OPTIMIZING TRAINING ARCHITECTURES FOR A HIGH-DIMENSIONAL, MULTI-AGENT SYSTEM IN MISSILE DEFENCE

A Thesis Submitted to the Division of Graduate Studies
of the Royal Military College of Canada
by

Callaghan Wilmott

In Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

July, 2023

© This thesis may be used within the Department of National Defence
but copyright for open publication remains the property of the author.

Acknowledgments

I extend my sincere gratitude to everyone who has supported my thesis journey. I'm deeply appreciative of my supervisor, Dr. Francois Rivest, for his unwavering guidance and substantial impact on my academic growth.

My heartfelt thanks go to my thesis committee for their insightful input and constructive criticism which greatly improved this work. I also wish to acknowledge my lab colleagues for their cooperation, with a special mention to Karla Gonzalez for her indispensable contribution to the missile environment simulator, and Masih Hashemi for his meticulous code reviews.

I am thankful to my family and friends for their steadfast support and encouragement throughout this journey. Lastly, I recognize the broader academic community whose dedication to knowledge advancement made my research possible. Thank you all for your invaluable roles in my academic journey.

Abstract

This study is focused on identifying an effective state representation, neural network architecture, and multi-agent training paradigm for high-dimensional missile defence scenarios. We follow a progressive approach, beginning with simpler environments and problem structures, then extending our findings to more complex settings. Initially, we analyze the impact of state representations on agent learning in the MountainCar environment. We demonstrate that higher resolution representations, such as the Radial Basis Function (RBF) transformation paired with convolutional deep learning architectures, enhance agent performance. Furthermore, we provide evidence that even lower resolution representations with higher noise can be trained effectively when a higher resolution representation is used as input to the critic network. We then broaden our exploration to a multi-agent particle environment, where we investigate the relative merits of centralized and decentralized execution paradigms, finding the decentralized paradigm to consistently outperform the centralized one. Lastly, we introduce the custom missile defence environment, where we apply lessons learned and perform ablation studies to validate the generalizability of our findings. We compare the performance of our trained agents with hard-coded baseline agents, effectively demonstrating the success of our progressively complex approach in the domain of Deep Reinforcement Learning (DRL) methods for missile defence scenarios.

Résumé

Cette étude se concentre sur l'identification d'une représentation efficace des états, d'une architecture de réseau neuronal et d'un paradigme de formation multi-agents pour des scénarios de défense antimissile de grande dimension. Nous suivons une approche progressive, commençant par des environnements et des structures de problèmes plus simples, puis étendant nos découvertes à des contextes plus complexes. Dans un premier temps, nous analysons l'impact des représentations d'état sur l'apprentissage des agents dans l'environnement MountainCar. Nous démontrons que les représentations à plus haute résolution, telles que la transformation par fonction a base radiale, associées à des architectures d'apprentissage profond convolutif, améliorent les performances des agents. De plus, nous démontrons que même des représentations à plus faible résolution avec un bruit plus élevé peuvent être entraînées efficacement lorsqu'une représentation à plus haute résolution est utilisée comme entrée dans le réseau critique. Nous élargissons ensuite notre exploration à un environnement de particules multi-agents, où nous étudions les mérites relatifs des paradigmes d'exécution centralisés et décentralisés, trouvant que le paradigme décentralisé surpasse systématiquement le paradigme centralisé. Enfin, nous introduisons l'environnement de défense antimissile personnalisé, dans lequel nous appliquons les leçons apprises et effectuons des études d'ablation pour valider la généralisabilité de nos résultats. Nous comparons les performances de nos agents formés avec des agents de base codés en dur, démontrant ainsi le succès de notre approche de plus en plus complexe dans le domaine des méthodes d'apprentissage par renforcement profond (DRL) pour les scénarios de défense antimissile.

Contents

Acknowledgments **ii**

Abstract **iii**

Résumé **iv**

List of Tables **vii**

List of Figures **xi**

Acronyms **xvi**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Problem Formulation . . . 2
1.3 Contribution . . . 3
1.4 Organization of Thesis . . . 4

2 Literature Review **6**
2.1 Reinforcement Learning . . . 6
2.2 Deep Learning . . . 10
2.2.1 Neural Networks . . . 11
2.2.2 Activation Functions . . . 12
2.2.3 Loss Functions . . . 13
2.3 Deep Reinforcement Learning . . . 15
2.4 Feature Representation . . . 20
2.4.1 State Encoding . . . 20
2.4.2 Representation Learning . . . 22
2.5 Asymmetric Architectures . . . 23
2.6 Multi-Agent Reinforcement Learning . . . 26

2.7 Conclusion 30

3 Experimental Environments **32**
3.1 Introduction 32
3.2 Mountain Car Environment 33
3.2.1 Problem Definition 33
3.2.2 Method 34
3.2.3 Results & Discussion 48
3.3 Multi-agent Particle Environment 56
3.3.1 Problem Definition 56
3.3.2 Method 59
3.3.3 Results & Discussion 65
3.4 Conclusion 72

4 Missile Defence Environment **80**
4.1 Introduction 80
4.2 Related Work 81
4.3 Environment Background & Baseline Policies 82
4.3.1 Environment Background 82
4.3.2 Baseline Policies 84
4.4 Agent Representation & Architecture 87
4.4.1 Observation Representation 87
4.4.2 Action Representation 92
4.4.3 Neural Network Architecture 97
4.5 Experiments & Results 101
4.5.1 Experiment Background 101
4.5.2 Results & Discussion 102
4.6 Conclusion 107

5 Summary and Conclusions **109**
5.1 Summary 109
5.2 Limitations and Future Work 110

Bibliography **113**

List of Tables

3.1 A summary of precision by representation type. 42
3.2 Neural Network Architectures Utilized with MountainCar Experiments. The count of the number of layers includes convolutional, dense, and the output layer, but excludes the input layer. 47
3.3 Hyperparameters used with Proximal Policy Optimization (PPO) for the MountainCar experiments 48
3.4 The mean and standard deviation of the asymptotic performance, and the convergence characteristics, for each representation configuration over 10 training runs. Configurations are ordered by mean asymptotic reward. Success is defined as an agent that consistently reaches the goal position, and mean timesteps to converge is determined by averaging the timesteps of the first datapoints exceeding 93 units of reward from each successful training run. 53
3.5 Tukey's HSD Test Results comparing mean rewards between groups. The columns represent the following: 'Configuration Pair' are the two configurations being compared; 'Mean Diff.' is the difference in mean reward between the two groups; 'p-adj' is the adjusted p-value; 'Lower' and 'Upper' are the lower and upper bounds of the 95% confidence interval, respectively; 'Reject' indicates whether the null hypothesis of no difference in means can be rejected (True) or not (False). 55
3.6 The reference figures and parameter counts for the neural network utilized in each configuration. Note that configurations 2, 6 and 9 refer to figures in Section 3.2. For these configurations, the architectures utilized in this section differ from those referenced only in the input dimension, which is now 10 instead of 2. 62
3.7 Hyperparameters used with PPO for the Multi-agent Particle Environment (MPE) experiments 72

3.8 Configuration definitions for each experiment conducted in MPE. The *Actor Input* and *Critic Input* columns specify the inputs to the policy and value networks, respectively. The *Merged Other Agents* column is only relevant when utilizing an RBF representation and specifies whether a single channel is used to represent information pertaining to other agents or if it is split into one layer per other agent. The *Shared Feature Extractor* specifies whether the actor and critic networks have any shared parameters between them. Finally, the *Training & Execution Paradigm* column indicates whether the same policy network parameters are utilized in action selection for all agents (*CTCE*), or whether each agent in the environment has parameters specific to each agent in the environment (*CTDE*). 73
3.9 The mean and standard deviation of the final performance for each representation configuration, with the highest-performing learning rate shown. Results are averaged over 5 training runs with the given learning rate. 76
3.10 Shorthand introduced to more readily interpret experiment configurations when referenced in results. CTCE and CTDE refer to the training paradigm utilized; RBF/RC/RBF1D indicates *RBF (2D)*, *Raw-continuous* or *RBF (1D)* observation types respectively; SYM/ASYM indicate whether the actor and critic architecture or symmetric or asymmetric respectively; M/NM indicates whether the other agents are merged or not merged into a single channel respectively; S/NS indicates whether the feature extractor is shared or not shared between the actor and critic. 77
3.11 The performance difference and statistical comparison resulting from using the *Merged Other Agents* representation, as opposed to using separate channels for other agents. Paired configurations are identical except for this representation decision. A positive *Relative Performance* Δ indicates higher performance resulting from using *Merged Other Agents* as opposed to using separate channels. 77
3.12 The performance difference resulting from changing the *Training and Execution Paradigm*. Paired configurations are identical except for the use of a Centralized Training with Centralized Execution (CTCE) or Centralized Training with Decentralized Execution (CTDE) paradigm. A positive *Relative Performance* Δ indicates higher performance resulting from using a CTDE paradigm as opposed to a CTCE paradigm. 78

3.13 The performance difference resulting from changing the *Shared Feature Extractor*. Paired configurations are identical except for the use of a *Shared Feature Extractor* as opposed to entirely separate actor and critic networks. A positive *Relative Performance* Δ indicates higher performance resulting from using separate actor and critic networks as opposed to a *Shared Feature Extractor*. . . . 78
3.14 The performance difference resulting from changing the input representation to actor and critic. Paired configurations are identical except for the use of an *RBF (2D)* or *Raw-continuous* input representation. Note that two corresponding *RBF (2D)* runs exist for each *Raw-continuous* run due to the added configuration parameter *Merged Other Agents* in *RBF (2D)* experiments, so the best performing of the two is shown for this comparison table since no significant difference was found in experiments varying the *Merged Other Agents* configuration. Note that shorthand corresponding to *Merged Other Agents* is omitted since it does not apply to *Raw-continuous* representations. A positive *Relative Performance* Δ indicates higher performance resulting from using a *Raw-continuous* input representation as opposed to an *RBF (2D)* representation. . 79
3.15 The performance difference resulting from our additional experiments, including the ablation experiment utilizing *RBF (1D)* representations, and our experiment utilizing an asymmetric architecture. In the *RBF (1D)* experiments, a positive *Relative Performance* Δ indicates worse performance resulting from using a *RBF (1D)* input representation as opposed to the relevant alternative representation. In the asymmetric experiment, a positive *Relative Performance* Δ indicates worse performance resulting from using the asymmetric architecture. 79

4.1 Hyperparameters used with PPO for the missile defence environment experiments 100
4.2 Agent configurations with figure reference: Type, Training Paradigm, and Network Architecture. 102
4.3 Agent configurations with figure reference: Observation. 102
4.4 Agent configurations with shorthand reference. 103

4.5 Adversarial performance of attacker and defender policies over a constant 100-member set of test scenarios. All agent configurations are trained 5 times to produce the mean and standard deviations shown, with the standard deviation being calculated over all 5×100 test scenario performances for the given agent configuration. For the attackers, a higher mean represents better performance, while for the defenders, a lower mean represents better performance. . . 105
4.6 Adversarial performance of attacker and defender policies over a constant 100-member set of test scenarios. All agent configurations are trained 5 times to produce the mean and standard deviations shown, with the standard deviation being calculated from the means of the 5 iterations of test scenario performances for the given agent configuration. For the attackers, a higher mean represents better performance, while for the defenders, a lower mean represents better performance. 105

List of Figures

2.1 The typical loop structure utilized to demonstrate the interaction between environment and agent. The environment provides the agent with an observation, i.e. a representation of the state of the environment, from which an agent returns an action to be executed in the environment. The environment additionally provides a reward signal to the agent based on the state, along with the new observation resulting from the action and environment events. . . . 6

2.2 The Rectified Linear Unit (ReLU) and Gaussian Error Linear Unit (GELU) activation functions utilized in this work. 14

3.1 A selection of representations of the environment. (a): The floating point Raw-continuous representation. (b): The *Radial Basis Function (RBF) (1D)* representation, which is a representation consisting of 2 1-dimensional arrays as shown. The array centers are indicated by the red arrows to clearly show the shift in the entry of peak intensity in the velocity array. (c): An *RBF (2D)* image representation of the state. (d): The *One-hot* image representation of the state. (e): A *Human-img* representation, consisting of a stack of three grayscale renderings from the most recent three timesteps. 36

3.2 A representation of a 9x9 *RBF (2D)* observation showing the greater precision deduced through sampling of 3 pixel values. . . . 39

3.3 The architecture utilized in training with proximal policy optimization for the *Raw-continuous* representation (1,017,506 parameters). 43

3.4 The architecture utilized in training with proximal policy optimization for the *RBF (1D)* representation (1,014,306 parameters). . . . 44

3.5 The architecture utilized in training with proximal policy optimization for the image-based representations of *RBF (2D)* (1,010,210 parameters), *One-hot* (1,010,210 parameters), and *Human-img* (1,014,306 parameters). 45

3.6 The architecture utilized in training with proximal policy optimization for the *Hybrid* representation (1,016,386 parameters) where the actor receives the *Human-img* representation and the critic receives the *RBF (2D)* representation. 46
3.7 The mean episodic reward achieved by our agents in 10 independent training runs of 301,056 timesteps each, when utilizing a *Raw-continuous* representation. 49
3.8 The mean episodic reward achieved by our agents in 10 independent training runs of 301,056 timesteps each, when utilizing a *RBF (1D)* representation. 50
3.9 The mean episodic reward achieved by our agents in 10 independent training runs of 301,056 timesteps each, when utilizing a *RBF (2D)* representation. 51
3.10 The mean episodic reward achieved by our agents in 10 independent training runs of 301,056 timesteps each, when utilizing a *One-hot* representation. 52
3.11 The mean episodic reward achieved by our agents in 10 independent training runs of 301,056 timesteps each, when utilizing a *Human-img* representation. 53
3.12 The mean episodic reward achieved by our agents in 10 independent training runs of 301,056 timesteps each, when utilizing a *Hybrid* representation. 54
3.13 The final positions of agents around the landmark when following an effective formation policy. The agents are shown to be approximately equidistant from the landmark, and are separated by approximately $\frac{2\pi}{3}$ radians. 58
3.14 A selection of state representations of the MPE environment. The default *Raw-continuous* state representation is shown at the top. *RBF (2D)* is a 50x50 3-channel image. *RBF (1D)* consists of 10 vectors of length 50, with each vector representing a dimension of the *Raw-continuous* representation, in an x and y frame of reference. The RBF-based figures are with respect to the darkest agent in the *Render* image, and are transformed to be in an agent-centered coordinate frame. 61
3.15 The *CTCE* architecture utilized when training on *RBF (2D)* observations, without shared parameters between the actor and critic networks. The total number of trainable parameters in this architecture is 1,024,838. 64

3.16 The *CTCE* architecture utilized when training on *Raw-continuous* observations, without shared parameters between the actor and critic networks. The total number of trainable parameters in this architecture is 1,014,566. 65
3.17 The multi-head *CTDE* architecture utilized when training on *RBF (2D)* observations, with shared parameters between the actor and critic networks. The total number of trainable parameters in this architecture is 1,004,910. 66
3.18 The multi-head *CTDE* architecture utilized when training on *Raw-continuous* observations, with shared parameters between the actor and critic networks. The total number of trainable parameters in this architecture is 1,056,124. 67
3.19 The *CTDE* architecture utilized when training on *Raw-continuous* observations, without shared parameters between the actor and critic networks. The total number of trainable parameters in this architecture is 1,011,120. 68
3.20 The *CTDE* architecture utilized when training on *RBF (2D)* observations, without shared parameters between the actor and critic networks. The total number of trainable parameters in this architecture is 1,029,582. 69
3.21 The *CTDE* architecture utilized when training on *Hybrid* observations, without shared parameters between the actor and critic networks. The actor networks receive *Raw-continuous* observations, while the critic receives *RBF (2D)* observations. The total number of trainable parameters in this architecture is 1,023,180. . 70
3.22 The *CTDE* architecture utilized when training on *RBF (1D)* observations, without shared parameters between the actor and critic networks. The total number of trainable parameters in this architecture is 1,033,704. 71
3.23 The mean episodic reward achieved by all of our agent configurations through 302,400 timesteps of training in the multi-agent particle environment's formation task. The mean and standard deviation are calculated from 5 training runs at each configuration and learning rate combination. 74
3.24 The mean episodic reward achieved by our highest performing agents in each configuration through 302,400 timesteps of training in the multi-agent particle environment's formation task with corresponding learning rate shown. 75

4.1 A map showing a rendering of the environment state, with key entities labelled. The skew normal distributions sampled when initializing the x-positions for defenders (in blue) and targets (in green) are also shown. The targets' radii are proportional to their value. The defenders' detection and interception radii are pictured in yellow and red, respectively. 83
4.2 A full set of channels representing the *RBF (2D)* observation of an agent, with their corresponding indices shown below the image. The 'H' panel is not part of the observation, but represents the human-rendered representation of the environment. 93
4.3 The observation representation, consisting of a 9x84x84 image, utilized as input to the actor and critic when training symmetric attackers, and as input to the actor only when training asymmetric attackers. The channels are numbered in accordance with the previously listed set of possible channels. 94
4.4 The observation representation, consisting of a 15x84x84 image, utilized as input to the actor and critic when training symmetric defenders, and as input to the actor only when training asymmetric defenders. The channels are numbered in accordance with the previously listed set of possible channels. 94
4.5 The observation representation, consisting of a 13x84x84 image, utilized as input to the critic when training asymmetric attackers and asymmetric defenders. The channels are numbered in accordance with the previously listed set of possible channels. 95
4.6 A sample of the output action logits, generated from a symmetric policy network, demonstrating the higher likelihood actions surrounding high-value target locations, as seen in the human-rendered representation of the environment on the right-hand side. 96
4.7 The U-Net feature extraction architecture utilized in all agent configurations. 97
4.8 The actor-critic network architecture used in the configuration 0, 2, 4 and 6 CTDE PPO implementation, with entirely separate networks for the actor and critic, and **the same input** utilized for each. Attacker configurations 0 & 2 contain 2,684,825 trainable parameters, while defender configurations 4 & 6 contain 4,063,709 trainable parameters. 98

4.9 The actor-critic network architecture used in the configuration 1, 3, 5 and 7 CTDE PPO implementation, with entirely separate networks for the actor and critic, and **asymmetric input** representations utilized for the actor and critic. Attacker configurations 1 & 3 contain 2,693,081 trainable parameters, while defender configurations 5 & 7 contain 4,059,581 trainable parameters. 99
4.10 The train loss (left) and mean episodic reward (right) for attacker agents over 1,024,000 timesteps of training. 104
4.11 The train loss (left) and mean episodic reward (right) for defender agents over 1,024,000 timesteps of training. 104

Acronyms

A2C Advantage Actor Critic
A3C Asynchronous Advantage Actor-Critic
ACER Actor-Critic Experience Replay
AI Artificial Intelligence
ASM Anti-Ship Missile
BMDS Ballistic Missile Defense System
CE Cross-Entropy
CNN Convolutional Neural Network
COMA Counterfactual Multi-Agent
CTCE Centralized Training with Centralized Execution
CTDE Centralized Training with Decentralized Execution
DDPG Deep Deterministic Policy Gradient
DDQN Double Deep Q-Network
Dec-POMDP Decentralized Partially Observable Markov Decision Process
DL Deep Learning
DNN Dense Neural Network
DQN Deep Q-Network
DRL Deep Reinforcement Learning
DRQN Deep Recurrent Q-Network
GAE Generalized Advantage Estimation
GELU Gaussian Error Linear Unit
HSD Honestly Significant Difference
MADDPG Multi-Agent Deep Deterministic Policy Gradient
MANA Map Aware Non-uniform Automata
MAPPO Multi-Agent Proximal Policy Optimization
MARL Multi-Agent Reinforcement Learning
MDP Markov Decision Process
ML Machine Learning
MLR Mask-based Latent Reconstruction
MPE Multi-agent Particle Environment

MSE Mean Squared Error
NLP Natural Language Processing
PG Policy Gradient
POMDP Partially Observable Markov Decision Process
PPO Proximal Policy Optimization
QMIX Q-value Mixture
RBF Radial Basis Function
ReLU Rectified Linear Unit
RL Reinforcement Learning
SC2LE StarCraft II Learning Environment
SMAC StarCraft Multi-Agent Challenge
SoS System of Systems
SOTA State-of-the-art
ST-DIM Spatiotemporal DeepInfomax
TD Temporal Difference
TRPO Trust-Region Policy Optimization
UAV Unmanned Aerial Vehicle
WTA Weapon-Target Assignment

1 Introduction

1.1 Motivation

The field of Deep Learning (DL) involves learning functions capable of mapping input data to some useful output, where this output depends on the task at hand. Reinforcement Learning (RL) aims to develop optimal agents, where an optimal agent is defined as an agent capable of maximizing cumulative reward through interaction with their environment. Such an agent is able to output the action that is expected to progress the agent along a cumulative reward-maximizing path, given an input observation of the state. The confluence of the fields of RL and DL is present in the field of Deep Reinforcement Learning (DRL), whereby the function approximators used in the RL domain are represented as deep neural networks. This field has demonstrated significant promise at extending RL methods to higher-dimensional input spaces and at learning more expressive and hierarchical representations of the environment [50, 51, 98]. By leveraging the power of deep neural networks, DRL has shown remarkable success in various domains, including robotics, gaming, and natural language processing [13, 85, 57]. These models can learn directly from raw sensory inputs, enabling them to process complex and high-dimensional data, and extract meaningful features that aid in decision-making.

That said, in the context of RL, the data quantity and compute capacity for training a DL model capable of learning useful representations from real-world high-dimensional data are not always present [25]. The magnitude of these requirements is contingent on the input representation utilized. When working in real-world domains, the representation of an agent's state might be predetermined by the sensors available in a system. However, training agents purely on data collected from a real-world system is limiting due to the high relative time and cost required to gather such data [18]. Additionally, in certain systems, the cost of an agent following a non-optimized policy can lead to damage to the system itself, further contributing to the high cost of collecting data in this manner. For this reason, it is often desirable to recreate

the defining parameters and dynamics of an agent and its environment in a simulated setting. In such a setting, data can be gathered much faster and at a much lower cost, with a trade-off existing for decreased fidelity in the simulation environment when compared to the real world system [38]. When discussing simulated environments, along with reducing the cost and time burden of data generation, one also removes the constraint imposed by real-world system sensors as the representation of the agent's state. The observation, or state representation, that an agent has access to can take on a wide variety of forms in such a setting.

Although the use of simulated environments offers significant advantages, it is important to note the limitations of utilizing simulations [38]. One key limitation, as previously mentioned, is the fidelity gap between the simulation and the real-world system. Simulated environments may not perfectly capture the intricacies of real-world scenarios. Additionally, simulations rely on predefined models and assumptions about the environment, which may not fully capture the complexity and stochasticity of real-world systems. This limitation can introduce biases or inaccuracies in the learned policies. These issues ultimately create difficulty for agents to generalize effectively to real-world systems when trained in simulation. While these issues fundamentally relate to state representation differences between real-world and simulation, this work does not look to explore this angle of state representation issues. Instead, we focus on how the state representation can be modified in simulation environments to improve agent performance and expedite convergence to an effective/optimal policy.

1.2 Problem Formulation

In this work, the environment of greatest interest to us is a custom, OpenAI Gym-style missile defence simulator. This environment presents numerous challenges when trying to train effective DRL agents. Firstly, there are many environment features that might play a role in an effective policy, making the decision of what state representation to utilize essential in the agent training architecture. The choice of state representation additionally plays a significant role in the design of neural network architecture. Secondly, the mixed cooperative and competitive environment contains multiple agents, which raises an additional set of questions relating to the multi-agent training architecture and paradigm to utilize.

With our objective of finding an effective representation, neural network architecture and multi-agent training paradigm for a high-dimensional mis-

sile defence environment, we have formulated a problem structure of escalating complexity. We start with simpler problem formulations and scenarios in environments of lower dimensionality and complexity when compared to our missile defence environment, to establish a foundation of understanding. We then apply the lessons learned to our increasingly complex problem formulations and environments, thereby following a constrained and progressive approach.

Our first problem formulation is based in the MountainCar environment, a classic benchmark in the field of reinforcement learning. This relatively simple setting allows us to explore the fundamentals of state representation and neural network architectures without the intricacies of a multi-agent scenario. These experiments look at varying the state representation and neural network architecture on agent performance and convergence characteristics. This experiment also introduces an asymmetric architecture, to see if a high-performance representation can be utilized to train actor networks to infer on an otherwise low-performing representation.

Progressing further into the multi-agent problem, we turn our attention to the Multi-agent Particle Environment (MPE). Here, we extend our experimentation into a multi-agent system. We utilize the top performing representations from the MountainCar environment to keep the scope of experimentation manageable and to see if the performance results extend to this environment. While continuing to refine our understanding of the impact of state representations and neural network architectures on agent learning in this environment, we additionally experiment with both centralized and decentralized execution paradigms.

Finally, we extend the experimentation to a custom OpenAI Gym-style environment simulating a missile defence scenario. After having gained valuable insights from simpler domains, we are able to utilize lessons learned to keep the breadth of experimentation in this complex domain within a manageable scope. These insights guide us in crafting an effective state representation, neural network architecture, and training paradigm capable of managing a high-dimensional, multi-agent system, which is the ultimate objective of this thesis.

1.3 Contribution

The overarching objective of this thesis is to identify an effective state representation and training architecture to utilize in a high dimension, multi-agent missile defence simulation environment. To this end, we break the problem

into constituent parts, and define the following contributions:

- A study on the impact of varying state representation, and corresponding neural network architectures, on the performance and rate of convergence to an effective policy of agents trained in the simple MountainCar environment.
- A study of varying state representation, corresponding neural network architecture, and multi-agent training paradigm on agent performance in the MPE. This study utilizes the outcome from the prior study in the MountainCar environment to focus the experimental scope.
- A formulation of a missile defence simulation environment, with state representation and training architecture guided by findings from prior experiments. We contribute a study on the impact of utilizing asymmetric observations for the actor and critic on agent performance in this environment. We demonstrate that a pixelized action space coupled with a segmentation network, using image-like observation representations as input, can enable training of performant agents in the environment. We additionally contribute an ablation study comparing agents trained on *RBF (2D)* representations to those trained one *One-hot* representations, to demonstrate the merit of the former representation type.

1.4 Organization of Thesis

The remainder of this thesis is split into a literature review, 2 core chapters, followed by a conclusion. Chapter 2 contains the fundamental background and literature review, which provides the foundational concepts upon which the rest of this document is built. Chapters 3 and 4 are the core chapters. Chapter 3 is further divided into two subsections, the first of which focuses on the low dimension, single-agent MountainCar environment, and the second of which focuses on the slightly higher dimension MPE. This chapter explores representation and architecture selection in both environments through a selection of experiments using varying state representations and neural network architectures, and subsequent analysis of agent performance. Chapter 4 focuses on the multi-agent missile defence simulation environment. It begins by outlining the structure and dynamics of the environment, the baseline agent behaviors, and the justifications for parameter selections utilized in the environment. It then utilizes the lessons learned from Chapter 3 to train both attacking and defending agents, and concludes by analyzing agent performance against our selection of baseline agents, with particular focus on comparing symmetric and asymmetric training architectures. Furthermore, Chapter 4 incorporates

an ablation study of utilizing *One-hot* representations, to confirm the merit of *RBF (2D)* representations in the given environment. The thesis finishes with Chapter 5, which concludes and summarizes the findings of our experiments, and outlines potential future research directions.

2 Literature Review

2.1 Reinforcement Learning

A common problem of interest in artificial intelligence research is that of determining the actions an agent should perform in an environment in order to meet a defined objective. Success in solving this problem in a multitude of realms has been achieved through the application of RL algorithms. Typical RL implementations aim to develop a policy for an agent's decisions by defining an environment and an objective, having the agent sequentially interact with the environment, and improving the agent's policy through provision of reward for completion of the objective. The general framework by which RL methods operate is encapsulated by the loop as shown in Figure 2.1.

Figure 2.1: The typical loop structure utilized to demonstrate the interaction between environment and agent. The environment provides the agent with an observation, i.e. a representation of the state of the environment, from which an agent returns an action to be executed in the environment. The environment additionally provides a reward signal to the agent based on the state, along with the new observation resulting from the action and environment events.

A common framework utilized to represent the sequential decision-making processes studied in RL is the Markov Decision Process (MDP) [10]. An MDP can be viewed as an expansion of the agent/environment loop into the constituent states, actions, transition probabilities and rewards that are possible

in the given agent/environment configuration. More formally, any RL task that satisfies the Markov property, which requires that the conditional probability of future states of the process (conditioned on both past and present states) depends only upon the present state, not on the sequence of events that preceded it, can be represented as an MDP.

The standard MDP consists of the tuple (S, A, P, R, γ), where the constituents of this tuple are defined as follows:

- S: The state space, which equates to a finite set of states.
- A: The action space, which equates to a finite set of actions.
- P: The transition probabilities between states. This represents a function which, given the current state and selected action, provides the probability distribution over the set of possible subsequent states of transitioning to the given state. The domain of this function is a cross product of the current state (an element of S), the action (an element of A), and the next state (an element of S). Its value can be anywhere in the range 0 to 1. $P : S \times A \times S \rightarrow [0, 1]$
- R: The reward function, which provides the immediate reward from state transition given the provided action. The domain of this function is a cross product of the current state (an element of S), the action (an element of A), and the next state (an element of S). Its value can be any real number. $R : S \times A \times S \rightarrow \mathbb{R}$
- γ: The discount factor, which is used to temporally adjust the value of rewards in order to favor immediate reward signals over delayed reward signals. In the case of MDPs with infinite time horizons, this discount factor is required to ensure that the expected returns are well-defined and finite. This can take any value in the range 0 to 1. $\gamma \rightarrow [0, 1]$

Building upon the foundation of the MDP, one can view the Bellman Expectation Equation as a representation of an MDP that's driven by a certain policy. A Bellman equation is a common method of representing a recursive relationship in an equation, which is useful in the realm of dynamic programming and optimization to demonstrate the well-defined nature of optimal solutions to optimization problems. The Bellman Expectation Equation provides a recursive calculation of the expected value for each state under that policy, as shown in Equation 2.1.

$$V_\pi(s) = \sum_{a \in A} \pi(a|s) \sum_{s' \in S} P(s'|s, a)[R(s, a, s') + \gamma V_\pi(s')] \tag{2.1}$$

The Bellman Expectation Equation states that the value of a particular state under a certain policy is the expected immediate reward, plus the ex-

pected value of the future states, given the current state and action. It sums over all possible actions, taking the probability of each action into account, and for each action, it sums over all possible next states, weighted by their probabilities, considering the immediate reward and the discounted value of the future state.

A breakdown of the components of the Bellman Expectation Equation is as follows:

- $V(s)$: Represents the value function under a policy π for state s. It is the expected return from state s when actions are chosen according to π.
- $\sum_{a \in A} \pi(a|s)$: This is an expectation over all actions a that can be taken in state s, under the policy π. It signifies the average effect of all possible actions, weighted by their probability under the policy.
- $\sum_{s' \in S} P(s'|s,a)$: This is an expectation over all possible next states s'. It signifies the average effect of all possible next states, weighted by their transition probability when taking action a in state s.
- $P(s'|s,a)$: Represents the transition probability of reaching state s' after taking action a in state s.
- $[R(s,a,s') + \gamma V(s')]$: Represents the expected return from taking action a in state s and transitioning to state s'. It's composed of the immediate reward $R(s,a,s')$ plus the discounted (γ) expected return from the next state s' under the policy π.

While the Bellman Expectation Equation provides us with a method for calculating the value of a state based on a particular policy, it doesn't necessarily give us the optimal strategy. That's where the Bellman Optimality Equation comes in, shifting from averaging over all possible actions according to a certain policy to choosing the action that maximizes expected return. Essentially, the Bellman Optimality Equation represents the most advantageous action we can take at any given state to achieve the highest future rewards, thus defining the best policy. To optimally solve an MDP is to find a policy function, π, that maps states to the optimal action from that state, where optimality is as defined in the Bellman Optimality Equation, which results from taking the argmax over all possible policies in Equation 2.1, resulting in Equation 2.2.

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} P(s'|s,a)[R(s,a,s') + \gamma V^*(s')] \tag{2.2}$$

In RL algorithms, the value function of a state plays a critical role as it helps guide the decision-making process towards the optimal policy. The

Bellman Expectation Equation expresses the future return as $_{\pi}(s')$, the discounted value of the next state, averaged over all actions and next states. It has a recursive nature, which allows it to be used for dynamic programming methods such as value iteration and policy iteration. A common alternate representation of the value function expresses the future return as a sum over future rewards, explicitly taking into account the timing of each reward. It directly shows that the value of a state is an expected sum of future rewards. This perspective of the value function is as defined in Equation 2.3. The state value function under policy π, $V_{\pi}(s)$, represents the expected return from state s when actions are chosen according to policy π. This function encapsulates the agent's anticipations about future rewards, and forms the basis for the iterative improvement of the policy.

$$V_{\pi}(s) = \mathbb{E}\pi \left[\sum_{k=0}^{\infty} \gamma^t R_{t+k+1} \middle| S_t = s \right] \tag{2.3}$$

In addition to approaches based on the value function, another significant family of methods in reinforcement learning stems from the concept of the state-action value function. Also known as the Q-function, this is a fundamental concept that broadens the perspective beyond states alone, to consider the implications of specific actions taken within those states. The Q-function quantifies the expected return or cumulative discounted future reward of taking a particular action in a given state, under a specific policy. The state-action value function for a policy, denoted by $Q_{\pi}(s, a)$, is defined as the expected return when starting in state s, taking action a, and thereafter following policy π, as shown in Equation 2.4. A key point is that the state-action value function takes both a state and an action as its argument, compared to the value function which only considers the state. The state-action value function forms the basis for many RL algorithms, such as Q-learning [90] and SARSA [64].

$$Q_{\pi}(s, a) = \mathbb{E}_{\pi} \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a \right] \tag{2.4}$$

The value function and state-action value function form the foundation of many RL methods, collectively being referred to as value-based methods.

A core decision required when formulating an RL problem is how to represent the state of the environment to the learning agent. This decision can have significant implications for the agent's ability to identify important features in the environment, and consequently learn an effective policy.

A variety of common methods of representing a state to the value function exist, with a key distinction existing between tabular methods and function

approximation methods. Tabular methods are typically limited to small, discrete state spaces, as they involve representing the entire state space in a tabular format, with rows corresponding to states and columns corresponding to actions (or vice-versa). The objective of tabular methods is to learn a value for each state-action pair. Because of the requirement that each state and action be uniquely represented with tabular methods, its memory requirements can quickly become prohibitive in high-dimensional environments due to the exponential growth of the table with increasing state space and action space dimension. Additionally, the requirement that each state-action pair be visited to generate adequate value estimates for the given table entry becomes prohibitive with growing state and action space dimension.

Due to this combinatorial growth of state-action spaces when using tabular methods, a common generalization for environments that are high-dimensional, or with continuous state and action spaces, is to utilize function approximation methods. Function approximation methods involve parameterizing the value, V_π, or policy functions. When provided the state or state-action pair as input, the parameterized value or policy functions will output a value or action (or probability distribution over possible actions), respectively. The parameters of the functions utilized with these methods are iteratively updated through the RL process, with the objective of providing incrementally improved estimates of the value of a given state/state-action pair input, or the optimal action from the same input as is the case with a parameterized policy function. The primary benefit of function approximation methods when compared to tabular methods is their ability to handle larger state and action spaces, in particular due to their ability to generalize across similar unseen states or state-action pairs. When provided with an unseen input, these methods can rely on inputs with similar features that were seen in training to produce effective outputs. They do, however, come with some limitations, as it is oftentimes challenging to ensure that function approximation methods converge to a globally optimal solution, V^*. This is particularly the case with environments of greater complexity, or greater dimensionality in regard to their state and action spaces.

2.2 Deep Learning

Deep Learning is a subfield of Artificial Intelligence (AI), that leverages backpropogation to train neural networks to map a given input to a target output. These algorithms process data with a hierarchy of multiple layers of artificial neurons. In the last two decades, with the explosion of computational power

and the availability of large-scale datasets, DL has progressed from theoretical foundations to a wide variety of real-world applications.

This review will delve into the primary components and mechanisms of DL, focusing on the architectural components utilized later in this work. To begin, it will look at two fundamental neural network architectures - Dense Neural Networks (DNNs) and Convolutional Neural Networks (CNNs). Then, it will discuss the roles and implications of different activation functions, particularly the ReLU and the GELU. Finally, this review will discuss the applications of a key loss function, namely the Mean Squared Error (MSE).

2.2.1 Neural Networks

Neural network architectures provide the structural foundation for DL models. They are mathematically formulated algorithms that mimic the neural structure of the brain, allowing computers to learn from observed data. Two widely used types of neural network architectures are DNNs, often referred to as Fully Connected Networks, and CNNs.

DNNs are a type of artificial neural network where each neuron in a layer is connected to all neurons in the previous and next layer. They are termed *dense* due to the high number of connections between neurons. These architectures are a common choice for solving simple regression and classification tasks where the inputs are not spatially structured. The structure of a DNN consists of an input layer, one or more hidden layers, and an output layer. Each layer comprises multiple neurons or nodes, where the information from the previous layer is processed and passed onto the next layer, with an activation function applied to the sum of a node's inputs.

DNNs have been used successfully in a wide variety of applications, forming a core component of numerous architectures. They are flexible, easy to understand, and can model complex non-linear relationships.

However, DNNs also have their limitations. They often require a large amount of training data to avoid overfitting. Additionally, they do not consider the spatial hierarchy of data, which is why they are not as effective for tasks such as image recognition where the relative position of pixels carries significant information.

CNNs, on the other hand, are a specialized kind of neural network that introduce a strong inductive bias for spatial relationships. They achieve this by employing the convolution operation. In the broadest sense, convolution is a mathematical operation on two functions that produces a third function. This third function represents how the shape of one function is modified by the other. In the case of 2D convolution, the input might be an image (treated

as a matrix of pixel values), and the kernel is a smaller matrix of weights that slides over the image (i.e., it's convolved with the image). The output is a new matrix (often called a feature map or convolutional layer) that represents features detected in the input image by the kernel. For example, a kernel might learn to detect edges, in which case the output feature map would have high values in areas of the image where there are edges.

The architecture of CNNs includes convolutional layers that use filters or kernels to perform local operations, allowing them to detect spatial patterns such as edges, corners, and textures. These convolutional layers are followed by pooling layers that downsample the spatial dimensions, retaining the most important information. By repeatedly applying these operations, CNNs can effectively learn hierarchical representations of spatial features in the input data. The final fully connected layers then map these features to the final output, such as class labels in a classification task.

CNNs have proven to be highly effective for image and video processing tasks, largely owing to their distinctive spatial invariance and locality properties. Unlike DNNs, CNNs are designed to automatically learn spatial hierarchies of features, which allows them to handle the high dimensionality of raw image data and exploit spatial correlations. This spatial invariance, combined with spatial locality, results in CNNs having a significantly lower number of parameters than DNNs of comparable performance in the spatial domain.

2.2.2 Activation Functions

One of the key components in neural networks is the activation function applied to the weighted sum of the inputs to a given node in a neural network layer. Activation functions introduce non-linear properties to the network, enabling them to learn from the complex patterns and dependencies in the data. Without activation functions, neural networks would only be able to learn linear relationships, which are often insufficient for real-world tasks. Two commonly used activation functions in DL, which are utilized in this work, are the ReLU and the GELU.

ReLU is perhaps the most widely used activation function in DL models [53]. It's mathematically defined as shown in Equation 2.5, which defines a function that returns x if x is positive, and 0 otherwise.

$$\text{ReLU}(x) = \max(0, x) \tag{2.5}$$

ReLU has several advantageous properties. Its simplicity makes it computationally efficient, reducing the time needed for training deep neural networks.

Moreover, it helps mitigate the vanishing gradient problem, a common issue in training neural networks where gradients tend to get closer to zero as they backpropagate through the layers, slowing down the learning process or causing it to stop completely. This alleviation of the vanishing gradient issue is due to the derivative of the ReLU function. Specifically, for positive input values, the derivative of ReLU is 1, resulting in a sustained gradient during backpropagation and preventing the gradients from diminishing for these inputs. For negative inputs, the ReLU function's output is zero, nullifying their contribution to the gradient. This property facilitates more effective gradient propagation through deep networks.

However, ReLU is not without its limitations. It suffers from a problem known as the *Dying ReLU* issue. Since the output is zero for all negative inputs, during the training process, some neurons might end up always producing negative outputs, causing their weights to not update. As a result, these neurons become unresponsive to variations in error, essentially leading to a portion of the model being inactive or *dead*.

GELU is another activation function that has been gaining popularity in recent years [35]. It is defined in Equation 2.6, which defines a function that approximates the cumulative distribution function of a Gaussian distribution.

$$\text{GeLU}(x) = 0.5x\left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}\left(x + 0.044715x^3\right)\right)\right) \tag{2.6}$$

The continuous and smooth nature of the GELU function helps in dealing with the vanishing gradient problem, similar to the ReLU function. Moreover, GELU does not suffer from the *Dying ReLU* problem as it activates for both positive and negative inputs, reducing the likelihood of neurons becoming unresponsive.

However, the more complex nature of the GELU function compared to ReLU means it is computationally more expensive. This might be a factor to consider when training large models or when computational resources are limited. For our purposes, we limit usage of GELU to the higher dimensional models utilized in the missile environment experiments, and use the ReLU activation for experiments in the lower dimensional environments. A visual comparison of the two activation functions can be seen in Figure 2.2.

2.2.3 Loss Functions

Loss functions play a critical role in the training of neural networks as a function approximator to map a given data input to a desired output. They

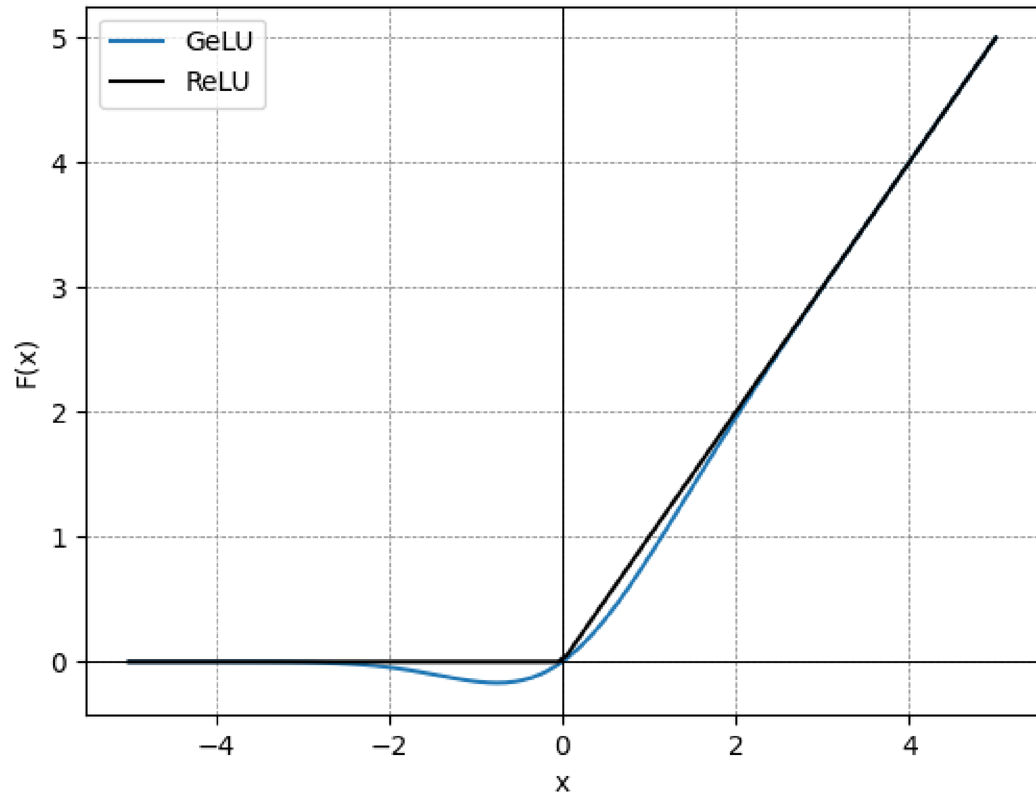


Figure 2.2: The ReLU and GELU activation functions utilized in this work.

measure the discrepancy between the model's predictions on a given input and the actual data's expected output, and this information is used to adjust the model's parameters during training using backpropagation. A widely used loss function that forms a component of our RL objective function is MSE.

MSE is a popular loss function primarily used in regression tasks. It calculates the average squared difference between a network's predicted value, and the actual value corresponding to the given input. Mathematically, it is defined in Equation 2.7, where y represents the actual values, $\hat{y}$ represents the predicted values, and n is the number of data points over which the mean is

being taken.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.7}$$

The intuition behind the MSE is simple: by squaring the difference, one ensures that the error is always positive, and by averaging these squared errors, we provide a single measure of how well the model is performing to guide parameter updates. The goal during the training process is to minimize this error through gradient descent.

One of the strengths of the MSE is its simplicity and ease of computation. Additionally, by squaring the errors, it penalizes large errors more than smaller ones, making it suitable for tasks where large errors are particularly undesirable.

The loss function to utilize depends on the specific problem being solved, the nature of the data, and the output requirements of the model. While MSE is popular in regression problems, alternatives, such as Cross-Entropy (CE) loss, are popular in classification tasks. For our own purposes, we utilize a custom loss function as described in the relevant section on PPO. Though this custom loss function makes use of MSE in its calculation, it is specifically formulated to fit the structure of the PPO algorithm.

2.3 Deep Reinforcement Learning

Fundamentally, RL algorithms rely on the learning of a function to develop an effective policy. The parametrization of these functions can take a variety of forms, however recent success has been found coupling the advances in the field of deep learning with the RL domain, by utilizing neural networks for this function parameterization. The application of DL techniques to RL problems has demonstrated great promise at adapting RL to high-dimensional input observations [6]. Such strategies are collectively known as DRL.

A common approach used by algorithms is to focus on learning the value, or as is the case with Q-learning and SARSA, the state-action value function. A notable early success in the field of value-based DRL can be seen in the work of Mnih et al. with the advent of the Deep Q-Network (DQN) algorithm [50, 51]. Their implementation makes use of a deep convolutional network to approximate the optimal state-action value function.

Previous analysis of utilizing a function approximator to represent the state-action value function demonstrated instability in training [81]. To counter this, DQN training utilizes experience replay and a separated target state-action value function from the evaluation function, with the target

updated iteratively at a lower frequency to the evaluation function. Updates are then performed by sampling randomly from the experience replay to generate batches, thereby removing the instability issues present in training without this strategy, due to correlation between sequential observations.

Utilizing the value-based paradigm described, Mnih et al. were able to successfully achieve human-level performance in 49 Atari games, with an identical training architecture used in each case. Hence, they were able to achieve this performance just from the rendering of the games to the screen using this DRL architecture, without the requirement of task-specific feature engineering.

Traditional value-based methods are known to be ineffective at dealing with scenarios where the action space is continuous or where the optimal policy is stochastic. Policy Gradient (PG) methods [93, 79] are a particular class of RL algorithms that look to address the issues present in value-based methods by instead representing the policy as a function approximation. This is accomplished by parameterizing the policy. Typically, this parameterization of the policy is in the form of a neural network where the policy network generates a probability distribution over the action space. PG methods function by performing updates to the policy based on the gradient of policy parameters with respect to some performance metric, such as expected return.

While PG-based algorithms have yielded promising results in a variety of domains, they do have drawbacks. Vanilla PG methods, such as REINFORCE [79], are entirely on-policy, meaning that they update their policy while interacting with the environment using the current policy, and utilize episodes just once for parameter updates. This makes the issue of sample inefficiency significant. Once the policy is updated, the collected data becomes outdated as the agent's behavior has changed. This leads to inefficient use of collected data, requiring the agent to collect new samples for each update. Additionally, since the policy network is continuously being updated, problems of stability in training can arise whereby the impact of a parameter update is too large and leads to complete deterioration of performance in subsequent episodes.

Mitigation strategies for the sample-inefficiency issue of vanilla PG methods have been proposed. One such example can be seen with Trust-Region Policy Optimization (TRPO) [69], which incorporates a trust region approach, limiting the extent of policy updates. It constrains the policy update to a certain neighborhood around the current policy, preventing large policy changes that could lead to instability or catastrophic performance degradation. By enforcing a more conservative update strategy, TRPO achieves more stable and incremental improvements, leading to improved sample efficiency. Another improved approach can be seen in Actor-Critic Experience Replay (ACER) [89], which utilizes experience replay, a technique popularized by deep Q-learning,

to improve sample efficiency. Instead of immediately using the collected experiences to update the policy, ACER stores them in a replay buffer and samples batches of experiences during training. This allows for more efficient use of the data and reduces the variance of updates.

Due to the importance from the standpoint of state/observation information encoding, in particular their allowance of asymmetric information during training and inference, we now wish to discuss in greater detail a popular subset of these algorithms known as actor-critic methods [8], of which TRPO and ACER are examples. As previously discussed, traditional RL algorithms often are subdivided into value-based and policy-based methods. Actor-critic methods are unique in the fact that they combine policy-based learning and value-based learning, in an attempt to harness the benefits of both paradigms. Actor-critic methods are named as such due to their combination of function approximation for the policy, defined as the actor, and the function approximation for the value, defined as the critic.

The Advantage Actor Critic (A2C) algorithm is one such algorithm. The training paradigm of A2C reinforces actions that were found to be better than expected, and weakens actions that were found to be worse than expected. This is accomplished through the following Temporal Difference (TD) error evaluation:

$$\delta_t = r_t + \gamma V_\theta(s_{t+1}) - V_\theta(s_t) \tag{2.8}$$

where V_θ is the current value function evaluated by the critic. By utilizing this TD error, or advantage estimate, the critic is able to determine the quality of the action, a_t, that causes the agent to transition from state s_t to state s_{t+1}. If the advantage is positive, and therefore indicates that the action result was better than expected, the action leading to that state is reinforced by increasing the probability of selecting that action from the given starting state. The converse is true should the TD error be negative.

Due to its proven success in multi-agent domains [97, 61], and our extensive usage of the algorithm in this work, an actor-critic algorithm that we wish to define in greater detail is PPO. PPO aims to address the instability of vanilla PG and improve sample efficiency, while remaining easy to implement [71]. This is achieved by having the agent interact with the environment with its current policy to generate a batch of data samples, and using this batch to optimize a surrogate objective function with stochastic gradient ascent over multiple epochs. Typical implementations additionally clip the objective function to be within a defined range to avoid too large of a step at each policy parameter update as in TRPO. This procedure is then repeated

for a certain number of iterations with the new policy being used to generate new batches of data samples each iteration. This procedure improves sample efficiency by using a batch of data samples in multiple epochs of parameter updates. The fact that a batch of data samples is used for parameter updates, along with the clipping of the objective function, improves the stability characteristics of the algorithm. PPO incorporates the Generalized Advantage Estimation (GAE) for effective variance reduction, further enhancing the stability and performance of the algorithm [70]. GAE computes a weighted sum of n-step estimators, with weights determined by λ raised to the power of the time-step, making it a kind of exponentially-weighted moving average of n-step estimators. On one side, you have the Temporal Difference (TD) error when $\lambda = 0$, which has low variance but can be highly biased. On the other side, you have the Monte Carlo (MC) method when $\lambda = 1$, which is unbiased but can have high variance. This amalgamation of multiple advantage estimates smoothes the policy update, facilitating more stable learning. The PPO algorithm utilized is defined as follows:

$$L_t^{CLIP}(\theta) = \hat{E}_t[min(\rho_t(\theta)\hat{A}_t, clip(\rho_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t)] \tag{2.9}$$

$$L_t^{VF}(\theta) = (V_\theta(s_t) - V_t^{targ})^2 \tag{2.10}$$

$$V_t^{\text{targ}} = G_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)} \tag{2.11}$$

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \ldots + (\gamma\lambda)^{T-t+1}\delta_{T-1} \tag{2.12}$$

$$L_t^{CLIP+VF+S}(\theta) = \hat{E}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)] \tag{2.13}$$

Algorithm 1 Proximal Policy Optimization (PPO)

Require: Actor-critic model with policy $\pi_\theta(a_t|s_t)$ and value function $V_\phi(s_t)$, environment with transition dynamics $p(s_{t+1}, r_t|s_t, a_t)$, hyperparameters ϵ, K, T
Ensure: Trained policy π_θ
1: Initialize actor and critic network parameters θ_0, ϕ_0
2: Set old policy parameters $\theta_{\text{old}} \leftarrow \theta_0$
3: **for** iteration $n = 1, \ldots, N$ **do**
4: Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
5: Compute advantages A_t using GAE as in Equation 2.12
6: Compute λ-return G_t^λ
7: Compute action probabilities $a_t \sim \pi_\theta(a_t|s_t)$
8: Compute old action probabilities $a_t^{\text{old}} \sim \pi_{\theta_{\text{old}}}(a_t|s_t)$
9: Compute surrogate objective function, L, using Equation 2.13
10: Update actor and critic network parameters using gradient ascent on calculated L for K epochs and with minibatch size M $\leq$ T
11: Set $\theta_{\text{old}} \leftarrow \theta$
12: **end for**
13: **return** Trained policy π_θ

Where equation and algorithm parameters are defined as follows:

- θ: the policy or value function neural network parameters
- $\hat{E}_t$: the empirical expectation over timesteps
- $\rho_t(\theta)$: the ratio of the probabilities in the new policy to those in the old policy, defined as: $\rho_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$
- $\hat{A}_t$: the advantage estimate at time t, as calculated by Equation 2.12
- ε: the clipping parameter, to prevent too large a policy update step in a given update
- $V_\theta(s_t)$: The parameterized value function as defined by our network output
- V_t^{targ}: The empirical, or target, value, as defined in Equation 2.11
- G_t^λ: The λ-return, used as the target in value function optimization
- G_t^n: The n-step return of following a trajectory
- r_t: The immediate reward after taking an action at time t
- c_1: the value loss coefficient
- c_2: the entropy loss (bonus) coefficient
- $S[\pi_\theta](s_t)$: the entropy loss (bonus) of the current policy given a state
- δ_t: The temporal difference error as calculated by Equation 2.8

- λ: The GAE lambda parameter.
- K: Number of epochs when optimizing the surrogate loss
- N: The number of iterations to run the optimization procedure to update actor and critic network parameters
- T: The number of timesteps in a batch, or a single iteration, N
- M: The minibatch size, where a minibatch is a subset from the T-length rollout collected in the current iteration

When performing stochastic gradient ascent on the surrogate objective function defined in Equation 2.13, it is important to note that while the value function contributes to the calculation of advantage, A_t, and the target value, V_t^{targ}, which in turn contribute to the policy gradient loss, $L_t^{CLIP}(\theta)$, the gradient does not flow through these values. In the PyTorch implementation of this algorithm utilized, they are detached from the computation graph prior to being used in the loss calculations. Similarly, the old log probabilities, $\pi_{\theta_{old}}$, utilized in calculated ρ_t, are detached. The gradient only flows through $V_\theta(s_t)$ in $L_t^{VF}(\theta)$, and π_θ in $L_t^{CLIP}(\theta)$.

2.4 Feature Representation

2.4.1 State Encoding

A linear function approximation involves representing a function as a weighted sum of input features. The effectiveness of linear approaches to function approximation heavily depends on how the states are portrayed via features. Ideally, these features should encapsulate regions of the state space where generalization is appropriate. However, a significant constraint of linear methods is their inability to handle interrelations between different features. To capture these interrelations, feature construction becomes an essential avenue for domain-specific knowledge of feature relationships to be incorporated into a linear solution to a task [78]. For this reason, a variety of state space representations have been explored in literature, such as those utilizing Fourier basis functions [42], coarse coding [88, 36], tile coding [3], and radial basis functions [60].

When utilizing non-linear methods, interactions between features don't need to be explicitly encoded in the state representation, because these methods can inherently capture and model the complex relationships through non-linear function approximation. The decision of how to encode a state is largely influenced by the problem at hand, and the simulation environment in question. In certain environments, such as the games of chess, Go, and shogi, or

the simulation environment utilized to train AlphaDev [48], the state space is both discrete, and sufficiently small, that information pertaining to the agent's state can be completely captured by the state representation. In other cases, such as in Atari games, Starcraft II, Dota 2, and self-driving car simulations, the state space is too large for this to be practical, and hence approximate representations of the state need to be utilized.

Google's DeepMind had success in training agents to play the games of chess, Go, and shogi [74, 72, 68]. These successful agents relied on state representations that were exact rather than approximate. Taking, as an example, their success in the game of chess [74], the state is encoded as an image of multiple layers. The 8x8 board results in an image of 8x8 pixels, with each of the 119 channels in the image encoding a different piece of information relevant to the game covering the most recent 8 timesteps of play. Encoded information includes binary-encoded piece locations for each player at each of the 8 timesteps, and a single channel representing a count of the total moves made so far in the game.

AlphaDev, another algorithm trained by Google's DeepMind, demonstrated the ability of DRL methods to discover faster sorting algorithms [48]. In their work, the action space of the environment consists of selecting an assembly language instruction, and the the state of the environment is represented as a combination of two elements. The first of these elements is a list of assembly language instructions, corresponding to the prior actions taken by the agent, and the second is the state of CPU memory and registers. These two elements are passed through independent encoders to generate embeddings for CPU state and assembly algorithm respectively, which are then combined to create a state representation of the environment.

In the groundbreaking work introducing DQN applied to Atari games, the capability of learning directly from high-dimensional state representations was demonstrated [50]. Their representation of the state is a grayscale image of 84x84 pixels. To make computations simpler, these grayscale representations of the Atari environment state are created by preprocessing raw Atari frames. Initially, these are 210 x 160 pixel images with a palette of 128 colors. They transform these into grayscale and down-sample to 110 x 84 pixels. Finally, an 84 x 84 area is cropped from the image, which generally contains the gameplay area. This procedure reduces input dimensionality, producing an approximate state representation that enhances computational efficiency.

OpenAI were able to find success in the high-dimensional, multi-agent setting of the video game Dota 2 [56]. However, a significant degree of feature engineering and specialist knowledge was utilized to enable the results achieved. Unlike the Atari environment, where success was found utilizing

high-dimensional sensory input, OpenAI Five utilized a vector representation of the environment, consisting of approximately 16,000 float and categorical variables defining various characteristics of the state, such as unit positions, health, and time remaining before major scheduled events.

2.4.2 Representation Learning

While appropriate hand-crafted features can yield effective model performance [56], it is oftentimes undesirable [12]. Firstly, generating such representations entails a high degree of manual effort, coupled with domain expertise, to produce high-quality features [39]. Additionally, it can lead to reduced expressivity of the generated agent model due to the fact that the feature engineer's biases are pre-encoded into the representation, at the potential expense of more abstract relationships in the data that are not captured in the feature set. Together, these issues can result in models with a high cost to implement, that may produce suboptimal results, and might not generalize well to different domains [73].

Recent research effort in deep learning has been put towards algorithms and methods that are able to extract useful representations automatically through the training process, rather than relying on specialized formulations of problems to develop effective models [75, 28, 34, 5]. One such example of a novel model-based approach to deep reinforcement learning in high-dimensional environments can be seen in the work of Hafner et al. [32]. Their paper introduces Dreamer, a reinforcement learning agent that learns long-horizon tasks by 'imagining' in a more compact, and therefore more efficient, latent space derived from high-dimensional sensory inputs. Dreamer uses a unique actor-critic algorithm to optimize the predicted state values and actions within this latent space, maximizing their value using analytic gradients. The agent outperforms existing model-based and model-free approaches on the DeepMind Control Suite, showing improved data-efficiency, computation time, and final performance when compared to alternatives such as PlaNet [33] and Asynchronous Advantage Actor-Critic (A3C) [49].

The solutions to the issue of learning useful representations from input data in the deep learning domain, and the solutions to these issues in the DRL domain often align. An example of this can be seen in the work of Anand et al. where unsupervised methods are adapted to a DRL domain [4]. The paper presents a new method for unsupervised state representation learning in the domain of DRL named Spatiotemporal DeepInfomax (ST-DIM), taking inspiration from advances in DL literature on estimation of mutual information [9, 82, 37, 83]. The ST-DIM technique maximizes mutual information across

both spatial and temporal axes of neural encoder observations, exploiting the spatio-temporal nature of visual observations in an RL setting.

Another example of adapting DL methods to the DRL domain can be seen in recent efforts to adapt self-supervised learning methodologies into DRL. As seen in the work of Yu et al. [98], success has been found with a Mask-based Latent Reconstruction (MLR) objective, whereby an auxilliary objective is defined with the aim of augmenting the representational understanding of the trained model. Specifically, they utilize a method inspired by successful methods in the Natural Language Processing (NLP) domain. In order to generate State-of-the-art (SOTA) results in NLP tasks, models typically undergo a pre-training stage whereby tokens are masked, and the model is trained to fill in the masked words [24, 76, 96]. Similarly, in the case of MLR, the model is given an auxilliary objective of reconstructing masked data in a latent space representation of a given visual input. The latent space reconstruction is utilized as opposed to reconstruction of the raw visual input data due to the relatively high dimensionality, and inclusion of excessive noise in the visual input representation.

In summary, the challenge at the heart of DL and more specifically, DRL, is the ability to learn efficient representations of data. Hand-crafted features, while powerful, have significant drawbacks including the need for domain expertise, reduced expressivity, and potential bias. Novel methods from the realm of DL are seeking to address these issues through automated feature extraction and learning representations, showcasing significant improvements in performance and efficiency.

While novel methods exist of ensuring the learned representations are effective and expressive, such as the aforementioned ST-DIM and MLR, these architectures will not be explored in this work. Instead, we focus on the impact of state encoding and the corresponding neural network architecture on the performance of an RL agent across a selection of tasks. The ultimate objective of this work will be to find an effective state representation and neural network architecture to be used in the training of agents in our custom missile defence simulation environment.

2.5 Asymmetric Architectures

The observation embedding utilized for model inference is often subject to restrictions. In the real world, these restrictions can be viewed from the limitations imposed by the sensors available to observe the system. In the multi-agent domain, this can be viewed from the standpoint of each agent's

knowledge of the policies of the other agents in the environment [47]. However, the same limitations need not apply during training. This fact has been utilized advantageously in the actor-critic paradigm to develop policies that demonstrate effective performance at a variety of tasks. A standard approach in actor-critic methods is to utilize symmetric architectures, whereby both actor and critic receive the same state representation as input. To benefit from the reduced limitations on state representation in simulation environments, asymmetric architectures can be utilized, whereby the actor and critic portions of the architecture receive differing inputs.

The utility of exploiting the presence of additional information during training was demonstrated by Pinto et al. in the realm of robotics [58]. In the field of robotics, real world training can be both time and cost prohibitive due to the number of episodes often required for policies to converge, and the damage due to cycling this would entail for a physical system. For this reason, a common tool in reinforcement learning for robotic control is to train policies in a simulation environment prior to testing the policies in the real world. The observability of the environment in simulation is not the same as in reality. In a simulation setting, it is possible to obtain full state representations of a robot and its environment. In reality, the environment is only partially observable, limited by the sensors available. Training a DRL algorithm on the fully observable states of the robot and environment in the simulation setting yield effective policies, however these policies are not extendable to a real world setting where the actor network can only receive partial observations. If instead only the partial observations, for example in the form of RGBD images of the robot and its environment, are utilized for training, algorithms often fail to converge on tasks involving complex behaviors. Pinto et al. address this issue utilizing a hybrid system much like the strategies employed in the multi-agent domain [47], whereby the observation representation utilized in training is not identical to that used during testing. Specifically, they utilize an actor-critic method, allowing them to differ the information received by the critic to that received by the actor. This allows them to benefit from the full observability of the simulator during training to guide policy updates, while still allowing the policies developed by the actor portion of the algorithm to be applicable in a real world scenario where full observability is not available. In the architecture they utilize, the critic network receives the complete state (full observation) of the system from the simulation environment, while the actor network receives an image rendered from the simulation from a viewpoint that is aligned with the camera positioning in the real world robotic system.

The concept of advantageously utilizing information available during training that is not available during inference to develop policies that are demon-

strably more effective during inference was also a core component of the success of Vinyals et al. in their StarCraft II reinforcement learning agent [84]. In their work, they utilize DRL to create an agent that has been shown to beat advanced human players in StarCraft II. Their training pipeline begins by utilizing a dataset of human-played games to generate policies that mimic human behavior in a supervised manner. This is due to the difficulty, due to the high dimensionality of the search space, of starting with a randomly initialized agent and relying on exploration to find effective strategies. From this baseline agent, they then utilize DRL to improve the agent performance.

The success of their agent at winning in StarCraft II against advanced human players was due to a plethora of DRL tricks and details in their DRL training architecture, including their novel league play approach, but a significant component of their success was their utilization of asymmetric information during training and inference.

The DRL algorithm utilized is a policy gradient method similar to A2C. During training, the critic network (value function) of their model receives both the player info along with the opponent info. A significant component of their design paradigm involved avoiding providing too great an advantage to their agent on the basis of it being non-human. This led them to include features such as a limiting the action rate and simulated latency for action execution to the system. For this same reason, despite its theoretical availability during inference, the inclusion of opponent information during inference would lead to a significant advantage to the agent-based player over a human player, and hence must be omitted during inference. Since only the policy networks, which are trained without this information, are utilized during inference, while the critic network is omitted, they are able to take advantage of the availability of opponent information in the critic network during training to develop better policy networks. This is just a small detail in their design decision, but is shown to have significant performance implications. According to the results they report, the inclusion of opponent information in their observations for the critic increased the average win rate percentage of their agents from 22% to 82% [84].

An example of the utility of asymmetric information in a multi-agent domain, as is the focus of later portions of this work, can be seen in the work of Lowe et al. where DRL methods were successfully applied to both cooperative and competitive multi-agent domains. The approach described in their work, named Multi-Agent Deep Deterministic Policy Gradient (MADDPG), takes advantage of the asymmetry of information available during training and inference. A novel component of their approach was to utilize centralized training of the agents with decentralized execution. The architecture utilizes

an actor and a critic for each agent in the environment. Hence, for N agents, a total of 2N networks are trained. The actor networks receive an observation and output an action for that agent to take. The critic networks receive the observations of each agent, along with the actions taken by each agent.

An advantage of this approach, as noted by Lowe et al., is the applicability of the architecture to a variety of domains. Specifically, whether the environment in question is cooperative, competitive, or mixed, the architecture is applicable. With varying reward structures, or even conflicting reward signals as would be the case in a competitive environment, each agent is able to independently approximate their state-action value function by means of their independent critic network.

In this work, we will look to explore the utility of applying the technique of utilizing an asymmetric architecture, whereby the actor and critic inputs differ, to our selection of tasks. In Chapter 3, this will be explored by providing the critic with a different state encoding that the actor. In Chapter 4, we deal with an adversarial scenario, where a natural asymmetry of information exists during inference in a similar manner to the StarCraft II environment. The attacker and defender agents might only have partial observability of each other's state. In this chapter, we will explore the merit of asymmetric architectures by providing the critic with greater observability of the opponent's state, to see if doing so allows for the training of more performant policy networks.

2.6 Multi-Agent Reinforcement Learning

Traditional applications of RL have focused on single-agent environments, however many environments of interest cannot be modelled as such. In particular, certain environments may need to be modelled as multi-agent systems, where the agents within the environment must take the behavior of other agents into account to learn an effective policy. These agents may need to exhibit a combination of cooperative and competitive behavior in order to maximize their reward. This fact has led to increasing interest in the field of Multi-Agent Reinforcement Learning (MARL), where the concepts developed in single-agent environments have been adapted and extended to solve multi-agent problems. Notable examples include real-world coverage control of Unmanned Aerial Vehicle (UAV)s whereby the environment is purely cooperative [99], and AlphaStar whereby the environment is competitive [84].

The training procedure for MARL systems, much like single-agent RL, relies on agents sequentially interacting with their environment, and learning

from their experiences. This is accomplished through agents observing their environment, actions taken, and resulting reward received, with the objective being to develop policies from this experience that favor the actions that led to high expected reward when provided with a particular observation. Research into MARL has often focused on extending concepts from single-agent RL to suit a multi-agent domain.

Many of the problems present in the single-agent RL setting are still present, and in many cases augmented, when translated to multi-agent domains [29]. In single-agent domains, the issue of credit assignment exists, whereby difficulty in determining which actions or events should receive credit for the agent's performance arises. This is oftentimes thought of in temporal terms [80]. However, in multi-agent domains, the dimensionality of the issue increases due to the added structural credit assignment [2]. In multi-agent domains, there exists an additional difficulty of assigning credit to the relevant agent's action among multiple interacting agents [92, 19, 94]. An additional issue arises when looking at the non-stationarity of the environment with the presence of multiple agents. Since the policies of the other agents in the environment are simultaneously changing during the learning process, an issue of non-stationarity of the environment arises. The changing policy of the other agents in the environment leads to an environment that appears non-stationary from the perspective of a single agent. Consequently, the Markov assumption of an MDP no longer holds, presenting agents with a challenge known as the *moving target problem* [16, 95].

Given the prevalence of multi-agent system problems ranging from guidance of UAV swarm behavior [20] to learning to play video games such as StarCraft II [84], there is increasing interest in the applications of RL methods to multi-agent systems. The environments that can be modelled as multi-agent systems come in a variety of configurations, but are most often divided into the broad categories of cooperative, competitive and mixed environments. Cooperative environments are those environments where the optimal policies of the agents involve cooperation to accomplish a shared objective. Competitive environments are typically zero-sum games, where two agents compete to maximize reward in their respective directions. Mixed environments, generally considered the most computationally and theoretically complex of the three environment types [99, 17], are those where optimal policies require a combination of competitive and cooperative behavior. Further subdivisions, of course, exist within each of these divisions of MARL. Important configurations in these subdivisions include the degree of observability that the agents have of the environment and their peers, whether the training is centralized or distributed, and whether the execution is centralized or decentralized.

In terms of observability in MARL, real-world scenarios often involve environments with varying degrees of observability. Agents may have full observability, where they can see the entire state of the environment and the actions of other agents, or they may operate under partial observability, where they can only observe a portion of the environment or the actions of others. The latter scenario, which turns the problem into a Partially Observable Markov Decision Process (POMDP), makes learning more complex [100], as agents must make decisions based on incomplete information and need to predict or estimate the unseen parts of the environment or the actions of their peers.

Training schemes in MARL can be broadly classified into centralized and distributed methodologies. In centralized training, agents' policies are updated based on mutual information. An example of such mutual information can be in the form of centralized value estimation during training [65, 77]. In contrast, distributed training involves each agent learning independently using its own experiences, without relying on any exchange of information between agents during the training procedure.

Execution in MARL can either be centralized or decentralized. Centralized execution implies that there is a central controller that decides the actions of all agents. On the other hand, decentralized execution allows each agent to make its own decisions based on their individual policies. Extending on the POMDP concept, the Decentralized Partially Observable Markov Decision Process (Dec-POMDP) is a framework for decision-making in multi-agent settings, where each agent has partial and possibly different observations of the environment, and the overall system dynamics are influenced by the combined actions of all agents [11].

Value-based methods have seen success being applied to multi-agent settings. The first such example can be seen in the Q-value Mixture (QMIX) algorithm [61]. QMIX is a value-based multi-agent reinforcement learning algorithm designed for cooperative scenarios, aiming to overcome challenges related to credit assignment and exploration. It uses a decentralized setup where each agent has its own Q-network, producing individual state-action value functions, which are then combined using a mixing network to approximate the joint state-action value function. An additional example of value-based methods succeeding in multi-agent scenarios is demonstrated by the work of Samvelyan et al. [67]. The environment utilized in their work is a custom-designed selection of cooperative StarCraft II scenarios, which they collectively refer to as the StarCraft Multi-Agent Challenge (SMAC). In this environment, they demonstrated the effectiveness of a selection of Deep Recurrent Q-Network (DRQN) architectures, including one policy gradient architecture with an additional Q-value critic network, at learning decentral-

ized policies for agents with the objective of winning a small scale two-team combat scenario. While the results were impressive, with the trained agents consistently outperforming the built-in heuristic-based AI, it is worth noting a couple of limitations of their approach. Firstly, they countered the issue of having a sparse reward system if they were to only reward wins, by instead establishing the default reward system to utilize a shaped reward that, aside from rewarding wins, additionally rewarded damage and kills of enemy agents. This is an understandable design decision given the environment context, but has implications on the generalizability of their results. Additionally, they opted to test only one on-policy, policy gradient architecture, Counterfactual Multi-Agent (COMA) policy gradients [27]. That said, their decision to focus on value-based methods does make sense given the discrete action space and mostly deterministic environment [85].

Vanilla policy gradient methods have seen limited success in multi-agent settings, which has been attributed to the fact that high variance in policy gradient estimation arises when applying them to multi-agent settings [67, 44, 47]. Additionally, the sample inefficiency of policy-gradient versions has been limiting to their performance when compared to value-based, off-policy alternatives [67]. However, variations of the algorithm have demonstrated promising results in multi-agent settings, as demonstrated by the work of Lowe et al. [47] which also serves as the origin of the MPE framework. In their work, they demonstrated the success of their multi-agent adaptation of Deep Deterministic Policy Gradient (DDPG) [46], which they refer to as MADDPG. This is an off-policy actor-critic algorithm that combines both Q-learning and PG methods to learn a centralized critic. They compared their MADDPG algorithm to DDPG with decentralized critics, and consistently found it to outperform across a variety of multi-agent scenarios.

In the case of the SMAC, Samvelyan et al. demonstrated that their Multi-Agent Proximal Policy Optimization (MAPPO) was able to perform at a level superior to the off-policy QMIX algorithm [61] across a majority of scenarios tested. This result is notable given the complete failure of COMA on challenging tasks in the work of attributed to the poor sample efficiency of on-policy methods, and the fact that the original work found QMIX to be, on average, the best performing agent. Furthermore, they rated a selection of their scenarios as *Super Hard* on the basis of QMIX being the only algorithm able to learn a winning policy from the algorithms they tested. This failure was attributed to insufficient exploration. MAPPO, however, was able to learn effective policies in these environments.

Yu et al. were further able to demonstrate the effectiveness of MAPPO in the MPE framework [97]. Of the three cooperative tasks tested, they demon-

strated performance on par with MADDPG on two tasks and superior to MADDPG on the third task. Agarwal et al. demonstrated a novel PPO implementation in the MPE framework that utilized an agent-entity graph embedding of a selection of cooperative scenarios, where graph edges were used to represent communication channels [1]. The algorithm demonstrated superior performance on the selection of cooperative tasks when compared to all algorithms tested, including MADDPG and QMIX.

Due to its demonstrated success in multi-agent settings, a multi-agent implementation of PPO will be utilized in the multi-agent experiments in this work.

2.7 Conclusion

This chapter has provided an extensive review of the current literature in the foundational subject areas that form the basis of the experiments to come. Specifically, we provide a review of relevant literature in the fields of Reinforcement Learning (RL), Deep Learning (DL), Deep Reinforcement Learning (DRL), and Multi-Agent Reinforcement Learning (MARL). The discussion commenced with an exploration of fundamental RL principles, advancing further into the more sophisticated DL techniques, which are critical to the success of DRL.

From the literature, we can see that the state representation or state encoding, neural network architecture, and multi-agent training methodology are all important decisions with significant impact on the success of agent training. In the case of neural network architecture, in particular, the symmetry or asymmetry of information provided to the actor compared to that provided to the critic is seen to have a role on agent performance. In the case of the multi-agent training paradigm, we see that various training paradigms exist, particularly in relation the presence of a single policy network controlling all agents, or one policy network per agent.

Each of these areas is critical to our upcoming experiments. In the following chapters, we will apply the concepts discussed in this literature review to our set of experimental environments. Specifically, in Chapter 3, we will experiment with two simple RL environments to gain an understanding of the dynamics and challenges in a simpler setting. Using the lessons-learned from these experiments, in Chapter 4, we will extend these concepts to a specific application of greater complexity, due to its higher dimensional state-space and multi-agent nature - the missile defence Environment. Through the upcoming experimental work, we hope to contribute insights to the effective training of

agents in this complex, multi-agent missile defence scenario. This literature review will act as a reference point, underlining the theoretical foundations on which our experiments are based.

3 Experimental Environments

3.1 Introduction

Recent work has demonstrated the capabilities of DRL to generate effective reward-maximizing agents in environments where the state representation is of high dimension. Given an observation of high dimension, DRL methods have been demonstrated to be capable of learning useful representations that enable effective decision making and control in these environments, in some tasks to a level of performance competitive with the highest-performing humans [56, 72, 74, 85]. However, agent performance on high-dimensional observations is contingent on the learning algorithm's ability to learn these useful representations of their environment [78]. A limitation of this work that should be noted is that it focuses on the PPO actor-critic architecture in all experiments across all environments tested.

One objective of this chapter is to systematically elucidate how the choice of state representations and the design of neural network architecture, utilized with the PPO actor-critic algorithm, can impact agent learning in RL environments. Specifically, we look to focus on raw variable representations coupled with dense networks, compared to image-like transformations of the raw variables coupled with convolutional networks. To do this, we will utilize the low-dimensional single-agent MountainCar environment.

The MountainCar environment serves as a representative low-dimensional, single-agent environment which allows us to conduct a focused analysis on the effect of various state representations on agent performance. Moreover, we also aim to investigate whether utilizing asymmetric architectures can help enhance agent performance, especially on low-performing representations. The goal here is to examine if the provision of a high-performing representation to the critic can lead to an improved overall performance.

The objective of Section 3.3 is to dissect the challenges associated with DRL in multi-agent systems, and further determine the efficacy of our proposed state representations and architectures under such circumstances. To

do this, we utilize the MPE. The MPE offers a slightly more complex scenario, due to its higher dimensionality and the presence of multiple agents, as compared to the MountainCar. In this setting, we are able to confirm whether findings regarding representation, neural network architecture and asymmetric architectures from our MountainCar experiments extend to this new environment, and we are able to expand the realm of experimentation to consider different multi-agent training architectures.

3.2 Mountain Car Environment

3.2.1 Problem Definition

Experiment Background

To better understand the impact of state representation on agent performance, a selection of popular representation options will be detailed, with results of training a DRL agent on each presented. Throughout this experiment, we will additionally define the precision of each representation we utilize, in order to identify whether a relationship exists between the precision of the information provided to the agent in training, and their ultimate performance in the environment.

Finally, to better understand whether an asymmetric architecture can augment performance on low-performing representations, an asymmetric architecture will be presented, with the actor and critic input representations for this architecture being selected based on the results of the representation performance experiment.

Environment Background

The popular MountainCar environment, utilized in this work, contains a car that is initialized in the valley of a sinusoidal terrain. Reward is provided to the agent for reaching an objective at the peak of the terrain. The amplitude of the terrain and the power of the car are defined such that the car cannot directly drive to the goal, and instead, must first build momentum to overcome the effect of gravity.

The continuous action space of the environment is of dimension 1, with the value corresponding to the force applied to the car in the given timestep. This action, a_t, can be any floating point value between -1 and 1. The state space is 2-dimensional and the state is updated according to Equations 3.1 and 3.2, with resultant values clipped to remain in their valid ranges of -1.2 to 0.6 for x_{pos} and -0.07 to 0.07 for x_{vel}. These variable ranges are taken as default from

the MountainCar environment. The car's velocity in a given timestep, $x_{vel,t}$ is calculated as shown in Equation 3.1 using the car's velocity in the prior timestep, $x_{vel,t-1}$, the selected action for the given timestep, a_t, the car's position in the previous timestep, $x_{pos,t-1}$, along with the car's power, P, and gravity, g, which are constants equal to 0.0015 and 0.0025 respectively. The car's position in a given timestep, $x_{pos,t}$, is then updated according to Equation 3.2 by taking the sum of the position in the previous timestep, $x_{pos,t-1}$, and the velocity in the current timestep, $x_{vel,t}$.

$$x_{vel,t} = x_{vel,t-1} + a_t \times P - g \times cos(3x_{pos,t-1}) \tag{3.1}$$

$$x_{pos,t} = x_{pos,t-1} + x_{vel,t} \tag{3.2}$$

At the start of an episode, the car is initialized with an x_{pos} between -0.6 and -0.4 by sampling a uniformly random distribution, and an x_{vel} of 0, with the goal state being located at $x_{pos} = 0.45$. Reaching this goal state results in a reward of 100 and terminates the episode. A negative reward is also assigned at each timestep according to the energy utilized by the agent, with a 0.1 unit penalty applied for taking a non-zero action in each timestep.

3.2.2 Method

State Space Representation

In order to evaluate the impact of the state space representation, we defined six different ways the state could be represented, observed or encoded by the agent (see Figure 3.1):

- **Raw-continuous**: The default observation space from MountainCar consisting of two float values, x_{pos} and x_{vel}, corresponding to the car's x-position and x-velocity (See Figure 3.1(a)).
- **RBF (1D)**: The popular RBF representation [15, 59], which has seen particular success when applied to the MountainCar problem [45]. RBF (1D) utilizes a 1-dimensional variation of the RBF, utilizing Equations 3.3 and 3.7 to create two 1-dimensional arrays to represent the state. A sample of the two arrays created using this method can be seen in Figure 3.1 (b).
- **RBF (2D)**: A 2-dimensional representation akin to a smoother version of the *One-hot* representation, using an RBF transformation of the *Raw-continuous* variables, consisting of a single-channel image (See Figure 3.1(c)).

- **One-hot**: A table-based version that discretizes the continuous space into a 50x50 grid and sets as 1 the table location corresponding to the x_{pos} and x_{vel} variables' values, which correspond to the axes of the grid (See Figure 3.1(d)).
- **Human-img**: A stack of three single-channel images corresponding to the renderings from the OpenAI gym environment's *render* function, from the three most recent timesteps (See Figure 3.1(e)).
- **Hybrid**: A pair of representations containing asymmetric [58] entries for the actor and critic, whereby the actor receives the *Human-img* representation and the critic receives the *RBF (2D)* representation. These two were selected for the *Hybrid* experiment based on the high performance of the *RBF (2D)* representation and the low performance of the *Human-img* representation, as will be demonstrated in the *Results & Discussion* section.

Radial Basis Function Transformations

Figure 3.1 (b) shows a sample of the two vectors that comprise an RBF (1D) representation. For the vector representing position on the left of the figure, the standard equation to generate an RBF is utilized as detailed in Equation 3.3. The index of an element in this vector corresponds to the x-position of the environment, and the mean of the RBF is $x_{pos,t}$. The value of a given pixel, i, is then calculated using the center-point of the given pixel, c_i, the mean of the RBF, $x_{pos,t}$, and a standard deviation of 1/16 of the axis range. This standard deviation was selected based on qualitative assessment of the generated vectors with this value, and equates to a σ_{pos} of 0.1125.

$$\phi_{RBF1D,pos}(c_i) = (\frac{1}{\sigma_{pos}\sqrt{2\pi}} e^{\frac{-(x_{pos,t}-c_i)^2}{2\sigma_{pos}^2}}) \tag{3.3}$$

For the vector representing x-velocity generated with Equation 3.7, a transformation is performed to ensure the axis of this vector also corresponds to the x-position axis of the environment, instead of corresponding to the x-velocity axis. This transformation brings the velocity vector representation into the same reference frame as the position vector, so the 1-dimensional convolution operation is performed in the same reference frame. To accomplish this, a blur effect is applied to the x-position representation according to the magnitude and direction of x_{vel}.

The mean of the RBF, μ_{vel} is generated by calculating the center point between the current position and the expected next position, assuming current

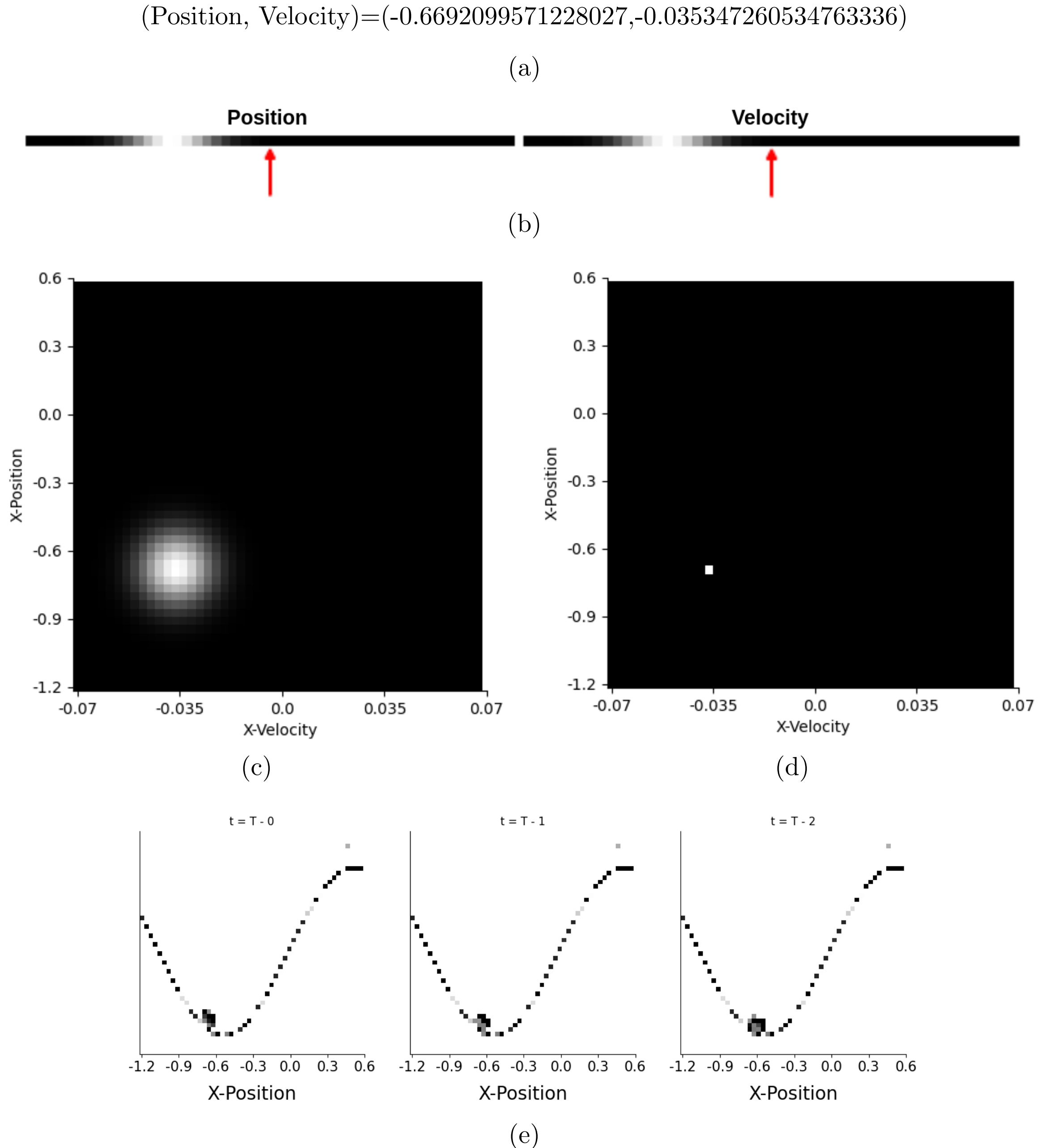


Figure 3.1: A selection of representations of the environment. (a): The floating point Raw-continuous representation. (b): The *RBF (1D)* representation, which is a representation consisting of 2 1-dimensional arrays as shown. The array centers are indicated by the red arrows to clearly show the shift in the entry of peak intensity in the velocity array. (c): An *RBF (2D)* image representation of the state. (d): The *One-hot* image representation of the state. (e): A *Human-img* representation, consisting of a stack of three grayscale renderings from the most recent three timesteps.

velocity, $x_{vel,t}$, were maintained in the next timestep. This shift is demonstrated in Equation 3.5, where Equation 3.4 is utilized to estimate $x_{pos,t+1}$. The blur of the vector to represent the velocity is then accomplished with Equation 3.6, where the standard deviation for the velocity vector, σ_{vel}, is calculated by stretching the standard deviation of the position vector, σ_{pos}, by an amount that increases in proportion to velocity.

$$\hat{x}_{pos,t+1} = x_{pos,t} + x_{vel,t} \tag{3.4}$$

$$\mu_{vel} = \frac{x_{pos,t} + \hat{x}_{pos,t+1}}{2} = x_{pos,t} + \frac{x_{vel,t}}{2} \tag{3.5}$$

$$\sigma_{vel,RBF1D} = (1 + |\hat{x}_{pos,t+1} - x_{pos,t}|) \times \sigma_{pos} \tag{3.6}$$

Once we have a mean, μ_{vel}, and standard deviation, σ_{vel}, calculated for our RBF vector representing velocity, we then generate the pixel values of the vector using these values in conjunction with the center-point of each pixel, c_i.

$$\phi_{RBF1D,vel}(c_i) = (\frac{1}{\sigma_{vel,RBF1D}\sqrt{2\pi}} e^{\frac{-(\mu_{vel}-c_i)^2}{2\sigma^2_{vel,RBF1D}}}) \tag{3.7}$$

For the *RBF (2D)* representation, the position blur is not utilized. Instead, Equation 3.8 is utilized to generate the single-channel image, where one axis represents the environment's x-position axis, as previously seen in the *RBF (1D)* representation, but the other axis represents the environment's x-velocity axis. Variables c_i and c_j are the coordinates along these respective axes of the center of a given pixel. The *Raw-continuous* observation values for x-position and x-velocity are used as the mean values, x_{pos} and x_{vel}, which correspond to the coordinates of the center of the RBF. We utilize a standard deviation of 1/16 of the respective axis ranges based on qualitative assessment of the generated image with this standard deviation, which equates to a σ_{vel} of 0.00875 and a σ_{pos} of 0.1125.

$$\phi_{RBF2D}(c_i, c_j) = (\frac{1}{\sigma_{pos}\sqrt{2\pi}} e^{\frac{-(x_{pos}-c_i)^2}{2\sigma^2_{pos}}}) \times (\frac{1}{\sigma_{vel}\sqrt{2\pi}} e^{\frac{-(x_{vel}-c_j)^2}{2\sigma^2_{vel}}}) \tag{3.8}$$

Information on the underlying state of the environment is limited by the precision of the selected representation. For this reason, we look to quantify

the impact on precision of each representation in order to utilize this quantification when comparing agent performance.

To begin with, we will look at the *Raw-continuous* representation from which all other observations are derived. The limiting factor for this representation's precision is the datatype precision of the variables representing position and velocity. In our case, these variables are represented using scalar values, and hence the precision is limited to 15 decimal places. As a percentage of the magnitude of our variable ranges, this provides a precision percentage, $Prec_{Raw-continuous}$, as defined by Equation 3.9:

$$Prec_{Raw-continuous} = \frac{1\text{e}{-15}}{|I|} \tag{3.9}$$

With an x_{pos} range magnitude, $|I_{x_{pos}}|$, of 1.8 and an x_{vel} range magnitude, $|I_{x_{vel}}|$, of 0.14, this yields resolutions of approximately 5.56e−14% and 7.14e−13% for the respective variables.

In the case of the *RBF (1D)* and *RBF (2D)* representations, each pixel intensity in a given representation is derived from a calculation utilizing the *Raw-continuous* values, with precision as defined in Equation 3.9. As such, by inverting the arithmetic that led to a selection of pixels having their given value, one can deduce the original variable values precisely.

Let us first consider the position representation, x_{pos}, as defined in Equation 3.3. We can rearrange this equation to obtain an expression for the underlying variable value, x_{pos}, as follows:

$$x_{pos,t} = c_i \pm \sqrt{-2\sigma_{pos}^2 ln(\sigma_{pos}\sqrt{2\pi}\phi_{RBF1D,pos}(c_i))} \tag{3.10}$$

By sampling two possible values for $\phi_{RBF1D,pos}(c)$ and obtaining the roots above, one can deduce the original value for $x_{pos,t}$ from which the representation was generated by taking the root that remains constant across the two samples. That said, *RBF (1D)* utilizes Equation 3.7 in its creation of the vector representing velocity. This derived representation utilizes both $x_{pos,t}$ and $x_{vel,t}$ in its creation. Hence, at least three points would have to be sampled to recover the original $x_{vel,t}$ representation through regression.

Similarly, when performing the same analysis in 2-dimensional space, one ends up with a solution space that corresponds to the equation of an ellipse. This can be seen in the rearrangement of Equation 3.8 as seen in Equation 3.11.

$$\frac{(x_{pos} - c_i)^2}{-2\sigma_{pos}^2 ln(2\pi\sigma_{pos}\sigma_{vel}\phi_{RBF2D}(c_i, c_j))} + \frac{(x_{vel} - c_j)^2}{-2\sigma_{vel}^2 ln(2\pi\sigma_{pos}\sigma_{vel}\phi_{RBF2D}(c_i, c_j))} = 1 \tag{3.11}$$

By sampling three points and identifying the intersection point of these solution spaces, one can deduce the coordinate corresponding to the original *Raw-continuous* representation. This can be seen in Figure 3.2.

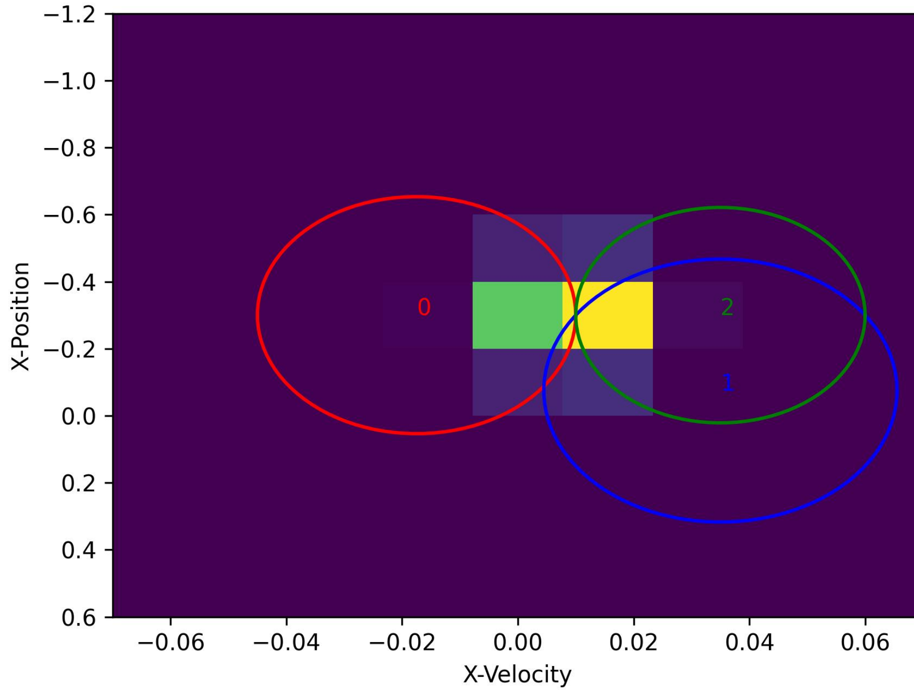


Figure 3.2: A representation of a 9x9 *RBF (2D)* observation showing the greater precision deduced through sampling of 3 pixel values.

The following parameters are utilized in Equation 3.11 to obtain the ellipses shown:

- Ellipse 0 Parameters:
  - Point(c_i, c_j) = (-0.3, -0.0175)
  - $\phi(c_i, c_j) = 1.158181450274851$
  - $\sigma_{pos} = 0.1125$

    - $\sigma_{vel} = 0.00875$
- Ellipse 1 Parameters:
    - Point(c_i, c_j) = (-0.075, 0.035)
    - $\phi(c_i, c_j) = 0.36935165886629207$
    - $\sigma_{pos} = 0.1125$
    - $\sigma_{vel} = 0.00875$
- Ellipse 2 Parameters:
    - Point(c_i, c_j) = (-0.3, 0.035)
    - $\phi(c_i, c_j) = 2.7291601267802785$
    - $\sigma_{pos} = 0.1125$
    - $\sigma_{vel} = 0.00875$

Following similar logic with *RBF (1D)* representations, the original precision of the $x_{pos,t}$ and $x_{vel,t}$ values can be recovered using regression on 2 vector values for each state variable. Hence, with a total of 4 vector values, 2 from each RBF vector representing $x_{pos,t}$ and $x_{vel,t}$, the original precision of the variables can be recovered. Ultimately, this results in an equivalent precision between the *RBF (1D)*, *RBF (2D)* and *Raw-continuous* representations.

$$Prec_{RBF1D} = Prec_{RBF2D} = Prec_{Raw-continuous} \tag{3.12}$$

When looking at the *One-hot* representation, the limiting factor of our precision is the grid resolution selected in generating the table representation. Since the continuous domain is discretized according to the image resolution, Res_{image}, selected for the other representations (50x50), the value will be translated to the pixel for which that value lies closest in the discretized domain. We can then calculate the precision as a percentage of the x-range, $Res_{image,x}$, provided by this representation in the following manner (note the same calculation holds for both variables since it is square):

$$Prec_{One-hot} = \frac{1}{Res_{image,x} - 1} = 2.04\% \tag{3.13}$$

In the case of *Human-img* observations, position is captured in the pixel location of the rendered car, while velocity is captured by stacking renderings from consecutive timesteps. The pixel location of the car will change between images in the stack, thereby capturing the motion of the car. Higher velocity will lead to larger differences in pixel location of the rendered car between consecutive renderings.

As shown by Equations 3.1 and 3.2 detailing the transition dynamics of the environment, the current velocity is determined by the velocity from the

previous timestep $x_{vel,t-1}$, the action taken in the current timestep a_t and the position in the previous timestep $x_{pos,t-1}$. In the *Human-img* representation, we have access to renderings that give us information on $x_{pos,t}$, $x_{pos,t-1}$ and $x_{pos,t-2}$. Using this information along with the fact that our timestep size between consecutive renderings a single environment timestep, we can approximate $x_{vel,t-1}$ to a level of precision determined by the rendered resolution, however we do not have any information that allows us to approximate a_t. Because of this, we can only approximate $x_{vel,t-1}$ with this representation, and not $x_{vel,t}$.

The precision of the approximation of $x_{pos,t}$ is determined by the pixel resolution utilized. In this work, we utilize an image resolution of 50x50, so our x_{pos} range is divided into 50 grid squares. Following an identical calculation as was utilized for the *One-hot* representation, the precision can be determined to be 2.04%.

$$Prec_{Human-img,x_{pos,t}} = \frac{1}{Res_{image,x} - 1} = 2.04\% \tag{3.14}$$

In the case of approximating $x_{vel,t-1}$, since we do not have the necessary information to approximate $x_{vel,t}$, the determining factor for precision is the precision of the approximations of $x_{pos,t}$ and $x_{pos,t-1}$ since the timestep size between consecutive renderings in the image stack is equivalent to a single environment timestep. We can approximate the velocity using Equation 3.16. More precisely, since our Δt is equal to a single timestep with no error associated with this time measurement, the calculation can be simplified to Equation 3.17. Since the precision error on $x_{pos,t}$ and $x_{pos,t-1}$ are both equivalent to 2.04%, the precision error on the result for $x_{vel,t-1}$ will be the sum of the precision error on these two values, or 4.08%.

$$x_{vel,t-1} = \frac{x_{pos,t} - x_{pos,t-1}}{\Delta t} \tag{3.15}$$

$$x_{vel,t-1} = \frac{x_{pos,t} - x_{pos,t-1}}{\Delta t} \tag{3.16}$$

$$x_{vel,t-1} = x_{pos,t} - x_{pos,t-1} \tag{3.17}$$

$$Prec_{Human-img,x_{vel,t-1}} = 2 \times Prec_{Human-img,x_{pos,t}} = 4.08\% \tag{3.18}$$

Representation	Precision	
	$Prec_{x_{pos}}$	$Prec_{x_{vel}}$
Raw-continuous	$5.56 \times 10^{-14}\%$	$7.14 \times 10^{-13}\%$
RBF (1D)	$5.56 \times 10^{-14}\%$	$7.14 \times 10^{-13}\%$
RBF (2D)	$5.56 \times 10^{-14}\%$	$7.14 \times 10^{-13}\%$
One-hot	2.04%	2.04%
Human-img	2.04%	4.08%

Table 3.1: A summary of precision by representation type.

Neural Network Architectures

The objective of this work is to compare agent performance with varying representation, and neural network architecture. Dense layers were utilized for the *Raw-continuous* representation. One-dimensional convolution was utilized for the *RBF (1D)* representations. Two-dimensional convolution was utilized in architectures that receive image-like inputs. Specifically, it is utilized with *RBF (2D)*, *One-hot*, and *Human-img* representations, and the actor network of the *Hybrid* representation.

The influence of parameter count in a given architecture is often overlooked when comparing network performance, potentially skewing results and interpretations. To ensure the credibility and fairness of our results, we acknowledged the potential bias induced by a large difference in the number of parameters. An important aspect of our work was to align the number of trainable parameters across all architectures to approximately 1M in each case, mitigating the confounding influence of architectural differences and providing a more balanced comparison between representations and architectures. In the case of our MountainCar architectures, there is less than a 1% difference between parameter counts of each network, which we deem to be close enough to not have a significant influence on agent performance. Details regarding the parameter alignment are presented in Figures 3.3-3.6.

For the default *Raw-continuous* representation of dimension two, the actor-critic architecture utilized is detailed in Figure 3.3. A shared feature extractor of three dense layers, with each layer containing 480 units, precedes the actor and critic heads. The actor and critic heads have two layers each of the same architecture, consisting of a dense layer of 480 units followed by a dense layer of 96 units. The actor head finishes with a single unit output, corresponding to the selected action, which is a scalar value between -1 and 1 as previously detailed in the discussion of the environment action space. The critic head also finishes with a single unit as output, where the value of this scalar output

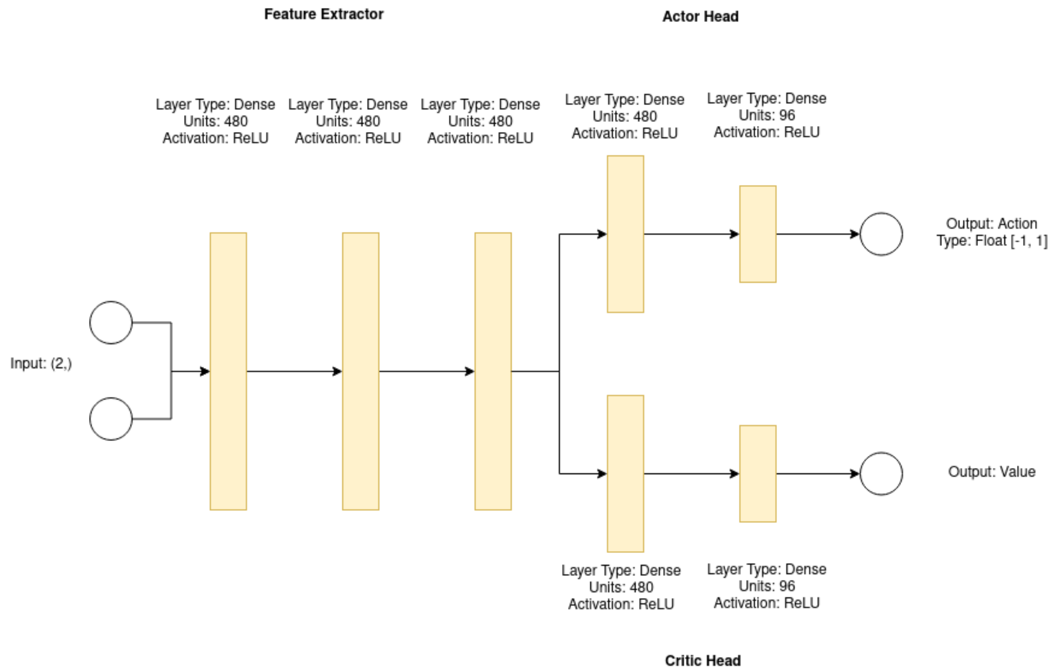


Figure 3.3: The architecture utilized in training with proximal policy optimization for the *Raw-continuous* representation (1,017,506 parameters).

corresponds to the value estimate of the input state representation. The activation function applied to all layers, with the exception of the output layer which does not utilize activation, is ReLU. The same outputs of a single scalar value each for the actor and critic, and the same ReLU activations on all other layers, are utilized across all architectures. Taking the quantity of weights and biases present in this architecture yields a total trainable parameter count of 1,017,506.

In the case of the *RBF (1D)* representation, the architecture utilized can be seen in Figure 3.4. As shown, a single 1-D convolution layer is utilized for feature extraction from the two input vectors of dimension 50. This 1-D convolution layer has 32 filters, a kernel size of 8, and a stride of 4. The output of this operation is then flattened to create a vector of dimension 352, which is then fed into separate actor and critic heads. The architectures of these heads are once again symmetrical, consisting of two dense layers. The first layer of

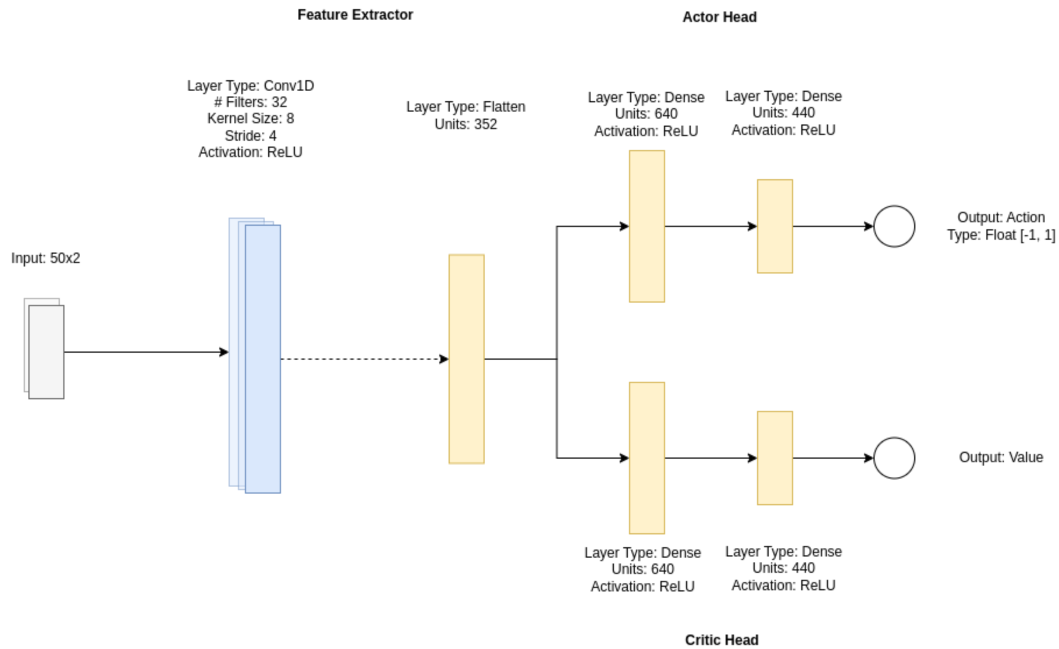


Figure 3.4: The architecture utilized in training with proximal policy optimization for the *RBF (1D)* representation (1,014,306 parameters).

640 units is followed by a second of 440 units. The calculation of the number of trainable parameters in this architecture yields a result of 1,014,306.

The next architecture utilized can be seen in Figure 3.5. This architecture is the most commonly utilized in our experiments, as it is used in training with *One-hot*, *RBF (2D)* and *Human-img* representations. To extract features from the input images, a single 2-D convolution layer is present containing 32 filters, a kernel size of 8, and a stride of 4. The result of this operation is flattened to create a vector of dimension 3872, which is then processed by the actor and critic heads separately. These symmetrical heads contain a dense layer of 128 units, followed by another dense layer of 64 units. The result of calculating the number of trainable parameters in this architecture differs slightly depending on the representation utilized, since the number of channels in the input varies. For the single-channel *RBF (2D)* and *One-hot* representations, there are 1,010,210 trainable parameters, while for the three-channel *Human-img* representation, there are 1,014,306. The architectures used for both single-channel and three-channel images are identical aside from

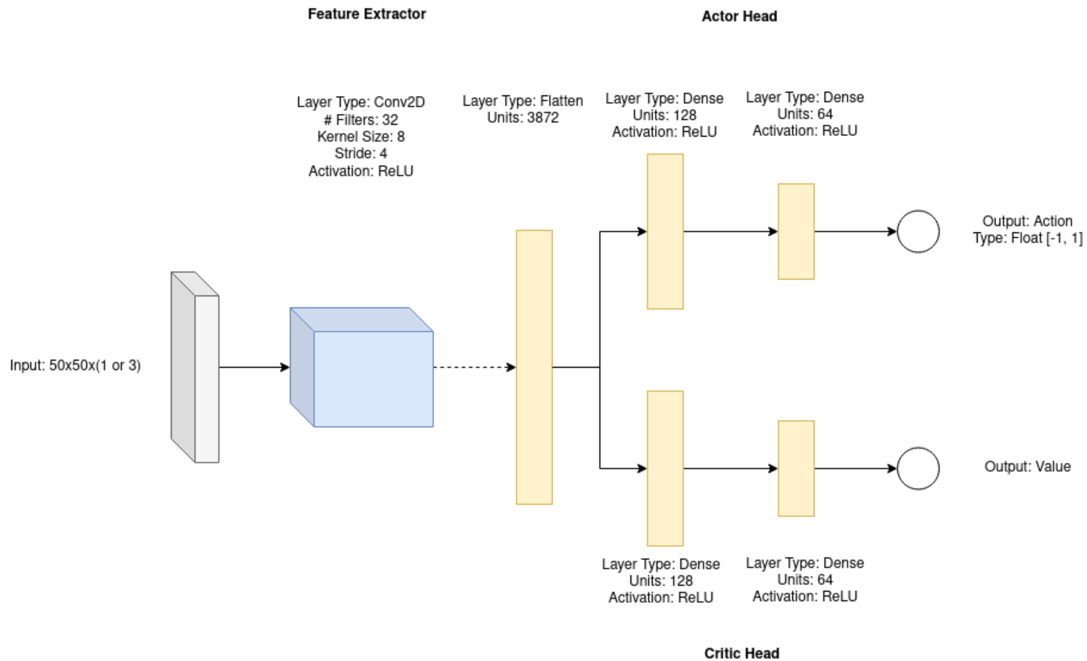


Figure 3.5: The architecture utilized in training with proximal policy optimization for the image-based representations of *RBF (2D)* (1,010,210 parameters), *One-hot* (1,010,210 parameters), and *Human-img* (1,014,306 parameters).

the difference in number of input channels.

The final architecture utilized in our experiments can be seen in Figure 3.6. Rather than sharing a feature extractor between the actor and critic, this architecture completely separates the two components of the algorithm into their own neural network architectures. The actor and critic are entirely symmetrical, with the exception of the input layer, since this architecture is utilized with *Hybrid* representations. The actor receives the *Human-img* representation, a three-channel image, while the critic receives the *RBF (2D)* representation, a single-channel image. Following this input is a single 2-D convolution layer with 32 filters, a kernel size of 8, and a stride of 4. The result of this operation is then flattened into a vector of dimension 3872, which is then passed through two dense layers prior to the output layer. The first of these layers contains 128 units, while the second contains 64 units. In total, this architecture contains 1,016,386 trainable parameters, taking into account

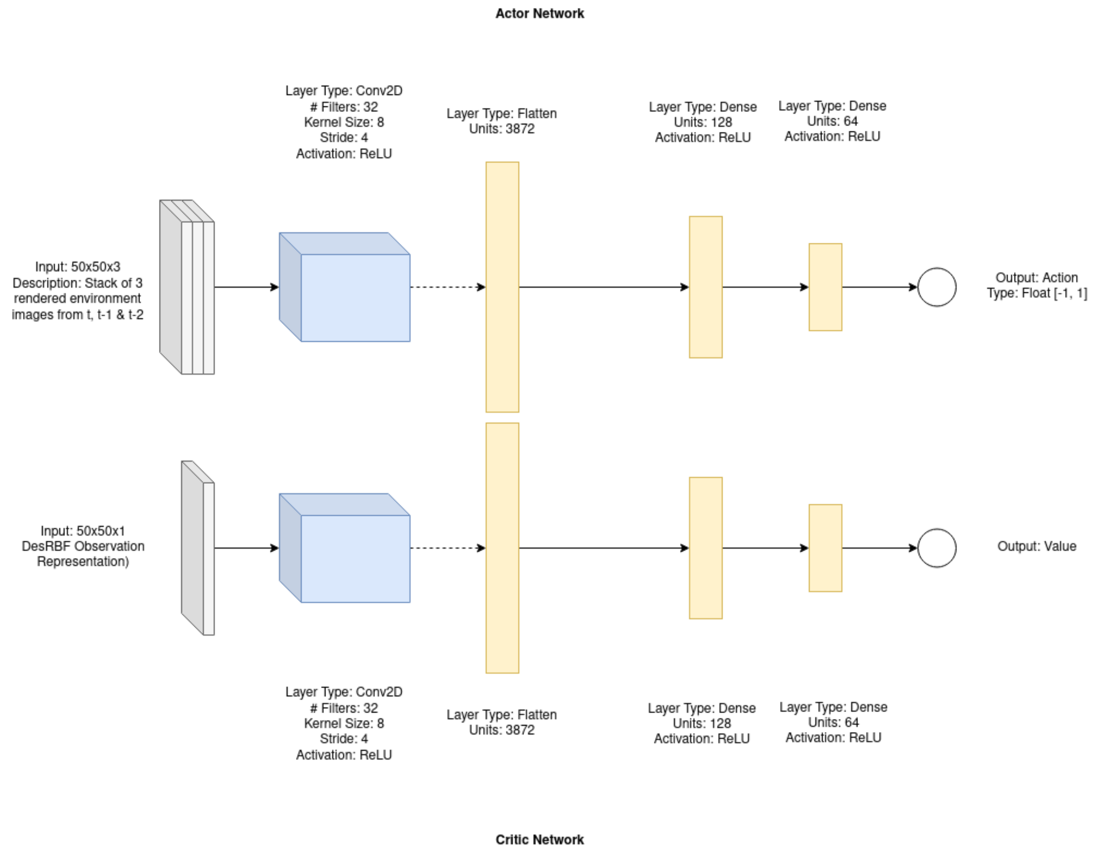


Figure 3.6: The architecture utilized in training with proximal policy optimization for the *Hybrid* representation (1,016,386 parameters) where the actor receives the *Human-img* representation and the critic receives the *RBF (2D)* representation.

both the actor and critic networks.

Table 3.2: Neural Network Architectures Utilized with MountainCar Experiments. The count of the number of layers includes convolutional, dense, and the output layer, but excludes the input layer.

Model	**Input Type**	**Input Shape**	**# of Layers**	**# of Parameters**
Raw-continuous	Dense	(2,)	6 (3 shared by actor & critic)	1,017,506
RBF (1D)	Vector	(2,50)	4 (1 shared by actor & critic)	1,014,306
RBF (2D) & One-hot	Image	(1,50,50)	4 (1 shared by actor & critic)	1,010,210
Human-img	Image	(3,50,50)	4 (1 shared by actor & critic)	1,014,306
Hybrid	Image	Actor: (3,50,50) Critic: (1,50,50)	4 (0 shared by actor & critic)	1,016,386

Training Configuration

Our agents were trained until total rollout timesteps exceeds 300,000. Due to our use of 2048 timesteps in each rollout, this equates to a total of 301,056 timesteps of training. We utilize PPO [71] for 10 training runs per agent configuration. It should be noted that we negate the terms in the surrogate objective function detailed in Equation 2.13, and perform stochastic gradient descent on the resulting surrogate objective function. This approach is taken in all subsequent experiments in this work, and should be noted when observing figures of loss curves. The average episode length varies through the training process from a minimum of approximately 66 timesteps when the agent is consistently solving the task, to a maximum of 1000 when the agent is unable to solve the task. On this basis, the 301,056 timesteps of training equates to between 500-600 episodes of training. The Adam optimizer is utilized for parameter updates utilizing a learning rate of 3×10^{-4}, and the remainder of hyperparameters set to default PyTorch values (notably, $\beta_1 = 0.9$,

$\beta_2 = 0.999$). Other important hyperparameter selections are detailed in Table 3.3.

Table 3.3: Hyperparameters used with PPO for the MountainCar experiments

Hyperparameter	Value
Learning Rate	3×10^{-4}
Rollout Timesteps, T	2048
Total Timesteps	301056
Minibatch Size, M	64
Iterations, N	147
Epochs, K	10
Discount Factor, γ	0.99
GAE Lambda, λ	0.95
Value Loss Coefficient, c_1	0.5
Entropy Loss Coefficient, c_2	0
Clipping Ratio, ε	0.2

3.2.3 Results & Discussion

The mean episodic reward achieved by agents in the 10 training runs, for each experiment configuration, are shown in Figures 3.7. The asymptotic reward and convergence characteristic results are further summarized in Table 3.12. In the case of the *Raw-continuous* representation, 7 out of 10 training results produced agents capable of consistently solving the task, while 3 training runs resulted in agents that learned to take the *do-nothing* action consistently to avoid the penalty for selecting the other actions, as shown in Figure 3.7. This results in an agent that remains in the trough of the sinusoidal terrain. Of all representations from which agents were able to solve the task, the *Raw-continuous* representation ranks 4th in convergence success rate, and was the slowest to converge, taking 238,629 timesteps on average.

Looking next at the *RBF (1D)* results in Figure 3.8, it can be seen that a higher success rate is achieved, with 8 of 10 agents learning to consistently reach the objective location, which ranks the representation at a tied 3rd in success rate with *Hybrid*. It also has the 2nd fastest mean timesteps to converge, at 160,400.

RBF (2D) proves to be the highest performing representation, ranking 1st in both success rate at 100%, and mean timesteps to converge at 122,240, as demonstrated in Figure 3.9.

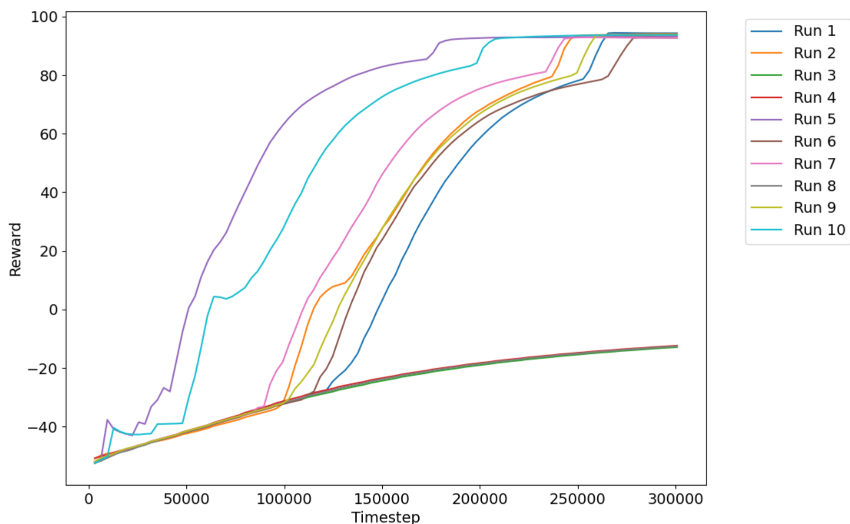


Figure 3.7: The mean episodic reward achieved by our agents in 10 independent training runs of 301,056 timesteps each, when utilizing a *Raw-continuous* representation.

The results for the *One-hot* representation, as seen in Figure 3.10, show just 6 of the 10 agents learning to consistently reach the objective location, ranking the representation 5th on this metric. The mean timesteps to converge over these 6 successful runs was calculated at 203,733, ranking this representation 4th on this metric.

The last representation utilized is the *Human-img*, which yielded the worst results of all those tested, as can be seen in Figure 3.11. None of the agents were able to successfully learn to reach the objective, and instead, all agents learned to consistently select the *do-nothing* action.

The preceding results guided our choice of representations to utilize in the *Hybrid* representation. Since the clear highest performer on both convergence metrics was shown to be the *RBF (2D)* representation, and the clear worst-performing agent on these same metrics was shown to be *Human-img*, we opted to utilize these two representations for our *Hybrid* configuration. Since only the actor network is required at inference time, and since an effective actor network could not be produced when utilizing the *Human-img* representation alone, we configured the *Hybrid* representation to utilize a *Human-img* representation as input to the actor network, and *RBF (2D)* as input to the

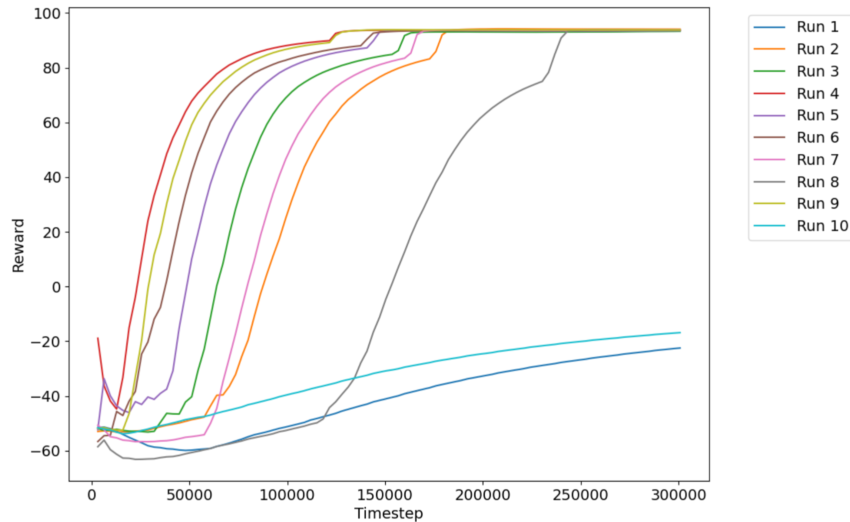


Figure 3.8: The mean episodic reward achieved by our agents in 10 independent training runs of 301,056 timesteps each, when utilizing a *RBF (1D)* representation.

critic network. As shown in Figure 3.12, this configuration is tied for 3rd in convergence success rate, and is 3rd in mean timesteps to converge, showing that one can generate an actor network capable of inferring on *Human-img* representations by utilizing the high-performing *RBF (2D)* representation as input to the critic network during training.

Utilizing the *Hybrid* architecture would generally entail smaller actor and critic network sizes when viewed in isolation if many parameters were shared between the networks. In our case, however, only the first convolution layer was shared between the actor and critic in the image-based architectures utilizing *RBF (2D)*, *One-hot*, and *Human-img* representations. This layer contains just 6,176 parameters in the *Human-img* architecture, and 2,080 parameters in the *RBF (2D)* architecture, so the separation of the actor and critic networks in the *Hybrid* architecture does not require downsizing of the actor and critic network size to keep the total parameter count in alignment across architectures.

For the final reward result groups, we run a one-way ANOVA test to assess the statistical significance of the differences in the mean rewards of each configuration. The ANOVA found a statistically significant difference in reward

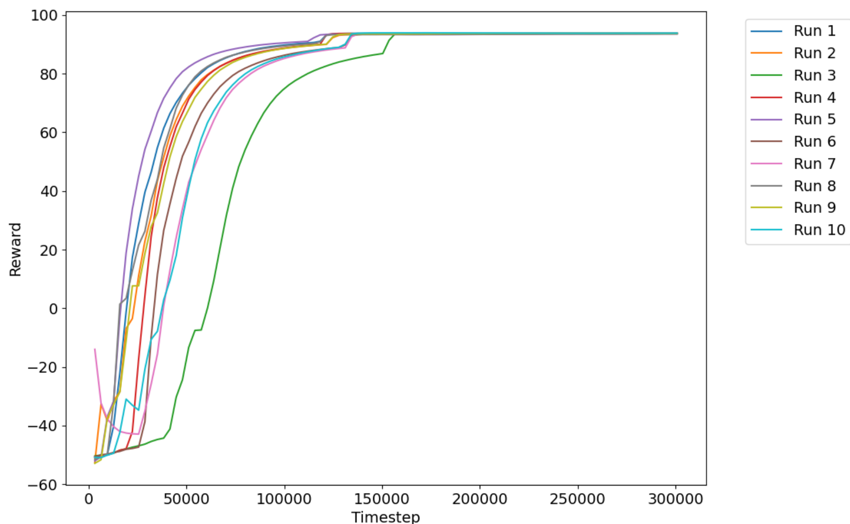


Figure 3.9: The mean episodic reward achieved by our agents in 10 independent training runs of 301,056 timesteps each, when utilizing a *RBF (2D)* representation.

between at least two groups ($F(4, 45) = 8.0241$, $p < 0.001$). Additionally, we ran a post-hoc Tukey's Honestly Significant Difference (HSD) test, the results of which are shown in Table 3.5. In the conducted Tukey's HSD test, statistically significant differences in mean reward were observed between all pairings involving the *Human-img* representation. *Human-img* is found to be the worst performing representation on this basis.

While we made an attempt to make a fair comparison between various representations and corresponding neural network architectures by balancing the parameter count between them, there are many other design decisions that could have played a role. A limitation can be seen in the fact that layers are of different dimension throughout the architectures, and in the case of the architecture utilized with *Raw-continuous*, contain different numbers of layers. These design decisions ultimately could have an impact on relative agent performance.

To summarize, we have demonstrated that *RBF (2D)* coupled with convolutional neural networks to parameterize the actor and critic is by far the best performing architecture, in terms of the the success rate and mean asymptotic reward, along with the rate of convergence. While the literature lacks

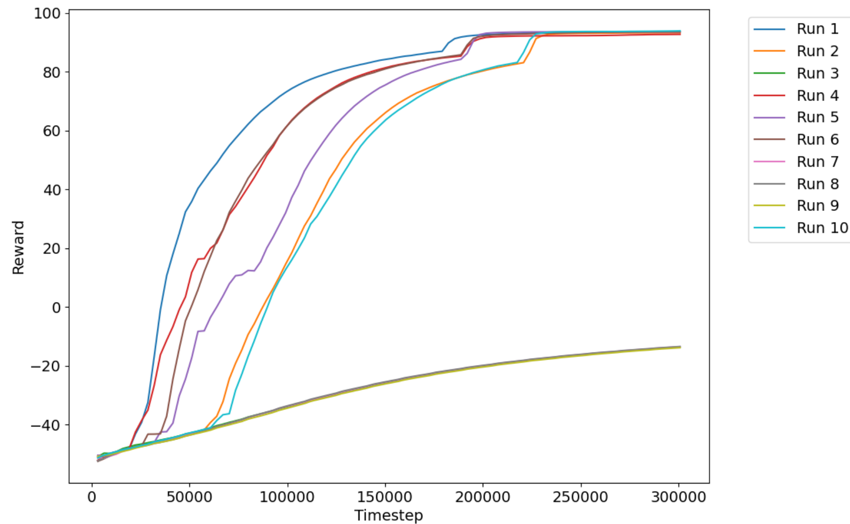


Figure 3.10: The mean episodic reward achieved by our agents in 10 independent training runs of 301,056 timesteps each, when utilizing a *One-hot* representation.

direct benchmarks for our specific training architecture and observation encoding, the performance of RBF-based methods in our study aligns with the documented success of applying RBF-based methods to the MountainCar environment [45]. Overall, the results of the symmetric architectures indicates that higher precision observations yield more performant agents. Additionally, we have demonstrated the merit of utilizing high-performing, high-precision observations as input to the critic, to train actor networks on low-performing, low-precision representations, as can be seen in our *Hybrid* training runs. These findings will be utilized to guide the formulation of our subsequent experiments.

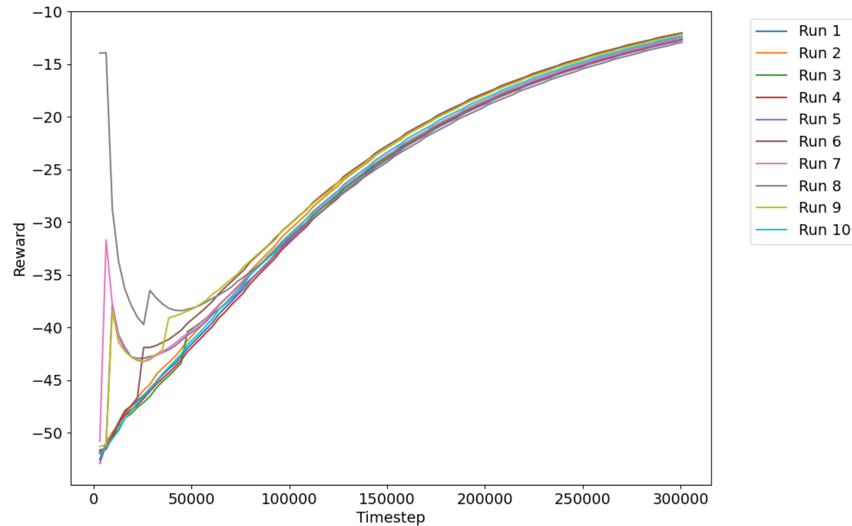


Figure 3.11: The mean episodic reward achieved by our agents in 10 independent training runs of 301,056 timesteps each, when utilizing a *Human-img* representation.

Table 3.4: The mean and standard deviation of the asymptotic performance, and the convergence characteristics, for each representation configuration over 10 training runs. Configurations are ordered by mean asymptotic reward. Success is defined as an agent that consistently reaches the goal position, and mean timesteps to converge is determined by averaging the timesteps of the first datapoints exceeding 93 units of reward from each successful training run.

Configuration	**Mean Final Reward**	**Convergence**		**Input Precision**	
		Mean Timesteps	**Success Rate**	$Prec_{x_{pos}}$	$Prec_{x_{vel}}$
RBF (2D)	93.76 ± 0.08	122,240	100%	5.56×10^{-14}%	7.14×10^{-13}%
Hybrid	72.02 ± 42.25	195,600	80%	–	–
RBF (1D)	71.02 ± 45.36	160,400	80%	5.56×10^{-14}%	7.14×10^{-13}%
Raw-continuous	61.68 ± 48.62	238,629	70%	5.56×10^{-14}%	7.14×10^{-13}%
One-hot	50.62 ± 52.44	203,733	60%	2.04%	2.04%
Human-img	-12.45 ± 0.28	N/A	0%	2.04%	4.08%

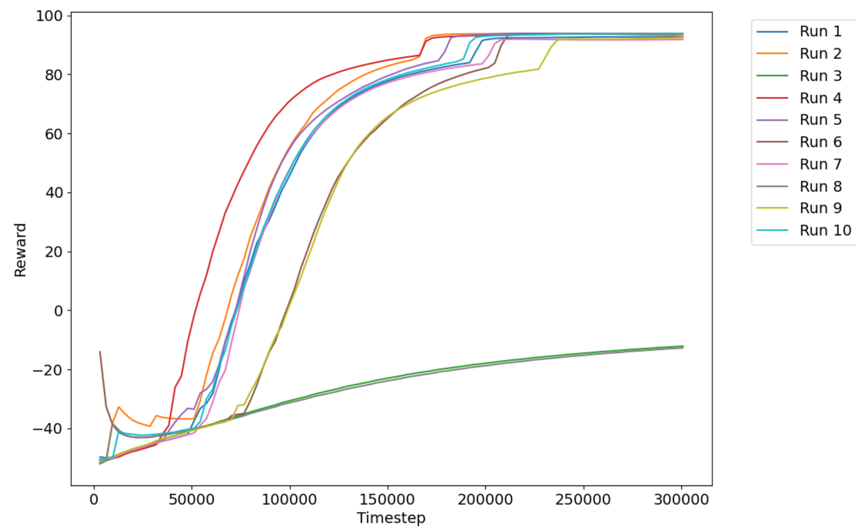


Figure 3.12: The mean episodic reward achieved by our agents in 10 independent training runs of 301,056 timesteps each, when utilizing a *Hybrid* representation.

Table 3.5: Tukey's HSD Test Results comparing mean rewards between groups. The columns represent the following: 'Configuration Pair' are the two configurations being compared; 'Mean Diff.' is the difference in mean reward between the two groups; 'p-adj' is the adjusted p-value; 'Lower' and 'Upper' are the lower and upper bounds of the 95% confidence interval, respectively; 'Reject' indicates whether the null hypothesis of no difference in means can be rejected (True) or not (False).

Configuration Pair	**Mean Diff.**	**p-adj**	**Lower**	**Upper**	**Reject**
Human-img, One-hot	63.0685	0.0128	9.2572	116.8797	True
Human-img, RBF (2D)	106.2105	< 0.001	52.3992	160.0218	True
Human-img, RBF (1D)	83.472	0.0004	29.6607	137.2833	True
Human-img, Hybrid	84.4655	0.0003	30.6542	138.2767	True
Human-img, Raw-continuous	74.1316	0.002	20.3203	127.9429	True
Hybrid, Raw-continuous	-10.3339	0.9927	-64.1452	43.4774	False
Hybrid, One-hot	-21.397	0.8468	-75.2083	32.4143	False
Hybrid, RBF (2D)	21.745	0.8378	-32.0662	75.5563	False
Hybrid, RBF (1D)	-0.9935	> 0.999	-54.8048	52.8178	False
Raw-continuous, One-hot	-11.0631	0.9901	-64.8744	42.7481	False
Raw-continuous, RBF (2D)	32.0789	0.4987	-21.7324	85.8902	False
Raw-continuous, RBF (1D)	9.3404	0.9955	-44.4709	63.1517	False
One-hot, RBF (2D)	43.1421	0.1857	-10.6692	96.9533	False
One-hot, RBF (1D)	20.4035	0.8709	-33.4078	74.2148	False
RBF (2D), RBF (1D)	-22.7385	0.8109	-76.5498	31.0728	False

3.3 Multi-agent Particle Environment

3.3.1 Problem Definition

Experiment Background

We now wish to expand our domain of experimentation to include multi-agent environments. The primary focus of this section will be to compare two popular multi-agent paradigms. Specifically, we will compare the two configurations of training with centralized training - CTDE and CTCE. We focus on these two paradigms as they both present potential advantages.

In both the CTCE and CTDE paradigms, a centralized training scheme is utilized, due to the reported superiority of such methods [31] over their distributed counterparts. In the centralized training scheme implemented in this work, a centralized critic network in the actor-critic architecture predicts the value of all agents' states. The sum of the critic network's output for each of the agents' observations is utilized in calculating the value loss.

An additional advantage that led us to focus on these paradigms is that the CTDE paradigm has been reported as SOTA for learning in environments with multiple agents [43, 55, 29], making it an obvious choice for our own experiments.

However, due to the fact that the CTDE paradigm allows for agent-specific parameters in the actor networks, either through multiple actor heads or entirely separate actor networks for each agent, the paradigm allows for specialized policies to be learned for each agent. This does not allow for new agents to be introduced to the environment while maintaining performance, since this would require the instantiation of new parameters for each additional agent.

The benefit of the CTCE paradigm that we believe warranted its inclusion in our experiments is that the same actor parameters control all agents in the environment. While the learned policy may still be conditioned on the number of agents during training, we believe it to be worth experimenting with this paradigm due to its potential generalizability to changing numbers of agents without the need for new parameters to be instantiated for each additional agent.

Our findings from Section 3.2 indicate that higher precision representations, in particular, the *RBF (2D)* and *RBF (1D)* representations, have the fastest rate of convergence to their asymptotic performance levels. An objective of our next set of experiments is to see whether the findings from the very low-dimensional, single-agent MountainCar environment extend to a higher dimensional, multi-agent setting. These experiments will focus on the perfor-

mance of the top two fastest converging representations in MountainCar, *RBF (2D)* and *RBF (1D)*, compared against the *Raw-continuous* representation.

Finally, in these experiments, we want to know the impact of learning rate on agent performance. In the MountainCar experiment, we utilized a constant learning rate throughout all experiments. Now, we wish to run the experiments with a variety of learning rates, to see if any relationships can be derived between learning rate and agent performance, taking the other experiment configuration parameters into account.

The presence of multiple agents in the environment leads us to an increased dimensionality in the possible configurations of our experiments when compared to the single-agent MountainCar problem. The purpose of this next set of experiments is to move the realm of experimentation into a multi-agent domain, with the ultimate objective of applying these lessons in our multi-agent missile environment in Chapter 4. To this end, we define a selection of experiment configurations to help in understanding the impact of varying certain key experiment configuration parameters on final agent performance.

Ultimately, this section aims to answer three questions. Firstly, we wish to know the impact on agent performance of using a CTDE, when compared to a CTCE paradigm. Secondly, we wish to see the impact of representation on agent performance. Lastly, we wish to run the experiments with a range of learning rates, to see how the variation of this hyperparameter impacts our final agent performances.

Environment Background

An environment that has seen significant attention in literature due to its adaptability and simplicity is OpenAI's MPE [47, 52]. This environment provides a framework for experimenting with both competitive and collaborative multi-agent scenarios in a simple, computationally-efficient, 2-dimensional setting.

The environment consists of a selection of particles in a 2-dimensional plane. These particles, or environment entities, can either be agents or landmarks. Agents in the environment are controlled by a policy, whereas landmarks are static entities. A selection of scenarios have been created in this environment, where the key determinant in a scenario is the configuration of agents and landmarks, the presence or absence of communication channels between agents, and the corresponding reward function for the given scenario.

While a variety of scenarios exist, the scenario of focus in our work is that of the purely cooperative formation task, without communication channels existing between agents. In this formation task, N agents are initialized in the

environment, along with a single landmark. An example of a formation with 3 agents can be seen in Figure 3.13.

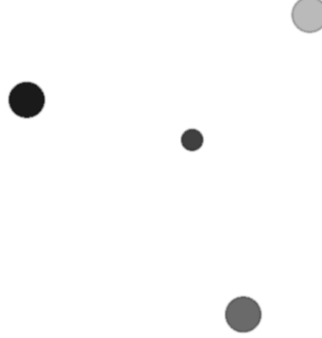


Figure 3.13: The final positions of agents around the landmark when following an effective formation policy. The agents are shown to be approximately equidistant from the landmark, and are separated by approximately $\frac{2\pi}{3}$ radians.

In this scenario, the collaborating agents are assigned a single, collective reward. This reward is calculated by first calculating the relative positions of each agent with respect to the landmark, along with their corresponding angles relative to the horizontal line passing through the landmark. The agent with the minimum calculated angle is taken as the anchor, and then optimal locations for the other agents are calculated based on the anchor location and the target radius of the formation. From these optimal locations, along with the actual positions of the agents, a Hungarian matching algorithm is utilized to find the pairing of optimal locations and agents that minimizes the distance between each paired location and agent. The mean distance found in this pairing is then clipped to be within the range of 0 and 2, and the negative of this value is assigned as reward to the agents.

The frame of reference for the *Raw-continuous* observation space for this environment is adjusted according to each agent, and is calculated based on the individual agents' locations. It is of dimension 10 for an environment containing 3 agents. The first two entries, at index locations 0 and 1, are the x and y components of the agent's velocity, taking values in the range [-1,1]. The next values, at index locations 2 and 3, are the x-position and y-position of the agent in a global coordinate frame, in the range [-1,1]. Index positions

4 and 5 then contain the landmark x and y positions, in the given agent's coordinate frame, in the range [-2, 2]. The remainder of the observation, which in the case of a 3-agent scenario consists of index locations 6-9, contain the x and y coordinates of the other agents in the environment, in the subject agent's coordinate frame, in the range [-2,2].

3.3.2 Method

State Space Representation

In addition to the *Raw-continuous* observation space of dimension 10 described in Section 3.3.1, we implement *RBF (2D)* and *RBF (1D)* transformations of the default observation space.

The *RBF (2D)* is a 50x50 image, where the *Raw-continuous* representation of dimension 10 is condensed into a single agent-centered frame of reference. The representation is additionally divided into two unique formats, yielding final observation dimensions of either 50x50x4 or 50x50x3, depending on the format utilized. In the first format of 50x50x4, the information pertaining to the two other agents in the environment, that are not observers of the given observation, exists in two separate channels. In the second format, this information is condensed into a single channel where the pixel values of the single channel are determined by taking the maximum pixel value of the two channels.

Each channel in this image encodes coordinate variables from the *Raw-continuous* observation representation, in a similar manner to the position and velocity *RBF (2D)* representation of the MountainCar's state from Equation 3.8 in Chapter 3.2, however in an x and y coordinate plane. We additionally represent velocity in an x and y coordinate plane by employing a similar blur methodology as detailed for the *RBF (1D)* velocity representation in Chapter 3.2 Equations 3.4-3.6, but are additionally extended to include y-velocity, as detailed in Equations 3.19-3.24.

$$\hat{y}_{pos,t+1} = y_{pos,t} + y_{vel,t} \tag{3.19}$$

$$\mu_{vel,y} = \frac{y_{pos,t} + \hat{y}_{pos,t+1}}{2} = y_{pos,t} + \frac{y_{vel,t}}{2} \tag{3.20}$$

$$\sigma_{vel,y} = (1 + |\hat{y}_{pos,t+1} - y_{pos,t}|) \times \sigma_{pos,y} \tag{3.21}$$

$$\hat{x}_{pos,t+1} = x_{pos,t} + x_{vel,t} \tag{3.22}$$

$$\mu_{vel,x} = \frac{x_{pos,t} + \hat{x}_{pos,t+1}}{2} = x_{pos,t} + \frac{x_{vel,t}}{2} \tag{3.23}$$

$$\sigma_{vel,x} = (1 + |\hat{x}_{pos,t+1} - x_{pos,t}|) \times \sigma_{pos,x} \tag{3.24}$$

As in MountainCar experiments, we utilize a value of σ_{pos} equal to 1/16th the range of the variable. In this case, since we are shifting to an agent-centered coordinate frame, the [-1, 1] global coordinate frame no longer applies. Instead, we have a [-2, 2] coordinate frame, yielding a σ_{pos} of 0.25 in both the x and y direction. Utilizing the results of Equations 3.19-3.24, along with our x_{pos} and y_{pos} we are then able to calculate our pixel values in our image using Equations 3.25 and 3.26.

$$\phi_{pos,RBF2D}(c_i, c_j) = (\frac{1}{\sigma_{pos,x}\sqrt{2\pi}} e^{\frac{-(x_{pos}-c_i)^2}{2\sigma_{pos,x}^2}}) \times (\frac{1}{\sigma_{pos,y}\sqrt{2\pi}} e^{\frac{-(y_{pos}-c_j)^2}{2\sigma_{pos,y}^2}}) \tag{3.25}$$

$$\phi_{vel,RBF2D}(c_i, c_j) = (\frac{1}{\sigma_{vel,x}\sqrt{2\pi}} e^{\frac{-(\mu_{vel,x}-c_i)^2}{2\sigma_{vel,x}^2}}) \times (\frac{1}{\sigma_{vel,y}\sqrt{2\pi}} e^{\frac{-(\mu_{vel,y}-c_j)^2}{2\sigma_{vel,y}^2}}) \tag{3.26}$$

A sample of the representation where the other agent information has been merged into a single channel can be seen in Figure 3.14, where *Channel 0* utilizes Equation 3.26 in its generation, while *Channel 1* and *Channel 2* utilize Equation 3.25.

We make one important modification to the default environment to ensure the varying representations are compared in a fair manner. In the default environment, the agents are able to move anywhere in the 2-dimensional plane. However, in our implementation, the x and y positions of each agent are clipped to ensure they remain within the range [-1, 1] of the global coordinate frame. While the default *Raw-continuous* representation can capture the location of the agents accurately outside of this clipped range, the RBF representations require a defined range be utilized in their creation. Should the agents move outside of this range, the RBF would contain near zero values, leading to poorer sample efficiency when utilizing these representations.

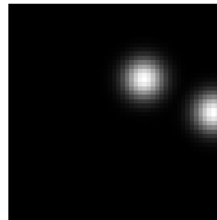
***Raw-continuous* Observation**

-0.21875, 0, -0.66792509, -0.49805831, 1.06178903, 0.97781595, 0.83366437, 0.48666542, 1.16444368, 0.48517012

***RBF (2D)* Observation**

Channel 0: x-velocity and y-velocity

Channel 1: Landmark x-position and y-position



Channel 2: Other agents' x-position and y-position

Render ***RBF (1D)* Observation**

Figure 3.14: A selection of state representations of the MPE environment. The default *Raw-continuous* state representation is shown at the top. *RBF (2D)* is a 50x50 3-channel image. *RBF (1D)* consists of 10 vectors of length 50, with each vector representing a dimension of the *Raw-continuous* representation, in an x and y frame of reference. The RBF-based figures are with respect to the darkest agent in the *Render* image, and are transformed to be in an agent-centered coordinate frame.

Table 3.6: The reference figures and parameter counts for the neural network utilized in each configuration. Note that configurations 2, 6 and 9 refer to figures in Section 3.2. For these configurations, the architectures utilized in this section differ from those referenced only in the input dimension, which is now 10 instead of 2.

Configuration	Figure	Parameter Count
0	3.15	1,024,838
1	3.20	1,029,582
2	3.5	1,016,614
3	3.17	1,004,910
4	3.15	1,014,566
5	3.20	1,021,390
6	3.5	1,014,566
7	3.17	1,002,862
8	3.16	1,018,666
9	3.3	1,021,734
10	3.19	1,011,120
11	3.18	1,056,124
12	3.21	1,023,180
13	3.22	1,033,704

Neural Network Architectures

In our experiments, a total of 14 configurations are trained, with 10 distinct neural network architectures utilized across these experiments. The neural network figure references for each configuration can be seen in Table 3.6. This table also demonstrates the alignment of parameter count to approximately 1M across all configurations. The variations in neural network architecture across experiments are centered around three distinct areas. Firstly, different architectures are utilized for CTDE as compared to CTCE configurations. Secondly, certain configurations utilize a shared feature extractor between actor and critic networks, while others have entirely independent actor and critic networks. Thirdly, since configurations use different input representations to the actor and critic networks, the architectures are modified to ensure the feature extraction methodology is suitable for the given input representation.

Training & Execution Paradigm The first defining characteristic of the neural network architectures relates to the training and execution paradigm utilized in the given configuration. In configurations 0, 2, 4, 6, 8 and 9 a CTCE

paradigm is utilized, while in all other configurations, a CTDE paradigm is utilized. In the CTDE configurations, each agent in the environment has parameters in their actor network that are only responsible for learning the given agent's policy. In the CTCE configurations, on the other hand, all parameters are shared between all agents. Due to this difference, the total parameter count for each individual actor network is decreased in CTDE experiments when compared to their CTCE network counterparts, to ensure alignment in total parameter count across configurations.

Shared Feature Extractor The second defining characteristic of the neural network architectures utilized in these experiments can be seen in whether they share a feature extractor between actor and critic networks. This design decision is tested in both CTCE and CTDE training paradigms. Examples of shared feature extraction architectures can be seen in Figures 3.17, 3.18, 3.3 and 3.5. Examples without shared feature extraction, where the actor and critic networks are entirely separated, can be seen in Figures 3.15, 3.16, 3.19 and 3.20.

Actor & Critic Inputs The third and final contributing factor in the neural network architecture designs utilized relates to the input representation for the actor and critic networks. These can individually be one of three options: *RBF (2D)*, *Raw-continuous*, or *RBF (1D)*. Since we saw success utilizing an asymmetric architecture with the *Hybrid* representation in the MountainCar experiments, we additionally choose to experiment with one such architecture in the configuration 12 experiment, where we utilize a *Raw-continuous* representation as input to the policy network, and a *RBF (2D)* representation as input to the critic. This specific combination was selected for the asymmetric experiment due to the success observed in the MountainCar experiments when utilizing an *RBF (2D)* representation for the critic's input, along with the success more generally seen in literature when utilizing radial basis functions with value-based methods. A final configuration utilizing *RBF (1D)* input representations is defined, in configuration 13, to confirm whether the *Raw-continuous* and *RBF (2D)* outperformance holds in this new environment.

Considering the variation and combination of each of these three determinants of neural network design, along with the *Merged Other Agents* representation option discussed in the prior section for *RBF (2D)* state representations, we define our 14 experiment configurations. Ultimately, our experiments make use of the distinct configurations detailed in Table 3.8.

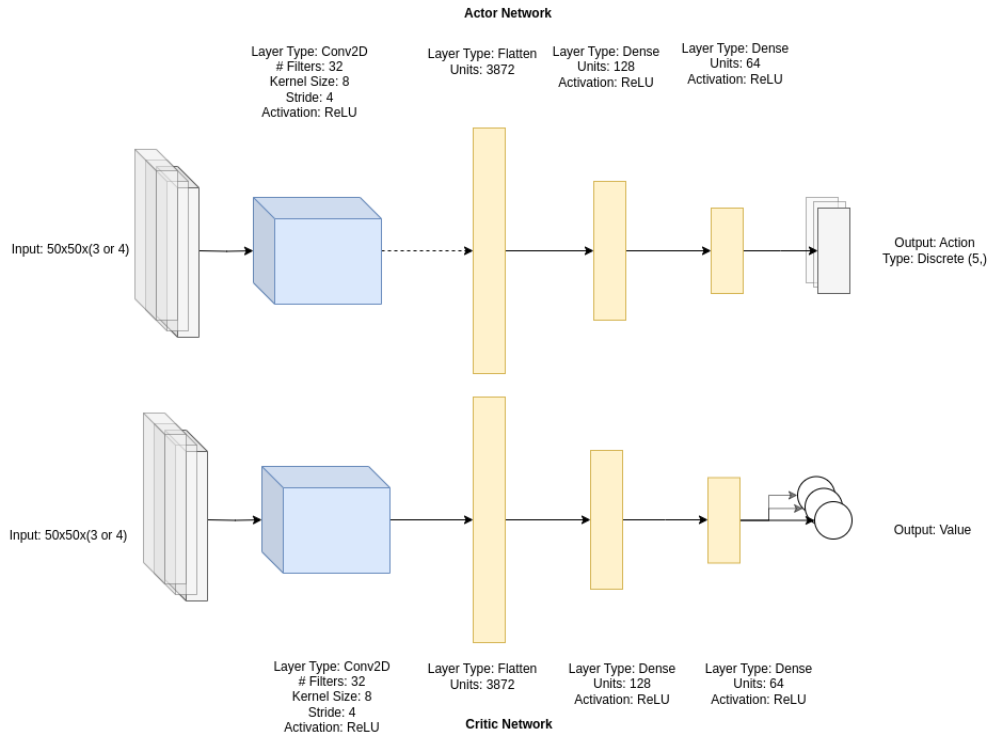


Figure 3.15: The *CTCE* architecture utilized when training on *RBF (2D)* observations, without shared parameters between the actor and critic networks. The total number of trainable parameters in this architecture is 1,024,838.

Training Configuration

Our agents were trained until total rollout timesteps exceeds 300,000. Due to our use of 4800 timesteps in each rollout, this equates to a total of 302,400 timesteps of training. We utilize PPO with a constant learning rate used for each training run, but varied between training runs on a base-10 log-interval between 1×10^{-5} and 1×10^{-1}. In total, 5 agents are trained for each configuration, at 5 different learning rates, for a total of 25 training runs per configuration. We report the mean and standard deviation of the five runs at each learning rate and configuration combination. Hyperparameters, with the exception of learning rate, are aligned across all training runs.

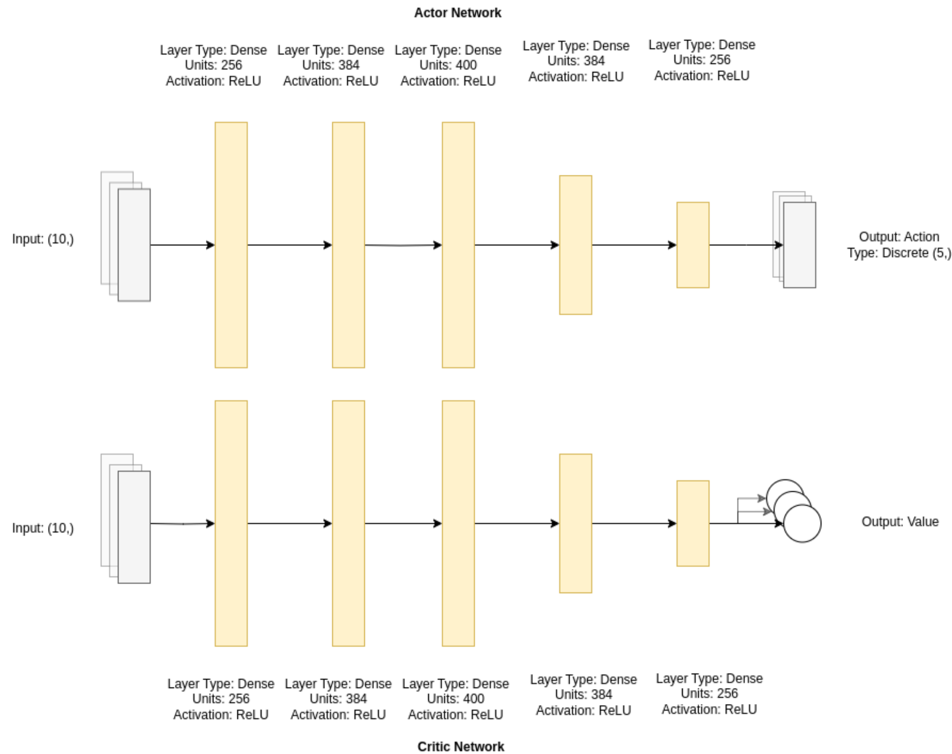


Figure 3.16: The *CTCE* architecture utilized when training on *Raw-continuous* observations, without shared parameters between the actor and critic networks. The total number of trainable parameters in this architecture is 1,014,566.

3.3.3 Results & Discussion

The result of training all 14 configurations with varying learning rates can be seen in Figure 3.23. In the case of our MPE architectures, there is less than a 5% difference between parameter counts, which we deem to be close enough to not have a significant influence on agent performance. For example the best performing agent is one of those with the smallest number of parameters. In general, we did not see any relationship between parameter count and agent performance.

Performance in each configuration can be seen to vary significantly based on the learning rate used during training. We take the best performing agent, based on the final mean episodic reward reached, from each configuration and plot their performances in Figure 3.24. In three of the configurations, 7, 8

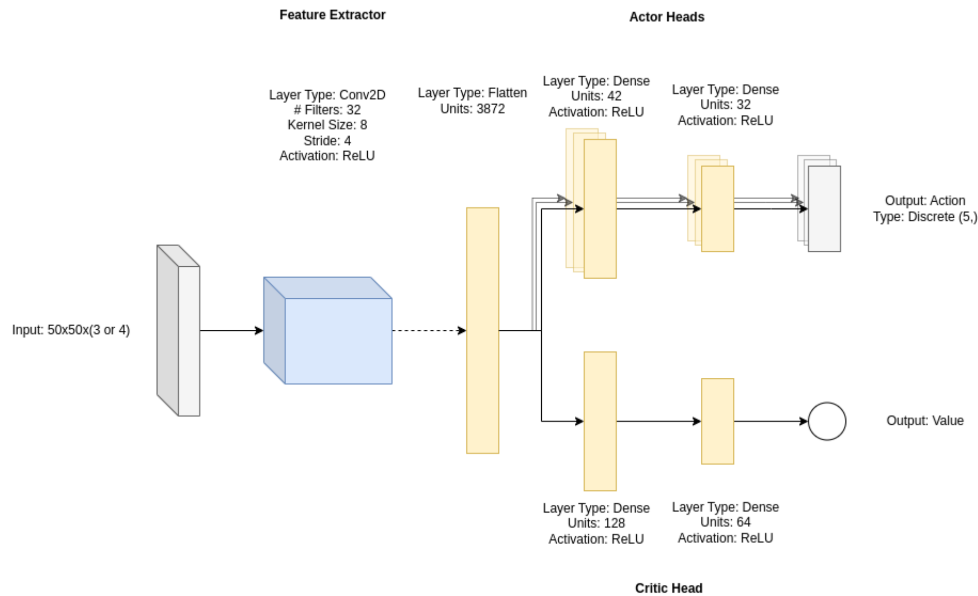


Figure 3.17: The multi-head *CTDE* architecture utilized when training on *RBF (2D)* observations, with shared parameters between the actor and critic networks. The total number of trainable parameters in this architecture is 1,004,910.

and 11, the best performance was seen with a learning rate of 1×10^{-3}. In three configurations, 0, 1 and 3, the highest performance was seen with a learning rate of 1×10^{-5}. Generally, however, a learning rate of 1×10^{-4} was found to produce the highest performing agents, with eight configurations seeing the best agent performance when using this learning rate. For the rest of this section, we consider the highest performing learning rate for each configuration, as detailed in Table 3.9.

For the 14 configuration groups, we conducted a one-way ANOVA test to evaluate the statistical significance of the differences in the means. The test revealed a statistically significant difference between at least two groups ($F(13, 56) = 63.74$, $p < 0.001$). We then run a post-hoc Tukey's HSD test. We omit the complete set of results due to the large number of combinations of configurations, and the fact that certain comparisons are not particularly illuminating, and instead choose to focus on certain key configuration combinations. These results are reported in Tables 3.11-3.13.

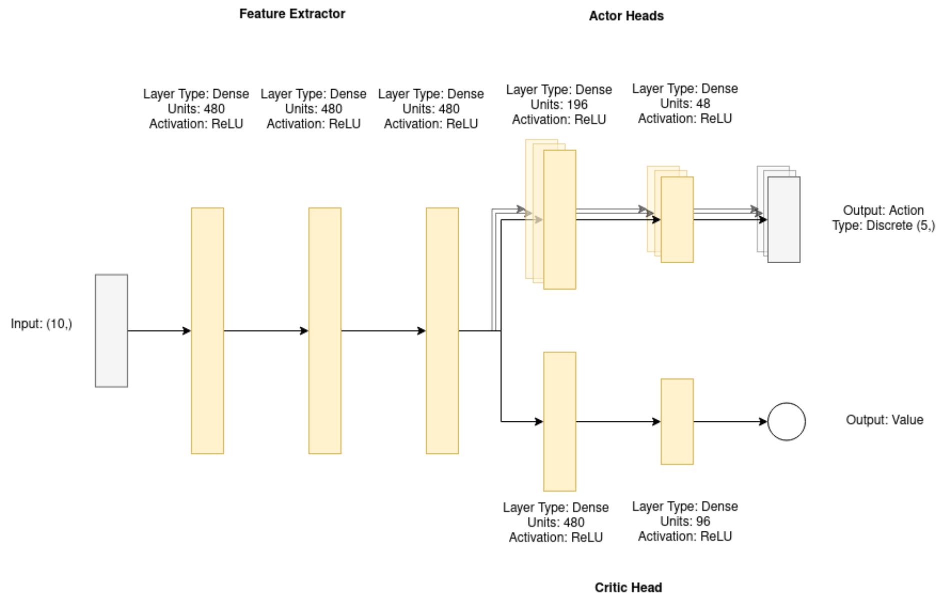


Figure 3.18: The multi-head *CTDE* architecture utilized when training on *Raw-continuous* observations, with shared parameters between the actor and critic networks. The total number of trainable parameters in this architecture is 1,056,124.

Merged Other Agents When looking at the impact of combining the channels that represent the two other agents in the environment into a single channel, no significant performance differences can be derived, as demonstrated by the low relative change in performance with this changing variable seen in Table 3.11, and the Tukey's HSD tests yielding no statistically significant differences in any of the pairings.

Training & Execution Paradigm Looking next at the impact of using CTDE as opposed to CTCE, we see significant differences, as detailed in Table 3.12. Across all experiment configurations, significantly higher performance is found when utilizing a CTDE paradigm. The Tukey's HSD tests yield statistically significant differences in all of the configuration pairings. This demonstrates that, in the MPE environment, having parameters in each actor network that are able to learn a policy specific to one agent in the environment yields more performant policies than when utilizing the same policy parameters for all agents in the environment. Our findings in this regard align with those from literature noting the performance of CTDE [43, 55, 29], in

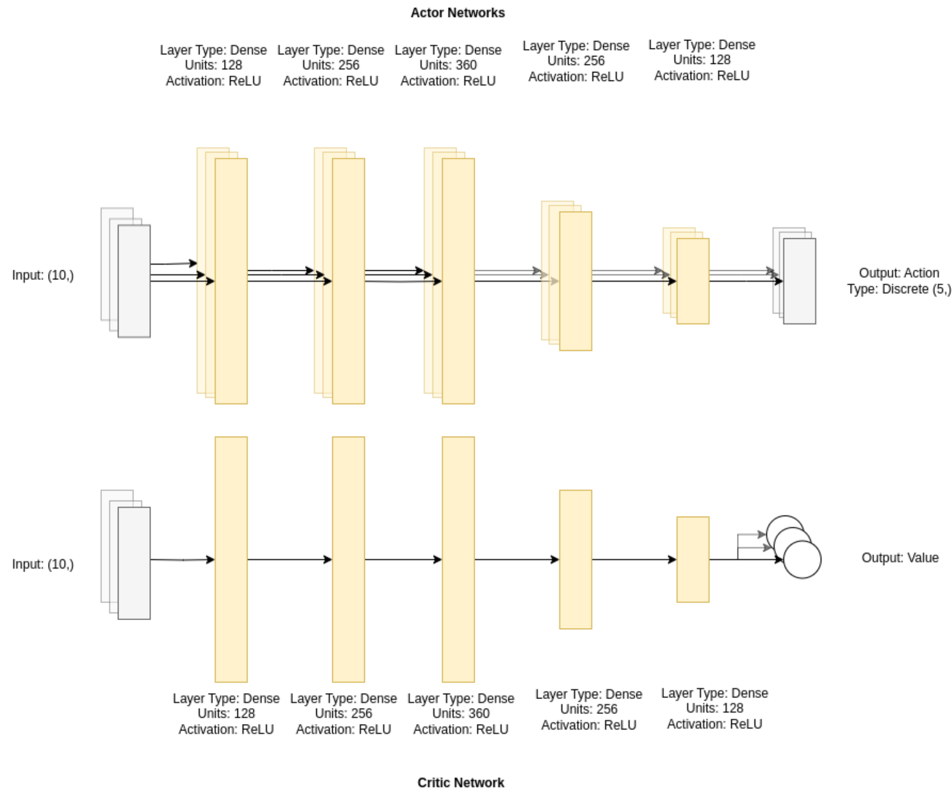


Figure 3.19: The *CTDE* architecture utilized when training on *Raw-continuous* observations, without shared parameters between the actor and critic networks. The total number of trainable parameters in this architecture is 1,011,120.

particular in the MPE domain [47].

Shared Feature Extractor The impact of utilizing a shared feature extractor between actor and critic networks is found to be minimal in most cases, but varies significantly. Based on the results shown in Table 3.13, there seems to be a slight improvement in performance when utilizing a shared feature extractor with *RBF (2D)* representations when comparing to an equivalent sized network with separate actor and critic networks. However, there seems a significant degradation in performance when utilizing a shared feature extractor, as opposed to separate networks, with *Raw-continuous* representations and the CTCE paradigm. The only statistically significant pairing is found to be

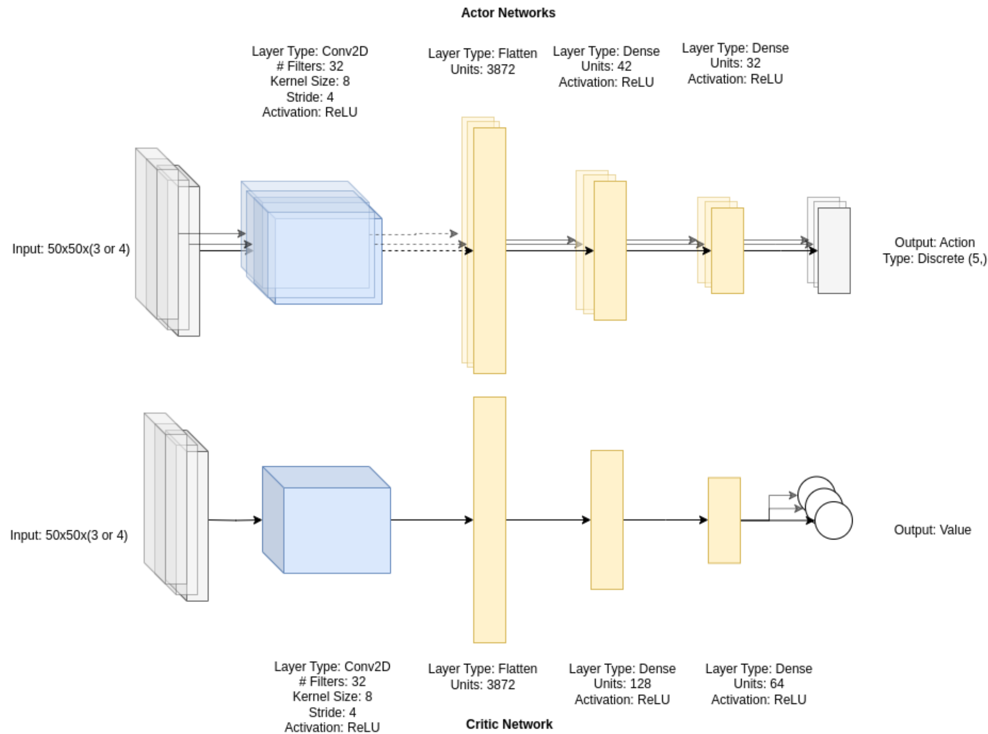


Figure 3.20: The *CTDE* architecture utilized when training on *RBF (2D)* observations, without shared parameters between the actor and critic networks. The total number of trainable parameters in this architecture is 1,029,582.

that of Configurations 8 and 9 based on the Tukey's HSD test performed.

Actor & Critic Inputs When looking at the impact of varying input representation on performance, it can be seen in Table 3.14 to be dependent on the configuration pairing. The second and fourth row of this table indicate that *RBF (2D)* representations and corresponding architectures significantly outperform in the CTCE configurations. The first and third rows of this table indicate that *Raw-continuous* representations and corresponding architectures outperform in the CTDE configurations. Looking at the top performing symmetric representation and architecture, configuration 10, compared to the asymmetric architecture in configuration 12, we find a comparable, but marginally worse performance when utilizing the asymmetric architecture, as shown in the fifth row of the results table.

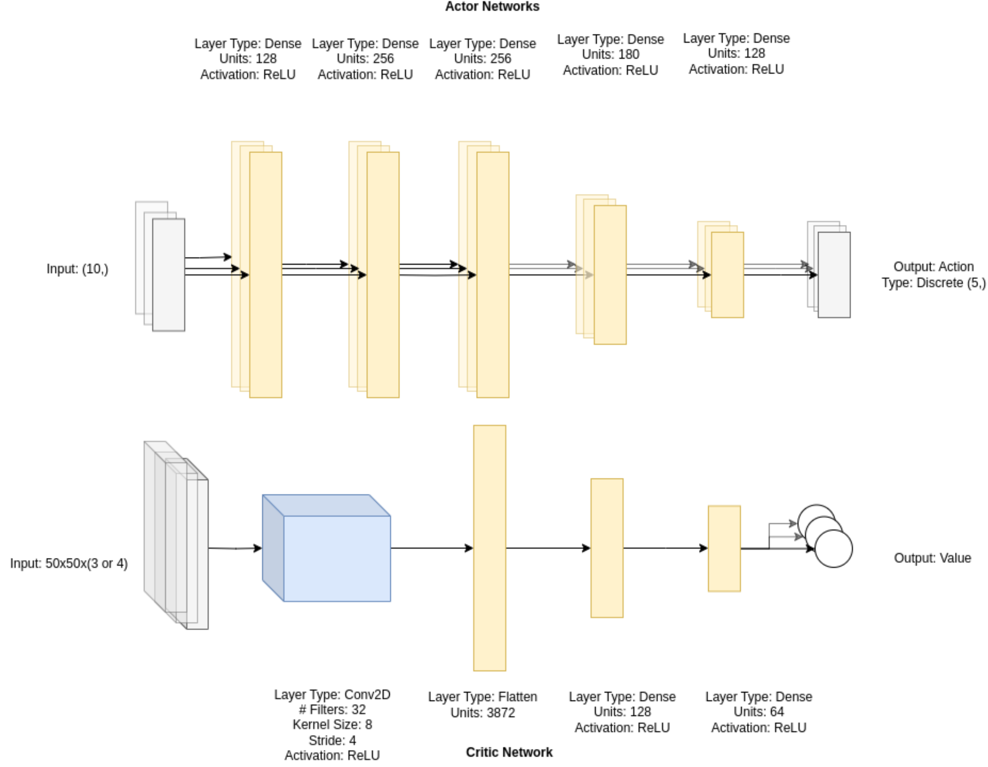


Figure 3.21: The *CTDE* architecture utilized when training on *Hybrid* observations, without shared parameters between the actor and critic networks. The actor networks receive *Raw-continuous* observations, while the critic receives *RBF (2D)* observations. The total number of trainable parameters in this architecture is 1,023,180.

Based on post-hoc Tukey's HSD tests performed on each configuration, we find just 1 pairing to have statistically significant performance differences based on the input representation. This is the pairing of Configuration 6 and 9. This seems to indicate that the *RBF (2D)* configuration with a shared feature extractor is more performant than a comparable *Raw-continuous* architecture when utilizing a CTCE paradigm.

These experiments leave us with two key takeaways that will guide future experimentation. Firstly, we note the relatively higher performance of CTDE architectures when compared to CTCE. Secondly, we note that the highest performing configuration is configuration 10, which utilizes a *Raw-continuous* representation, however we also note that architectures utilizing *RBF (2D)*

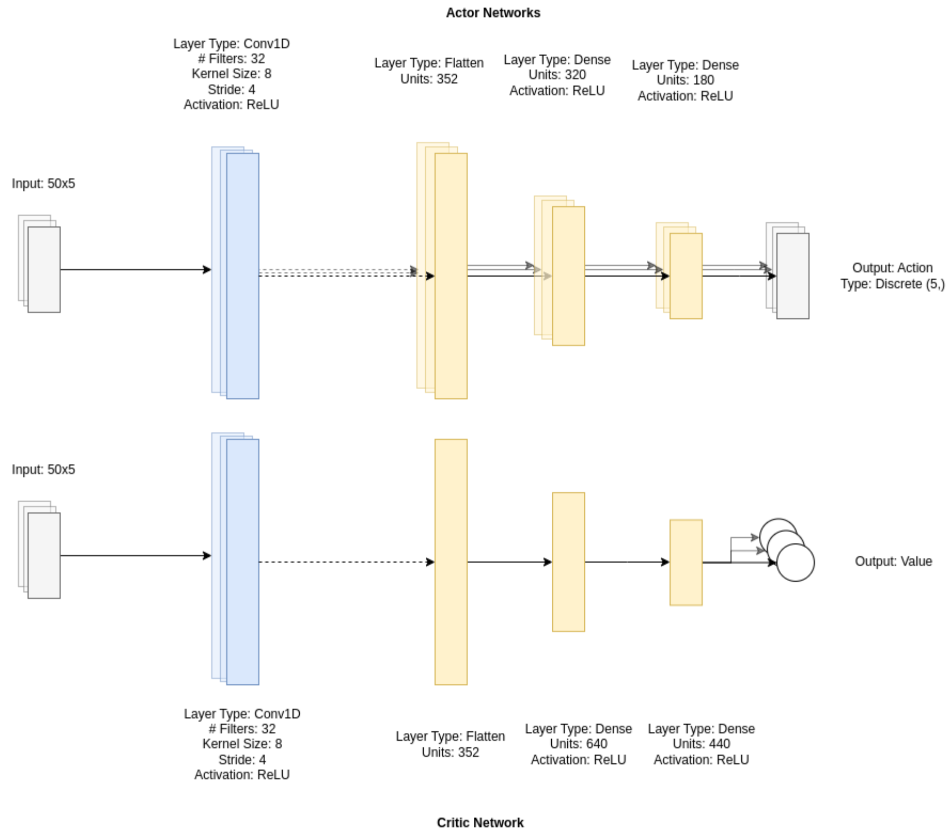


Figure 3.22: The *CTDE* architecture utilized when training on *RBF (1D)* observations, without shared parameters between the actor and critic networks. The total number of trainable parameters in this architecture is 1,033,704.

representations are highly competitive.

From our final set of miscellaneous experiment comparisons, detailed by the Tukey's HSD test results in Table 3.15, we can see that there is a statistically significant difference in performance when training with an architecccture that receives *RBF (1D)* representations as input. This can be seen to be true when compared against architectures utilizing both *Raw-continuous* and *RBF (2D)* representations. In the case of the asymmetric architecture, while the performance appears to slightly worsen based on the performance difference, the results show that the difference is not statistically significant.

Table 3.7: Hyperparameters used with PPO for the MPE experiments

Hyperparameter	Value
Learning Rate	[1e-5, 1e-4, 1e-3, 1e-2, 1e-1]
Rollout Timesteps, T	4800
Total Timesteps	302400
Minibatch Size, M	1600
Iterations, N	63
Epochs, K	10
Discount Factor, γ	0.95
GAE Lambda, λ	0.95
Value Loss Coefficient, c_1	0.5
Entropy Loss Coefficient, c_2	0.01
Clipping Ratio, ε	0.2

3.4 Conclusion

This chapter sought to explore low-dimensional simulation environments, focusing particularly on the MountainCar environment and the MPE. We conducted numerous experiments with diverse state representations to discern their combined influence with varying neural network architectures on the performance of trained agents. In the case of the MPE, we additionally explore the effects of varying learning rates and multi-agent execution paradigms - centralized versus decentralized - on performance outcomes.

The findings from our investigation in both the MountainCar environment and the MPE point towards the superior performance of decentralized execution and high-resolution state representations, specifically *RBF (2D)* and *Raw-continuous* representations. In Section 3.3, we further demonstrate the performance of agents trained using a CTDE paradigm, and demonstrate the consistent performance of agents trained in the MPE using a learning rate of '1×10^{-4}', across multiple experiment configurations.

These results suggest that the choice of state representation, neural network architecture, multi-agent training paradigm, and learning rate can significantly influence the effectiveness of an agent within a given environment. The apparent superiority of decentralized execution paradigms and high-resolution representations like *RBF (2D)* and *Raw-continuous* will be utilized to guide experimentation in the subsequent work.

Table 3.8: Configuration definitions for each experiment conducted in MPE. The *Actor Input* and *Critic Input* columns specify the inputs to the policy and value networks, respectively. The *Merged Other Agents* column is only relevant when utilizing an RBF representation and specifies whether a single channel is used to represent information pertaining to other agents or if it is split into one layer per other agent. The *Shared Feature Extractor* specifies whether the actor and critic networks have any shared parameters between them. Finally, the *Training & Execution Paradigm* column indicates whether the same policy network parameters are utilized in action selection for all agents (*CTCE*), or whether each agent in the environment has parameters specific to each agent in the environment (*CTDE*).

Configuration	Input Representation			Network Architecture	
	Actor Input	Critic Input	Merged Other Agents	Shared Feature Extractor	Training & Execution Paradigm
0	RBF (2D)	RBF (2D)	No	No	CTCE
1	RBF (2D)	RBF (2D)	No	No	CTDE
2	RBF (2D)	RBF (2D)	No	Yes	CTCE
3	RBF (2D)	RBF (2D)	No	Yes	CTDE
4	RBF (2D)	RBF (2D)	Yes	No	CTCE
5	RBF (2D)	RBF (2D)	Yes	No	CTDE
6	RBF (2D)	RBF (2D)	Yes	Yes	CTCE
7	RBF (2D)	RBF (2D)	Yes	Yes	CTDE
8	Raw-continuous	Raw-continuous	N/A	No	CTCE
9	Raw-continuous	Raw-continuous	N/A	Yes	CTCE
10	Raw-continuous	Raw-continuous	N/A	No	CTDE
11	Raw-continuous	Raw-continuous	N/A	Yes	CTDE
12	Raw-continuous	RBF (2D)	No	N/A	CTDE
13	RBF (1D)	RBF (1D)	No	No	CTDE

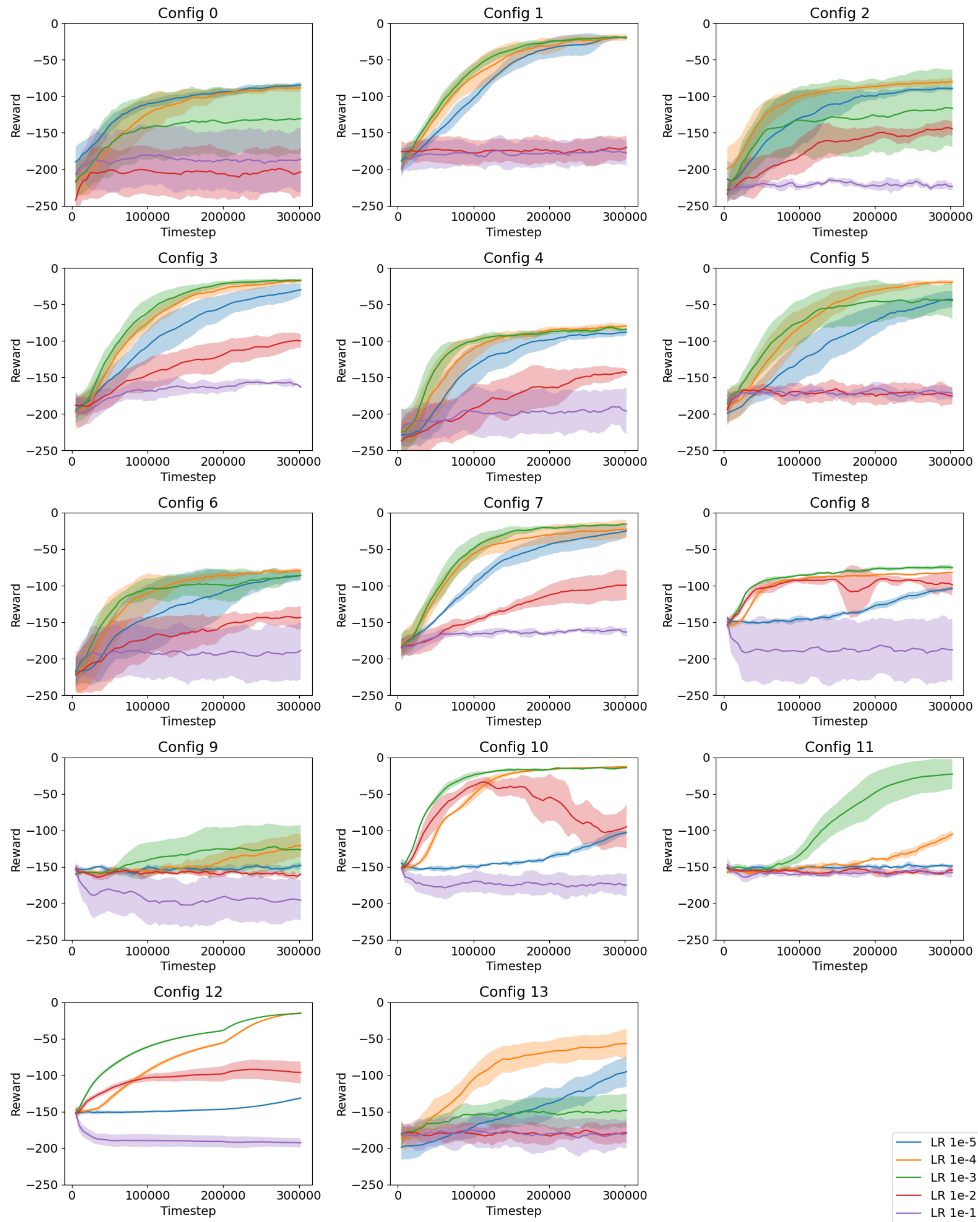


Figure 3.23: The mean episodic reward achieved by all of our agent configurations through 302,400 timesteps of training in the multi-agent particle environment's formation task. The mean and standard deviation are calculated from 5 training runs at each configuration and learning rate combination.

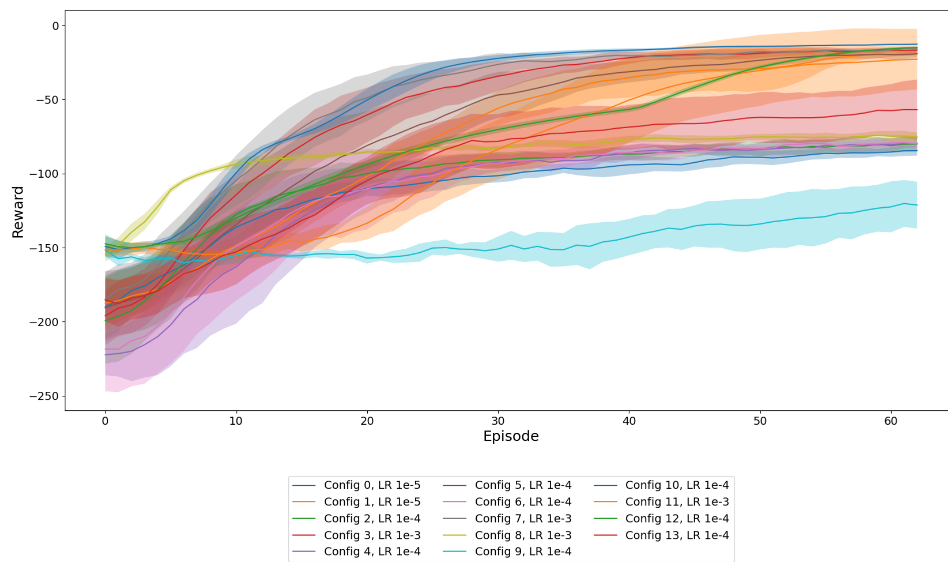


Figure 3.24: The mean episodic reward achieved by our highest performing agents in each configuration through 302,400 timesteps of training in the multi-agent particle environment's formation task with corresponding learning rate shown.

Table 3.9: The mean and standard deviation of the final performance for each representation configuration, with the highest-performing learning rate shown. Results are averaged over 5 training runs with the given learning rate.

Configuration	**Learning Rate**	**Final Reward**
0	1×10^{-5}	-84.53 ± 3.41
1	1×10^{-5}	-18.98 ± 2.01
2	1×10^{-4}	-80.09 ± 5.29
3	1×10^{-3}	-16.68 ± 2.03
4	1×10^{-4}	-79.72 ± 2.36
5	1×10^{-4}	-19.22 ± 2.42
6	1×10^{-4}	-79.94 ± 3.74
7	1×10^{-3}	-15.74 ± 0.49
8	1×10^{-3}	-75.38 ± 2.92
9	1×10^{-4}	-121.29 ± 15.77
10	1×10^{-4}	-12.70 ± 0.15
11	1×10^{-3}	-22.85 ± 20.48
12	1×10^{-4}	-14.85 ± 0.35
13	1×10^{-4}	-56.93 ± 20.32

Table 3.10: Shorthand introduced to more readily interpret experiment configurations when referenced in results. CTCE and CTDE refer to the training paradigm utilized; RBF/RC/RBF1D indicates *RBF (2D)*, *Raw-continuous* or *RBF (1D)* observation types respectively; SYM/ASYM indicate whether the actor and critic architecture or symmetric or asymmetric respectively; M/NM indicates whether the other agents are merged or not merged into a single channel respectively; S/NS indicates whether the feature extractor is shared or not shared between the actor and critic.

Configuration	**Shorthand**
0	CTCE-RBF-SYM-NM-NS
1	CTDE-RBF-SYM-NM-NS
2	CTCE-RBF-SYM-NM-S
3	CTDE-RBF-SYM-NM-S
4	CTCE-RBF-SYM-M-NS
5	CTDE-RBF-SYM-M-NS
6	CTCE-RBF-SYM-M-S
7	CTDE-RBF-SYM-M-S
8	CTCE-RC-SYM-NS
9	CTCE-RC-SYM-S
10	CTDE-RC-SYM-NS
11	CTDE-RC-SYM-S
12	CTDE-ASYM-NM
13	CTDE-RBF1D-SYM-NM-NS

Table 3.11: The performance difference and statistical comparison resulting from using the *Merged Other Agents* representation, as opposed to using separate channels for other agents. Paired configurations are identical except for this representation decision. A positive *Relative Performance* Δ indicates higher performance resulting from using *Merged Other Agents* as opposed to using separate channels.

Config. Pair	Shorthand, X = NM or M	Abs. Perf. Δ	Rel. Perf. Δ (%)	P-value	Reject
0,4	CTCE-RBF-SYM-X-NS	-4.81	5.69	> 0.999	False
1,5	CTDE-RBF-SYM-X-NS	0.24	-1.26	> 0.999	False
2,6	CTCE-RBF-SYM-X-S	-0.15	0.19	> 0.999	False
3,7	CTDE-RBF-SYM-X-S	-0.94	5.64	> 0.999	False

Table 3.12: The performance difference resulting from changing the *Training and Execution Paradigm*. Paired configurations are identical except for the use of a CTCE or CTDE paradigm. A positive *Relative Performance* Δ indicates higher performance resulting from using a CTDE paradigm as opposed to a CTCE paradigm.

Config. Pair	Shorthand, X = CTCE or CTDE	Abs. Perf. Δ	Rel. Perf. Δ (%)	P-value	Reject
0,1	X-RBF-SYM-NM-NS	-65.55	77.55	< 0.001	True
2,3	X-RBF-SYM-NM-S	-63.41	79.17	< 0.001	True
4,5	X-RBF-SYM-M-NS	-60.50	75.89	< 0.001	True
6,7	X-RBF-SYM-M-S	-64.20	80.31	< 0.001	True
8,10	X-RC-SYM-NS	-62.68	83.15	< 0.001	True
9,11	X-RC-SYM-S	-98.44	81.16	< 0.001	True

Table 3.13: The performance difference resulting from changing the *Shared Feature Extractor*. Paired configurations are identical except for the use of a *Shared Feature Extractor* as opposed to entirely separate actor and critic networks. A positive *Relative Performance* Δ indicates higher performance resulting from using separate actor and critic networks as opposed to a *Shared Feature Extractor*.

Config. Pair	Shorthand, X = NS or S	Abs. Perf. Δ	Rel. Perf. Δ (%)	P-value	Reject
0,2	CTCE-RBF-SYM-NM-X	-4.44	5.25	> 0.999	False
1,3	CTDE-RBF-SYM-NM-X	-2.30	12.12	> 0.999	False
4,6	CTCE-RBF-SYM-M-X	0.22	-0.28	> 0.999	False
5,7	CTDE-RBF-SYM-M-X	-3.48	18.11	> 0.999	False
8,9	CTCE-RC-SYM-X	45.91	-60.90	< 0.001	True
10,11	CTDE-RC-SYM-X	10.15	-79.92	0.9456	False

Table 3.14: The performance difference resulting from changing the input representation to actor and critic. Paired configurations are identical except for the use of an *RBF (2D)* or *Raw-continuous* input representation. Note that two corresponding *RBF (2D)* runs exist for each *Raw-continuous* run due to the added configuration parameter *Merged Other Agents* in *RBF (2D)* experiments, so the best performing of the two is shown for this comparison table since no significant difference was found in experiments varying the *Merged Other Agents* configuration. Note that shorthand corresponding to *Merged Other Agents* is omitted since it does not apply to *Raw-continuous* representations. A positive *Relative Performance* Δ indicates higher performance resulting from using a *Raw-continuous* input representation as opposed to an *RBF (2D)* representation.

Config. Pair	Shorthand, X = RBF or RC	Abs. Perf. Δ	Rel. Perf. Δ (%)	P-value	Reject
4,8	CTCE-X-SYM-NS	-4.34	5.44	> 0.999	False
6,9	CTCE-X-SYM-S	41.35	-51.73	< 0.001	True
1,10	CTDE-X-SYM-NS	-6.28	33.09	0.9993	False
7,11	CTDE-X-SYM-S	7.11	-45.17	0.9975	False

Table 3.15: The performance difference resulting from our additional experiments, including the ablation experiment utilizing *RBF (1D)* representations, and our experiment utilizing an asymmetric architecture. In the *RBF (1D)* experiments, a positive *Relative Performance* Δ indicates worse performance resulting from using a *RBF (1D)* input representation as opposed to the relevant alternative representation. In the asymmetric experiment, a positive *Relative Performance* Δ indicates worse performance resulting from using the asymmetric architecture.

Config. Pair	Type	Abs. Perf. Δ	Rel. Perf. Δ (%)	P-value	Reject
1, 13	RBF (1D)	37.95	-199.95	< 0.001	True
10, 13	RBF (1D)	44.23	-348.27	< 0.001	True
10,12	Asym	2.15	-16.93	> 0.999	False

4 Missile Defence Environment

4.1 Introduction

Modelling & Simulation (M&S) techniques have long been utilized in the defence community as a means of gaining insight into problems of interest without the limitations of real-world systems [54]. By developing computational models to the necessary level of fidelity to be a sufficient proxy for reality, a multitude of systems and scenarios can be explored and evaluated. This capability is especially valuable in optimization problems where real-world data is lacking or cost-prohibitive to obtain, or where the search space for parameters of interest is extensive.

In recent years, increasing interest has been devoted to modelling approaches that make use of dynamic programming [62], artificial intelligence (AI) and machine learning (ML) to develop outcomes that may not be achievable with traditional methods. The utility of AI/ML approaches has been highlighted in a wide variety of problems, including efficient resource allocation and task scheduling [14], and agent-based simulation. Simulations have found utility for scenarios aligned with the subject of this research, in missile defence applications [26, 87, 91].

In this chapter, we develop a multi-agent reinforcement learning framework for training of offensive and defensive strategies. Our mixed cooperative and competitive missile simulation environment incorporates a team of defensive agents with the ability to launch interceptor missiles at incoming attacker missiles. The environment additionally incorporates a single offensive agent. Our objective is to develop an effective state representation, and corresponding neural network architecture, to allow for deep neural network agent learning in our simulation environment, and to demonstrate this effectiveness through comparison of our trained agents' mean episodic reward to that achieved by a selection of hard-coded baselines.

4.2 Related Work

In this section, we delve into an examination of prior work pertinent to the topic of Deep Reinforcement Learning (DRL) for missile defence. We will first look at M&S techniques applied to defence problems more broadly, before moving into work relating to agent-based modelling in missile defence scenarios.

M&S techniques have found extensive use in predicting and strategizing against threats like improvised explosive devices (IEDs). Studies have shown the effectiveness of these techniques in modeling complex agent relationships that underpin IED placement, allowing for phase-specific countermeasures [30, 23]. Notably, Dekker et al. proposed the use of RL to imbue simulated agents with adaptive behavior.

Agent-based simulation, a key element in M&S, has also been utilized to assess security and efficiency in an airport setting, demonstrating that these two factors need not be in conflict [40]. Countermeasures against IEDs, such as ground-based and aerial jamming systems, have been evaluated using a comprehensive array of simulation parameters [7].

The defence-focused Map Aware Non-uniform Automata (MANA) simulation tool is used for broader applications beyond IED-related scenarios. For instance, MANA was used in simulating a maritime counter-piracy scenario, with automated red teaming and data farming methods employed to assess vulnerabilities and explore variable parameters [21, 22].

Prior work on missile defence applications of simulations have made use of surrogate modelling due to the computational complexity of a typical System of Systems (SoS) simulation architecture. These surrogate models utilize statistical methods to ensure representation of the design space and have found particular utility in Ballistic Missile Defense System (BMDS) simulations [26, 87, 91].

The air defence problem has been approached through Weapon-Target Assignment (WTA), with M&S methodologies aiding in defensive capability analysis of naval task groups against Anti-Ship Missile (ASM) fire [41]. The spatio-temporal dimensions and non-homogenous nature of the WTA problem have been further explored, revealing the superiority of non-heuristic methods, especially DRL, in managing combinatorially complex solution spaces. The DRL-based Double Deep Q-Network (DDQN) solution, in particular, demonstrated high performance and computational efficiency, outperforming evolutionary algorithms [66].

While the aforementioned surrogate modelling approaches look to efficiently model the complex dynamics of BMDS simulations, our work looks

to focus on learning strategies for attacking and defending agents, rather than focusing on efficient high-fidelity modelling. We look to take a novel approach of modelling the missile defence scenario as a multi-agent environment so that MARL methodologies and architectures can be utilized. This work is additionally differentiated from the DRL approaches to the WTA problem [66] that generate policies for an environment with a fixed number of weapons to be assigned to a fixed number of targets, as we look to generate policies that incorporate entity dynamics in the simulation environment.

4.3 Environment Background & Baseline Policies

4.3.1 Environment Background

The environment developed for the analysis of missile defence scenarios is modelled after an OpenAI gym-style architecture. In the mixed cooperative and competitive multi-agent simulation, a team of defenders is tasked with protecting a selection of targets of differing values from a single attacking agent, and the attacking agent is tasked with firing missiles at these targets. The team of defending agents are not able to communicate with each other to inform their action selection.

For the initialization of entities in the environment, probability distributions are utilized that aim to approximate the locations of such entities along a sea coast. We generate maps with both a length and width of 2000 units and divide the map into 4 quarters vertically, as can be seen in Figure 4.1. A total of 15 targets are initialized, with values uniformly randomly generated between 0 and 1 units of reward.

The attacker entity is uniformly, randomly positioned in the rightmost quarter. The defenders are positioned with uniformly random y-positions, and skew-normal distributed x-positions where the distribution spans the middle two quarters with a peak in the central-right quarter. Similarly, the targets are positioned with uniformly random y-positions, and skew-normal x-positions where the distribution spans the three leftmost quarters, with a peak in the center of the map, just behind the defenders' peak. The rightmost target position is clipped to ensure that it will never be farther right than the rightmost defender.

The defenders have a detection radius of 250 units, and an interception radius of 160 units. The detection radius corresponds to the zone within which a defender can begin computing interception opportunities on incoming missiles, while the interception radius corresponds to the zone within which these interceptions can take place. Each defender is initialized with 8 missiles

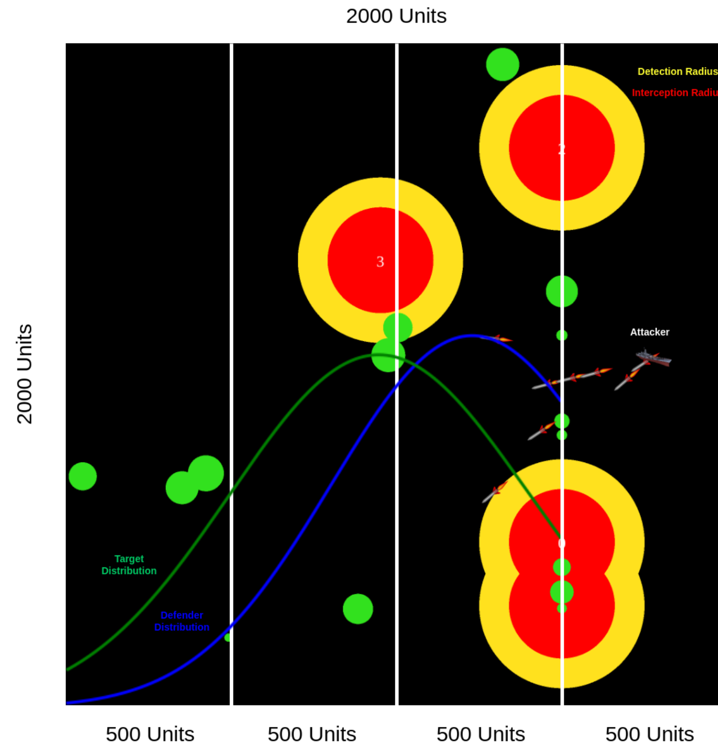


Figure 4.1: A map showing a rendering of the environment state, with key entities labelled. The skew normal distributions sampled when initializing the x-positions for defenders (in blue) and targets (in green) are also shown. The targets' radii are proportional to their value. The defenders' detection and interception radii are pictured in yellow and red, respectively.

to launch, with each missile able to travel at a speed of 42 units per timestep. The reload time required between a single defender's successive launches is set to 1 timestep. A total of 4 defenders are initialized in the environment at the start of a simulation. The attacker is initialized with 10 missiles to launch, with each missile able to travel at a speed of 15 units per timestep. The time required between the attacker's successive launches is 5 timesteps. The success rate of defender interception events being successful once taken is set to 100%.

The simulation environment can be viewed as a zero-sum game for the attacker and defenders, where the defenders are collectively penalized for targets being hit by the attacker, and the attacker is conversely rewarded an equal and

opposite amount. More specifically, the reward (penalty) signal is collectively received by the agents when a target is hit, and the magnitude of this signal is equal to the value of the target prior to the impact, multiplied by a constant to represent the percentage of damage done to the target on impact, which is set to 70%. The value of the target is then updated by subtracting this value from it's value prior to impact. Following this methodology implies targets will continuously decrease in value through consecutive missile impacts, and the reward (penalty) for successive impacts will be subject to decay, following a geometric progression.

Each step through the simulation environment progresses the environment clock forward a timestep. The actions available to the attacker agent consist of firing at a target of their choice, if they have the ammunition available and are not reloading, or doing nothing. The defenders, similarly, can take an interception opportunity if one is available, or they can do nothing. The environment tracks the opportunities available to the defenders at each timestep. The opportunities define where each defender has the ability to intercept an incoming missile if they fire at the current timestep. All agents act simultaneously and the actions received by the environment step function are added to the event queue. Scheduled events are then processed in order if they occur in the current timestep, invalidated future events are removed from the queue (i.e. if an attacker missile is intercepted, pending interception events on that missile are removed), and all relevant environment variables are updated.

4.3.2 Baseline Policies

To benchmark agent performance, the environment includes a selection of hard-coded baseline policies. These policies have been created to control both the attacker and defender behaviors according to predefined rules. The naming and corresponding behaviour definitions are as follows:

- **Attacker Baselines**
  - **Greedy**: The attacker will fire at the current highest value target on the map as soon as the action is available.
  - **SLS-Greedy**: The attacker will fire at the current highest value target on the map if the action is available and no other active missiles are present on the map. This baseline can be considered to be a Shoot-Look-Shoot (SLS) policy.
  - **Random**: The attacker will fire at a random target on the map as soon as the action is available.
- **Defender Baselines**

- **Greedy**: The defender will take an opportunity to intercept an attacker missile as soon as the opportunity is available.
- **SLS-Greedy**: The defender will take an opportunity to intercept an attacker missile as long as no other interception event is pending for the given attacker missile. This baseline can be considered to be a Shoot-Look-Shoot (SLS) policy.
- **Random**: When an opportunity is available to a defender, the defender has a 5% probability of taking it and a 95% probability of not taking it.

For the *Random Defender*, we cannot use a uniformly distributed random action, as it would almost always entail intercepting the missile. A 5% fire probability for the *Random* baseline was selected to give the defender an approximately 50% probability of firing at a missile when its path of travel passes through the interception zone of a defender. In this case, since defender missiles are faster than attacker missiles, the defender can intercept an attacker missile at any point that it is passing through the diameter of the interception zone of the given defender. To compute the selected probability assigned to taking an opportunity, we use the following schema.

The expected number of timesteps, n_{ts}, that an attacker missile will spend in the defender's interception zone when travelling along this diameter can be calculated as follows:

$$n_{ts} = \frac{2 \times r}{V_{att}} \tag{4.1}$$

Since we define the interception radius, r as 160 units, and the attacker missile velocity, V_{att} as $15\frac{units}{timestep}$, the expected number of timesteps that an attacker missile will spend in the interception zone can be determined as follows:

$$n_{ts} = \frac{2 \times 160 units}{(15\frac{units}{timestep})} = 21.33\overline{3} timesteps \tag{4.2}$$

We can then calculate the probability of the defender never firing at the attacker missile while it is travelling across the diameter of the interception zone as follows:

$$P_{no-intercept} = (1 - P_{fire,ts})^{n_{ts}} \tag{4.3}$$

With an expected number of timesteps in the interception zone of $21.33\overline{3}$ and a single timestep probability of firing, $P_{fire,ts}$, of 5%, the probability of a missile travelling through the center of the circle without an interception event taking place is then determined to be:

$$P_{no-intercept} = (1 - P_{fire,ts})^{n_{ts}} = (1 - 0.05)^{21.33\overline{3}} \approx 0.335 \tag{4.4}$$

We can expand this calculation to consider the average case, rather than the diameter trajectory case, by considering the expected distance that a missile must travel through the interception zone. This distance can be approximated by taking the area of the interception zone, and dividing by the diameter of the zone. This equates to the average distance a missile would travel if it were to pass through any trajectory over the interception zone parallel to a trajectory that passes along a diameter of the zone. In our case, we can calculate this average trajectory length, $\overline{D}$, as:

$$\overline{D} = \frac{\pi \times r^2}{2 \times r} = \frac{\pi \times r}{2} = \frac{\pi \times 160 units}{2} = 251.33 units \tag{4.5}$$

We then use this value along with the attacker's missile velocity, V_{att}, to determine the expected number of timesteps, $\overline{n_{ts}}$. The expected number of timesteps, $\overline{n_{ts}}$, that a missile will spend within the interception zone in the general case can be calculated as:

$$\overline{n_{ts}} = \frac{\overline{D}}{V_{att}} = \frac{251.33 units}{15 \frac{units}{timestep}} \approx 16.78 timesteps \tag{4.6}$$

In the simulation configuration using a 100% probability of an interception event being successful, $P_{s,t}$, we can then determine probability of no interception event taking place in a given timestep, $P_{ni,t}$, as the sum of the probability of a defender not firing, and the probability of the defender firing multiplied by the probability of that interception event being unsuccessful.

$$\begin{aligned} P_{ni,t} &= 1 - P_{fire,t} + P_{fire,t} \times (1 - P_{s,t}) \\ &= 1 - 0.05 + 0.05 \times (1 - 1) = 0.95 - 0 \\ &= 0.95 \end{aligned} \tag{4.7}$$

We can then calculate the probability that no successful interception is made over all timesteps that the attacker missile is in a defender's interception zone, P_{ni}, as follows:

$$P_{ni} = P_{ni,t}^{\overline{n_{ts}}} = 0.95^{16.78} \approx 0.45 \tag{4.8}$$

Therefore, there is an approximately 45% chance that a missile with a trajectory over a defender interception zone will not be taken down by the defender in the 100% interception success, *Random Defender* configuration. In other words, there is an approximately 55% chance that the missile will be taken down in this configuration when its trajectory travels over a defender interception zone.

4.4 Agent Representation & Architecture

4.4.1 Observation Representation

The observation representation utilized in our experiments is dependent on the agent being trained, with differing representations being utilized for attacker and defender agents. The difference between the attacker and defender representations can be seen in Figure 4.3, representing the attacker's observation representation, and Figure 4.4, representing the defenders' observation representation. Additionally, asymmetric architectures utilize a more complete representation of the state as input to the critic, as show in Figure 4.5. The motivation behind this asymmetric representation is discussed in Section 4.5.1, and relates to the natural asymmetry of information in adversarial scenarios. The representations for attacker, defender, and asymmetric critic share a common layer dimension of 84x84, meaning each pixel equates to a region of approximately 23.8×23.8 units. However, the observation representations differ in number of channels, where each channel represents a different piece of information in the environment. The attacker actor input consists of 9 channels, while the defenders' actor inputs consist of 15 channels. The observation input to the defenders' actor networks contain 5 channels to represent information pertaining only to the defender for which the network must generate an action. The asymmetric critic input representation contains 13 channels, consisting of state information pertaining to both the attacker and defender.

To generate these image-based observations, we utilize the *RBF (2D)* methodology. The decision to utilize an image-based representation was made due to its success at generating effective agents in Chapter 3, along with the success of comparable representations seen in literature on complex adversarial scenarios, in particular the StarCraft II Learning Environment (SC2LE) [85]. We additionally implement a *One-hot* version, in order to confirm whether the superior performance of *RBF (2D)* representations found in prior environments extends to the missile environment.

The following describes the content of each channel of an observation in the *RBF (2D)* and *One-hot* cases. The visual representations of *RBF (2D)* are additionally available in Figure 4.2. We utilize the general format of Equation 4.9 for the generation of *RBF (2D)* images, where the σ value varies by layer. Note than in all channels containing information from multiple agents, the maximum pixel value from overlapping *RBF (2D)* circles is kept in the final layer generated.

In encoding our states as 2-dimensional RBF-based representations, a key

parameter to consider is the spread of RBFs. While the primary application of varying RBF spread in our context was to represent variables of differing magnitude in the same channel, it could also serve to encode uncertainty. One such example of this could be to encapsulate the uncertainty in entity locations in the environment, or uncertainty in the likelihood of interception events being successful.

One key point throughout our experiments was that the agents assume a fully observable state observation that encapsulates all information relevant to the action selection of a given agent. For this reason, we include the defender locations explicitly in the observations of attacker agents, as this is deemed a core piece of information in developing an effective attacking strategy.

$$\phi_{RBF2D}(c_i, c_j) = (\frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x_{pos}-c_i)^2}{2\sigma^2}}) \times (\frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(y_{pos}-c_j)^2}{2\sigma^2}}) \tag{4.9}$$

Complete List of Possible State Representation Layers

0. **Target Value**
   - *RBF (2D)*: A single channel to represent the current value of targets on the map. Each circle is generated using Equation 4.9, where the radius of the circle is proportional to the value of the target, since we utilize a $\sigma = \frac{1}{16} \times$ Target Value. The center of each circle corresponds to the target x_{pos} and y_{pos}, and pixel intensity increases to a maximum of 1 at the center.
   - *One-hot*: A single pixel per target located at the targets' x_{pos} and y_{pos}, with pixel intensities equal to target values.
1. **Defender Reload Delay**
   - *RBF (2D)*: A single channel to represent the current reload delay of each of the defenders. Each circle is generated using Equation 4.9, where the radius of the circle is inversely proportional to the number of timesteps until the reload process is completed, since we utilize a $\sigma = \frac{1}{16} \times \frac{1}{\text{Defender Reload Delay}+1}$. The center of each circle corresponds to the defender x_{pos} and y_{pos}, and pixel intensity increases to a maximum of 1 at the center.
   - *One-hot*: A single pixel per defender located at the defenders' x_{pos} and y_{pos}, with pixel intensities equal to $\frac{1}{\text{Defender Reload Delay}+1}$.
2. **Defender Ammunition**
   - *RBF (2D)*: A single channel to represent the ammunition remaining for each of the defenders. Each circle is generated using Equation

4.9, where the radius of the circle is proportional to the number of interceptor missiles that the given defender has remaining, since we utilize a $\sigma = \frac{1}{16} \times \frac{\text{Defender Ammunition}}{\text{Defender Max Ammunition}}$. The center of each circle corresponds to the defender x_{pos} and y_{pos}, and pixel intensity increases to a maximum of 1 at the center.

– *One-hot*: A single pixel per defender located at the defenders' x_{pos} and y_{pos}, with pixel intensities equal to $\frac{\text{Defender Ammunition}}{\text{Defender Max Ammunition}}$.

3. **Defender Detection**
   – *RBF (2D)*: A single channel to represent the detection zone for each of the defenders. Each circle is generated according to the predefined 250-unit detection radius, which is uniform among all defenders. The center of each circle corresponds to the defender location, and pixel intensity is equal to 1 through the entire circle corresponding to the detection region. This region is calculated by taking the Euclidean distance from the center of each pixel to the center of the defender, and setting the pixel value to 1 if this distance is less that the 250-unit detection radius.
   – *One-hot*: Identical to *RBF (2D)* for this channel.
4. **Defender Interception**
   – *RBF (2D)*: A single channel to represent the interception zone for each of the defenders. Each circle is generated according to the predefined 160-unit interception radius, which is uniform among all defenders. The center of each circle corresponds to the defender location, and pixel intensity is equal to 1 through the entire circle corresponding to the interception region. This region is calculated by taking the Euclidean distance from the center of each pixel to the center of the defender, and setting the pixel value to 1 if this distance is less that the 160-unit interception radius.
   – *One-hot*: Identical to *RBF (2D)* for this channel.
5. **Defender Opportunities**

   – *RBF (2D)*: A single channel to represent the current opportunities available for all defenders. Each circle is generated using Equation 4.9, where the radius of the circle is constant, since we utilize a $\sigma = \frac{1}{16} \times 0.3$. The center of each circle corresponds to the x_{pos} and y_{pos} of the opportunities, and pixel intensity increases to a maximum of 1 at the center.

   – *One-hot*: A single pixel per opportunity located at the opportunities' x_{pos} and y_{pos}, with pixel intensities equal 1.

6. **Defender Opportunities Taken**
   - *RBF (2D)*: A single channel to represent the opportunities taken by all defenders. Each circle is generated using Equation 4.9, where the radius of the circle is constant, since we utilize a $\sigma = \frac{1}{16} \times 0.3$. The center of each circle corresponds to the x_{pos} and y_{pos} of the opportunities, and pixel intensity increases to a maximum of 1 at the center.
   - *One-hot*: A single pixel per opportunity taken located at the opportunities' x_{pos} and y_{pos}, with pixel intensities equal 1.
7. **Attacker Reload Delay**
   - *RBF (2D)*: A single channel to represent the current reload delay of the attacker. The circle is generated using Equation 4.9, where the radius of the circle is inversely proportional to the number of timesteps until the reload process is completed, since we utilize a $\sigma = \frac{1}{16} \times \frac{1}{\text{Attacker Reload Delay+1}}$. The center of the circle corresponds to the attacker x_{pos} and y_{pos}, and pixel intensity increases to a maximum of 1 at the center.
   - *One-hot*: A single pixel located at the attacker's x_{pos} and y_{pos}, with pixel intensity equal to $\frac{1}{\text{Attacker Reload Delay+1}}$.
8. **Attacker Ammunition**
   - *RBF (2D)*: A single channel to represent the ammunition remaining for the attacker. The circle is generated using Equation 4.9, where the radius of the circle is proportional to the number of missiles that the attacker has remaining, since we utilize a $\sigma = \frac{1}{16} \times \frac{\text{Attacker Ammunition}}{\text{Attacker Max Ammunition}}$. The center of the circle corresponds to the attacker x_{pos} and y_{pos}, and pixel intensity increases to a maximum of 1 at the center.
   - *One-hot*: A single pixel located at the attacker's x_{pos} and y_{pos}, with pixel intensity equal to $\frac{\text{Attacker Ammunition}}{\text{Attacker Max Ammunition}}$.

9-11. **Attacker Missile Positions (t=[T,T-1,T-2])**
   - *RBF (2D)*: Three channels to represent the positions of the attacker's missiles in the most recent three timesteps. Each circle is generated using Equation 4.9, where the radius of the circle is constant, since we utilize a $\sigma = \frac{1}{16} \times 0.15$. The center of each circle corresponds to the x_{pos} and y_{pos} of the active missiles, and pixel intensity increases to a maximum of 1 at the center.
   - *One-hot*: Three channels to represent the positions of the attacker's missiles in the most recent three timesteps. A single pixel is activated in each channel per active attacker missile in the given

timestep of the channel. These pixels are located at the x_{pos} and y_{pos} of the active missiles in the timestep of the channel, with pixel intensities equal 1.

12. **Target Value Loss if Pending Hits Completed**
    - *RBF (2D)*: A single channel to represent the value to be taken from a target and provided as reward to an attacker agent, or penalty to the defending agents, should the presently active missiles hit their intended targets. Each circle is generated using Equation 4.9, where the radius of the circle is determined by the value of the target and the incoming attacker missiles on a given target, since we utilize a $\sigma = \frac{1}{16} \times (\text{Target Value}) \times (1 - 0.3^{N_{att}})$. Note that N_{att} is equal to the number of incoming attacker missiles for a given target, and $(\text{Target Value}) \times (1 - 0.3^{N_{att}})$ is equivalent to the expected reward, or penalty, should the pending missiles reach their intended target. The center of each circle corresponds to the target x_{pos} and y_{pos}, and pixel intensity increases to a maximum of 1 at the center.
    - *One-hot*: A single pixel per target located at the targets' x_{pos} and y_{pos}, with pixel intensities equal to $(\text{Target Value}) \times (1 - 0.3^{N_{att}})$.
13. **Individual Defender Reload Delay**: A single channel to represent the current reload delay of the individual defender for which the network must generate an action. This layer uses the same generation methodology as the respective *RBF (2D)* and *One-hot* representations in the *Defender Reload Delay* layer, however it only contains a single defender's information.
14. **Individual Defender Ammunition**: A single channel to represent the ammunition remaining for the individual defender for which the network must generate an action. This layer is the same as the *Defender Ammunition* layer, however it only contains a single defender's information.
15. **Individual Defender Detection**: A single channel to represent the detection zone for the individual defender for which the network must generate an action. This layer uses the same generation methodology as the respective *RBF (2D)* and *One-hot* representations in the *Defender Detection* layer, however it only contains a single defender's information.
16. **Individual Defender Interception**: A single channel to represent the interception zone for the individual defender for which the network must generate an action. This layer uses the same generation methodology as the respective *RBF (2D)* and *One-hot* representations in the *Defender Interception* layer, however it only contains a single defender's information.

17. **Individual Defender Opportunities**: A single channel to represent the current opportunities available for the individual defender for which the network must generate an action. This layer uses the same generation methodology as the respective *RBF (2D)* and *One-hot* representations in the *Defender Opportunities* layer, however it only contains a single defender's information.

The attacker actor, defender actor, and asymmetric critic input representations consist of subsets from this set of possible layers. These subsets are detailed in the following list, and are further visualized in Figures 4.3, 4.4, and 4.5.

Components of Each Network Input

- Symmetric Attacker Actor and Critic Input, Asymmetric Attacker Actor Input: [0, 3, 4, 7, 8, 9, 10, 11, 12] (Figure 4.3)
- Symmetric Defender Actor and Critic Input, Asymmetric Defender Actor Input: [0, 1, 2, 3, 4, 5, 6, 9, 10, 11, 13, 14, 15, 16, 17] (Figure 4.4)
- Asymmetric Attacker and Asymmetric Defender Critic Input: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12] (Figure 4.5)

4.4.2 Action Representation

Another key decision in terms of representation is that of the action space. The environment is inherently spatial and dynamic. A traditional dense representation could struggle to capture the nuances of this spatial aspect and fall short in effectively representing the state of the environment at any given time, since it doesn't inherently encode relative positions or spatial dependencies. Additionally, while we utilize a constant number of entities in our training, in reality this number could vary. In order to generalize to varying numbers of valid actions, an index-based action representation could not be naively utilized.

One option for encoding the inputs and outputs of the policy network to allow for generalization to scenarios with a varying number of valid actions would be to utilize a pointer network [86], where each input would be a feature vector representing a possible action, and the output would be a probability distribution over these inputs. However, utilizing this method would take the observation and action representation out of the topographical domain.

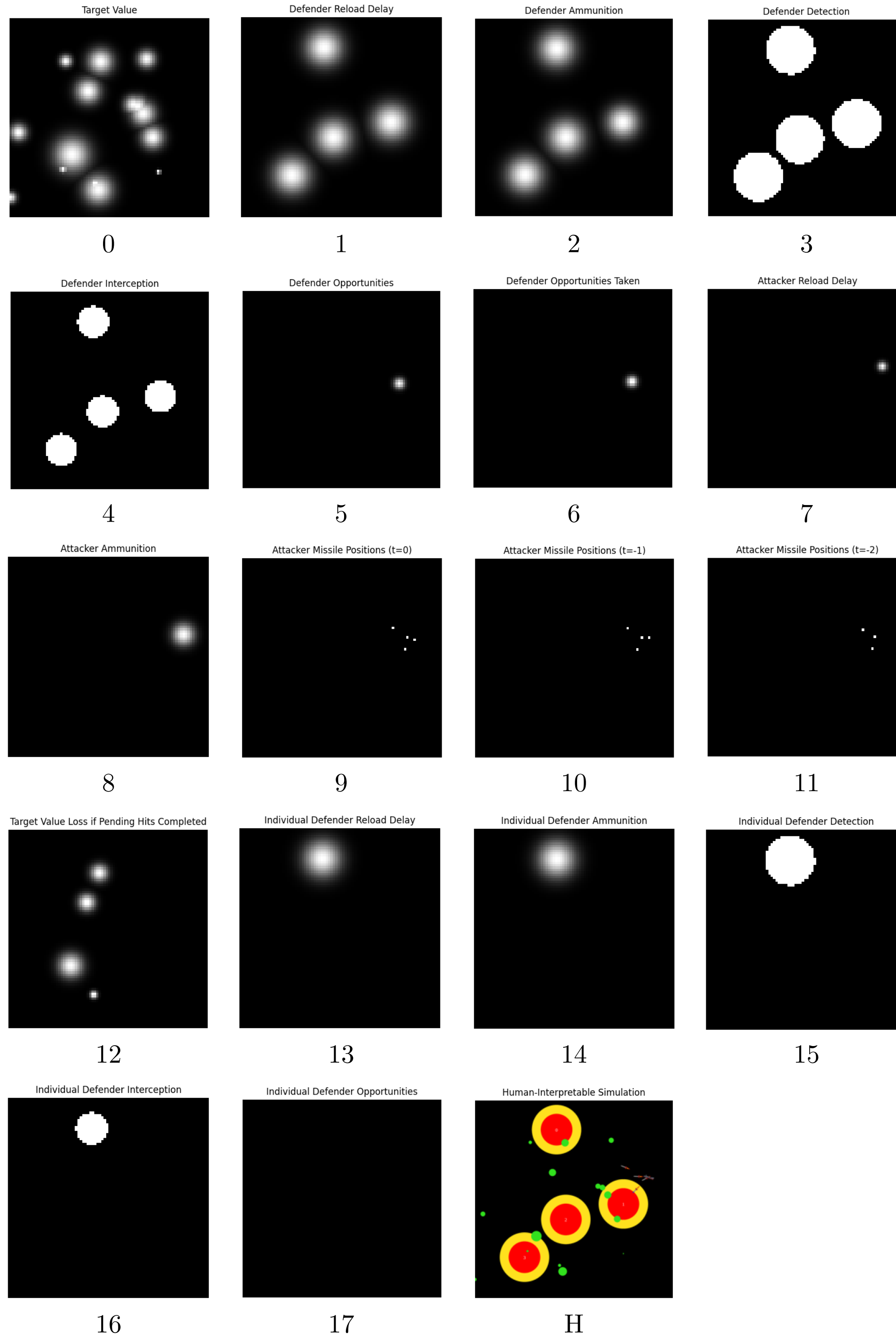


Figure 4.2: A full set of channels representing the *RBF (2D)* observation of an agent, with their corresponding indices shown below the image. The 'H' panel is not part of the observation, but represents the human-rendered representation of the environment.

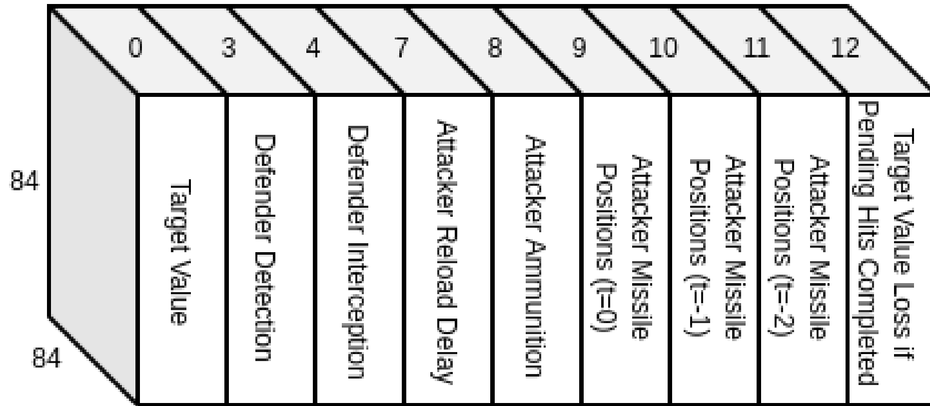


Figure 4.3: The observation representation, consisting of a 9x84x84 image, utilized as input to the actor and critic when training symmetric attackers, and as input to the actor only when training asymmetric attackers. The channels are numbered in accordance with the previously listed set of possible channels.

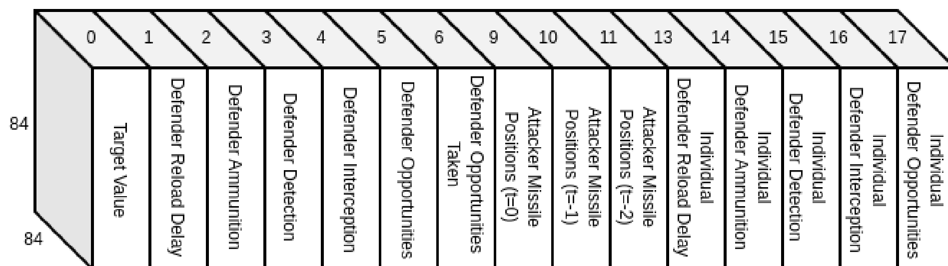


Figure 4.4: The observation representation, consisting of a 15x84x84 image, utilized as input to the actor and critic when training symmetric defenders, and as input to the actor only when training asymmetric defenders. The channels are numbered in accordance with the previously listed set of possible channels.

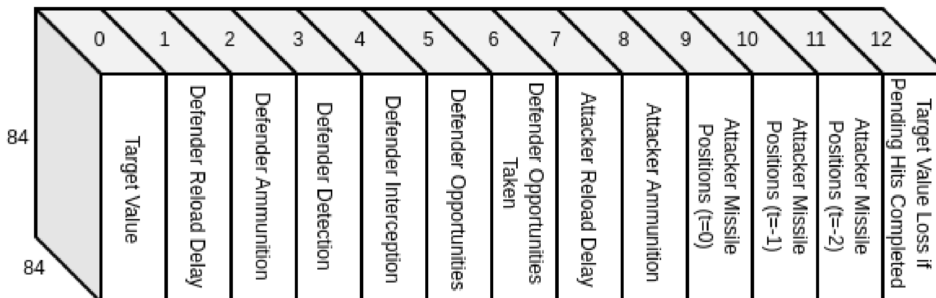


Figure 4.5: The observation representation, consisting of a 13x84x84 image, utilized as input to the critic when training asymmetric attackers and asymmetric defenders. The channels are numbered in accordance with the previously listed set of possible channels.

We instead choose to represent the action space as an image, similar to our input observation representation, resulting in action selection corresponding to a spatial action. By representing the action space as an image, in a similar manner to the observation space, this allows the agent to learn spatial dependencies directly, exploiting convolutional neural networks' proficiency in capturing local patterns in data. Ultimately, incorporating convolutional architectures' spatial bias and image-like representations enhances agent learning and generalization efficiency. The decision to utilize a spatial action space was made in part due to the success of a comparable architecture in the adversarial SC2LE [85]. Additionally, it allows us to maintain the spatial structure of the input, which encodes the valid actions directly as part of the input channels for both attacker and defender agents. While we can't say with certainty that a spatial observation and action space is the top performing agent configuration, a complete experimental analysis of the alternatives is beyond the scope of this thesis.

Aligned with the single-layer observation dimension of 84x84, we define the action space as a single 84x84 layer, where the pixels in this layer correspond to the action of firing at that location of the environment. In the case of the attacker, the agent can choose to fire at a target if they have remaining ammunition and are not reloading, or they can choose to do nothing and wait until the next timestep. The valid actions in pixel-space, therefore, are defined as one pixel in the location of each of the targets, along with one pixel at the location of the attacker itself to represent the *do-nothing* action.

Defenders, on the other hand, can either fire at an opportunity if present, assuming reload delay is zero and the defender has ammunition remaining,

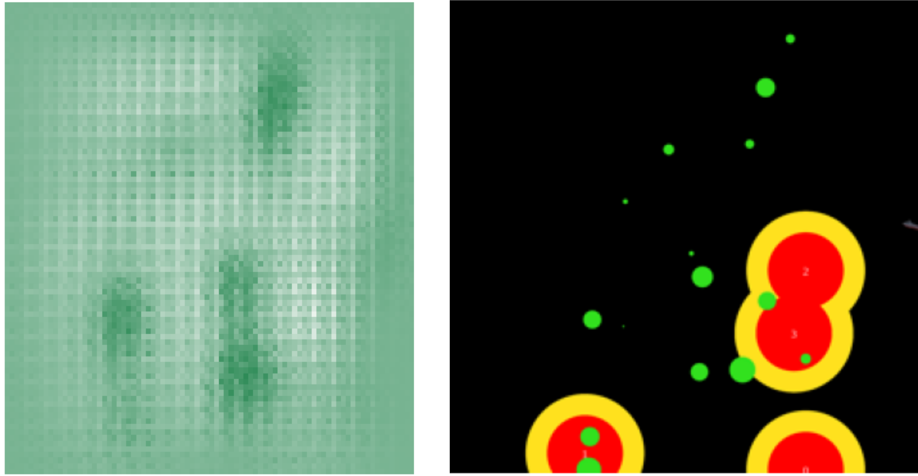


Figure 4.6: A sample of the output action logits, generated from a symmetric policy network, demonstrating the higher likelihood actions surrounding high-value target locations, as seen in the human-rendered representation of the environment on the right-hand side.

or they can do nothing. Hence, for each defender, the actions in pixel space are represented as one pixel for each interception opportunity available at the location of the forecast interception event, along with one pixel at the location of the defender to represent the *do-nothing* action.

In both attacker and defender cases, to generate a probability distribution over the action space, the logits output by the policy network are masked to replace invalid actions with large, negative values. By masking before selecting an action, we prevent outputs that would provide uninformative, high levels of negative error to the network when selecting invalid actions. Without masking, in any given timestep an agent would have mostly invalid actions to choose from. A sample of the environment state and action logits produced in the case of a trained attacker agent can be seen in Figure 4.6. A categorical distribution is created by applying *Softmax* activation to the masked logits, and action selection then proceeds by either sampling from the resulting distribution during training, or selecting the highest probability action during inference.

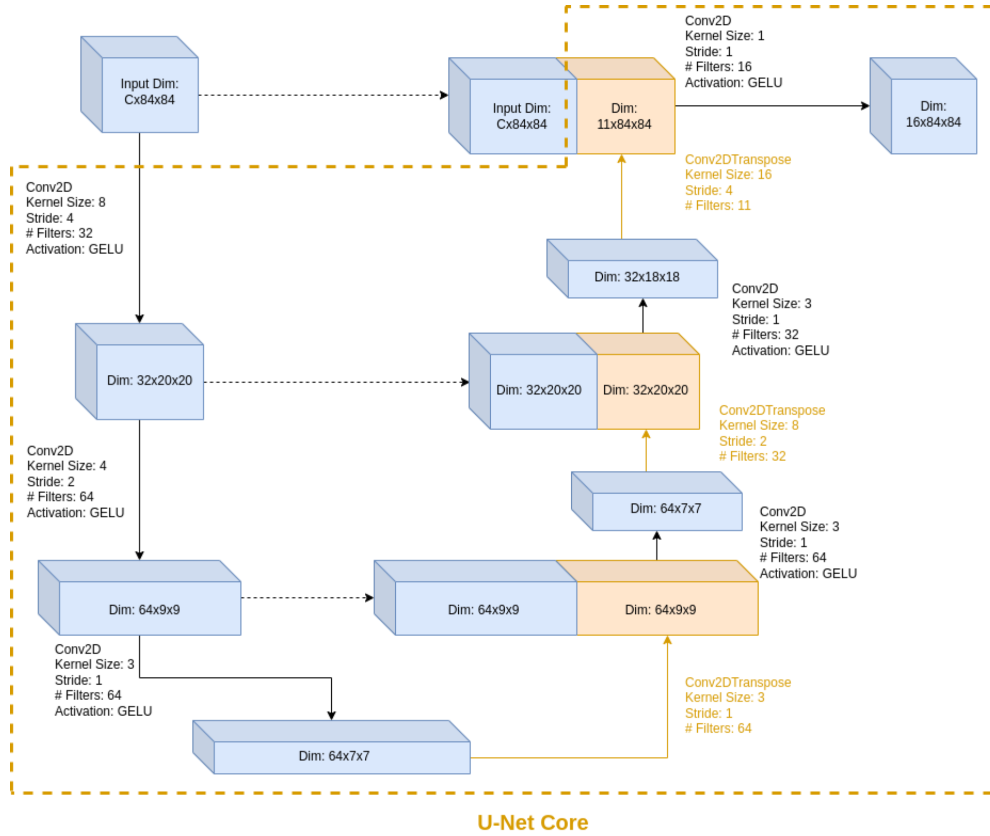


Figure 4.7: The U-Net feature extraction architecture utilized in all agent configurations.

4.4.3 Neural Network Architecture

In all experiments shown, the same core feature extraction architecture is utilized for both the actor and critic. This architecture is inspired by the popular U-Net architecture for image segmentation [63], and consists of a series of standard convolution layers which comprise the downward section of the U-Net, followed by a series of transpose convolution layers that comprise the upward section of the U-Net. There are skip-connections between the downward and upward portions of the U-Net, as the upward layers take as input the output of the corresponding tier from the downward section. In addition to these skip-connection inputs, the upward layers also take as input the output of the transpose convolution from the prior layer in the upward U-Net. This architecture is illustrated in Figure 4.7. GELU activation is

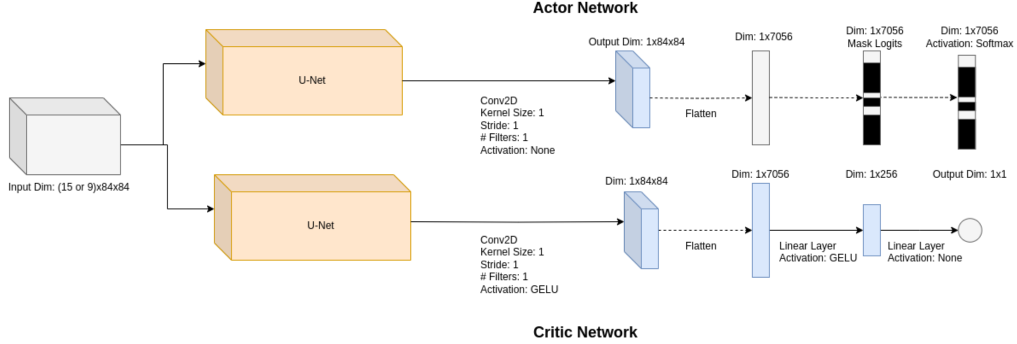


Figure 4.8: The actor-critic network architecture used in the configuration 0, 2, 4 and 6 CTDE PPO implementation, with entirely separate networks for the actor and critic, and **the same input** utilized for each. Attacker configurations 0 & 2 contain 2,684,825 trainable parameters, while defender configurations 4 & 6 contain 4,063,709 trainable parameters.

used throughout, with the exception of the output activation, where the actor networks utilize Softmax and the critic network utilizes no activation function. The decision to utilize GELU in these experiments was made to minimize the risk of the vanishing gradient and *Dying ReLU* problems, as discussed in Section 2.2.2.

As in the previous chapter, we are interested in comparing both symmetric and asymmetric architectures, due to the natural asymmetry of information in adversarial scenarios, along with the success of such architectures observed in the previous chapter. The symmetric and asymmetric architectures differ by the inputs utilized during training, as previously mentioned in Section 4.4.1.

In the case of the symmetric architecture, the actor and critic networks both receive the same representation. Each U-Net receives an observation as input, where the content of this observation is determined by whether the agent is controlling a defender or an attacker. The actor network then flattens the 84x84 dimensional output, and uses the resulting 7056 dimension tensor as the logits, which are then masked and utilized to create a categorical distribution with *Softmax* activation on the masked logits. The critic network applies GELU activation to the the output of the U-Net, and then flattens the 84x84 dimensional output. The resulting 7056 dimension output of this operation is then connected to a linear layer of dimension 256 with GELU activation, prior to being connected to a single output node, without activation, corresponding to the predicted value. This agent's architecture is shown in Figure 4.8.

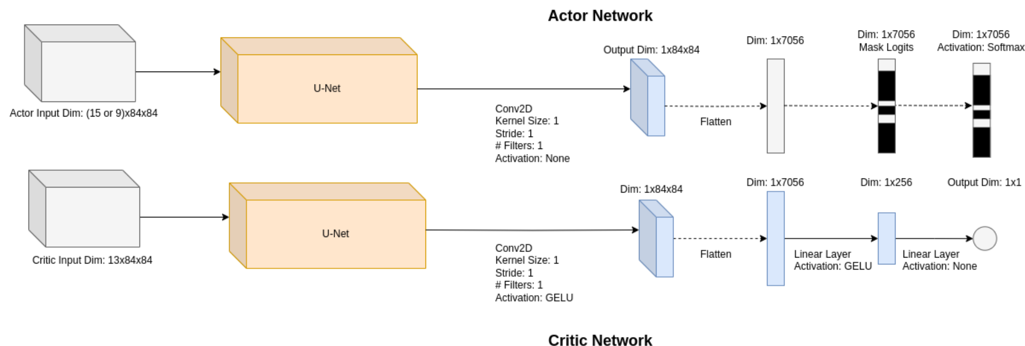


Figure 4.9: The actor-critic network architecture used in the configuration 1, 3, 5 and 7 CTDE PPO implementation, with entirely separate networks for the actor and critic, and **asymmetric input** representations utilized for the actor and critic. Attacker configurations 1 & 3 contain 2,693,081 trainable parameters, while defender configurations 5 & 7 contain 4,059,581 trainable parameters.

The asymmetric architecture, on the other hand, utilizes entirely separate actor and critic network inputs. The asymmetric architecture uses the same U-Net architecture for the actor and critic networks as described for the symmetric architecture, however the critic network receives additional information pertaining to the state of the environment, as detailed in Figure 4.5. This agent's architecture is shown in Figure 4.9, highlighting the different inputs to the actor and critic networks.

In the case of training defender agents, since we opt to test both the CTDE and CTCE training and execution paradigms. When utilizing the CTDE paradigm, each defender agent present in the environment has its own actor network. So, the actor network shown at the top of Figures 4.8 and 4.9 are duplicated 4 times for each of the 4 defender agents in the environment. In the case of attackers and CTCE defenders, just one actor network is utilized.

In our experiments with the missile defence environment, we once again trained our agents using PPO. The training process involved a total of 1,024,000 timesteps, where each rollout consisted of 25,600 timesteps, for a total of 40 training iterations. We used a minibatch size of 512. The discount factor γ was set to 1.0, ensuring equal importance to immediate and future rewards, since agents should be agnostic to the time of experiencing a reward or penalty. We employed the Generalized Advantage Estimation (GAE) with a lambda value of 0.95 to estimate advantages. The value loss coefficient c_1 was set to 1.0, emphasizing the importance of the value function in the overall loss,

Table 4.1: Hyperparameters used with PPO for the missile defence environment experiments

Hyperparameter	Value
Learning Rate	1e-4
Rollout Timesteps, T	25600
Total Timesteps	1024000
Minibatch Size, M	512
Iterations, N	40
Epochs, K	10
Discount Factor, γ	1.0
GAE Lambda, λ	0.95
Value Loss Coefficient, c_1	1.0
Entropy Loss Coefficient, c_2	0.01
Clipping Ratio, ε	0.2

while the entropy loss coefficient c_2 was set to 0.01 to encourage exploration. To control the update step size, we used a clipping ratio ε of 0.2.

Regarding the learning rate, we used a constant value of 1×10^{-4}. Although the experiments from MPE may not be indicative of performance in the missile environment, we opted to utilize this learning rate based on the performance of agents trained in the MPE experiments. In total, we trained five agents for each configuration, with hyperparameters kept constant to those listed in Table 4.1 across all training runs.

Regarding computational resource demands for training, there were observable disparities based on specific configurations. Defenders typically required a training duration spanning between 7 to 14 hours, while attackers, on the other hand, demanded shorter intervals, generally ranging from 3 to 6 hours. Training was executed on a Tesla V100 GPU, housed in a DGX system. At inference time, empirical evaluations of the networks found inference times on a single timestep consistently below 50 milliseconds. Inference operations were carried out on a RTX 3070 GPU. Such rapid processing allows for its application in real-world settings, as a decision support system would necessitate swift responses. However, it's pivotal to highlight that these runtimes are contingent on the hardware available.

4.5 Experiments & Results

4.5.1 Experiment Background

In Chapter 3.2, utilizing the simple MountainCar environment, we demonstrated the apparent correlation between observation resolution and resultant agent performance, with particularly high performance found when utilizing a 2-dimensional RBF transformation of the state. Additionally, we demonstrated the merit of utilizing an asymmetric architecture to enable the training of an effective actor on a poorer performing state representation by providing a higher performing representation as input to the critic during training. Subsequently, in Chapter 3.3, we demonstrated the merit of utilizing a centralized training with decentralized execution (CTDE) training paradigm in a multi-agent setting.

These findings guided the experiment configurations detailed in Tables 4.2 and 4.3, where the former details whether the configuration is attacking or defending, the training paradigm used, and the network architecture. The latter provides details on the input observation, both in terms of the encoding type as either *RBF (2D)* or *One-hot*, and the symmetry or asymmetry of the actor and critic network inputs.

Our decision to focus on the comparison of *RBF (2D)* and *One-hot* representation types is based on the results from Chapter 3. While *RBF (2D)* was shown to perform well, we wish to confirm whether these results hold in the more complex missile environment by comparing performance to agents trained using *One-hot* representations, which had inferior results in the prior experiments. We opt for these two representations as they both are image-based, which allows us to effectively design networks taking these representations as input for the image-based action space output.

The symmetric and asymmetric comparison is also chosen for experiments in this section due to the results seen using this method in Chapter 3, and the natural asymmetry of information in the adversarial missile environment. In a real-world scenario, it is likely that the attacking and defending agents would only have partial observability of the environment state. The agents might not have full observability of their adversary's state during inference. However, during training, it might be useful for the opposing agents to have their critics receive a more complete representation of state, while their actor receives the limited representation. This chapter aims to use the lessons learned from Chapter 3 to train effective attacking and defending agents in our missile defence scenario, while evaluating the merit of utilizing asymmetric architectures in the training procedure.

Table 4.2: Agent configurations with figure reference: Type, Training Paradigm, and Network Architecture.

Configuration	Agent Type	Training Paradigm	Network Architecture
0	Attacker	Single-agent	Figure 4.8
1	Attacker	Single-agent	Figure 4.9
2	Attacker	Single-agent	Figure 4.8
3	Attacker	Single-agent	Figure 4.9
4	Defender	CTDE	Figure 4.8 (With 4 Actor Networks)
5	Defender	CTDE	Figure 4.9 (With 4 Actor Networks)
6	Defender	CTDE	Figure 4.8 (With 4 Actor Networks)
7	Defender	CTDE	Figure 4.9 (With 4 Actor Networks)
8	Defender	CTCE	Figure 4.9 (With 1 Actor Network)
9	Defender	CTCE	Figure 4.8 (With 1 Actor Network)
10	Defender	CTCE	Figure 4.9 (With 1 Actor Network)
11	Defender	CTCE	Figure 4.8 (With 1 Actor Network)

Table 4.3: Agent configurations with figure reference: Observation.

Configuration	Observation Type	Actor Input	Critic Input	Symmetric? (Y/N)
0	RBF (2D)	Figure 4.3	Figure 4.3	Y
1	RBF (2D)	Figure 4.3	Figure 4.5	N
2	One-hot	Figure 4.3	Figure 4.3	Y
3	One-hot	Figure 4.3	Figure 4.5	N
4	RBF (2D)	Figure 4.4	Figure 4.4	Y
5	RBF (2D)	Figure 4.4	Figure 4.5	N
6	One-hot	Figure 4.4	Figure 4.4	Y
7	One-hot	Figure 4.4	Figure 4.5	N
8	One-hot	Figure 4.4	Figure 4.5	N
9	One-hot	Figure 4.4	Figure 4.4	Y
10	RBF (2D)	Figure 4.4	Figure 4.5	N
11	RBF (2D)	Figure 4.4	Figure 4.4	Y

To assess the effectiveness of our trained agents, we generate a set of 100 test scenarios. Agent performance will be compared directly against agent and baseline adversaries on this constant set of test scenarios. By using the same 100 scenarios for comparison, we minimize the impact of the high variance in episodic reward when utilizing randomly initialized scenarios on the analysis of results.

4.5.2 Results & Discussion

Symmetric and asymmetric attacker configurations were trained against a *Greedy* defender for 1,024,000 timesteps. This adversary was selected due to it being the middle performer amongst the defender baselines, as shown in the *Greedy* defender row of Table 4.5. Similarly, symmetric and asymmetric

Table 4.4: Agent configurations with shorthand reference.

Configuration	Agent Type	Training Paradigm	Observation Type	Symmetric? (Y/N)	Shorthand
0	Attacker	Single-agent	RBF (2D)	Y	SA-RBF-SYM
1	Attacker	Single-agent	RBF (2D)	N	SA-RBF-ASYM
2	Attacker	Single-agent	One-hot	Y	SA-OH-SYM
3	Attacker	Single-agent	One-hot	N	SA-OH-ASYM
4	Defender	CTDE	RBF (2D)	Y	CTDE-RBF-SYM
5	Defender	CTDE	RBF (2D)	N	CTDE-RBF-ASYM
6	Defender	CTDE	One-hot	Y	CTDE-OH-SYM
7	Defender	CTDE	One-hot	N	CTDE-OH-ASYM
8	Defender	CTCE	One-hot	N	CTCE-OH-ASYM
9	Defender	CTCE	One-hot	Y	CTCE-OH-SYM
10	Defender	CTCE	RBF (2D)	N	CTCE-RBF-ASYM
11	Defender	CTCE	RBF (2D)	Y	CTCE-RBF-SYM

defender configurations were trained against a *Random* attacker for 1,024,000 timesteps. This attacker baseline was selected due to it being the middle performer amongst the attacker baselines. The total loss, as defined in Equation 2.13, over the course of each training regime can be seen in the left-hand plots of Figures 4.10 and 4.11. Furthermore, the mean episodic reward over the course of training can be seen in the right-hand plots of these same figures.

The trained agents were then used for inference against the baseline *Greedy*, *SLS-Greedy* and *Random* defenders. Additionally, they were used for inference against each other. The baseline agents were also placed against each other for comparison. The mean and standard deviation of the reward obtained by the attacking team in the 100-scenario test set can be seen in Table 4.5. Note that instead of configuration numbers, we utilize shorthand, as defined in Table 4.4, to define each agent configuration in the result tables to make configuration variations more readily interpretable.

In addition to calculation of the standard deviation on a scenario basis, we also report the standard deviation of the mean performance on the set of 100 test scenarios of each of the 5 trained agents. These results are reported in Table 4.6. It can be seen by comparing the standard deviations reported in Tables 4.5 and 4.6 that agent performance between scenarios is of high variance, while performance between agents has much lower variance.

In the case of the defender baselines, the *SLS-Greedy* behavior, as seen in the first row in Table 4.5, was found to generally yield the best results, as evidenced by the relatively low values present in this row. Utilizing this strategy, wastage of ammunition is minimized. This is particularly expected to be the case due to the simulation parameters utilized. The combination of the interceptor missile speed being much greater than the cruise missile speed, along with the short reload delay of defenders, leads to defenders having multiple opportunities to fire at a single missile before an initial opportunity taken is concluded. Since the probability of an interception opportunity succeeding

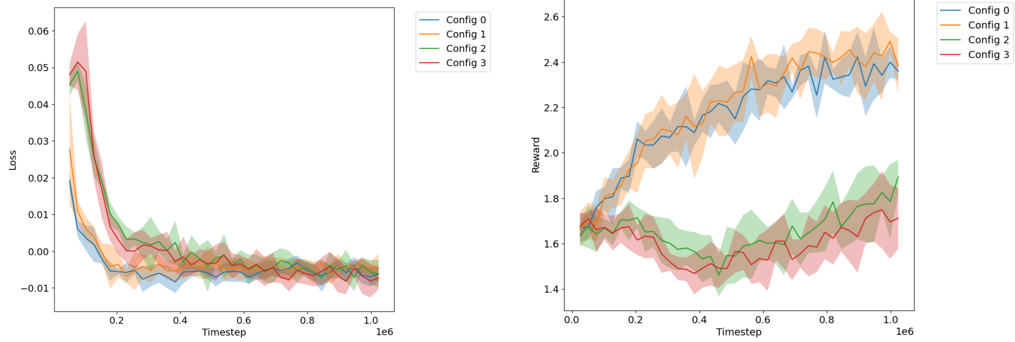


Figure 4.10: The train loss (left) and mean episodic reward (right) for attacker agents over 1,024,000 timesteps of training.

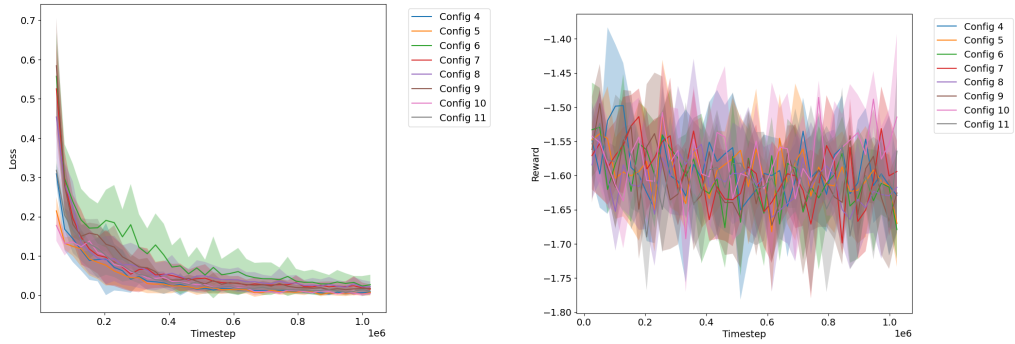


Figure 4.11: The train loss (left) and mean episodic reward (right) for defender agents over 1,024,000 timesteps of training.

once taken is 100%, firing multiple interceptor missiles at a single attacker missile is a wasteful, non-optimal policy. This is further demonstrated when looking at the performance of the *Greedy* defender baseline in the third row of the table, where poor performance can empirically be found to originate from scenarios where defenders run out of ammunition intercepting attacker missiles in the early stages of an episode, so are unable to intercept missiles in later stages of the episode. The worst performing defender baseline, however, can be seen in the *Random* results in the second row of the table, where attacker missiles are in many cases allowed to pass through defender interception zones unhindered.

Regarding the defence strategies, neural network defenders outperformed the *Random* and *Greedy* baselines, but were consistently outperformed by

Table 4.5: Adversarial performance of attacker and defender policies over a constant 100-member set of test scenarios. All agent configurations are trained 5 times to produce the mean and standard deviations shown, with the standard deviation being calculated over all 5×100 test scenario performances for the given agent configuration. For the attackers, a higher mean represents better performance, while for the defenders, a lower mean represents better performance.

	Attacker Mode						
Defender Mode	SLS-Greedy	Random	Greedy	SA-RBF-SYM	SA-RBF-ASYM	SA-OH-SYM	SA-OH-ASYM
SLS-Greedy	0.99 ± 1.04	1.21 ± 0.81	0.67 ± 0.38	1.73 ± 1.11	1.73 ± 1.10	1.41 ± 0.93	1.30 ± 0.85
Random	2.67 ± 1.19	1.90 ± 0.70	0.93 ± 0.21	2.50 ± 0.86	2.57 ± 0.85	2.04 ± 0.77	1.89 ± 0.71
Greedy	2.17 ± 0.95	1.41 ± 0.72	0.92 ± 0.19	2.06 ± 0.90	2.09 ± 0.91	1.65 ± 0.79	1.51 ± 0.76
CTDE-RBF-SYM	1.73 ± 1.13	1.36 ± 0.74	0.83 ± 0.32	1.94 ± 0.95	1.97 ± 1.00	1.54 ± 0.84	1.45 ± 0.80
CTDE-RBF-ASYM	1.71 ± 1.16	1.33 ± 0.76	0.82 ± 0.33	1.92 ± 1.00	1.96 ± 0.99	1.56 ± 0.85	1.46 ± 0.78
CTDE-OH-SYM	1.72 ± 1.16	1.37 ± 0.77	0.82 ± 0.32	1.98 ± 0.98	2.01 ± 0.97	1.57 ± 0.82	1.47 ± 0.78
CTDE-OH-ASYM	1.71 ± 1.13	1.33 ± 0.77	0.82 ± 0.32	1.94 ± 0.97	1.97 ± 0.99	1.57 ± 0.85	1.44 ± 0.79
CTCE-OH-ASYM	1.71 ± 1.13	1.30 ± 0.78	0.82 ± 0.32	1.92 ± 1.00	1.97 ± 1.00	1.54 ± 0.82	1.43 ± 0.77
CTCE-OH-SYM	1.72 ± 1.13	1.36 ± 0.78	0.82 ± 0.31	1.98 ± 0.99	2.00 ± 0.98	1.58 ± 0.85	1.44 ± 0.79
CTCE-RBF-ASYM	1.60 ± 1.16	1.34 ± 0.77	0.80 ± 0.34	1.90 ± 1.00	1.93 ± 1.00	1.53 ± 0.86	1.43 ± 0.80
CTCE-RBF-SYM	1.72 ± 1.15	1.34 ± 0.77	0.82 ± 0.33	1.94 ± 0.98	1.96 ± 1.00	1.54 ± 0.83	1.43 ± 0.78

Table 4.6: Adversarial performance of attacker and defender policies over a constant 100-member set of test scenarios. All agent configurations are trained 5 times to produce the mean and standard deviations shown, with the standard deviation being calculated from the means of the 5 iterations of test scenario performances for the given agent configuration. For the attackers, a higher mean represents better performance, while for the defenders, a lower mean represents better performance.

	Attacker Mode						
Defender Mode	SLS-Greedy	Random	Greedy	SA-RBF-SYM	SA-RBF-ASYM	SA-OH-SYM	SA-OH-ASYM
SLS-Greedy	0.99 ± 0.00	1.21 ± 0.08	0.67 ± 0.00	1.73 ± 0.06	1.73 ± 0.06	1.41 ± 0.08	1.30 ± 0.18
Random	2.67 ± 0.04	1.90 ± 0.04	0.93 ± 0.01	2.50 ± 0.09	2.57 ± 0.05	2.04 ± 0.16	1.89 ± 0.22
Greedy	2.17 ± 0.00	1.41 ± 0.05	0.92 ± 0.00	2.06 ± 0.08	2.09 ± 0.08	1.65 ± 0.09	1.51 ± 0.18
CTDE-RBF-SYM	1.73 ± 0.05	1.36 ± 0.04	0.83 ± 0.02	1.94 ± 0.07	1.97 ± 0.09	1.54 ± 0.1	1.45 ± 0.19
CTDE-RBF-ASYM	1.71 ± 0.04	1.33 ± 0.03	0.82 ± 0.02	1.92 ± 0.06	1.96 ± 0.08	1.56 ± 0.09	1.46 ± 0.19
CTDE-OH-SYM	1.72 ± 0.09	1.37 ± 0.05	0.82 ± 0.02	1.98 ± 0.05	2.01 ± 0.11	1.57 ± 0.08	1.47 ± 0.18
CTDE-OH-ASYM	1.71 ± 0.04	1.33 ± 0.04	0.82 ± 0.01	1.94 ± 0.08	1.97 ± 0.06	1.57 ± 0.12	1.44 ± 0.17
CTCE-OH-ASYM	1.71 ± 0.03	1.30 ± 0.02	0.82 ± 0.01	1.92 ± 0.07	1.97 ± 0.06	1.54 ± 0.09	1.43 ± 0.19
CTCE-OH-SYM	1.72 ± 0.04	1.36 ± 0.07	0.82 ± 0.01	1.98 ± 0.06	2.00 ± 0.07	1.58 ± 0.09	1.44 ± 0.15
CTCE-RBF-ASYM	1.60 ± 0.04	1.34 ± 0.05	0.80 ± 0.01	1.90 ± 0.07	1.93 ± 0.09	1.53 ± 0.13	1.43 ± 0.19
CTCE-RBF-SYM	1.72 ± 0.05	1.34 ± 0.03	0.82 ± 0.02	1.94 ± 0.06	1.96 ± 0.06	1.54 ± 0.09	1.43 ± 0.2

SLS-Greedy. The best performing defender can be seen in the *CTCE-RBF-ASYM* configuration row, as this agent has the lowest mean penalty of all defender training configurations when playing against 5 of the 7 possible attacker adversaries. Additionally, it is tied with *CTCE-OH-ASYM* and *CTCE-*

RBF-SYM configurations as the top performer against *SA-OH-ASYM* configuration attackers. That said, the differences in performance of *CTCE-RBF-ASYM* configuration to the next highest performing defender agent in each given column are not significant in most cases. The only column with a difference greater than 3% between the top two neural network defender agents is that of the *SLS-Greedy* attacker, where the *CTCE-RBF-ASYM* configuration has a mean approximately 6.9% lower than the next best configuration.

Overall, for defender agents, the representation being *One-hot* or *RBF (2D)*, the symmetry or asymmetry of observations used as input to the actor and critic networks, and the use of a CTDE or CTCE paradigm were not found to play a significant role in agent learning. While the performance differences were significant in the attacker case, they are fairly homogeneous in the defender case. When looking at the loss curve for the defenders in Figure 4.11, it can be seen to be decreasing through training. However, when looking at the mean episodic reward curves in the same figure, agent performance improvement is not evident. We believe this to be a result of the high variance in scenario initial conditions, resulting in high variance in mean episodic reward achievable. Because of this, defenders quickly learn to take opportunities when available, akin to a *Greedy* policy, regardless of observation.

When looking at the performances of the attacker baselines, *SLS-Greedy* can generally be seen to perform the best, as evidenced by the higher average values in that column. A notable exception can be seen when the *SLS-Greedy* attacker is facing an *SLS-Greedy* defender, where the baseline does not perform. The worst performing attacker baseline, against every defender adversary, can be seen in the *Greedy* column. This baseline empirically is found to dedicate multiple missiles to a small number of targets, resulting in diminishing reward for multiple impacts on the same targets.

The neural network attackers utilizing *RBF (2D)* representations, while performing slightly worse than the *SLS-Greedy* attacker against *Random* and *Greedy* baseline defenders, demonstrated superior performance against all other adversaries. Moreover, asymmetric experiment configurations, where the critic receives a more complete state representation than the actor, yielded marginally higher-performing attacking agents at inference time in the *RBF (2D)* representation case. They were, however, worse performing in the *One-hot* representation case. The attacker agents trained utilizing *RBF (2D)* representations, *SA-RBF-SYM* and *SA-RBF-ASYM* configurations, were on average found to perform 0.47 points higher than their *One-hot* counterparts.

Of final note is that with the exception of results in the *Random* column and *Greedy* row of Tables 4.5 and 4.6, all results reported are out-of-distribution, as the agents are trained against constant adversary policies. It

is expected that performance of agents would be higher should the adversary that the agent was trained against be in-distribution in all cases.

4.6 Conclusion

The objective of this chapter was to first develop a simulation environment conducive for the investigation of DRL methods in a missile defence task, and then to develop DRL agents and analyze their performance against our environment baselines.

We analyzed a range of DRL agents of varying architecture and input representation in our constructed missile defence scenario, through comparison of agent performance to the selection of hard-coded baselines. Following this evaluation, for defence strategies, *SLS-Greedy* emerged as the superior baseline. Neural network defenders were found to outperform the other baselines, however did not outperform *SLS-Greedy*. Of the 8 defender configurations tested, the *CTCE-RBF-ASYM* configuration defender agent utilizing an *RBF (2D)* state representation, asymmetric inputs to the actor and critic, and *CTCE* training and execution paradigm was found to perform marginally better against certain adversaries. However, no significant performance difference was observed amongst the configurations.

The highest performing attacker baseline was found to be *SLS-Greedy*. Similarly, the neural network attackers performed slightly worse than the *SLS-Greedy*, however outperformed the *Random* and *Greedy* baselines. Of note is that the asymmetric experiment configuration, where the critic receives a more complete state representation than the actor, yields higher-performing agents at inference time.

In our experiments on varying input representation in this chapter, we further demonstrated the merit of utilizing *RBF (2D)* representations. In the attacker agent case, these representations were found to once again outperform a comparable neural network architecture trained utilizing a *One-hot* representation of the state.

The architectures and training methodologies utilized demonstrated promise in their ability to create performant agents, however future work could look at different avenues to augment this performance. The addition of model-based methods, perhaps through Monte Carlo Tree Search (MCTS) being utilized in conjunction with the neural network [74, 72], might augment the neural network performance. Additionally, due to the lack of discernible difference in defender performance with varying configuration, more research is warranted to improve the performance of these agents. Effort should

be made to better understand the cause of the relatively homogeneous performance amongst defender agents. This could include experiments adjusting the frame of reference to be agent-centered in a similar manner to the MPE observation encoding, and experiments with adversarial examples where a policy of always firing at an opportunity is non-performant. The final recommendation for future work is to look at observation and action space representations that were omitted in this thesis. While we focused on spatial observation and action spaces, alternative approaches utilizing pointer networks, or attention mechanisms, to represent entities and actions warrant exploration.

5 Summary and Conclusions

5.1 Summary

In this study, we conducted an investigation into the impact of different state representations, corresponding neural network architectures, and multi-agent training and execution paradigms on agent learning, with the objective of training effective agents in a high-dimensional, multi-agent missile defence simulation environment. To explore this, we employed the MountainCar environment as our initial testbed. Our findings revealed that employing a *RBF (2D)* transformation of the default representation, coupled with a CNN, resulted in exceptional performance and convergence characteristics.

Furthermore, in the *Human-img* experiments of Chapter 3.2 we examined the potential of lower resolution representations with higher levels of noise. Such representations were experimentally found to hinder agents from learning effective policies when trained directly. However, we discovered that by utilizing an asymmetric architecture, where a higher resolution representation is utilized as input to the critic network as is the case in the *Hybrid* experiments, these lower resolution representations could still be effectively trained. This novel approach enabled the agents to overcome the limitations of lower resolution representations and achieve notable performance in the MountainCar environment.

Building upon our insights from the MountainCar experiments, we extended our research to a higher-dimensional, multi-agent particle environment. Here, we leveraged the most successful representations we had identified in the MountainCar environment—the *RBF (1D)*, *RBF (2D)* and *Raw-continuous* representations. However, in this new setting, we expanded our experimentation to include aspects of the multi-agent training paradigm. Specifically, we compared the performance of agents trained using a centralized versus a decentralized training paradigm. After thorough analysis, we consistently observed that the decentralized paradigm outperformed the centralized paradigm in terms of agent performance. This finding confirms the advantages of decen-

tralized execution in multi-agent settings, aligning with findings in literature [47, 97, 43, 55].

Lastly, we introduced a custom environment, which was specifically designed to explore the application of deep reinforcement learning (DRL) methods in a missile defence scenario. Leveraging the knowledge gained from our previous experiments in the MountainCar and multi-agent particle environments, we integrated our prior learnings into the training architecture for this new environment. By doing so, we aimed to showcase the merits of our trained agents in comparison to a selection of hard-coded baseline agents. Through a comprehensive performance evaluation, we demonstrated the effectiveness of our trained agents in tackling the challenges posed by the missile defence environment, further validating the effectiveness of our approach.

5.2 Limitations and Future Work

While our study on different state representations and agent learning yielded promising results, there are several limitations that should be acknowledged. Firstly, our initial experiments were primarily focused on the MountainCar environment and the higher-dimensional, multi-agent particle environment. Although these environments are representative of certain types of problems, they may not fully capture the complexity and diversity of real-world scenarios. Therefore, the generalizability of our findings to other domains should be approached with caution.

Secondly, our investigation mainly concentrated on the effects of state representations, with primary focus on the default *Raw-continuous* compared against *One-hot* and RBF-based representations. While these representations demonstrated consistent performance in our experiments, there may exist other state representations or transformations that could yield even better results. Exploring a wider range of state representations could provide additional insights and potentially uncover superior approaches.

Another notable aspect is that our experiments did not utilize communication between agents in both the MPE and missile defence environment. Communication, especially in multi-agent systems, can significantly influence agent performance and learning dynamics. Introducing inter-agent communication might allow agents to share valuable information, make collective decisions, and adapt to the environment more effectively. Future work should consider experimenting with various communication mechanisms to determine their impact on agent performance and cooperation.

Another pivotal assumption underpinning our experiments in the missile

environment was the utilization of a state observation designed to encapsulate information deemed pertinent to an agent's action selection. Yet, it's important to recognize that in real-world scenarios, the same level of information might not be accessible. As such, alternative methods might be worth exploration. For example, an LSTM or recurrence-based approach might allow agents to derive insights from intra-episode feedback by integrating observations and events from various timesteps into the current state. This might provide the agent with an intrinsic memory of past events, potentially filling the void of missing immediate observations. Another analytical approach worth exploring could be the direct updating of state observations from an agent's accumulated experience. The uncertainty on location could then be encoded using the spread of an RBF centered at the estimated location. For instance, by maintaining a record of locations where attacker missiles are intercepted, agents can adapt their strategies, even when lacking real-time defender location data. Both methodologies offer prospects for enhancing agent efficacy in less-than-ideal informational landscapes, and may be worth exploration to enhance the real-world applicability of the agents trained in the missile defence environment.

Additionally, our study focused on the comparison of centralized and decentralized training paradigms in the context of the multi-agent particle environment. While we consistently observed superior performance with the decentralized paradigm, it is important to note that the effectiveness of training paradigms can heavily depend on the specific problem and environment. This is evidenced by the homogeneous performance of CTCE and CTDE agents in the missile defence environment, despite the outperformance of CTDE agents in the MPE. Different scenarios may require tailored training approaches, and the decentralized paradigm may not always be the optimal choice. Further investigation is needed to explore the factors that influence the suitability of different training paradigms across various domains.

Furthermore, our evaluation of agent performance in the missile defence environment was primarily conducted through a comparison with hard-coded baseline agents. While this approach provides a benchmark for assessing the effectiveness of our trained agents, it does not capture the full spectrum of potential performance levels. Additional evaluations against other state-of-the-art DRL methods or human expert performance would offer a more comprehensive assessment of our agents' capabilities.

Lastly, our study did not extensively address the computational and resource requirements associated with training agents using different state representations and training paradigms. Training agents with higher resolution representations or employing decentralized paradigms may demand increased

computational power and longer training times. These practical constraints should be taken into account when considering the scalability and real-world applicability of our approaches.

In conclusion, while our study sheds light on the benefits of different state representations and decentralized training paradigms, it is important to recognize the limitations inherent in our research. Further exploration in diverse environments, consideration of alternative state representations, investigation of domain-specific training paradigms, comprehensive performance evaluations, and addressing practical constraints will contribute to a more comprehensive understanding of agent learning and its applicability in real-world scenarios.

Bibliography

[1] Akshat Agarwal, Sumit Kumar, Katia Sycara, and Michael Lewis. Learning transferable cooperative behavior in multi-agent teams. In *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '20, page 1741–1743, Richland, SC, 2020. International Foundation for Autonomous Agents and Multiagent Systems.

[2] Adrian K. Agogino and Kagan Tumer. Unifying temporal and structural credit assignment problems. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '04, page 980–987, USA, 2004. IEEE Computer Society.

[3] James S. Albus. A theory of cerebellar function. *Mathematical Biosciences*, 10(1):25–61, 1971.

[4] Ankesh Anand, Evan Racah, Sherjil Ozair, Yoshua Bengio, Marc-Alexandre Côté, and R Devon Hjelm. Unsupervised state representation learning in atari, 2020.

[5] Grigory Antipov, Sid-Ahmed Berrani, Natacha Ruchaud, and Jean-Luc Dugelay. Learned vs. hand-crafted features for pedestrian gender recognition. In *Proceedings of the 23rd ACM International Conference on Multimedia*, MM '15, page 1263–1266, New York, NY, USA, 2015. Association for Computing Machinery.

[6] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, Nov 2017.

[7] Umit Ayvaz, Murat Dere, and Yao Tiah. Using the mana agent-based simulation tool to evaluate and compare the effectiveness of ground-based and airborne communications jammers in countering the ied threat to ground convoys. pages 113–118, 01 2007.

[8] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846, 1983.

[9] Mohamed Ishmael Belghazi, Aristide Baratin, Sai Rajeswar, Sherjil Ozair, Yoshua Bengio, Aaron Courville, and R Devon Hjelm. Mine: Mutual information neural estimation, 2021.

[10] Richard Bellman. A markovian decision process. *Indiana Univ. Math. J.*, 6:679–684, 1957.

[11] Daniel S Bernstein, Shlomo Zilberstein, and Neil Immerman. The complexity of decentralized control of markov decision processes, 2013.

[12] John A. Bogovic, Gary B. Huang, and Viren Jain. Learned versus hand-designed feature representations for 3d agglomeration, 2013.

[13] Steven Bohez, Tim Verbelen, Elias De Coninck, Bert Vankeirsbilck, Pieter Simoens, and Bart Dhoedt. Sensor fusion for robot control through deep reinforcement learning, 2017.

[14] Anne-Claire Boury-Brisset and Jean Berger. Benefits and challenges of ai/ml in support of intelligence and targeting in hybrid military operations. *IST-190 Research Symposium (RSY) on Artificial Intelligence, Machine Learning and Big Data for Hybrid Military Operations (AI4HMO)*, Oct 2021.

[15] David Broomhead and David Lowe. Multi-variable functional interpolation and adaptive networks. *ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (UNITED KINGDOM)*, RSRE-MEMO-4148, 03 1988.

[16] Lucian Busoniu, Robert Babuska, and Bart De Schutter. A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 38(2):156–172, 2008.

[17] Lucian Buşoniu, Robert Babuška, and Bart De Schutter. Multi-agent reinforcement learning: An overview. *Innovations in Multi-Agent Systems and Applications - 1 Studies in Computational Intelligence*, page 183–221, 2010.

[18] Wendelin Böhmer, Jost Springenberg, Joschka Boedecker, Martin Riedmiller, and Klaus Obermayer. Autonomous learning of state representations for control: An emerging field aims to autonomously learn state

representations for reinforcement learning agents from their real-world sensor observations. *KI - Künstliche Intelligenz*, 29, 03 2015.

[19] Yu-han Chang, Tracey Ho, and Leslie Kaelbling. All learning is local: Multi-agent learning in global reward games. In S. Thrun, L. Saul, and B. Schölkopf, editors, *Advances in Neural Information Processing Systems*, volume 16. MIT Press, 2003.

[20] Jingjing Cui, Yuanwei Liu, and Arumugam Nallanathan. Multi-agent reinforcement learning based resource allocation for uav networks, 2018.

[21] James Decraene, Mark Anderson, and Malcolm Y. H. Low. Maritime counter-piracy study using agent-based simulations. page 165, 01 2010.

[22] James Decraene, Fanchao Zeng, Malcolm Y. H. Low, Suiping Zhou, and Wentong Cai. Research advances in automated red teaming. page 47, 01 2010.

[23] Anthony Dekker. Agent-based simulation for counter-ied: A simulation science survey. 05 2010.

[24] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[25] Gabriel Dulac-Arnold, Daniel Mankowitz, and Todd Hester. Challenges of real-world reinforcement learning, 2019.

[26] Tommer Ender, Ryan Leurck, Brian Weaver, Paul Miceli, W. Dale Blair, Phil West, and Dimitri Mavris. Systems-of-systems analysis of ballistic missile defense architecture effectiveness through surrogate modeling and simulation. In *2008 2nd Annual IEEE Systems Conference*, pages 1–8, 2008.

[27] Jakob Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. Counterfactual multi-agent policy gradients, 2017.

[28] Lauren E. Gillespie, Gabriela R. Gonzalez, and Jacob Schrum. Comparing direct and indirect encodings using both raw and hand-designed features in tetris. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, page 179–186, New York, NY, USA, 2017. Association for Computing Machinery.

[29] Sven Gronauer and Klaus Diepold. Multi-agent deep reinforcement learning: A survey. *Artif. Intell. Rev.*, 55(2):895–943, feb 2022.

[30] Jaff Guo, Joe Armstrong, and David Unrau. Predicting emplacements of improvised explosive devices. *The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology*, 10:75–86, 01 2013.

[31] Jayesh K. Gupta, Maxim Egorov, and Mykel Kochenderfer. Cooperative multi-agent control using deep reinforcement learning. In Gita Sukthankar and Juan A. Rodriguez-Aguilar, editors, *Autonomous Agents and Multiagent Systems*, pages 66–83, Cham, 2017. Springer International Publishing.

[32] Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination, 2020.

[33] Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels, 2019.

[34] Liqiang He, Dan Su, and Dong Yu. Learned transferable architectures can surpass hand-designed architectures for large scale speech recognition. In *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6788–6792, 2021.

[35] Dan Hendrycks and Kevin Gimpel. Bridging nonlinearities and stochastic regularizers with gaussian error linear units. *CoRR*, abs/1606.08415, 2016.

[36] Geoffrey E. Hinton, James L. Mcclelland, and David E. Rumelhart. Distributed representations. In David E. Rumelhart and James L. Mcclelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*, pages 77–109. MIT Press, Cambridge, MA, 1986.

[37] R Devon Hjelm, Alex Fedorov, Samuel Lavoie-Marchildon, Karan Grewal, Phil Bachman, Adam Trischler, and Yoshua Bengio. Learning deep representations by mutual information estimation and maximization, 2019.

[38] Sebastian Höfer, Kostas Bekris, Ankur Handa, Juan Camilo Gamboa, Melissa Mozifian, Florian Golemo, Chris Atkeson, Dieter Fox, Ken Goldberg, John Leonard, C. Karen Liu, Jan Peters, Shuran Song, Peter Welinder, and Martha White. Sim2real in robotics and automation: Applications and challenges. *IEEE Transactions on Automation Science and Engineering*, 18(2):398–400, 2021.

[39] Christian Janiesch, Patrick Zschech, and Kai Heinrich. Machine learning and deep learning. *Electronic Markets*, 31(3):685–695, 2021.

[40] Stef Janssen, Alexei Sharpanskykh, and Richard Curran. Agent-based modelling and analysis of security and efficiency in airport terminals. *Transportation Research Part C Emerging Technologies*, 100:142–160, 01 2019.

[41] Orhan Karasakal. Air defense missile-target allocation models for a naval task group. *Computers OR*, 35:1759–1770, 06 2008.

[42] George Konidaris, Sarah Osentoski, and Philip Thomas. Value function approximation in reinforcement learning using the fourier basis. 01 2011.

[43] Landon Kraemer and Bikramjit Banerjee. Multi-agent reinforcement learning as a rehearsal for decentralized planning. *Neurocomputing*, 190:82–94, 2016.

[44] Jakub Grudzien Kuba, Muning Wen, Yaodong Yang, Linghui Meng, Shangding Gu, Haifeng Zhang, David Henry Mguni, and Jun Wang. Settling the variance of multi-agent policy gradients, 2021.

[45] Chun-Gui Li, Meng Wang, Zhen-Jin Huang, and Zeng-Fang Zhang. An actor-critic reinforcement learning algorithm based on adaptive rbf network. In *2009 International Conference on Machine Learning and Cybernetics*, volume 2, pages 984–988, 2009.

[46] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019.

[47] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6382–6393, Red Hook, NY, USA, 2017. Curran Associates Inc.

[48] Daniel Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, Thomas Köppe, Kevin Millikin, Stephen Gaffney, Sophie Elster, Jackson Broshear, Chris Gamble, Kieran Milan, Robert Tung, Minjae Hwang, and David Silver. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618:257–263, 06 2023.

[49] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016.

[50] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning.

[51] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. 518(7540):529–533, 2015.

[52] Igor Mordatch and Pieter Abbeel. Emergence of grounded compositional language in multi-agent populations. *arXiv preprint arXiv:1703.04908*, 2017.

[53] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, page 807–814, Madison, WI, USA, 2010. Omnipress.

[54] Klaus Niemeyer. Modeling and simulation in defense. *Information & Security An International Journal*, 12:19–42, 2003.

[55] F. A. Oliehoek, M. T. J. Spaan, and N. Vlassis. Optimal and approximate q-value functions for decentralized POMDPs. *Journal of Artificial Intelligence Research*, 32:289–353, may 2008.

[56] OpenAI, :, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique P. d. O. Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning, 2019.

[57] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback.

[58] Lerrel Pinto, Marcin Andrychowicz, Peter Welinder, Wojciech Zaremba, and Pieter Abbeel. Asymmetric actor critic for image-based robot learning. In *Robotics: Science and Systems XIV*. Robotics: Science and Systems Foundation, 2018.

[59] T. Poggio and F. Girosi. Networks for approximation and learning. *Proceedings of the IEEE*, 78(9):1481–1497, 1990.

[60] M. J. D. Powell. *Radial Basis Functions for Multivariable Interpolation: A Review*, page 143–167. Clarendon Press, USA, 1987.

[61] Tabish Rashid and Mikayel Samvelyan. QMIX: Monotonic value function factorisation for deep multi-agent reinforcement learning.

[62] M. Rempel and J. Cai. A review of approximate dynamic programming applications within military operations research. 8:100204, 2021.

[63] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation.

[64] G. Rummery and Mahesan Niranjan. On-line q-learning using connectionist systems. *Technical Report CUED/F-INFENG/TR 166*, 11 1994.

[65] Stuart Russell and Andrew L. Zimdars. Q-decomposition for reinforcement learning agents. In *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*, ICML'03, page 656–663. AAAI Press, 2003.

[66] Alex Salgo, Jeremy Banks, and François Rivest. Exploring decision support systems in task scheduling. 11 2021.

[67] Mikayel Samvelyan, Tabish Rashid, Christian Schroeder de Witt, Gregory Farquhar, Nantas Nardelli, Tim G. J. Rudner, Chia-Man Hung, Philiph H. S. Torr, Jakob Foerster, and Shimon Whiteson. The StarCraft Multi-Agent Challenge. *CoRR*, abs/1902.04043, 2019.

[68] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, dec 2020.

[69] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2017.

[70] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2015.

[71] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[72] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[73] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[74] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.

[75] David Silver, Satinder Singh, Doina Precup, and Richard S. Sutton. Reward is enough. 299:103535, 2021.

[76] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. Mass: Masked sequence to sequence pre-training for language generation, 05 2019.

[77] Peter Sunehag, Guy Lever, Audrunas Gruslys, Wojciech Marian Czarnecki, Vinicius Zambaldi, Max Jaderberg, Marc Lanctot, Nicolas Sonnerat, Joel Z. Leibo, Karl Tuyls, and Thore Graepel. Value-decomposition networks for cooperative multi-agent learning, 2017.

[78] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[79] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999.

[80] Richard Stuart Sutton. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, 1984. AAI8410337.

[81] J.N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, 1997.

[82] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding, 2019.

[83] Petar Veličković, William Fedus, William L. Hamilton, Pietro Liò, Yoshua Bengio, and R Devon Hjelm. Deep graph infomax, 2018.

[84] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, L. Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander Sasha Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom Le Paine, Caglar Gulcehre, Ziyun Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy P. Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, pages 1–5, 2019.

[85] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. StarCraft II: A new challenge for reinforcement learning.

[86] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks, 2017.

[87] Brian Wade. Creating surrogate models for an air and missile defense simulation using design of experiments and neural networks. *The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology*, 18:154851291987798, 09 2019.

[88] M. Waltz and K. Fu. A heuristic approach to reinforcement learning control systems. *IEEE Transactions on Automatic Control*, 10(4):390–398, 1965.

[89] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, and Nando de Freitas. Sample efficient actor-critic with experience replay, 2017.

[90] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.

[91] Brian Weaver. A methodology for ballistic missile defense systems analysis using nested neural networks. 01 2008.

[92] Gerhard Weiß. Distributed reinforcement learning. In Luc Steels, editor, *The Biology and Technology of Intelligent Autonomous Agents*, pages 415–428, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[93] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.

[94] David H. Wolpert and Kagan Tumer. An introduction to collective intelligence, 1999.

[95] Erfu Yang and Dongbing Gu. Multiagent reinforcement learning for multi-robot systems: A survey. 06 2004.

[96] Gokul Yenduri, Ramalingam M, Chemmalar Selvi G, Supriya Y, Gautam Srivastava, Praveen Kumar Reddy Maddikunta, Deepti Raj G, Rutvij H Jhaveri, Prabadevi B, Weizheng Wang, Athanasios V. Vasilakos, and Thippa Reddy Gadekallu. Generative pre-trained transformer: A comprehensive review on enabling technologies, potential applications, emerging challenges, and future directions, 2023.

[97] Chao Yu, Akash Velu, Eugene Vinitsky, Yu Wang, Alexandre Bayen, and Yi Wu. The surprising effectiveness of ppo in cooperative multi-agent games, 2021.

[98] Tao Yu, Zhizheng Zhang, Cuiling Lan, Yan Lu, and Zhibo Chen. Mask-based latent reconstruction for reinforcement learning, 2022.

[99] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. *Multi-Agent Reinforcement Learning: A Selective Overview of Theories and Algorithms*, pages 321–384. Springer International Publishing, Cham, 2021.

[100] Åström, Karl Johan. Optimal Control of Markov Processes with Incomplete State Information I. 10:174–205, 1965.