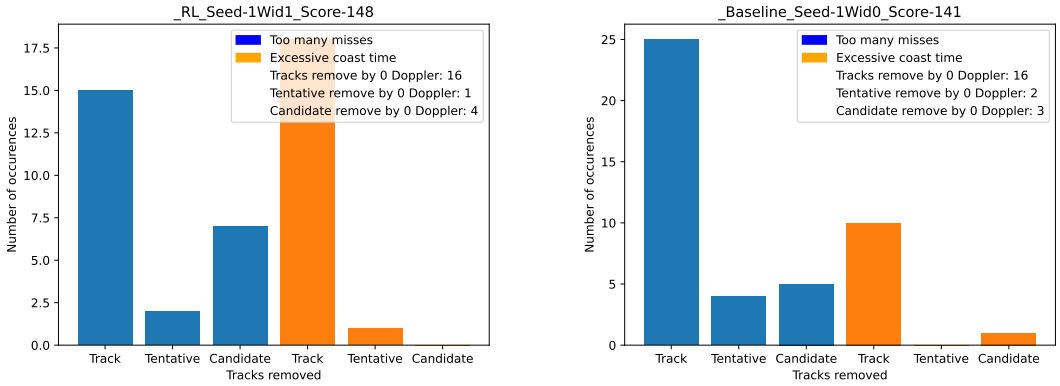


Figure 5.6: Histogram of the time taken to upgrade tracks for the agent and baseline respectively.

for removal. The legend also displays how many removals occurred in zero-Doppler blindness (Equation 3.9). The total number of removed tracks for the RL agent and the baseline implementation is quite similar, totaling 33 and 35 removed tracks, respectively. Around half of those were in the zero-doppler blind zone. Note that a track is removed automatically without adding to the histogram if it exits the search area. Compared to the baseline implementation, the only considerable difference we can see is that the RL Agent more often loses tracks to coast time rather than missing. Some trained agents (which performed similarly to the one evaluated here) would never terminate a track by missing and would instead leave a track alone before the last allowed miss, presumably to delay the impending reward penalty in case the last reillumination is a miss. This behavior seems to come from a trade-off between delaying a penalty using γ-discount versus risking the last allowed reillumination to be a miss.



(a) (b)

Figure 5.7: Removed tracks over the course of the trajectory for the RL Agent (a) and the baseline implementation (b)

5.3 Added Realism

While most of the project was focused on the heavily simplified simulation environment to make the development of the training algorithm easier, a few more realistic properties of the radar model were added to the simulation and evaluated to see how the agent would react and perform. We ran completely new training sessions on new trajectories with the added features for this. The first three of the below subsections adds one layer of realism each, with resolving measurements, tracker delay and false alarms. The final subsection adds all three layers of realism and with this we reach the realism of Nathansson [8]. The performance is compared to the same baseline implementation and a search-only policy. However, the RL training algorithm and the baseline implementation have yet to be optimized for these scenarios. Thus they are primarily optimized for the simplistic setting of above. This section aims to analyze the additional challenges a more realistic environment brings.

5.3.1 Resolving Measurements

In the simplified environment considered, measurements were immediately resolved without associating with an existing track. This section considers an agent trained in an environment where the required number of detections is raised to a whole number $n > 1$ before a non-associated measurement is considered resolved. In order to make this practical, the number of lobes overlapping during the search control input is changed to a whole number $m > n$. The parameters n and m are expected to be suitable for a radar of this type. Integration times are set to $I_{128} = 0.015, I_{256} = 0.024, I_{512} = 0.042$ seconds. The agent can also learn quite well in this environment, as seen in the training scores in Figure 5.8. After the training was complete, 20 trajectories of the agent, baseline, and search-only policy were obtained, and their results are shown in Figure 5.9.

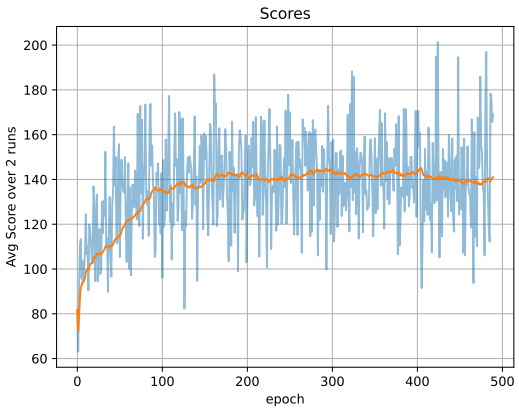


Figure 5.8: Training scores for the agent.

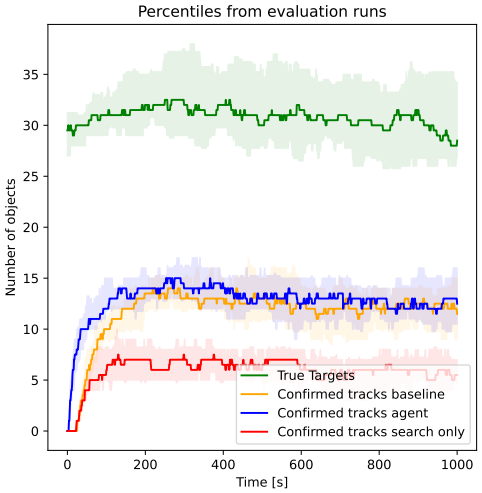


Figure 5.9: Tracker history over 20 trajectories.

5.3.2 Tracker Delay

In the real world, updating the track state requires processing time. However, ideally, we want to perform the following action immediately after completing the previous one. This means we may be unable to act using the most recent track state. For this section, the tracker delay is set to 400 ms, and the agent trained as usual obtained the training scores shown in Figure 5.10. Another 20 trajectories each are plotted in Figure 5.11. We remark that 400 ms is a very long time for processing, but since we discard the time from resolving the measurements we add this extra long time.

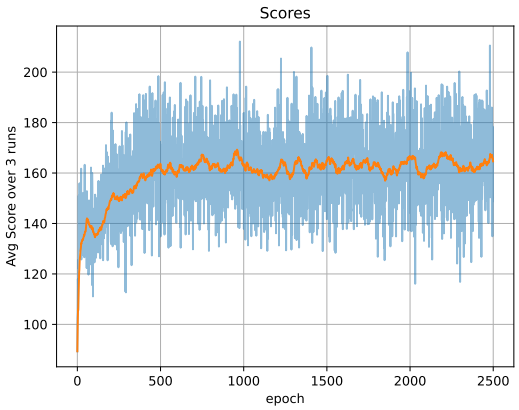


Figure 5.10: Training scores for the agent.

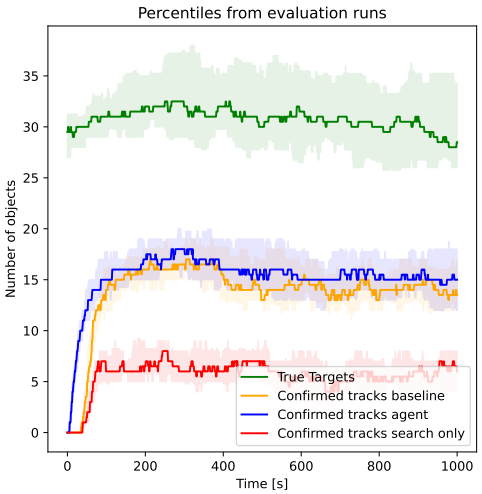


Figure 5.11: Tracker history over 20 trajectories.

5.3.3 False Detections

False detections arise from random spikes in signal noise, causing the tracker to initialize a track that does not correspond to a real target. In this section, the false alarm rate was set to 1% for each action, resulting in a false detection every few seconds. The reward function was also changed to grant a 0 reward for candidate tracks. This was done in order to keep the reward function fair, as otherwise, false candidates could be granting the agent rewards. Figure 5.12 shows the agent's training scores in this environment. The tracker history over 20 trajectories each is shown in Figure 5.13

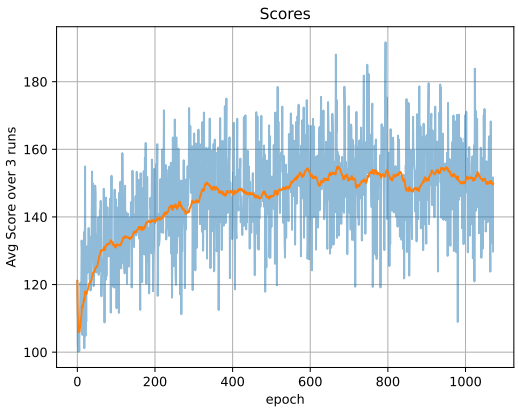


Figure 5.12: Training scores for the agent.

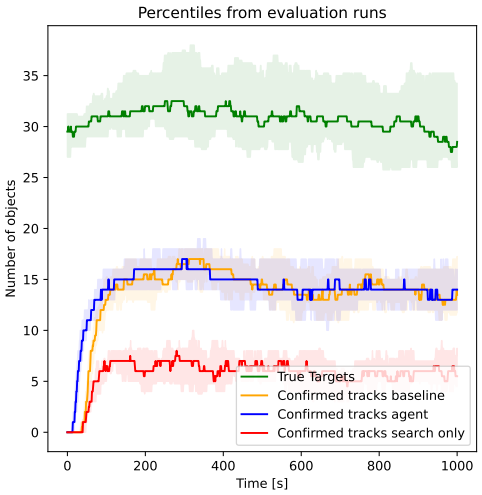


Figure 5.13: Tracker history over 20 trajectories.

5.3.4 Resolving measurements, tracker delay and false detections

An agent was trained on a simulator version where all three extra realistic aspects in Section 5.3.1 through 5.3.3 were added simultaneously. The training and evaluation results are shown below in Figure 5.14 and 5.15. We see that the training algorithm generally handles this fairly well, but again does not convincingly outperform the baseline algorithm. As such, we expect that a lot of improvements can be made in order to tailor the training algorithm for this level of realism, but nevertheless we have shown that it is at the very least viable to train an agent in this way, even with high levels of realism.

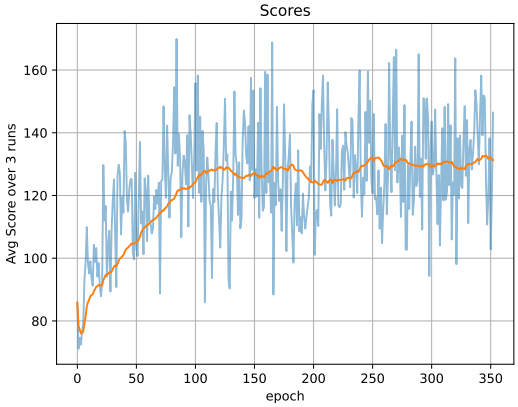


Figure 5.14: Training scores for the agent.

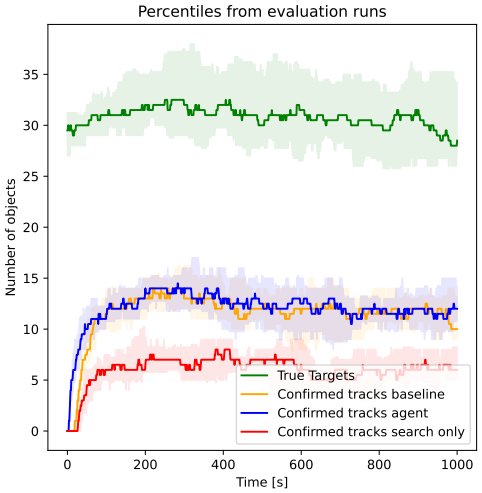


Figure 5.15: Tracker hitosry over 20 trajectories.

5.3.5 General Remarks

After modifying the simulation to include various realistic features, the training remains relatively stable, and the agent obtains a decent score even without largely modifying the training algorithm for each of the different increases of realism. However, the performance difference between the agent and the baseline implementation tends to diminish. Neither the training algorithm nor the baseline implementation is specifically designed to deal with these environments. Given the last implementation with all the realism factors implemented, the performance was still on par with the baseline, and therefore, there is reason to believe that further improvements can be made to keep the performance higher in realistic scenarios. This possibility leaves interesting future work to be done in these environments where the training algorithm has not been optimized to the same extent. but also that there exists more realism that can be introduced into the environment to come closer to solving the real-world scenario of the problem.

6
Conclusion

In this project, we investigated the feasibility of training an agent which could outperform our baseline implementation. Our results demonstrate that it is possible for an agent trained in a simple environment to perform better than the baseline implementation. The agents displayed improved behavior, as shown from the distribution of actions, re-illuminations, the time before confirming tracks, and the reduction in lost tracks shown in Chapter 5.2. These improvements show that the agent's ability to make informed decisions and effectively track targets in a simple environment.

The agent was also trained after introducing additional realism into the environment, such as resolving measurements, tracker delay, and false detections. By incorporating these improvements into the simulation, we were able to simulate a more challenging and realistic environment and assess the training loops and network structures' adaptability. This scenario shows that the agent is capable to adapt to a more realistic environment. This finding is significant as it demonstrates the potential for using reinforcement learning to achieve an agent which can be a valuable tool for decision-making and target tracking in surveillance radar.

6.1 Future Improvements

The most interesting improvement to be made is the inclusion of waveform selection while also achieving good tracking performance, which probably requires a different network architecture.

A much larger step forward may be to consider the tracking problem as a whole, evaluating the applicability of the ρPOMDP framework, for example, rather than relying on traditional association algorithms.

Evaluating the agent on environments with added realistic elements shows some potential areas for improvement in the training algorithm, which may not be obvious in a simple environment where those flaws are not as obvious. One such weakness is the tendency that the agent is not able to wait for rewards available too far into the future. For instance, when rescaling the action time to reflect the time necessary to perform resolving measurements, the agent seems to largely undervalue searching with the likely explanation that searches take too much of a time investment so that initiating a search requires looking far enough into the future to see the potential reward.

Bibliography

[1] Mauricio Araya et al. "A POMDP Extension with Belief-dependent Rewards". In: *Advances in Neural Information Processing Systems*. Ed. by J. Lafferty et al. Vol. 23. Curran Associates, Inc., 2010. URL: https://proceedings.neurips.cc/paper_files/paper/2010/file/68053af2923e00204c3ca7c6a3150cf7-Paper.pdf.

[2] S.S. Blackman. "Multiple hypothesis tracking for multiple target tracking". In: *IEEE Aerospace and Electronic Systems Magazine* 19.1 (2004), pp. 5–18. DOI: 10.1109/MAES.2004.1263228.

[3] Alan J Fenn et al. "The development of phased-array radar technology". In: *Lincoln laboratory journal* 12.2 (2000), pp. 321–340.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. "6.2. 2.3 softmax units for multinoulli output distributions". In: *Deep learning* 180 (2016).

[5] Kevin Gurney. *An introduction to neural networks*. CRC press, 1997.

[6] Hao Jiang, Sidney Fels, and James J. Little. "A Linear Programming Approach for Multiple Object Tracking". In: *2007 IEEE Conference on Computer Vision and Pattern Recognition*. 2007, pp. 1–8. DOI: 10.1109/CVPR.2007.383180.

[7] Harold W. Kuhn. "The Hungarian Method for the Assignment Problem". In: *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art*. Ed. by Michael Jünger et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 29–47. ISBN: 978-3-540-68279-0. DOI: 10.1007/978-3-540-68279-0_2. URL: https://doi.org/10.1007/978-3-540-68279-0_2.

[8] Axel Nathanson. "Exploration of Reinforcement Learning in Radar Scheduling". In: *Chalmers ODR* 20.500.12380/304144 (2022).

[9] "News & information: Stimson's introduction to Airborne radar, 3rd edition". In: *IEEE Aerospace and Electronic Systems Magazine* 29.5 (2014), pp. 41–41. DOI: 10.1109/MAES.2014.140036.

[10] J. Peters. "Policy gradient methods". In: *Scholarpedia* 5.11 (2010). revision #137199, p. 3698. DOI: 10.4249/scholarpedia.3698.

[11] Jan Peters and Stefan Schaal. "Natural Actor-Critic". In: *Neurocomputing* 71.7 (2008). Progress in Modeling, Theory, and Application of Computational Intelligenc, pp. 1180–1190. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2007.11.026. URL: https://www.sciencedirect.com/science/article/pii/S0925231208000532.

[12] *Pytorch Official Documentation*. https://pytorch.org/docs/stable/. Accessed: 2023-05-16.

[13] Simo Särkkä. *Bayesian Filtering and Smoothing*. Institute of Mathematical Statistics Textbooks. Cambridge University Press, 2013. DOI: 10.1017/CBO9781139344203.

[14] John Schulman et al. "Proximal Policy Optimization Algorithms". In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[15] John Schulman et al. *Trust Region Policy Optimization*. 2017. arXiv: 1502.05477 [cs.LG].

[16] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (Jan. 2016), pp. 484–489. ISSN: 1476-4687. DOI: 10.1038/nature16961. URL: https://doi.org/10.1038/nature16961.

[17] Matthijs TJ Spaan. "Partially observable Markov decision processes". In: *Reinforcement learning: State-of-the-art* (2012), pp. 387–414.

[18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[19] Oriol Vinyals et al. *AlphaStar: Mastering the Real-Time Strategy Game StarCraft II*. https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/. 2019.

[20] Oriol Vinyals et al. "Grandmaster level in StarCraft II using multi-agent reinforcement learning". In: *Nature* 575.7782 (Nov. 2019), pp. 350–354. ISSN: 1476-4687. DOI: 10.1038/s41586-019-1724-z. URL: https://doi.org/10.1038/s41586-019-1724-z.

[21] Alexander Zai and Brandon Brown. *Deep reinforcement learning in action*. Manning Publications, 2020.

[22] Pengfei Zhu et al. *On Improving Deep Reinforcement Learning for POMDPs*. 2018. arXiv: 1704.07978 [cs.LG].

A

POMDP & ρPOMDP

A POMDP[17] is described by six values (S, A, T, R, Ω, O) where T is a set of conditional transition probabilities between states, much like P earlier. Ω is the set of observations and $O(o|s, a)$ is a set of observation probabilities. The POMDP is typically used to model problems where the agent receives noisy or incomplete measurements of the state and has to act based on limited information. It turns out the POMDP framework relies on a memory with all previous observations and actions in order to act optimally, since multiple observation-action pairs say more about the true state compared to only the last observation. It is therefore valuable to gather as much information about the true state as possible in order to accurately predict rewards, and updating the belief state becomes a part of the optimal policy through information-gathering actions.

Even if the O and S are discrete spaces, storing the agent's entire past quickly grows to unmanageable sizes. Instead, all necessary information can be captured in a *belief state* $b(s)$, transforming the POMDP into a *belief MDP* where

$$b(s') = \eta O(o|s', a) \sum_{s \in S} T(s'|s, a) b(s). \tag{A.1}$$

Here, η is a normalization constant. The belief state needs to be initialized with an initial belief $b(s_0)$ which could be any distribution. Since the reward is still fully dependent on the true state, it is the same as in the standard MDP. However, we would like a value which is based on the belief state to help us determine the optimal policy. We introduce the value function

$$V_\pi(b) = E_\pi \left[\sum_{t=0}^{\infty} \gamma^t R(b_t, \pi(b_t)) | b_0 = b \right], \tag{A.2}$$

where $R(b_t, \pi(b_t)) = \sum_{s \in S} R(s, \pi(b(s_t)))b(s_t)$. We can solve the POMDP by solving for the optimal policy $\pi^*(b)$ which maximizes the value

$$\pi^*(b) \arg\max_a V_\pi(b). \tag{A.3}$$

There is one large problem with the POMDP framework in regards to solving the tracking problem; the reward function's dependence on $R(s, a)$. For surveillance tasks, the agent can not generally interact with the environment directly, and we may instead want to restrict the task to only updating the belief state. In such cases, we may be more interested in a reward function which depends on the ceratinty of the belief state, or the correlation between the belief state and the true state. The framework known as *ρPOMDP* [1] allows for incorporation of such a reward function and may be the definitive way to solve this problem.

B

Simulator parameters used

Variable	value	decription
TIME_HORIZON	5000	# time steps for training
TERMINATION_NR	[15,12,9]	allowed misses before removed
TERMINATION_TIME	30	allowed coast time
SECTOR_RMIN/MAX	[20e3, 450e3]	surveillance area [m]
SECTOR_ANGLE	75	sector area angle [deg]
INIT_TIME	500	initialization time [s]
N_INITIAL_TARGETS	30	# initial targets
SPEED_MIN/MAX	[400,800]	min/max target speed [m/s]
AVG_SPEED	600	average speed [m/s]
RCS_MIN/MAX	[-10, 10]	min/max RCS [dB m^2]
MANUV_RATE_MIN/MAX	[1e-4, 1e-3]	target maneuver rate [s^{-1}]
MANUV_END_RATE	[0.05, 0.1]	target maneuver rate end frequency [s^{-1}]
TURN_RATE_MIN/MAX	[0.001, 0.02]	Turn rate [rad/s]
RAD_VEL1/2	[15,40]	Radial velocity for zero doppler
FALSE_ALARM_RATE	0.0	False alarm rate
[R_VAR,U_VAR,VX_VAR	[100,0.12,	Variance for
VY_VAR,RCS_VAR]	5,5,2]	measurments
[R_MEAS_VAR, U_MEAS_VAR,	[100., 0.12,	noise for diagonal matrix
VX_MEAS_VAR , VY_MEAS_VAR]	5., 5.]	in kalman filter
N_PULSES	[128, 256, 512]	pulses for specific waveform
MEAS_TIME	[0.13, 0.22, 0.38]	Integration times for the available waveforms
BURN_IN_TIME	86	pulses to be discarded 0.07ms/pulse
LOBE_WIDTH_BORESIGHT	2.	Lobe width in degrees
WAVE_LENGTH	10e-2	wavelength for sent pulses
PRI_MIN/MAX	[1/10000, 1/5000]	Pulse repetition interval
LOBE_DECAY	0.5	Lobe decay based on azimuth angle
K	225	K faktor in SNR calculations
DETECTION_THRESHOLD	5.	SNR threshold [dB]
DUTY_FACTOR	0.1	time transmission fraction [0,1]
DECAY_EXPONENT	1.5	greater then 1, decreased range for increased azimuth angle
ATHM_DAMPING	4e-6	greater then 0, atmospheric attenuation
SP_DELAY	0.	Signal processing delay
LOBE_STEPS (m)	[1,1]	nr of overlapping lobes search/track
N_OUT_OF (n)	1	times found before its a detection
N_TENT_2_TRACK	2	detections before tent become track
N_CAND_2_TENT	2	detections before cand become tent

Table B.1: Parameters used when running the simplest version of our environment

DEPARTMENT OF SOME SUBJECT OR TECHNOLOGY
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY