HIERARCHICAL REINFORCEMENT LEARNING FOR ADAPTIVE AND AUTONOMOUS DECISION-MAKING IN ROBOTICS

BY

NEALE VAN STRALEN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Aerospace Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Advisers:

Research Assistant Professor Huy Tran
Assistant Professor Girish Chowdhary

ABSTRACT

In recent years, Reinforcement Learning has been able to solve extremely complex games in simulation, but with limited success in deployment to real-world scenarios. The goal of this work is create an ecosystem in which Reinforcement Learning algorithms can be deployed onto real robots in complex games. The ecosystem begins with the creation of a development pipeline which can be used to progressively train Reinforcement Learning Algorithms in increasingly realistic scenarios, culminating with the deployment of these algorithm onto a real system. The pipeline is paired with the novel Reinforcement Learning algorithms that are better able to adapt to new scenarios than traditional methods for autonomy and robotic planning We implement two techniques to enable this adaptation First, we implement a hierarchical Reinforcement Learning architecture that uses differentiated sub-policies governed by a hierarchical controller to enable fast adaptation. Second we introduce a confidence-based training process for the hierarchical controller which improves training stability and convergence times. These algorithmic contributions were evaluated using our development pipeline.

ACKNOWLEDGMENTS

I'd like to thank my advisors, Dr. Huy Tran and Dr. Girish Chowdhary, for guiding me through my thesis research. They have helped develop and realize many of my different research ideas, and provided guidance to improve the quality of my work. Finally, I want to express my gratitude to my family, friends, and colleagues for always supporting me.

TABLE OF CONTENTS

LIST OF TABLES v

LIST OF FIGURES vi

LIST OF ABBREVIATIONS viii

CHAPTER 1 INTRODUCTION 1
1.1 Decision-making in Robotics 1
1.2 Contributions 3
1.3 Capture the Flag 3

CHAPTER 2 BACKGROUND AND RELATED WORK 5
2.1 Reinforcement Learning 5
2.2 Hierarchical Reinforcement Learning 15
2.3 Adaptable RL 17

CHAPTER 3 RL DEVELOPMENT PIPELINE 18
3.1 Capture the Flag Game 19
3.2 CTF Development Pipeline 20
3.3 2D Abstracted Simulation 23
3.4 3D Physics-based Simulated Environment 28
3.5 Real-world CTF 41

CHAPTER 4 HRL FOR ADAPTATION 50
4.1 HRL for Adaptation 50
4.2 Confidence based HRL 53
4.3 Experimental Setup 54
4.4 Results 58

CHAPTER 5 CONCLUSIONS 63

REFERENCES 64

APPENDIX A: CODE AVAILABILITY 67

LIST OF TABLES

4.1 Sub-policy environment settings 57
4.2 Adaptation Results 59

LIST OF FIGURES

2.1 Agent Environment Interactions 6
2.2 Artificial Neuron 12
2.3 Fully Connected Layer 13
2.4 Convolution Layer 14
2.5 Hierarchical Reinforcement Learning Architecture 15

3.1 CTF Game-play Example 20
3.2 CTF Deployment Pipeline 21
3.3 CTF Observation 22
3.4 2D Abstracted Simulation Visualization 23
3.5 2D Abstracted CTF MDP Formulation 24
3.6 CTF Environment Wrappers 26
3.7 3D Physics-based Simulated Architecture 29
3.8 Example ROS2 Node Structure 31
3.9 ROS2 Architecture for the 3D Physics-based Simulation 32
3.10 CTF being played in the Physics-based simulator Gazebo . . 32
3.11 Gazebo functionality and interfaces 34
3.12 Comparison of native software, virtual machines and Docker . 35
3.13 Terrasentia simulated functions and interfaces 36
3.14 UAV simulated functions and interfaces 37
3.15 Emulated CTF observations and interactions in the 3D physics based simulation 38
3.16 CTF Planning Node 39
3.17 Hardware in the loop setup for CTF 40
3.18 Machine vision algorithm integration 41
3.19 Real-World CTF Architecture 42
3.20 Terrasentia Hardware Architecture 43
3.21 UAVs used in Real-world CTF 44
3.22 3DR Solo Architecture 45
3.23 Custom UAV Architecture 46
3.24 Real-world communication infrastructure 48

4.1 HRL architecture used for adaptation 52
4.2 Confidence-based training example 55
4.3 Hyperparameter Tuning 57

4.4 Evaluation Metrics 58
4.5 Adaptation performance comparison 60
4.6 Transient adaptation performance of HRL 61
4.7 Transient behavior of policies during adaptation 62

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
ML	Machine Learning
RL	Reinforcement Learning
UAV	Unmanned Aerial Vehicle
CTF	Capture the Flag
PPO	Proximal Policy Optimization
HC	Hierarchical Controller
HRL	Hierarchical Reinforcement Learning

CHAPTER 1

INTRODUCTION

The developments in computational hardware have allowed for Artificial Intelligence (AI) to scale to problems that were computationally intractable only a decade ago; and in that time AI has become a backbone for modern technology. AI has many implications in robotics for creating fully autonomous systems as the two primary tasks of these systems, processing environment observations and making complex decisions to interact with the environment, require highly intelligent processes. There has been significant development in both systems with machine vision algorithms accurately identifying objects and contextual information from images and Reinforcement Learning (RL) algorithms playing complex games such as DOTA II and StarCraft II better than humans [1, 2]. However, these algorithms struggle in real-world environments, such as self-driving cars in which vision algorithms can misinterpret the environment and decision-making algorithms fail to safely react to noisy observations or unknown situations, resulting fatal crashes.

In this thesis we will be addressing improving the ability to deploy AI algorithms, specifically those for autonomous decision-making, into real-world environments. Current RL algorithms have shown great promise, but suffer under even minor environment changes. Addressing this shortcoming will help improve the ability of these algorithms to be deployed onto real-world systems.

1.1 Decision-making in Robotics

Making sequential decisions in uncertain environments is incredibly difficult, especially for robots with no natural intelligence. One of the more popular methods to solve these tasks is to train in a simulated environment with a technique called RL where a policy is created to control the decision-making.

The policy is optimized to maximize the performance at a specific task based on experimentation in the environment, which is similar to how humans experiment to learn how to solve a task. Policies like AlphaStar and OpenAI Five, have shown that RL can create policies that are able to beat World Champions in complex video games StarCraft II and DOTA II, respectively [1, 2].

The difficulty with RL is that the method is sample intensive and requires significant exploration in the environment, which if performed on real robots would take hundreds of years or cause thousands of crashes while naively exploring the environment. These restrictions are avoided by primarily training in simulated environments which significantly improves data collection and eliminates risk in high stakes environments. This knowledge can then be applied to the real robot, and in the case of simple policies such as robotic manipulation, this approach has shown great success [3, 4, 5, 6, 7]. This success may lead one to wonder why we do not use these models in critical pieces of infrastructure. The short answer is that the real-world has two key differences than a simulated environment:

- Changing Dynamics: Often there exist changes in dynamics between a simulated environment and a real-world environment.
- Observation Noise: There is often more noise in the observations due to issues in sensor processing, which results in more strain on the planner.

One possible way to handle these differences would be to improve the pipeline from simulation to real-world. The real-world system would be a game which can be used to demonstrate a variety of complex decision-making problems. The simulated environment should allow for high sampling efficiency to train RL methods while maintaining similarities to the final system so the policies can be deployed in a real world environment.

A more generic way to deal with these differences is to modify the RL models to be more robust to noise and to have adaptable features which can be updated to overcome the changing dynamics. However, classic RL methods even with modifications to improve robustness are based on the assumption that their environment is static and will not change during operation. This assumption leads policies to suffer from extreme performance loss when they are applied to even a slightly different environment. Novel methods for improving the adaptability of RL methods are required to deploy these policies

to real-world environments as conditions are continually changing and the policy must be able to cope with the changes successfully.

1.2 Contributions

The main contributions of this thesis are as follows:

Creating a modular pipeline that can be used for testing and development of AI algorithms to real-world Robotic systems.

We created a progressive training and development infrastructure for machine vision and RL tasks. The infrastructure has three main components, a 2D abstracted simulation, a 3D physics-based simulation and a real-world environment, which can be used progressively to deploy planning and vision algorithms onto real-world robotics.

Development of two RL techniques to improve the adaptive performance of policies.

We propose two RL techniques which improve the training and adaptive performance of RL. The first technique is to use differentiated sub-policies governed by a hierarchical controller to support adaptation in dynamically changing scenarios. The second technique is a confidence-based training process for hierarchical controllers which improves training stability and convergence times. We demonstrate that these method improve adaptive performance compared to traditional RL schemes, when adapting to different agents in the 2D abstracted Capture the Flag (CTF) game.

1.3 Capture the Flag

Adversarial games and tasks are a primary part of human growth, where we learn to compete and improve with our peers. In this thesis we use the game CTF to demonstrate my RL development pipeline and RL adaptation techniques. We chose this game because it is an adversarial game in which

policies need to adapt to changing maps, dynamics, and opponent strategies. Some of the main features which make this an interesting environment:

- Adversarial: Learning to play against an intelligent enemy is difficult as they can pursue an ever evolving strategy to counter your actions.
- Multi-agent: Learning to coordinate with other agents to achieve optimal performance.
- Partially Observable: Learning optimal decisions when you have imperfect information in the environment is difficult, especially when combined with adversarial and multi-agent characteristics.

CHAPTER 2

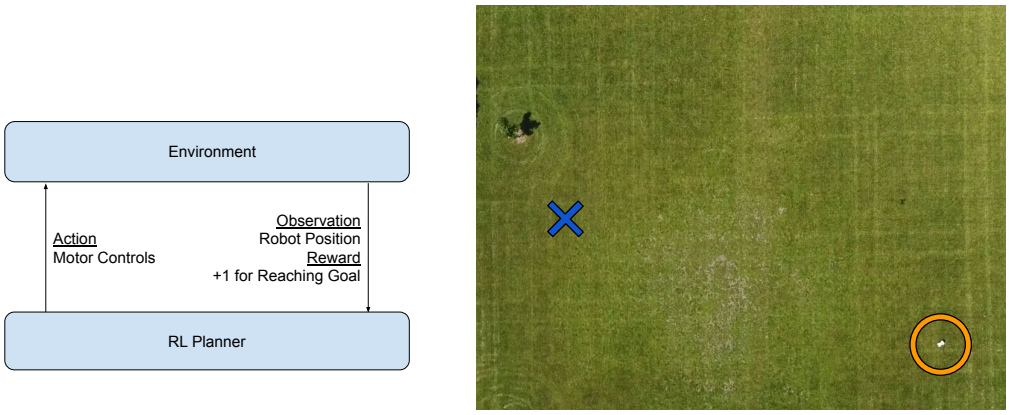
BACKGROUND AND RELATED WORK

One of the main fields within AI is machine learning (ML), in which algorithms are created to find complex patterns in data-sets, which are then leveraged to make decisions and predictions. ML relies on large sets of labeled data to train the algorithms, but this is often infeasible in many tasks, such as playing a video game or operating a robot, which are either difficult to label the data or infeasible to collect sufficient data. One subset of ML that addresses this problem is RL in which an agent collects its own data through experimentation, and labels it based on a simple reward function defined by the operator. A major limitation of most RL methods is that the algorithm is fit to one specific task and has limited generalizability or applicability to even similar tasks, leading to failure of the algorithm in changing environments. Several methods have addressed this by improving the ability of the algorithms to transition to new tasks either by remapping information or creating a generalizable policy.

This chapter begins by formulating the generic RL problem and discussing the basic solution process. The formulations are then built for Hierarchical RL which provide modelling benefits over traditional RL schemes. The chapter finishes with a summary of traditional adaptation techniques.

2.1 Reinforcement Learning

As stated earlier the main paradigm in RL is to learn while experimenting and collecting it's own data, which is a very generic process, but requires many assumptions to be placed on the problem. The general framework for the problem is composed of two core pieces, the environment which is the game or task which one wants to learn and the agent which is the intelligent algorithm that interacts with the environment. The agent interacts with the



(a) (b)

Figure 2.1: (a) A block diagram describing the RL framework, in which the agent uses experimental actions to interact with the environment and receives observations and rewards, which it uses to train a control policy. (b) An example RL problem in which the robot (circled) has a goal to navigate to a specific location (cross). The RL planner must learn to control the robot's motors to navigate to the goal.

environment by taking actions which effect the state of the environment. The environment is then observed by the agent, creating an observation, which the agent uses as data. The goal of the agent is to maximize a reward signal or a performance metric, which is defined by the user. This interaction is shown in Figure 2.1a.

An example of this operation would be learning a policy to control the motors that drives a robot to a goal, shown in Figure 2.1b. In this case we can define the actions as the motor movements, the observation may be the GPS location of the robot and a reward would be given when the agent reaches the goal. What we would want is the agent to explore the environment, and finally create a policy which would define what motor movements to make at a GPS coordinate.

The rest of this section defines the formal process for solving this type of problem. It begins with a structured definition of the environment and the agent, called a Markov Decision Process. This is followed by defining a formal solution to the MDP, and preliminary methods of solution based on dynamic programming. The section then addresses different ways to accomplish the solution with neiral networks and complex update algorithms.

2.1.1 Markov Decision Process

The Markov Decision Process (MDP) is a model for environments, which provides a structured definition which many theories in RL are based on. A MDP is defied by a tuple of information (S,A,P,R) in which;

- S defines a set of all possible states, $s \in S$
- A defines a set of all possible actions, $a \in A$
- P defines the transition between different states, conditioned on an action, $P(s'|s,a)$
- R defines a reward structure when transitioning between states, $R = S' \times S \times A$

During the RL training it is often assumed that the MDP is unknown a-priori and is learned through the agent's experimentation within the environment, with the goal to maximize the total reward. For temporally extended tasks the reward becomes increasingly complicated as the agent must balance receiving immediate rewards and future rewards. In these scenarios the reward is often discounted into the future with a parameter γ, typically 0.9 or 0.99, creating a discounted return defined as,

$$G = r_1 + \gamma r_2 + \gamma^2 r_3 + ... = \sum_{i=0}^{\infty} \gamma^i r_{i+1} \tag{2.1}$$

2.1.2 Agent Policies

To formally solve the environment the agent creates a policy, $\pi(a|s)$ which defines optimal actions to be taken at every state in the environment. Currently, there are two primary ways to create these policies, model-based and model-free methods. In model-based methods the agent explicitly learns all of the environment transitions, states and rewards, and uses the information in an optimization process to create a $\pi(a|s)$ which maximizes the rewards. One drawback of model-based methods are that they are inflexible to environment changes as the learned information is often tailored around the environment. In contrast, the more popular and flexible option in RL are the model-free methods, in which the agent directly learns the policy, $\pi(a|s)$,

which maximizes the rewards, at the cost of sample inefficiency and longer training times.

This work primarily deals with model-free methods, which are based on the value function,

$$V(s) = E[G|s] = E[\sum_{i=0}^{\infty} \gamma^i r_{i+1}|s], \tag{2.2}$$

which represents the expected discounted reward, from Equation 2.1 when starting in a given state. This quantity effectively measures the success on the agent and can be used in two ways to solve the environment: value-based and policy-based methods. In value-based methods the value function can be learned explicitly and used to define the policy, i.e. the agent will select actions which maximize the value function. In policy-based methods the agent can implicitly learn the value function inside of a policy, where the actions happen to maximize the value function. These methods are discussed in more detail in Section 2.1.3

These value functions must be learned, based on environment, but the infinite horizon presents a complex problem to solve. This limitation is dealt with by breaking the infinite horizon into a series of one step approximation which are called the Bellman equation,

$$V(s) = E[R_{t+1} + \gamma V(s_{t+1})|s], \tag{2.3}$$

which states the value of a state is equivalent to the next expected reward plus the value of the next state. These equations appear as iterative quantities and are used with dynamic programming algorithms which are used to update the value function.

In some methods the value is parameterized as a state-action value function,

$$Q(s,a) = E[G|s,a] = E[\sum_{i=0}^{\infty} \gamma^i r_{i+1}|s,a] \tag{2.4}$$

$$Q(s,a) = E[R_{t+1} + \gamma Q(s_{t+1}, a_{t+1})|s,a], \tag{2.5}$$

which represents the value of taking a particular action at a given state, and is useful to compare the value of different actions at a given state.

2.1.3 Reinforcement Learning Algorithms

RL algorithms are the process of iteratively updating a policy $\pi(a|s)$ to maximize the value, V. The value of the policy is often represented as,

$$V^{\pi}(s) = E[G|s, \pi], \tag{2.6}$$

which is the expected value that is obtained by following the policy in the environment. The policy is updated until it reaches the maximum value, denoted as

$$V^{*}(s) = max(V^{\pi}(s)). \tag{2.7}$$

These iterative updates are made with Monte Carlo sampling performed by the agent to explore the environment. Due to the requirements of Monte-Carlo sampling, there often has to be some level of exploration tied to the agent's actions when exploring the environment. This leads to the balancing of exploration vs exploitation of the environment, which is addressed differently with different methods.

As the value functions can be extremely complex they are often parameterized with neural networks, which are described in Section 2.1.4. The way of notated these for the purpose of he algorithms is with θ, which represents the all of parameters of the network.

RL algorithms can broadly be split into three main categories, value-based and policy-based methods, mentioned earlier, and actor-critic methods which combine aspects of the other methods.

Value-based Methods

In value based methods a value function, often the state-action value function, is learned and parameterized in a neural network. A policy is then described based on the value function as,

$$\pi(a|s) = \max_{a}(E[G|s, a]) = \max_{a}(Q(s, a)), \tag{2.8}$$

where an action is selected which maximizes the value function.

To explore the environment the Value-based methods often use a method

called E-Greedy which balances the exploration and exploitation. This is done by selecting random actions occasionally based on the following rule:

$$a(s) = \begin{cases} \max_a(Q(s,a)) & \text{with probability } 1-\epsilon \\ \text{a random action} & \text{with probability } \epsilon \end{cases} \tag{2.9}$$

A common algorithm for these methods is the Q-learning algorithm [8] described in Algorithm 1, which updates a neural network to approximate $Q(s,a)$.

Algorithm 1: Q-learning Algorithm

```
Initialize Q(s,a) parameterized by θ ;
repeat
    for Until Episode Ends do
        Select a from s in policy derived from Q (E-Greedy);
        Take a, observe r and s' ;
        Q(s,a) ← Q(s,a) + α[r + max(γQ(s',a) − Q(s,a)];
        s ← s'
    end
until;
```

Policy-based methods

In policy-based methods the policy is learned explicitly. This has benefits over value based methods when being applied to continuous or large action spaces, which drastically decrease performance of value-based methods. The exploration comes from the fact that an initialized policy is naturally random, and allows exploration. However some implementations use an E-greedy approach to improve long-term exploration of the environment. The policy-based method REINFORCE [9] is described in Algorithm 2 where parameters of the network are updated iteratively resulting in a final policy.

Actor-Critic Methods

Actor-critic methods blend both value-based and policy-based methods into one framework, which results in efficient and stable updates, similar to value-based policies, and good parameterization of continuous spaces, similar to

Algorithm 2: REINFORCE Algorithm

Initialize $\pi(a|s)$ parameterized by θ ;
repeat
  Generate trajectory ;
  Calculate discounted rewards: $G \leftarrow \sum_{i=0}^{\infty} \gamma^i r_{i+1}$;
  Update parameters of π: $\theta \leftarrow \theta + \alpha\gamma G[\nabla ln\pi(A|S,\theta)]$;
until;

policy-based methods [10]. These methods operate by learning a value representation of the state $V(s)$ and then using that to more efficiently train the actor, which is a policy. The Vanilla Actor-Critic algorithm [11] is described in Algorithm 3.

Algorithm 3: Vanilla Actor Critic Algorithm

Initialize $\pi(a|s)$ parameterized by θ ;
Initialize $V(s)$ parameterized by w ;
repeat
  for *Until Episode Ends* **do**
    Select a from s in policy $\pi(a|s)$;
    Take a, observe r and s' ;
    $\delta \leftarrow R + \gamma V(S') - V(S)$;
    $w \leftarrow w + \alpha\gamma\delta[\nabla V(s,w)]$;
    Update parameters of π: $\theta \leftarrow \theta + \alpha\delta[\nabla ln\pi(A|S,\theta)]$;
    $s \leftarrow s'$
  end
until;

2.1.4 Neural Networks

Modern RL algorithms leverage neural networks which are used as flexible function approximators for the above value functions and policies. Given the Universal Approximation Theory [12], a neural network with carefully selected weights can recreate any continuous function. Neural networks are composed of sets of artificial neurons which mimic synaptic responses of neurons to various input signals. An example artificial neuron is shown in Figure 2.2, in which a vector of input signals is multiplied by an internal weights and biases resulting in an output. This model lacks the ability to model

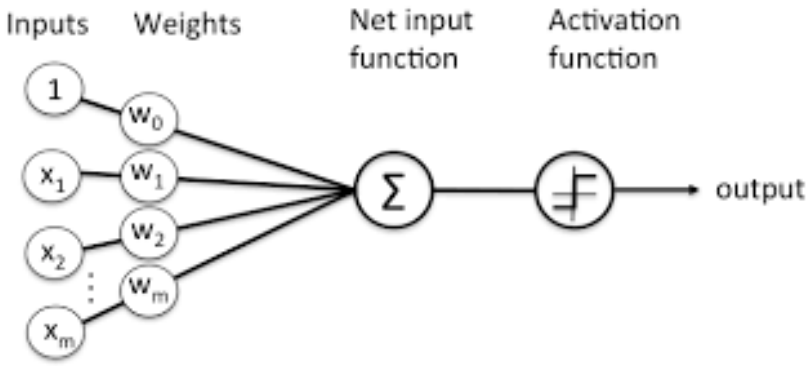


Figure 2.2: An artificial neuron which mimics neuron firing of a traditional cell. This is replicated by taking a linear combination of input signals and applying a complex activation function to the output.

non-linear transformations that exist in biological neurons, such as activation potentials. Activation functions are used to model activation potentials, by using non-linear functions, such as the sigmoid, hyperbolic tangent and the Rectified Linear Unit. And with an arbitrarily large number of these neurons with activation functions one is able to create any desired response to an input.

Although the Universal Approximation Theory states that the neurons can approximate the input with a single I/O layer of neurons, they are often arranged together in sequential groups, called layers, to increase the computational performance of the neurons. The most common types of layers are Fully-Connected and Convolution layers and are combined linearly to create a deep neural network. These networks are updated with back-propagation algorithms.

Fully-Connected Layers

The Fully-Connected layer is a group of neurons which each share the same input and create a vector output. An example Fully Connected layer is shown in Figure 2.3

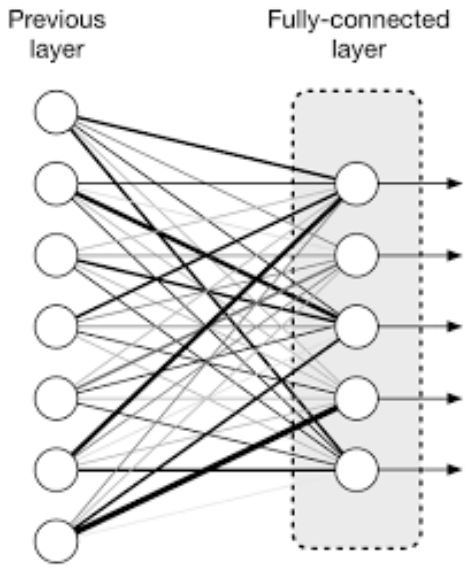


Figure 2.3: A fully connected layer, shown on the right is a set of neurons that which are each connected to a set on inputs, in this case a previous set of neurons. This representation provides a compact way of representing sets of neurons, and is computationally simplified to a matrix notation.

Convolution Layers

A Convolution layer is used to process image-like data, in which the inputs exist is a spatial construct. The layer works by applying a set of neurons to a small chunk of the input data to get an output. This set of neurons is then applied to overlapping sections of the input data to create multiple inputs. This is shown in Figure 2.4

Back-Propagation

The Universal Approximation Theory does not address how the weights should be selected to model a function but several methods exist to update the parameters in the neurons. The most common method the back-propagation algorithm which iteratively updates the weights to converge to minimize an arbitrary loss function. For RL based tasks the loss is typically a value or reward function that is defined in the previous section.

For successive weights the chain rule can be applied,

... (2.10)

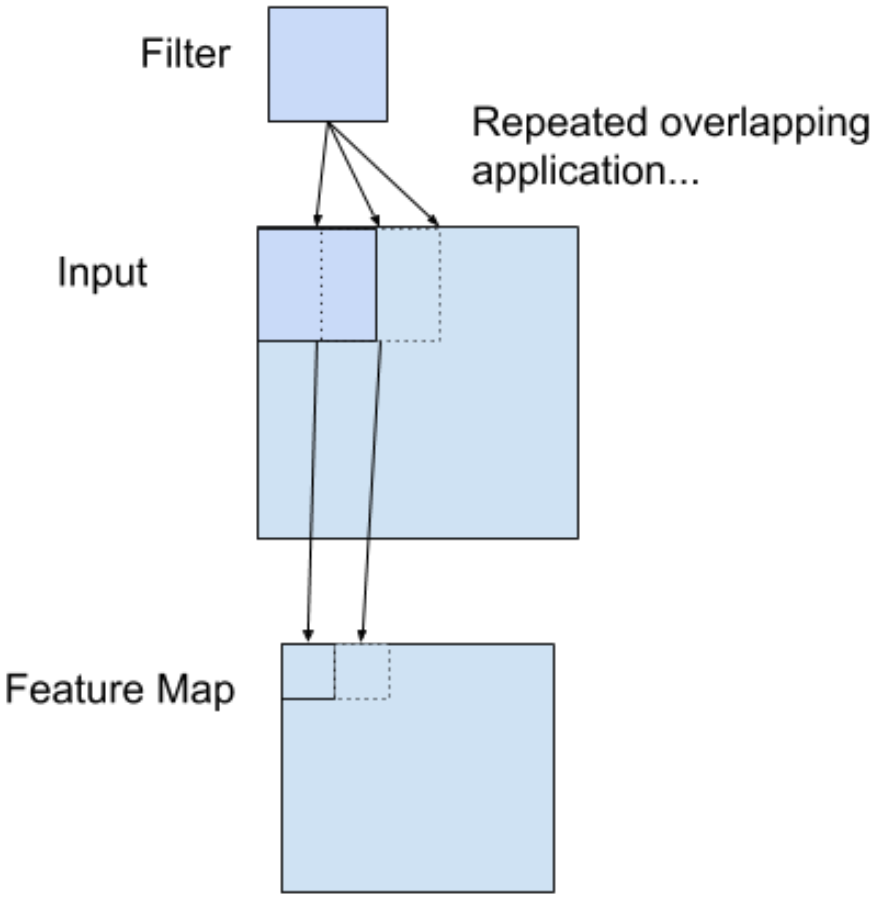


Figure 2.4: A Convolution Layer is a set of neurons arranged in a spatial format called a filter. The filter is then applied successively to small sections of an input signal, typically an image, resulting in an output which captures spatial features of the input.

For an arbitrary dense layer this is done by calculating the gradient on the parameter with respect to the final output. Then the parameter is updated based on the loss based on an update parameters. If the function is non transient and the learning parameters are tuned, this will result in a close approximation of the value function.

Algorithm 4: Back-Propagation

Initialize network f_θ parameters θ
Have set of inputs and outputs [X,Y]
Cost Function C
Learning Rate β
repeat
  Calculate loss, $\alpha = \mathrm{C}(f_\theta(\mathrm{x}), \mathrm{Y})$
  Calculate gradient w.r.t. loss. $\frac{\delta C}{\delta \theta}$
  Update parameters $\theta^+ \leftarrow \theta^- + \beta\alpha\frac{\delta C}{\delta \theta}$
until *done*;

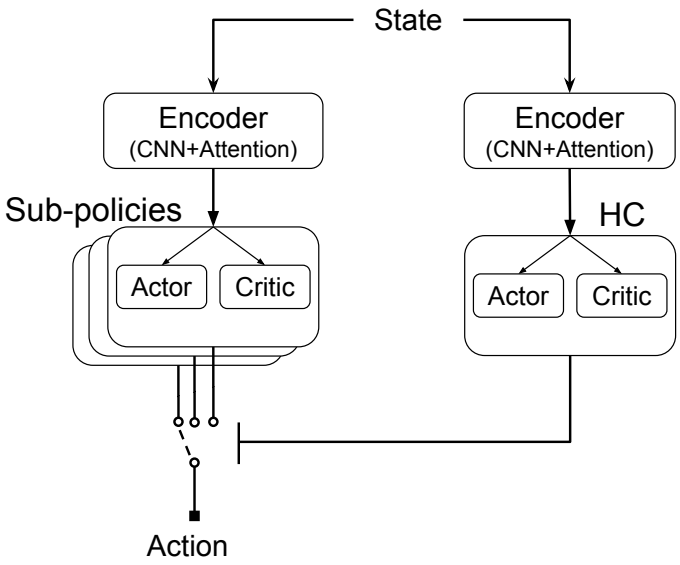


Figure 2.5: An example HRL architecture in which the hierarchical controller, shown on the left, selects between different sub-policies.

2.2 Hierarchical Reinforcement Learning

Hierarchical Reinforcement Learning(HRL) is the process of decomposing the policy into a set of hierarchical representations. For example one could decompose the policy into a set of sub-policies, which are used by a hierarchical controller (HC), which is shown in Figure 2.5. This methodology has been used successfully in temporally abstracted games such as Atari [13], walking environments [14], robotic path planning [15], and for high-level coordination in multi-agent scenarios [16, 17]. The framework is theoretically qualified by creating a temporally extended MDP, which allows for rigorous definitions of the associated value function.

2.2.1 Temporally Extended MDP and the Option Framework

HRL is based on the framework proposed by [18], in which a policy is abstracted using Markovian options. A Markovian option is a $(\mathcal{I}_\omega, \pi_\omega, \beta_\omega)$ tuple, where $\mathcal{I}_\omega$ is an initiation condition, π_ω is a sub-policy, and β_ω is a termination condition. Sets of these options are created where each option has a unique sub-policy π_ω representing a distinct mapping of how to traverse the MDP; these sub-policies can then be composed together based on initiations and terminations to create an overall control policy.

2.2.2 Hierarchical Reinforcement Learning Methods

HRL creates an overall control policy by using an HC to control the initiations and terminations of these sub-policies. Classically, this framework has been used to help solve complex environments by using options to create temporal abstractions that decompose the MDP space [19]. Note that we use the term sub-policies instead of options because options are typically used to define a subset of the MDP space, whereas we use sub-policies that can be defined over the entire MDP space.

One limitation of HRL is that discounted reward assignment is difficult when switching between sub-policies [16]. A common method for addressing this is to operate the HC at a higher temporal scale where sub-policies are switched at a fixed-step interval. Fixed-step intervals stabilize training by creating longer sub-policy trajectories that are easier for the network to learn from. However, fixed-step training often fails when the selected interval does not temporally align with environment time scales. An alternative method is the option-critic framework, which switches sub-policies based on learned activation and termination functions [19]. These functions create extended sub-policy trajectories that allow sub-policies to be trained end-to-end; however, these sub-policies are typically not globally defined, limiting their value towards adaptation.

2.3 Adaptable RL

Two main lines of work have addressed adaptation in RL: transfer learning and one-shot learning.

2.3.1 Transfer Learning

In instances of a well-defined target environment, transfer learning methods aim to correlate information between the source and target environments. One common method is manifold alignment, in which features of the source and target environment are correlated to serve as an intermediate between the trained policy and the target environment [20, 21]. Another method, progressive networks, adds layers to the trained policy so that features from the source environment can be extracted and applied to the target [3]. While successful, these methods are often computationally expensive, as they require significant exploration of the target environment.

2.3.2 Meta Learning

One-shot learning extends transfer learning to consider transfer to unknown environments. These methods focus on enhancing the online training process with sample reuse and augmentation or controlled periods of rapid training. A common method is meta-learning, where a policy is trained to maximize its ability to adapt with limited samples from the target environment, resulting in an ability to update a policy on-line [22, 23, 24, 25]. These methods show promise, but often show drops in short-term performance following environment changes and struggle with target environments that are drastically different from those seen during training.

CHAPTER 3

RL DEVELOPMENT PIPELINE

Training RL algorithms requires 10,000 to 1,000,000 game-plays of data depending on the task complexity and algorithm efficiency, which cannot be accomplished in real world settings. For example, the real-world implementation of the CTF game discussed in this section requires 5 minutes to execute, and combined with setup would result only 1,000 games in 7 straight days of running. This limitation can be overcome with the use of computers which can simulate multiple environments simultaneously, often faster than real experiment, leading to orders of magnitude being shaved from training time. The RL algorithm learned in the simulated environment then can be applied to a real world scenario, and would only require minor fine-tuning to account for differences between the simulated and real environments.

Several simulators utilize this progressive structure, such as walking and grasping robot environments, have shown great performance when transitioning from simulated environments to real environments [26, 7]. However, these pipelines represent niche manipulation and control scenarios, and no such framework exists for applying RL to more complex tasks, such as coordination of heterogeneous agents in partial observability operation, to real world robots. Additionally many of these real-world systems and robots are comprised of multiple RL components such as machine vision and RL planning algorithms, which are not entirely simulated in existing architectures and require a separate framework to train and deploy them.

This chapter presents a progressive framework for AI development. The most basic simulation is computationally simple and can be used to generalize planning models, and complex decision making and coordination. The second point in the pipeline is a physics based simulation, which can be used to transition the initial control policies to more complex dynamics. This provides an initial test-bed for AI integration with robotics platform. This environment also allows for the testing of sensor based observation technolo-

gies, such as image classification and localization. Finally a full robotics environment was developed which allows the AI to deployed onto real world robotics.

3.1 Capture the Flag Game

The game Capture the Flag (CTF) was selected to serve as the basis of the pipeline. CTF is a game in which two teams compete with each-other to capture the enemy's flag while simultaneously defending their own flag. There are several major rules and assumptions that define this game:

1. Teams are comprised multiple agents, which are split into two categories, ground agents and flying agents.

2. Agents are not created equal, some may be faster or stronger than others. This primarily applies to flying agents which cannot interact with other agents or flags, but can provide observations to there allies.

3. The game board is comprised of territories for each team and obstacles, which can only be navigated by flying agents.

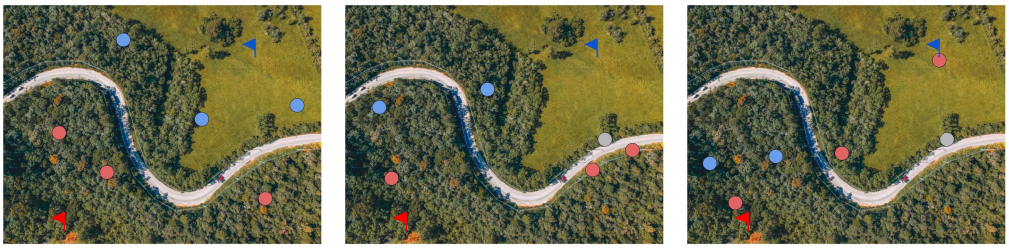
4. When agents come with proximity to one-another there is a stochastic interaction in which agents can die. This is biased based on the agents' strength factors, the number of nearby allies, and whether the agent is on their home territory.

5. A flag can only be captured by a ground robot when the agent reaches the flag, at which point the game immediately ends.

An example game-play, showing agent interactions and the win-condition, is shown in Figure 3.1.

On first inspection this game appears simple with a well defined goal, however CTF addresses three major research questions in RL:

- Multi Agent Control and Coordination - With multiple agents on different teams, how does one optimally control them to solve a task? If agents each have their unique logic, how do they coordinate with one another to solve the task?



(a) (b) (c)

Figure 3.1: (a) Initial Starting position of CTF game. Agents of two teams begin in an environment with a flag for each. They must capture the enemies flag to win, but allowing their flag to be captured results in a loss. (b) Agents have a mechanism to capture, or kill, one another. In this CTF there is a stochastic probability based on proximity to other agents and their strength metric. Here one can see the Blue agent being captured by the two red agents who have an advantage because they are coordinated. (c) The red team wins the game by capturing the enemies flag.

- Partial Observability Decision-making - In partially observable scenarios, there is uncertainty in enemy locations, which play a large role in the environment. How do you create a planning algorithm to explore unknown environments and successfully combat complex enemies?
- Adaptable Planning - This environment can change extremely quickly with the changing of the maps or more complex changes like introducing faster opponents. How does one adapt to the different agents to successfully play the game?

3.2 CTF Development Pipeline

To address this training and deployment issue of RL, three distinct training and development environments, a 2D abstracted environment, a 3D physics based simulation environment and a real-world system, will vary the level of realism allowing RL algorithms to be progressively trialed in more complex and realistic scenarios. The pipeline begins with the 2D abstracted environment, which allows for fast environmental results without the need to fully simulate robots, which allows for the training and development of complex strategic algorithms. This learned information can then be applied to a 3D simulated environment, which will allow for fine-tuning the algorithm

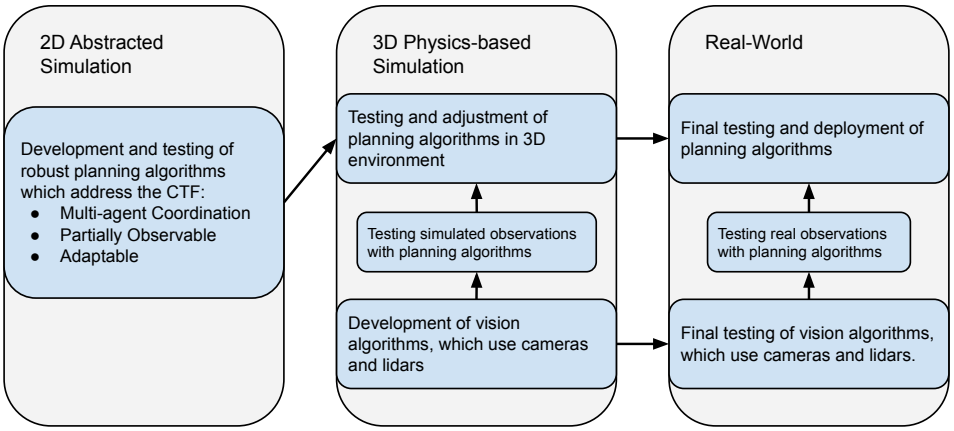
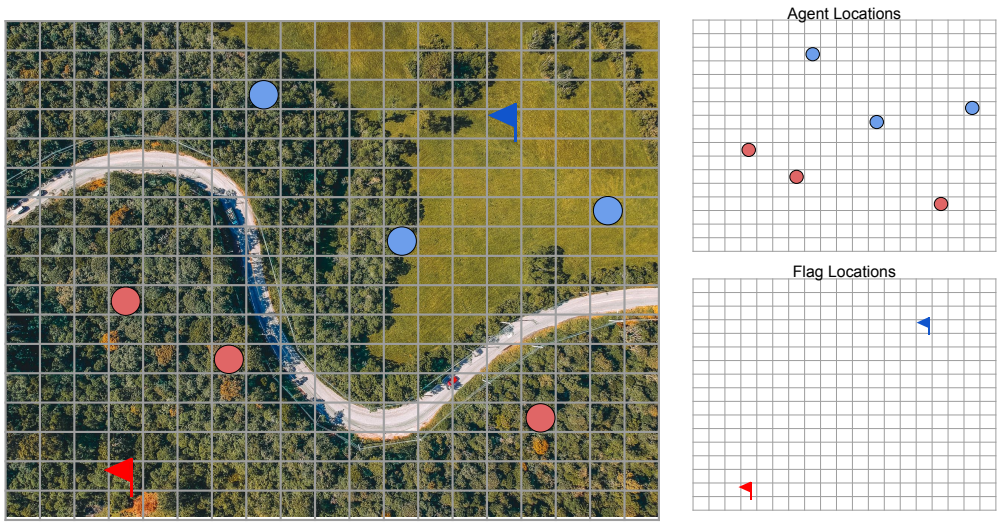


Figure 3.2: The CTF Development pipeline is comprised of three main sections: the 2D Abstracted Simulation, the 3D Physics-based Simulation and the Real-World Game. The 2D Abstracted Simulation is designed for rapid testing of RL algorithms to address fundamental problems in decision-making and control. These learned representations can then be deployed into the 3D Physics-based simulation to adapt the algorithms to a 3D, time based representation. The algorithm can then be deployed onto a real robotic system and evaluated in a final target environment. This pipeline also allows for development and evaluation of vision based algorithms for object detection and localization, which can be used in conjunction with the final decision-making algorithms.

to match physics and temporal scales, as well as allowing for the preliminary testing of vision and sensing algorithms that can be used to create the observations autonomously. The data from this environment can then be applied to an similar real-world scenario that would require small calibration. This pipeline is diagrammed in Figure 3.2.

CTF Representation

The major consideration in the progressive simulation architecture is to maintain a consistent representation between the environments for the RL algorithms, which is difficult as it needs to be applicable to a 2D abstracted environment while maintaining sufficient information to represent the 3D simulation and the real-world system. The observation space chosen for this task is a 2D grid representation of the environment, shown in Figure 3.3a, because it represents the CTF map with little information loss. The final



(a) (b)

Figure 3.3: (a) The full CTF Game shown above, with a grid overlay, showing an example of how the environment may be segmented. (b) Observations split between two separate channels, one for agents locations and one for flag locations. This maintains the positional encoding of each object in the environment in a simple format that allows for

representation of the grid environment segments information into different classes. For example information on obstacles, agent positions and flags are segmented into different data channels, which gives contextual information that an algorithm can use to solve the problem effectively. An example of this channelization is shown in Figure 3.3b in which information on the agents and flags is split into different channels. Finally, this structure also has practical implications as on the real robots they can process their sensor information to the 2D representation and share it with other robots over limited bandwidth connections.

There are several limitations with this representation, that we acknowledge:

- Actions are based upon this grid structure as the there are only slight performance losses based on orthogonal actions when compared to a continuous action space.
- There is no mechanism to represent the orientation of the agents which can have effects upon the observations and movement of the robot in the environment.

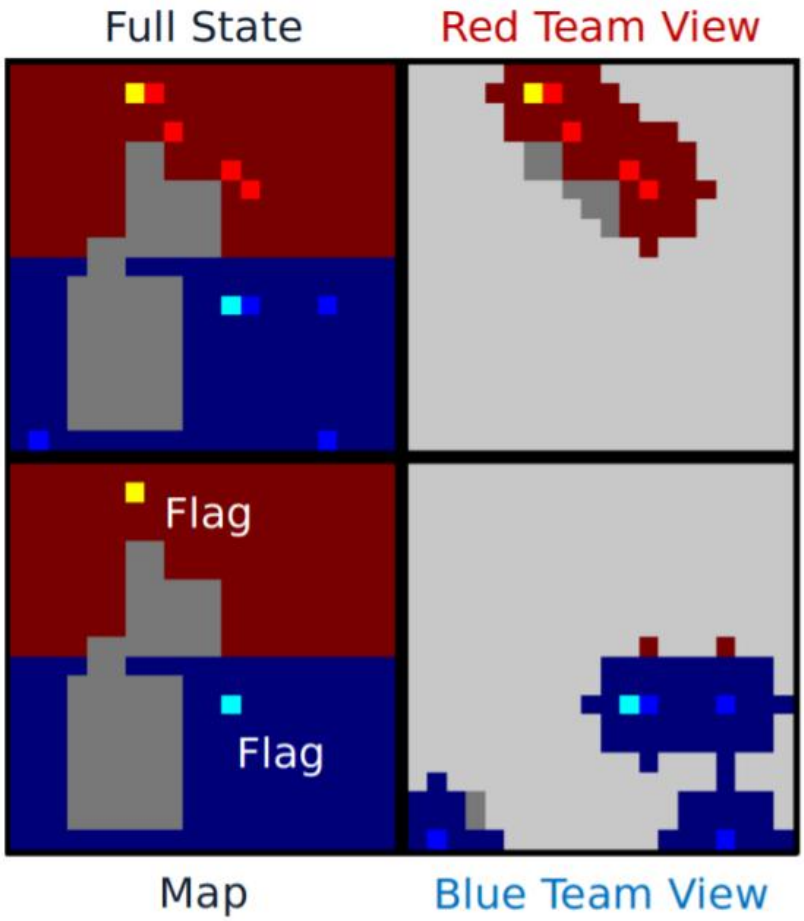


Figure 3.4: An example CTF match in a 4 vs 4 setting. The right side of the figure shows the simulated observations generated by each team, emulating the limited sensor ranges of the robots.

However, based on practical testing these limitations do not have significant consequences on the outcome of the environment.

3.3 2D Abstracted Simulation

The pipeline begins with a 2D abstracted simulation based on the 2D CTF representation, which allows for extremely fast training in a computationally simple environment. The environment is designed to be easily interfaced with RL algorithms to evaluate the research questions posed above. A visualization of the environment is shown in Figure 3.4

This section highlights the features of the 2D abstracted CTF simulation.

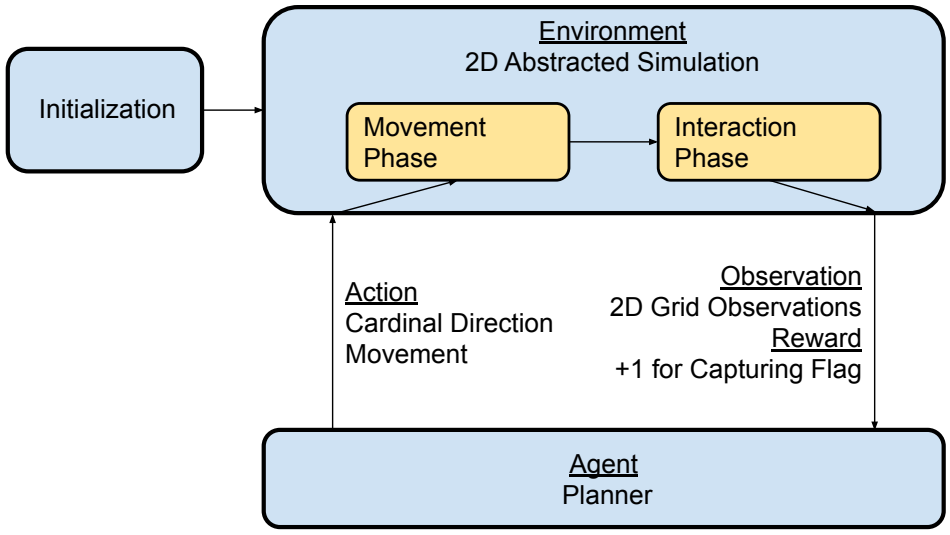


Figure 3.5: The CTF environment operated on the principles of an MDP, which is base assumption for many RL formulations. In this architecture the MDP operates in a cyclic fashion with the agent, taking in an action, updating the state and then returning an observation and reward signal to the RL Planner.

It begins with a discussion of the simulations architecture, which is based on the 2D channelized representation of CTF, where object locations and movements are exclusively represented as discrete 2D locations. Then there are several sections describing the simulation environment and it's opportunities for rapid prototyping of RL algorithms, which are applicable to the real-world environment.

3.3.1 Simulator Architecture

The environment is built with the OpenAI Gym architecture because it provides a modular framework for representing an MDP, which is assumption used in RL formulations. Following the MDP structure this environment takes in an action, desired movements of the agents, updates the internal state and outputs an observation and reward. This structure can be seen in 3.4. Overall, the environment can be broken into two main functional components, initialization, where the game is initialized and operation, where the environment is performing MDP steps.

Initialization

Similar to how a real-life CTF game may begin the initialization phase begins with placing agents, obstacles, and flags. This is performed either performed automatically or based on a user defined configuration file. The environment defaults to creating a randomized environment, with balanced obstacles and starting positions where the map is spit evenly between the two teams.

The user defined inputs for map generation are very diverse, with options to control spawning locations of every object. This control can also be used to create user-defined scenarios to test specialized RL, such as playing against adverse scenarios in which there are more enemy agents or a smaller home territory. The configuration can also control the capabilities of the agents allowing for definition of their speed, strength, and observations. Speed controls how many steps on the grid they can take in a specified time, the strength impacts the combat effectiveness, and the observations impact how many adjacent squares the agent is able to see.

Operation

There are two main phases that occur in the environment when an MDP step: the movement phase and the interaction phase.

The movement phase comprises of moving the representations of the agents based on the action inputs, and assessing the outcome of collision events with obstacles and other agents. The movement of the agents is typically conditioned on agent capabilities, for example the UAV agents can fly over other obstacles and other agents, while the ground based agents

The interaction does two main things. Firstly it emulates the combat between the different agents. The combat is based on a survival equation,

$$P_{Survival} = \frac{\sum_{i=1}^{N_{Friendly}} S_i}{\sum_{i=1}^{N_{Friendly}} S_i + \sum_{j=1}^{N_{Enemy}} N_j}, \tag{3.1}$$

where the chance of survival is dependent upon the strength factors of the friendly and enemy agents within a certain range. This equation gives motivation to for agent coordination, both offensively and defensively as single agents are put at disadvantages during combat. The second purpose of the interaction phase is to generate the simulated observations of the different

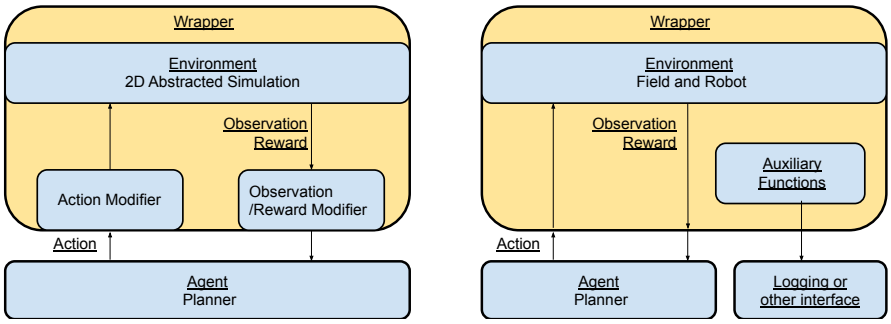


Figure 3.6: Wrappers are a construct of the OpenAI Gym Architecture which allow for easy modification to the environment. (a) The modifications can be used to insert functions to modify actions, observations, and rewards, to simulate a wide variety of systems, such as observation noise. (b) The wrappers can also be used in conjunction with RL based training to serve as logging functions.

agents. This is done by using a simulated observation radius and providing the agent with full observations within the range. While this mimics the certain capabilities such as lidar, which have limited effective ranges, this model does not fully model how some sensors, like a camera, would perform. A camera, paired with a machine vision scheme have high accuracy when identify close targets, with a decrease in accuracy with distance, which will introduce sensor error into the observation. For the sake of preliminary training this was not included into the environment and would be adapted to in later stages in the training pipeline where the machine vision systems come into play.

3.3.2 CTF Modifications

The OpenAI Gym structure supports the use of wrappers, which surround the IO of the environment and can be used to modify the actions, observations and rewards. The most obvious use case of this is to modify the observations or actions to include noise or to center the observations around the different agents. However the wrappers also allow one to define a number of additional functions, such as logging rewards or trajectories, which allow for tighter integration with RL training. These examples are shown in Figure 3.6.

3.3.3 Features and Use Cases

The main features of this environment is simple integration with RL codes, where data is easily generated to train an RL algorithm. The pseudo-code for running an experiment is shown in Algorithm 5, and follows an extremely simplified loop to train the RL a generic RL algorithm

Algorithm 5: RL Integration Pseudo-code

```
Initialize Planning Algorithm
Initialize Environment
repeat
    Observation ← Reset the Environment
    for Until Episode Ends do
        Select action with Algorithm, based on observation
        Observation ← Input Action into Environment
    end
    Update Algorithm
    Perform Logging Functions
until;
```

The simulation framework can also be leveraged in the later stages of the training pipeline to simulate agent interactions and observations, if required. The simulation of interactions allows for consistent performance between the three environments. This environment can also be leveraged to emulate agent observations on physical robots if there is no hardware or software to identify and localize data. This allows for purely the algorithms to be simulated in the 3D environments without the need to create machine vision algorithms and worry about the integration with the system. These methods were accomplished by writing these portions of the code as separate sub-routines that allows them to be called from separate processes.

3.3.4 Adaptations

One of the main considerations for the environment was the ability to test in situ adaptations to different environmental factors. There are four main factors that can be changed to test the adaptability of the agents, which are all reflective of real world scenarios.

1. Environment Obstacles - The simplest thing to change in the environment is the map structure, which is equivalent of testing the algorithm

in a new geographical location. The RL would be evaluated on how well it can generalize different terrain features to adapt it's policy.

2. Enemy agent capabilities - One major military difficulty is adapting to changing military technology. We simulate this with parameters of strength and speed in the environment. The adaptation in this scenario is evaluated on how fast the agent can generalize the new agent capabilities and create a policy to solve the environment.

3. Team Compositions - Team compositions play an integral role in how a team approaches the game, and adding or removing an agent drastically changes that approach. The evaluation of this adaptation is based on how well the policy can use the new composition of agents. Another change that can occur is changing agent capabilities which also has a profound impact on how the policy can solve the environment

4. The last adaptation scenario implemented in CTF is changing the observation structure. The major implementation is reducing the sensing radius, due too environmental factors such as fog. Another impact would sensor error in which observations are misplaced or missing. This posses a challenge for adapting to deal with the new uncertainties in the environment.

3.4 3D Physics-based Simulated Environment

The next progression in the simulation environment is the 3D physics based simulation, which allows for testing of the algorithms in a near realistic environments. This begins to allow the RL algorithms to handle problems such as movement time, where agents will take slightly different times to move. Additionally, this environment introduces emulated sensors, such as cameras, which allow for machine vision algorithms to process data to create observations for the RL algorithms.

The simulation is comprised of five main parts, the communication infrastructure, the simulation environment, the simulated robots, the CTF interface, and the RL interface. Robot Operating System 2 (ROS2) was selected as the common communication interface because it provides a modular

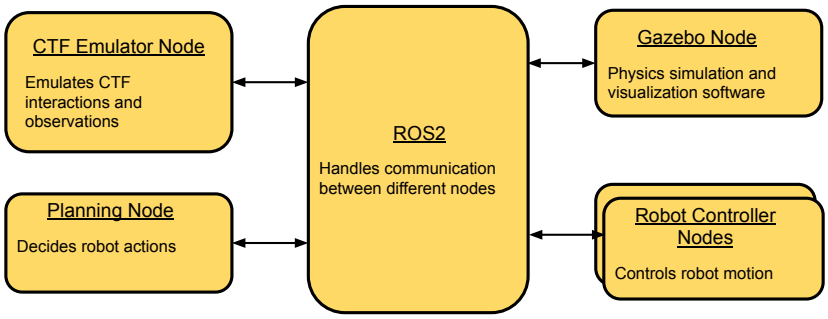


Figure 3.7: The 3D Physics-based simulation is comprised of five main components. The backbone of the infrastructure is ROS2, which serves as a message passing protocol between the other components in the system. The other components operate together to simulate teams of robots playing the CTF game in a simulated environment.

messaging system that can interface with C++ and python code, which are the basis of machine learning code and most embedded code used in robots. The physics engine, Gazebo, was selected foremost for it's design to simulate robots with custom plugins to simulate robotic physics and because it natively supports ROS2 based messaging, i.e. the physics messages are sent and interpreted in ROS2. The robots control infrastructure was then integrated with ROS2 and Gazebo to and were modified to be completely simulated. The python interface described in Section 3.3 was modified to provide to simulate interactions and observations Finally, a node is created that interfaces with different ROS messages and serves as a central processor for all AI based activities. The basic interaction between these components is shown in Figure 3.7.

The components of the simulator are discussed in this section along with several features of the infrastructure, such as HITL capabilities.

3.4.1 ROS2 Communication Infrastructure.

One difficulty in autonomous systems is the ability for different components on robots to communicate with one another, as piping signals to different processes requires careful resource management. This becomes increasingly

complicated when inter-robot communications are required, as the robots need to handle piping of internal and external messages. With this in mind, we selected ROS2 to serve as a modular framework which cleanly handles message passing between different components and systems.

ROS2 is an open-source messaging system, designed to provide robust communication infrastructure between different systems. ROS2 uses the Data Distribution Standard (DDS), a reliable, real-time messaging standard, to handle low level communication protocol between different objects. ROS2 then layers on a robust API which allows for easy integration into existing systems or development of new systems. This API is largely based on a node structure, in which multiple processes, called nodes are linked together to accomplish a goal. In this model the nodes connect together in a publisher-subscriber model, in which publisher nodes broadcast they are sending data on particular channels and subscriber nodes define which data channels to listen too. ROS2 handles connecting the subscribers to the publishers and creates reliable pipelines for data to be transmitted between nodes.

To illustrate the power of ROS2, consider the problem of where a robot autonomously explores an area. This problem can be decomposed into three main tasks: observe its surroundings, process observations and decide navigation goals, and use motor commands to navigate to the goal. In ROS2 we can represent these tasks as individual Nodes, and the data passed between the Nodes as messages. The information begins when the Observation Node reads all data from the cameras, lidars and other sensors, and combines the data into an observation. This observation is sent to the Planning Node which processes the data and decides where the robot should navigate to. This goal is sent to the Navigation Node which drives the robot to the goal, where the cycle begins again. Figure 3.8 shows the example node scheme.

This node based structure has two notable benefits for development of systems. Firstly, different components can be added to the node graph, subscribing to existing messages, which eases integration of new components. For example, say we wanted to record all of the observations, we could add a Node which reads the observation messages and saves them to a file. Secondly, the node structure allows for the substitution of nodes, because the model creates a consistent messaging protocol which allows the Nodes become easy to swap. In the previous example one could swap out the Planning Node with a more robust Planning Node with more observation processing.

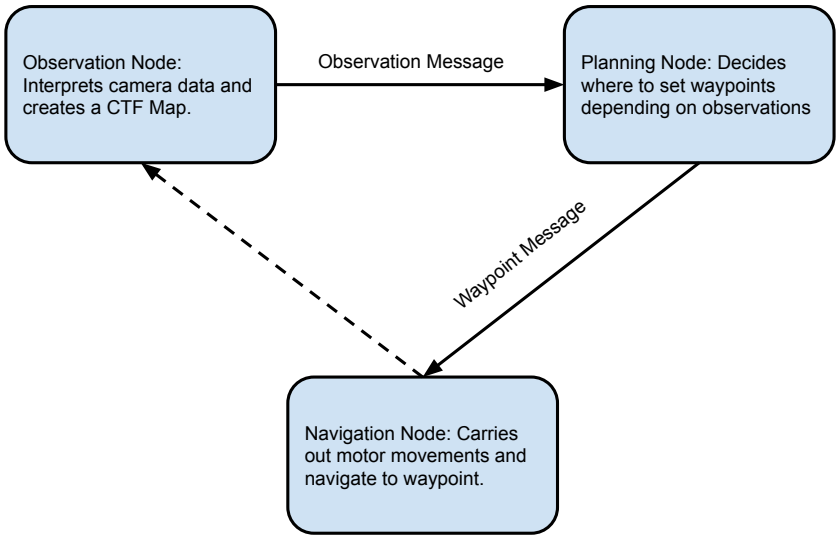


Figure 3.8: An example node structure which decomposes the task of navigating a complex environment into three main operations, which communicate with one another. The task begins with observing the environment with sensors in the Observation Node, which sends processed data to a Planning Node. The Planning Node then determines where the robot should navigate to, and send the information to the Navigation Node which executes the movement. The loop is closed when the robot moves and the processed begins again.

Within ROS2 each of the other components of the simulation can be reparameterized as nodes, allowing us to simplify the architecture diagram, and draw direct message connections between the different systems. The architecture for 3D CTF Simulation is described in Figure 3.9. The purposes of different nodes and messages are elaborated in the following sections.

3.4.2 Gazebo Simulation Software

Gazebo is a physics based simulation platform that was designed to simulate robots, by allowing the use of custom plugins to simulate robotic functions such as motor movement or sensor emulation. An example Gazebo simulation is shown in Figure 3.10, in which robots drive around a simulated

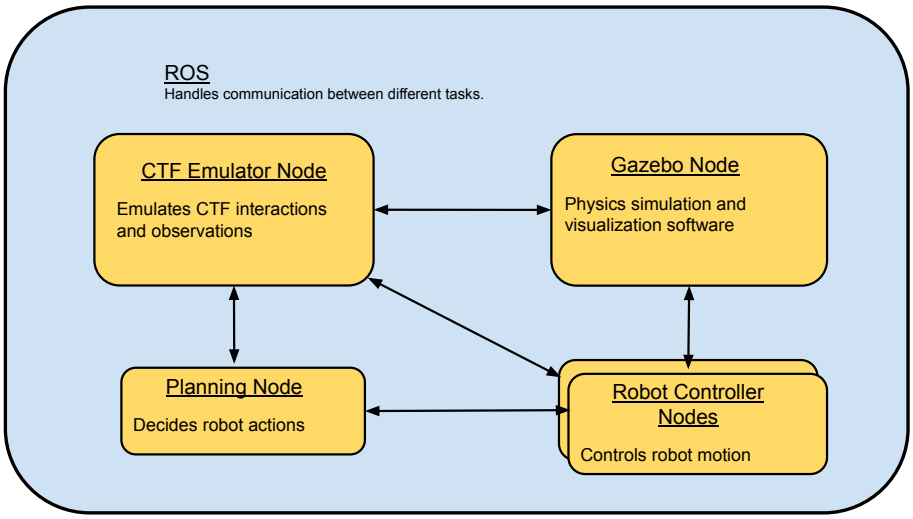


Figure 3.9: The simulation architecture defined with respect to ROS2 message passing. The diagram shows the interconnected nodes and describes their respective purpose in context of the simulation

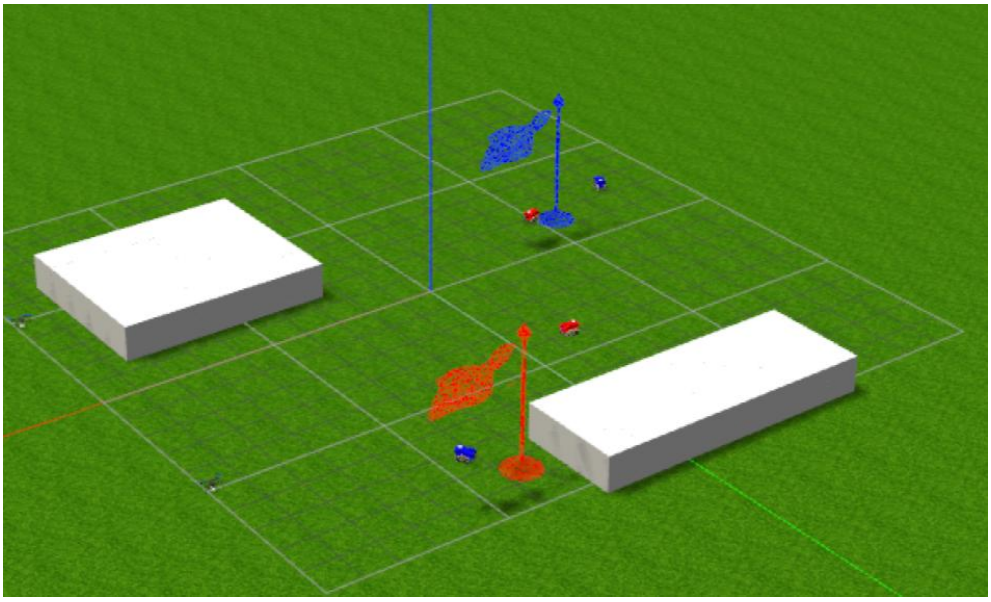


Figure 3.10: A simplified CtF Game played in Gazebo. Simulated robots use the same software stack as physical robots.

environment playing a game of CTF.

Like most physics based simulations a Gazebo simulation is parameterized with a set of physical objects, such as the ground, robots or obstacles, which are defined with contact surfaces. The first step of the Gazebo simula-

tion is simulating collisions and interactions between the surfaces of different components, which result in forces and moments on the different objects. Concurrently, Gazebo runs plugins which simulate force and moment applying devices such as motors, which are summed with the collision forces and moments. The simulator then uses finite time steps to calculate object motion, based on the above forces and moments.

the objects in the simulation are created in XML format which defines three attributes of the object of the robot. The first is the physical properties of the object, such as contact surfaces, such as the exterior shell, and physical properties, such as mass and friction, which govern the interaction with other components. The second is to define joints of the object, such as wheel's rotational joint, which allow for constrained motion to be simulated in the interactions. Finally, any plugins associated with the object or robot are specified.

Plugins serve two major purposes: to simulate the motion of different components and to emulate the operation of complex sensors. The motion based plugins are described as control inputs, such as motors which move different components such as wheels or propellers of the vehicle. These plugins often take an external control signals which creates proportional motor responses, similar to how a real system would operate. In the case of the UAV's there are plugins that simulate the aerodynamics of the propellers based on lift and drag coefficients. The other plugins emulate the operation of sensors, such as the GPS and IMU of the robots, which use physical properties of the simulated environment, such as the object's current location and velocity, and calculates outputs comparable to traditional sensors. Other components such as cameras and lidars can also be simulated in Gazebo which can be used to run machine vision and localization algorithms.

These plugins control the sending and receiving of ROS2 messages to other components in the 3D CTF Simulation. In this setup the motor plugins for the UAV and the Terrasentia robots takes in ROS2 messages to move propellers and wheels, respectively. On the output side sensors plugins output verbose sensor information to specific ROS2 topics.

The different components, plugins, and interfaces of the gazebo simulation engine are shown in Figure 3.11.

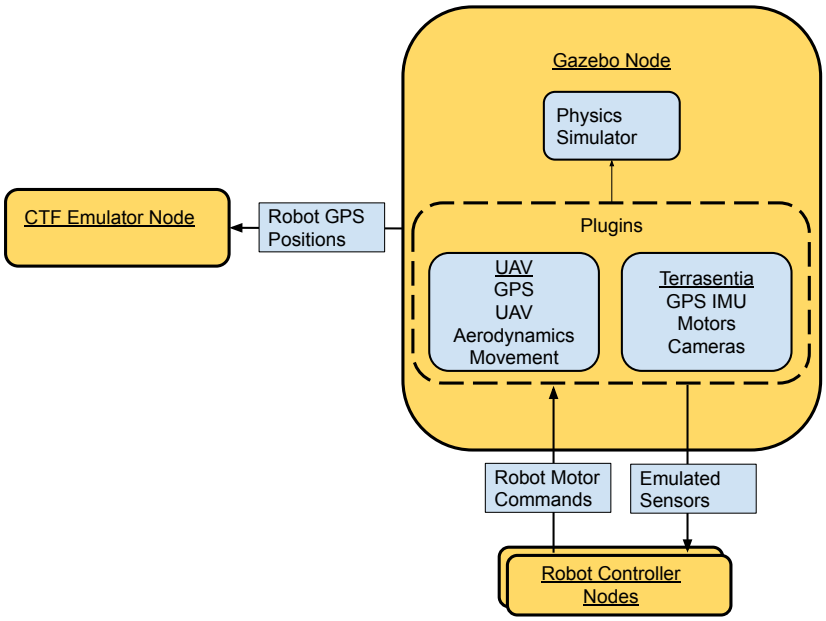


Figure 3.11: The Gazebo Node in the 3D physics-based simulation is primarily dependent upon the operation of the physics simulator and the plugins which simulate motion and the sensors of the robots. The node outputs sensor data, specifically GPS, and motor encoders, which are sent to the CTF emulator and the robot controllers. Motion plugins which control motion of wheels and propellers in the simulation take an input from the robot controllers.

3.4.3 Simulating Robot Control Software with Docker

To simulate the software of the robots, several changes needed to be made to the control architectures to allow them to interface with the Gazebo simulator. Changes to simulate the Terrasentia Robot and the UAV are described in the following section.

Terrasentia Robot

The main controller of the robot is a custom controller which robustly monitors the sensors from the Robot and can be used to autonomously control the robot. The main component of this system is a robust MPC controller which can be used for path planning and following, which takes a way-point

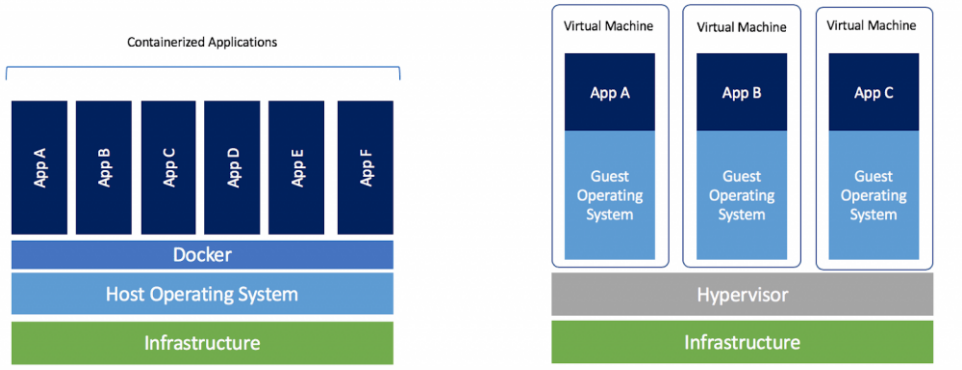


Figure 3.12: A docker deployment of software, shown on the left requires minimal infrastructure as all of the software is self-contained in separate containers. In a traditional Virtual Machine deployment of systems each software requires a separate guest OS which increases simulation overhead.

input and navigates to the desired location. This controller is traditionally ran on a Raspberry Pi computer, which provides it access to sensors on the robot through the use of Bluetooth, WiFi and GPIO pins. To allow this to existing controller operate in the 3D simulation, two main changes were made to the operation of the controller.

The first was to allow the controller to run be run on the same machine as the simulator, through the use of Docker software. Docker is serves as a software abstraction layer, allowing one to emulate different software stacks on one host machine, with less performance loss than a traditional virtual machine. A comparison between native software, virtual machine, and a Docker image is described in Figure 3.12. Docker was used to create a software stack equivalent to that of the original controller, but formatted to be compatible with the host machine. Another thing to note about Docker is that it allows the emulation of 32bit software on 64bit machines, which allows for a more accurate representation of the controller to be simulated.

With the controller accessible on the host machine, the second integration point was to create the interface with the robot to ROS2, as the original controller was hard-coded to accept specific sensor data from physical sensor inputs. This required creating alternate channels with which data could be passed to the controller which would take the simulated sensors from Gazebo. This allowed the controller to use both emulated and real signals based off a configuration file. This change of including a ROS2 interface also brought

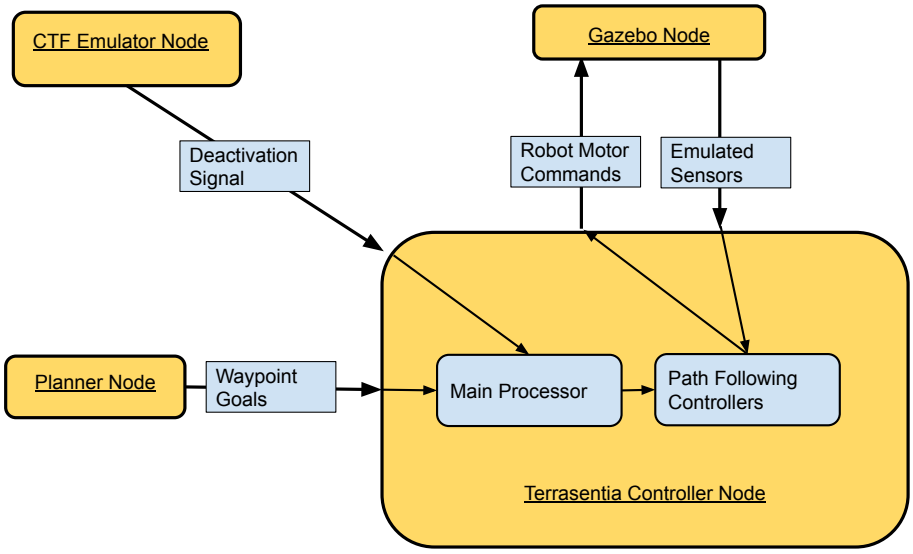


Figure 3.13: The main processes of the Terrasentia are the main processor which in simulation mainly controls the use of the way-point following controller. The Terrasentia interfaces with all of the other nodes in the system to take in way-point commands from the planning node and output motor commands to the gazebo plugins.

a modularization of the communication interface, which allowed for external messages to be sent to the robot, such as observation messages from other robots. The IO of the system is shown in Figure 3.13

UAV Robot

Similar to the Terrasentia Robot, the software controller for the UAVs is simulated within a Docker Image, allowing the controller to be simulated on the host Machine, in parallel with the other software components. ArduPilot control software, an open-source flight control software, which can be used to control a variety of vehicles, such as ground robots, boats and UAVs, was used to control the UAV's. Our specific implementation uses a specialized branch, ArduCopter, designed for rotor-craft control and navigation. It primarily uses PID controllers to fly and control the robot, but has a variety of different options based on the high level control features such as way-point navigation and setting flight parameters.

One aspect of the software that is different with the UAV is that the

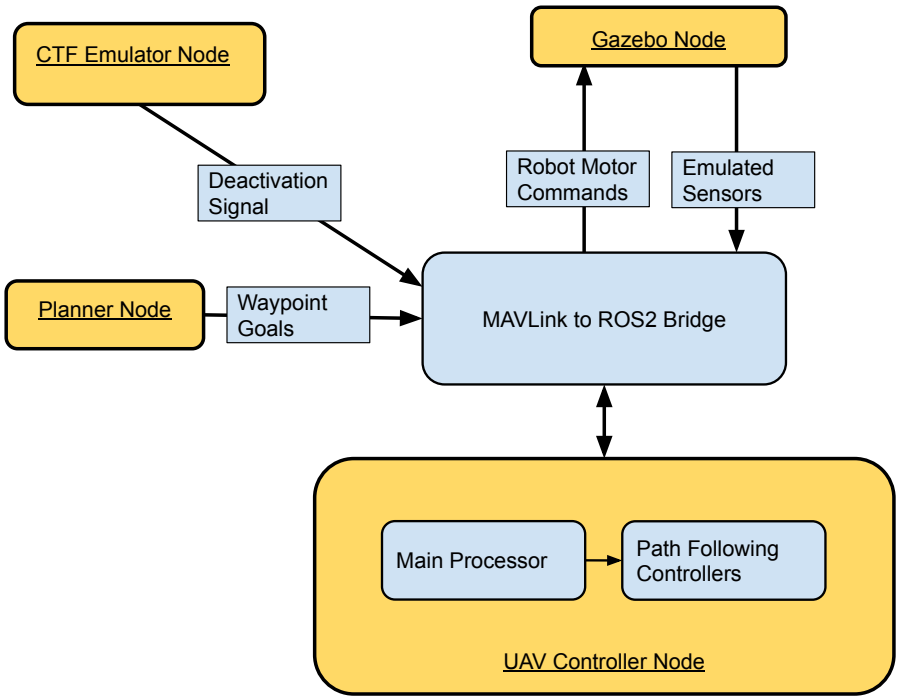


Figure 3.14: UAV functions are similar to that of the Terrasentia, in that there is a main controller that decides the use of the path-following controller. The main difference of this system is the use of a MAVLink to ROS2 Bridge which converts the ROS2 messages from the other nodes to MAVLink messages than can be interpreted by the Ardupilot controller.

communication infrastructure with the software is extremely well defined with the MAVLink messaging protocol, which is a standard messaging system to define information packets specifically for UAV's. This messaging system is discussed in more detail in Section 3.5.2 with the physical implementation of the UAV's. In simulation these messages are emulated using the open-source flight-simulator for ArduCopter. To interface with the robot a bridge was created which converts messages between MAVLink and ROS2. A similar approach is deployed in the real robot.

The full simulated architecture for the UAV is shown in Figure 3.14.

3.4.4 CTF Emulator Node

An ROS2 interface was created for the python environment which allows data from simulated or real robot to be inputted into the simulator. This

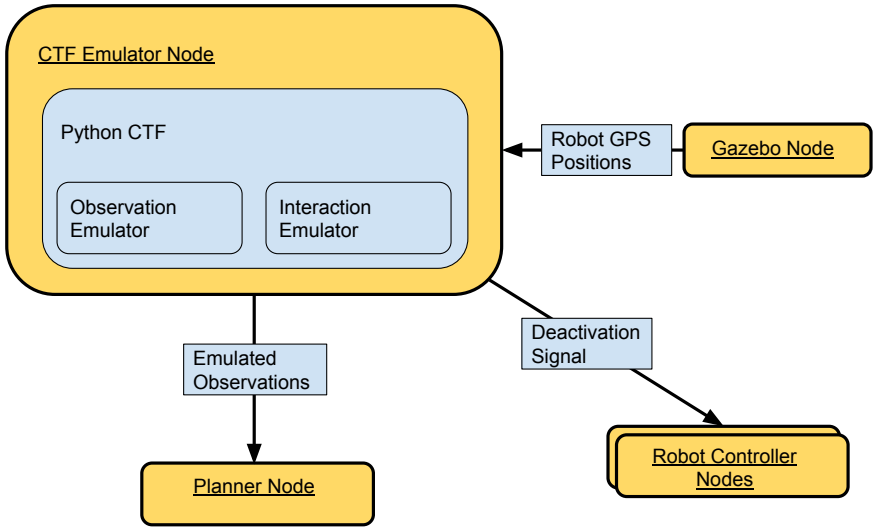


Figure 3.15: In the 3d physics-based simulation the CTF interactions and observations are simulated with functions derived from the 2D abstracted simulation. The message interfaces with other nodes are shown.

input, typically a GPS signal, is then localized onto a pre-defined 2D grid, updating the agents' position within the simulated environment. This localized representation is then used to simulate environment interactions, which is performed every time an agent position update is made. The result of the interactions are then sent to the different robots to disable their movements if they die in the game. The localized 2D representation is also be used to emulate the observations of different agents. The architecture is fully drawn out in Figure 3.15.

3.4.5 Planning Node

The main purpose of the planning node is to process observations from the robots and decide the optimal actions for each of the robots to take. This node mounts algorithms trained in the 2D simulated environment to evaluate their performance in the 3D simulation, and they can be updated to account for any differences introduced with the 3D simulation. The algorithm selects primitive actions for each of the robots to take in the 2D environment.

The node processes the actions into interpretable messages for each of the robots and sends them via ROS2 to the simulated controllers. This node

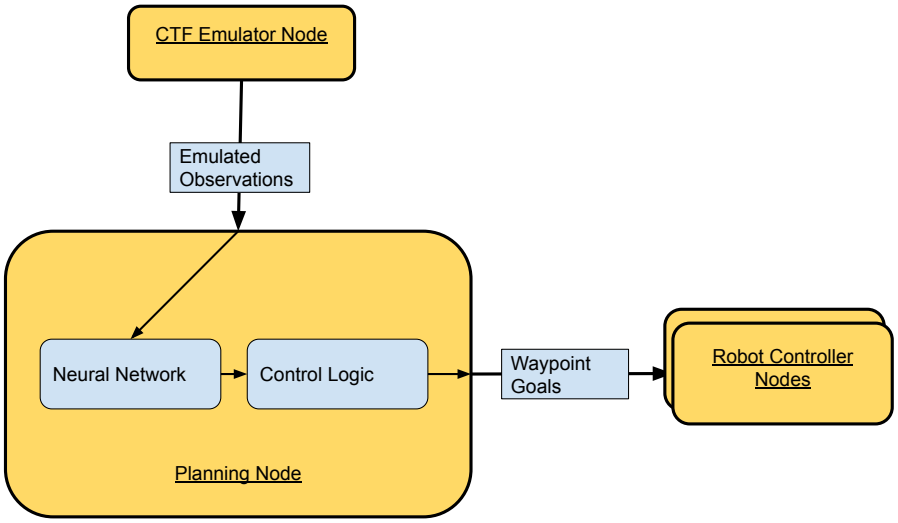


Figure 3.16: The CTF planning node serves one main purpose, to mount the RL algorithms, based on neural networks, which take select an way-point based on an observation. The node also contains some control logic which monitors way-point following.

also contains protocols which verify that the messages were received by the individual controllers. To further verify the way-point following performance of the robots the node contains several safety mechanisms which monitor the progress of the robots, and is used to detect faults in way-point following which are often bit-errors in the messages. This architecture is shown Figure 3.16.

3.4.6 Hardware in the loop Capabilities

Another key component of the setup, with the Docker images is that there is potential to run the Docker image on any computer. This allowed us to run the robot controllers on remote machines and allow use to test the networking infrastructure that was used in the real demo, as described in Section 3.17. It also allows us to evaluate if there are any computational limitations that exist in the desired hardware, such as the Raspberry Pi 3B+ which is used as the main controller for the Terrasentia Robots. This HITL structure is shown in Figure 3.17

This is analogous to moving the controller to the physical robot, where a

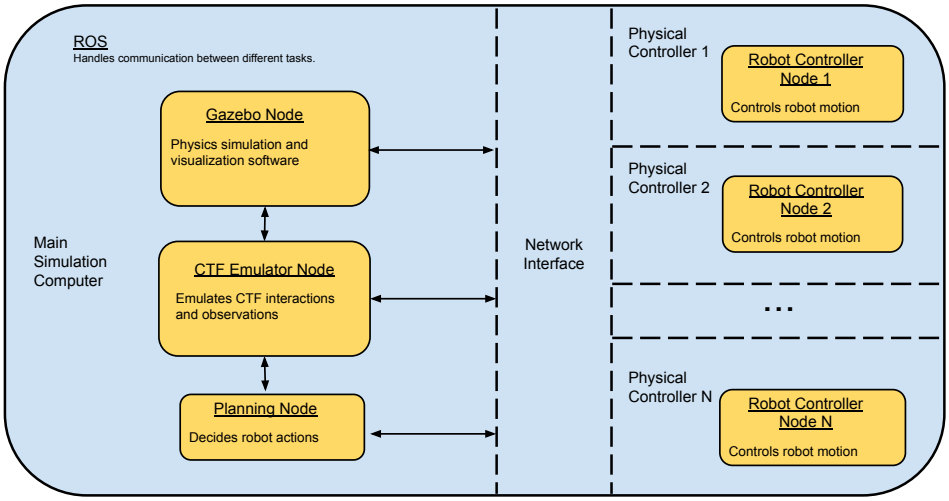


Figure 3.17: The simulation architecture offers a hardware in the loop option. This involves operating the individual robots on remote machines, which will test the networking connections between the different robot entities.s

communication infrastructure must be deployed to allow the remote robot communicate with the host machine.

3.4.7 Machine Vision Integration

The 3D CTF Simulation also supports the testing of machine vision algorithms for the use of classifying and localizing objects. In this framework the simulated camera and lidar outputs from Gazebo can be passed into these algorithms, and they can be directly compared to ground truth object locations. This allows for complex evaluation of the algorithms performance in situ, which can't be performed on classical image data-sets, which have little heuristic information for evaluation. For example, one can test an algorithm to detect and estimate the distance to an enemy robot, and can correlate classification error to distance or orientation of the camera and use the information to fine tune their method.

These observation algorithms can also be integrated with the rest of the CTF pipeline by feeding these observations into the CTF environment instead of the simulated observations from the 2D Emulator node. This change results in an end to end localization and planning architecture for simulated

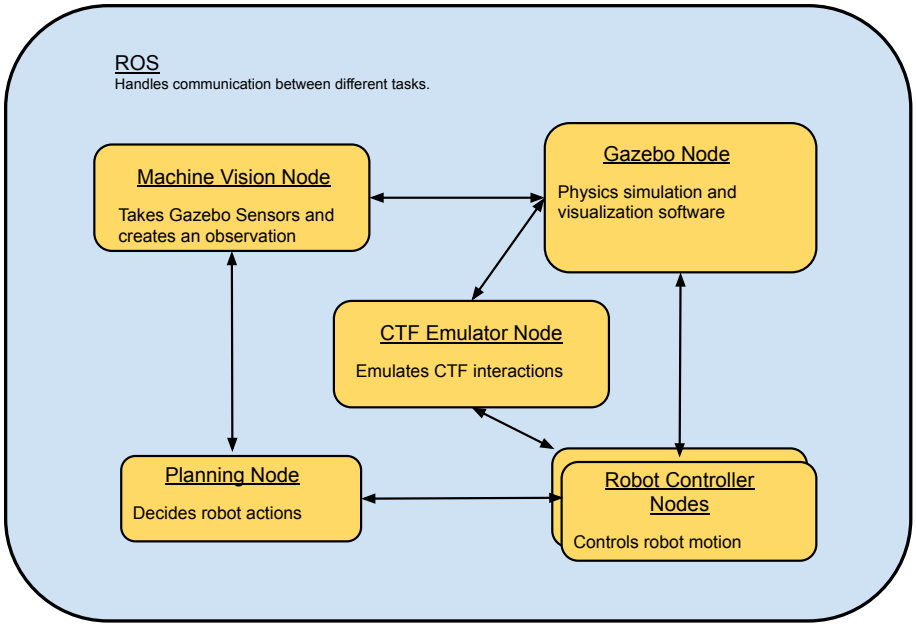


Figure 3.18: To test Machine Vision algorithms, the observations from the CTF Emulator is replaced by the localization algorithms.

robots, which is shown in Figure 3.18

3.5 Real-world CTF

Because this environment is similar to the 3D physics simulation in practical implementation with controlling the robots and message sharing, it is able to reuse the software components described above. However, several new components are required for the introduction of physical hardware. The full environment structure is shown in Figure 3.19.

This section discusses the hardware components of the environment, the Terrasentia robots, UAVs, and the physical communication infrastructure.

3.5.1 Terrasentia Robots

The ground robots used in the CTF game are the Terrasentia Robot, a robotic platform developed in partnership with UIUC's DASLab and EarthSense Robotics. Only a small changes were made to the existing robotic platform.

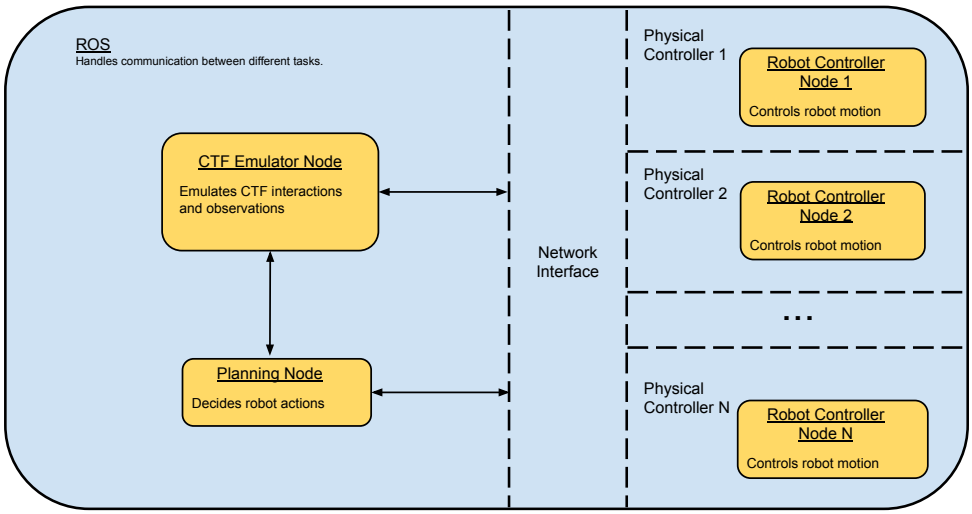


Figure 3.19: The real-world simulation is similar to the hardware in the loop simulation, but operates without the Gazebo node. The CTF Emulator and Planner communicate over complex networking connections which leads to the control of the individual robots.

The main control software is run on a Raspberry Pi, which provides real-time interface with a variety of sensors, but secondary tasks, such as observation processing can be performed on the auxiliary computer that is also in the robot.

The Terrasentia robots are a modular platform that has a wide variety of sensory capabilities, and in this implementation we utilize three main packages. The first package on the robot is the gimbal stabilized cameras, which can be used with AI algorithms to detect and localize different objects, such as enemy robots. The second is a lidar sensor, which primarily can be used to identify objects and map the environment, but can also be used in conjunction with the cameras to help localize objects. The final package is the positioning system which uses a differential GPS receiver which reduces GPS uncertainty to a few centimeters, which allows for extremely accurate paths to be followed.

To interface with the robot, a WiFi receiver with a high gain antenna was installed on the robot. This was used to connect to local WiFi networks and allow the robots to communicate with one another as well as the base station. The base station is used to decide the movements of the robots, as

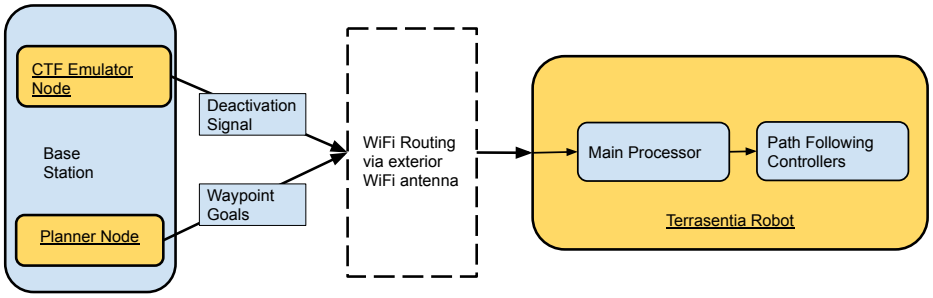


Figure 3.20: The Terrasentia Robot contains the same software controller used in the 3D simulated environment. The main progression in this environment is the WiFi receiver that handles the communication with the other nodes in the system.

well as simulate the CTF Observations and interactions.

The entire hardware and software infrastructure of the Terrasentia robot is described in Figure 3.20.

3.5.2 UAVs

Two UAV systems were integrated with the Real-World CTF Game, the 3DR Solo and a custom made Pixhawk-based UAV, which are shown in Figure 3.21. Both of these UAVs serve the same purpose of being a UAV system that can identify objects and their respective locations, but they have several key differences in their software and hardware capabilities, which differentiate there roles.

3.5.3 **3DR Solo**

The 3DR Solo's was integrated into the architecture with as shown in Figure 3.22. One of the most important things to see here is the specialized connection interface through the controller. This connection is a high bandwidth 5g WiFi connection that enables the Solo to stream High Quality video short distances to the controller and by proxy, the base station. This architecture can be leveraged in a centralized processing scheme, in which all of the observations are processed by one algorithm, which can improve information. The benefit comes with several trade-offs; firstly there is no processing capabilities

(a) (b)

Figure 3.21: (a)The 3DR Solo (b) Custom built UAV with a Pixhawk flight controller

available on the Solo, and no way to interface external computing into the system. This means that no edge processing can occur on the Solo, unless an entirely separate system is attached to the system with a separate camera, battery and processing module, which would limit the flight capabilities of the system.

This limitation also extends into the software side of the robot, which uses a heavily tailored version of ArduPilot, which optimizes performance of the 3DR with respect to it's hardware components. But this comes with several limits to the configuration of the UAV. The foremost is that the customization created several differences in how several core systems, primarily the way-point navigation, a changed to use different messaging conventions, which need to be dealt with on the planner side of the system. The second is that no customization can be performed in the software to either change flight characteristics or allow additional information to be passed to the base station.

This UAV can be used in close range missions to the base station to feed video data to the Base station over a high bandwidth 5g WiFi. However it cannot perform many other tasks as the software and hardware interfaces with the UAV are quite limited.

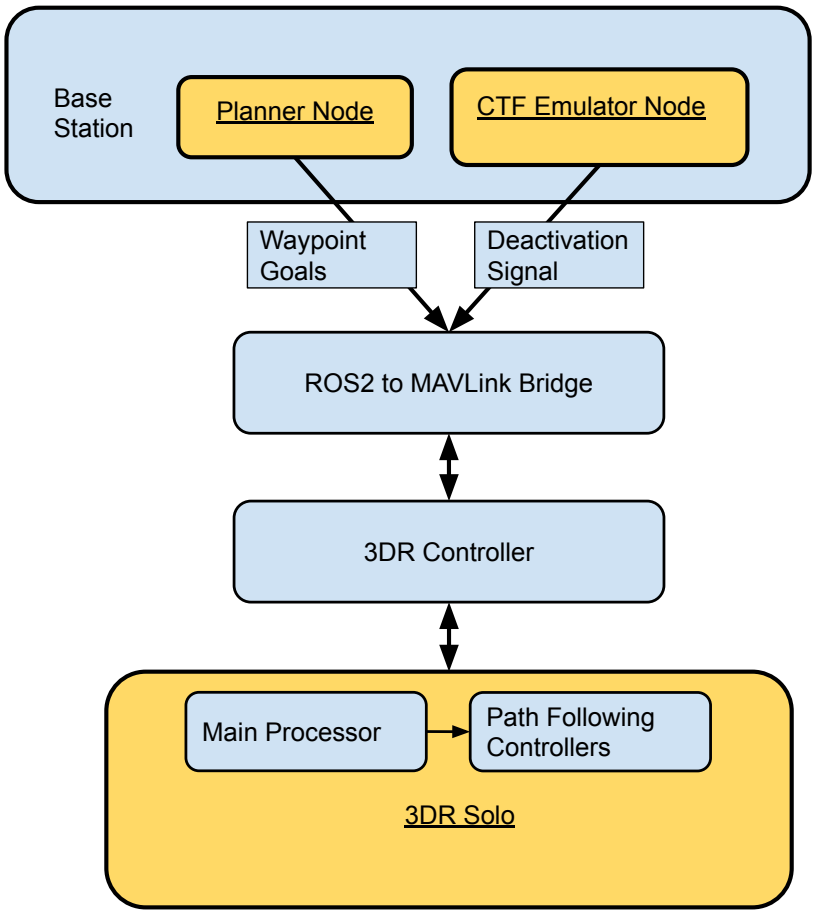


Figure 3.22: The 3DR Solo communicates with the other nodes in the system through the controller which creates a 5GHz WiFi connection to the UAV.

Custom UAV

The Custom UAV is integrated into the CTF system with a slightly different configuration, which allows for a more modular approach to observing and interacting with the system. This interface is shown in Figure 3.23.

Compared to the 3DR Solo, the custom UAV is unable to stream high quality data with it's current hardware setup, but has the ability to interface with small external compute modules, which allow for the collection and processing of data on-board. In the setup we mount a camera and Raspberry Pi package to the UAV, which allows for. The Raspberry Pi with an external VPU, the Intel Movidius, is able to process images and make appropriate detect ions for the CTF game using either blob detection algorithms or complex

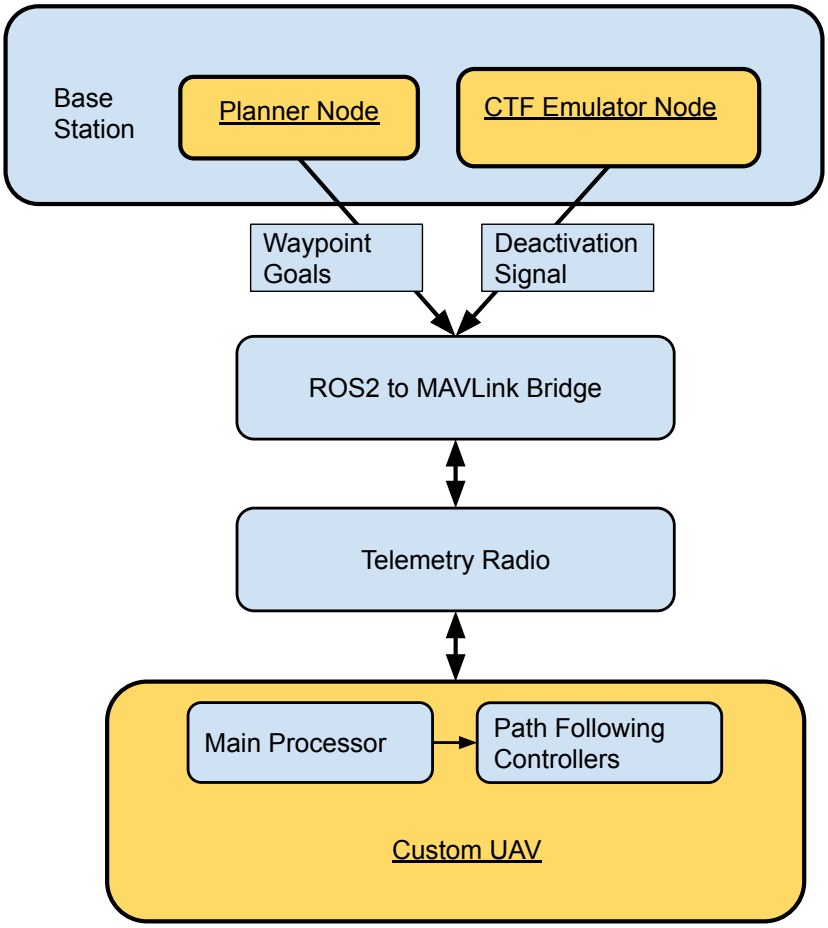


Figure 3.23: The Custom UAV uses a Telemetry radio connection to communicate between the UAV and the other architecture components.

neural networks.

Additionally the system uses the default version of Ardupilot, which is open-source and allows for the definition of custom messages, and the ability to send them over the low bandwidth telemetry radio connection. This tight integration of systems allows for reduced weight on the UAV, and simplifies the software connection to control the UAV from the planner's perspective.

To demonstrate the system for future use we defined a MAVLink message type to be used to share observation information with the base station. Due to the limited bandwidth of the telemetry radio we minimized the observation to consist of the identification and localization of unique objects. The

message is described with a tuple of information:

[Latitude, Longitude, Identification],

In which the location of the object is presented in a GPS coordinate system and there is an agreed upon identification standard a-priori. If multiple objects are detected in the image, then the multiple messages are sent. The planner is designed to interpret this and populate them into a 2D CTF grid representation.

3.5.4 Physical Control Architecture: Networking

The networking between the different robots is similar to the simulated architecture in that it relies on ROS2 to send the data between the different processes. The main difference is that the messages need to be routed to different physical systems instead of being simulated software controllers on the same computer. This physical infrastructure differs between the ground robots and UAVs and required several different infrastructure upgrades to support the game. This architectures interaction with different systems is diagrammed in Figure 3.24.

Wifi Setup

The Terrasentia Robots were only controllable with TCP/IP data schemes, i.e. WiFi and Ethernet, which severely limit the bandwidth and transmission range for data without dedicated hardware infrastructure. To satisfy the bandwidth requirements of ROS2, outdoor WiFi access points were installed in the demonstration area, which would allow for high speed communication between the robots and team base stations.

This architecture was tested with the WiFi receivers, for latency, bandwidth, and packet loss across the field. The heat-map below shows that there were one or two areas with interference from trees, but the connection was still good enough to send ROS2 messages reliably. There was an issue with a low static packet loss over the entire field, where less than 1% of packets would net reach the intended robots, which could result in slight losses in data. This was addressed with the control infrastructure which uses

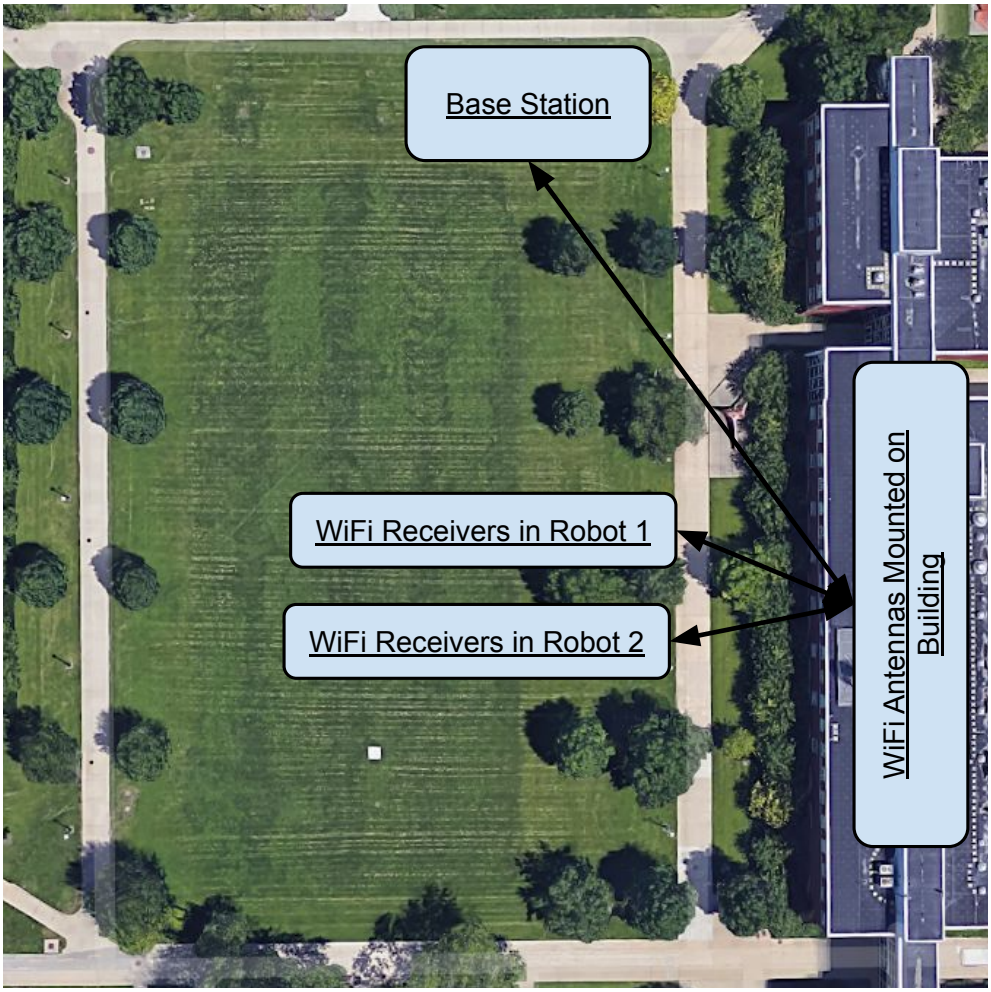


Figure 3.24: The communication infrastructure consists of three major pieces, the base station, the antenna and the robot receivers. The base station runs all of the planning and CTF emulation, which is sent through the WiFi antenna to individual robots.

a send and reply type execution where the robots would verify that messages where received properly. Additionally the system was designed to monitor locations of the robot to ensure that it was going to the destination.

The other issue experienced with the networking was that it operated over a controlled Network, which limits the ability of the underlying DDS of ROS2 from connecting to the robots. This was addressed by creating a sub-net which was used to route information between the designated robots. Specifically we used Weave Net which is used to network Docker Containers across the internet.

Controlling the UAVs

UAVs are controlled with a Telemetry Radio. A bridge transfers the ROS based messages to encoded messages into MAVLink messages which are the standard for UAV control and reporting.

Custom messages where created which could send state information, such as current location and observations. Telemetry output from the Raspberry Pi would be passed through the base station. This could be used to run algorithms at the edge and autonomously control the aircraft.

An flight controller was used as an override to the system. The connection is made using a separate channel so there is no interference of messages. The controller is able to switch the mode of the UAV to a manual flight mode which can then be used to maneuver or land the UAV. In the control software the planning infrastructure would detect this switch and then cease operation of the UAV.

CHAPTER 4

HRL FOR ADAPTATION

HRL has is a useful structure which can be leveraged for adaptation. In this format the policy is distributed between different sub-policies, which allows the HC to switch how it uses the sub-policies to create adaptation. Consider an example scenario from the CTF game in which you trained to play against an agent, and at game time the enemy agents were suddenly faster. Following a standard RL policy and adaptation scheme you would have to re-learn every decision to make in the environment to address the difference in the enemy agent. A better way to add address the new enemy is to learn a variety of different strategies beforehand, and when playing the new enemy identify which combinations of basic strategy performs the best against the new agent.

This use of HRL presents one major difficulty during training: the HC has difficulty assigning value to reward-rich sub-policies with short trajectories, i.e. rapidly switching between sub-policies makes it difficult to discern if a specific sub-policy was useful. Towards this, this section discusses a confidence based training method, which improves training speed and consistency when compared to traditional methods.

This chapter begins by discussing formulation behind using HRL for adaptation and formulating a confidence-based training method for HRL which improves training time and stability. Finally results run on in the 2D Abstracted CTF environment demonstrate the performance of the proposed methods.

4.1 HRL for Adaptation

HRL originally was developed to allow for temporally extended decomposition of the MDP, leading to success in temporally extended and sequential

tasks. We aim to extend this framework to help with adaptation by creating higher-level strategic abstractions (i.e., sub-policies) that can be exploited by an HC during adaptation. We focus our policy updates on the HC to achieve more drastic changes in explored trajectories than updating a lower-level policy that only controls primitive actions.

During adaptation HRL can leverage the multiple sub-policies, π_ω, by switching the termination and initiation sets, which drastically effects the overall policy, π. On the surface this immediately changes primitive actions, but it also results in drastic changes in explored trajectories, improving training sampling. This is desirable to quickly explore and learn a new HC to create an optimal policy for the environment change. We propose, and our experiments indeed show, that using HRL in adaptation scenarios improves the policies' ability to change compared to traditional DRL.

Our HRL architecture is shown in Figure 4.1. We model the sub-policies and the HC with separate neural networks because they operate on different temporal scales, requiring unique spatial and temporal feature encodings of the state. Temporal features are captured by stacking four consecutive frames, which eliminates the need for recurrent elements in our networks. Spatial features are captured using a convolution neural network with a non-local attention block, which can correlate distant features to enhance the understanding of abstracted information, such as tactical position and team coordination [27]. We use a shared feature encoder to allow the sub-policies to share features for improved encoding density. Our networks terminate with two fully-connected layers to create the actor and critic used for training with PPO.

We train a policy for the source environment in two phases, by separately training the sub-policies and the HC. The first phase trains each sub-policy independently. We train these sub-policies using heuristically defined sub-rewards for this effort, as discussed in Section 4.3.2. We leave the exploration of alternative methods for defining and training sub-policies, such as unsupervised discovery, as future work. The second phase then fixes these trained sub-policies and uses them as a basis for training the HC. During adaptation, the trained sub-policies and HC are directly applied to the target environment and only the HC is updated online. We use confidence-based training for the HC to select sub-policy usage, which is used the source and target environments. All network updates are made using the PPO algorithm.

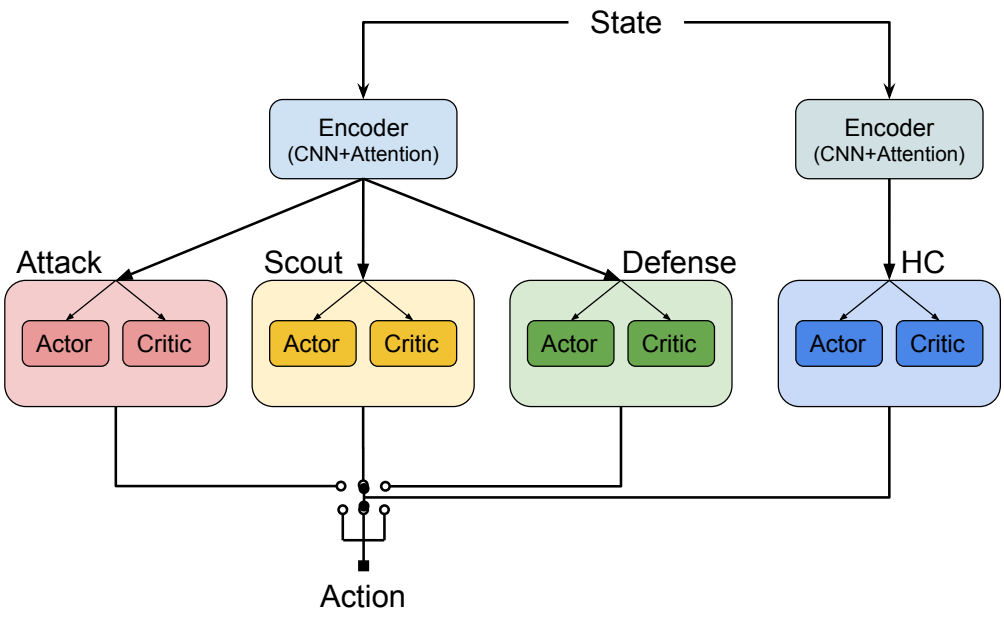


Figure 4.1: HRL architecture where an HC selects which sub-policy each agent should use at a given state.

4.1.1 Sub-policies

We place two assumptions on the sub-policies used within our architecture. First, we assume that the sub-policies overlap and are distinct in the MDP, which allows the HC to switch between distinct sub-policies to enable adaptation as described earlier. We satisfy this assumption by independently training sub-policies with different engineered rewards. Training sub-policies in parallel (as done in many existing HRL implementations) could lead to sub-policies that only span a localized region of the MDP. Second, we assume that optimal adapted policy can be achieved with the initially defined sub-policies since we do not update sub-policies during adaptation. This assumption is difficult to satisfy in many scenarios and we do not propose formal guarantees for meeting it; we hope to address this point further in future work.

There are several different methods to create sub-policies that satisfy the different assumptions. The simplest is to train unique strategies or behaviors in environments with tailored reward functions, creating sub-policies with diverse information encoded by optimizing for different reward structures. This also has the benefit of creating interpretable sub-policies as the sub-

policy is directed based on a human controlled reward function. Another method is to decompose the environment based on geometrical or task based factors and create policies based on the decomposition, allowing for implicit sub-policies based on the environment to be learned, improving the diversity of information. However they are not often not human interpretable, leading to issues with practical implementation as some level of knowledge on the goal of the system is desired.

For this work we use the first option of training the sub-policies with tailored reward functions. We discuss specific sub-policies used in Section 4.3.2.

4.2 Confidence based HRL

A challenge with classic HRL is reward assignment for the HC. If the HC makes decisions every step during training, advantages from selected sub-policies cannot be distinguished from each other well enough to provide stable updates to the HC. On the other hand, if the HC makes decisions every n steps, it may provide better reward assignment, but with the potential for poor convergence due to mis-matched temporal scales and poor sample inefficiency.

We propose a confidence-based training method for improved training of the HC, summarized in Algorithm 6. Our algorithm uses an intrinsic confidence measure and associated switching rules to control when the HC is allowed to switch sub-policies during training. This approach allows for longer trajectories to be sampled early in training when confidence is low in the HC policy, resulting in more stable updates. However, the network will switch more frequently later in training when confidence is high in the HC, resulting in more exploitation of the environment. We calculate the confidence measure as the information entropy of the HC's policy output. We then define a constant confidence threshold to limit when the HC is allowed to switch sub-policies. We focus on a constant threshold for this effort, but note that dynamic thresholds can be easily implemented. An notional example of our method is shown in Figure 4.2.

Our method is simplistic, but supports HRL adaptation because confidence drastically decreases when the network experiences new scenarios (thus pro-

Algorithm 6: HC Confidence Based Training

```
for Episode do
    Initialize Switching Threshold;
    while Episode Running do
        Evaluate internal confidence metric if Confidence ¿ Threshold
         then
            Switch sub-policy;
            Set new threshold;
        else
            Modify threshold;
        end
        Compute Segmented Reward for HC;
    end
    Update Controller based on Segmented Trajectory;
end
```

moting exploration), but then gradually increases given more samples (thus promoting exploitation). A possible extension of the method is to incorporate metrics that directly capture uncertainty in the environment into the confidence measure, for example by implementing anomaly detection techniques or comparing expected and actual rewards.

Confidence-based training provides two advantages to common alternatives for addressing HC reward assignment in HRL: reduced network complexity and more exploration during adaptation. For example, compared to termination functions in option-critic architectures, our method does not use additional networks, which reduces the number of parameters to update during training. The adaptation performance of option-critic architectures is also limited because their termination functions are heavily tied to the HC, making it difficult to effectively update them during adaptation.

4.3 Experimental Setup

The main objective of our experiments was to compare the adaptation performance of our HRL architecture to alternative solutions. We compared three methods for creating optimal control policies in our environment: our HRL architecture trained with confidence-based training for the HC (confidence HC), our HRL architecture trained with fixed-step updates for the HC

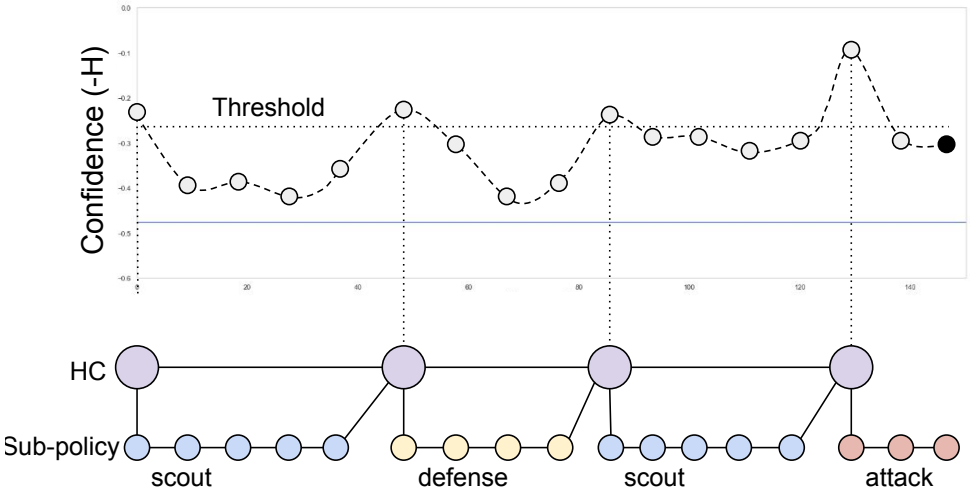


Figure 4.2: An example implementation of confidence-based training. The HC begins by assigning the "scout" sub-policy to an agent, which then implements that sub-policy for several steps. The HC is not allowed to assign a new sub-policy to the agent until the confidence threshold has been reached, at which point it assigns the "defense" sub-policy, etc.

(fixed HC), and a baseline non-hierarchical PPO algorithm (PPO) [28]. We introduced changes to the environment by varying the strength and speed of red team agents after initial training, while keeping the capabilities of blue team agents fixed. We then compared the ability of each method to adapt to these new target environments. The following sections describe details of our experiments.

4.3.1 Training

We used the PPO algorithm to train all networks used in our experiments, using the same hyperparameters throughout for consistency. The fixed step parameter for the fixed HC method was selected through experimentation within the source environment. As discussed in Section 4.1, we trained our HRL architectures in two phases, one for the sub-policies and one for the HC. The sub-policies were trained until they converged to steady-state performance in their respective partial game settings; more details about sub-policy training are provided in Section 4.3.2. Once the sub-policies converged, the HC was trained in the source environment until it converged to

steady-state performance. The baseline PPO was also trained in the source environment until it converged to steady-state performance. All methods generally reached stead-steady performance around episode 150,000. After fully training the HC and baseline PPO, enemy capabilities were changed to create a target environment for the HC and baseline PPO to adapt to. We considered eight target environments, where enemy agents were given some combination of faster or slower speed and weaker or stronger strength. We ran four replicates for each source and target environment and calculated mean performance metrics from those replicates. Most replicates showed win rates within a couple percent of the mean.

We trained policies to control the blue team. During training, the red team policy was randomly selected from a set of heuristically defined policies to support generalization of blue team policies. We considered random, patrol (randomly navigate within own territory), and A^* (search for enemy's flag using A^*) policies for the red team. Policies were trained in game maps randomly selected from a set of 2000 symmetrical maps (i.e., maps providing no advantage to either team). Table 4.1 summarizes the game settings and rewards used for training policies in the source and target environments.

4.3.2 HRL Sub-policies

We created sub-policies for our HRL methods by heuristically defining three intrinsic sub-rewards and independently training corresponding sub-policies to optimize each of those sub-rewards. We defined these sub-rewards such that some combination of them could be used to represent the overall task of winning the game. We focused on attack, scout, and defend sub-rewards for this effort. We trained each sub-policy in a partial game setting with randomized maps (including asymmetric ones), using only patrol and A^* red team policies to play against. Table 4.1 summarizes the partial game settings and sub-rewards used for training sub-policies.

4.3.3 Hyperparameter Tuning

To achieve optimal performance with the networks, we began with several experiments centered on tuning PPO and confidence HC hyperparameters.

	# Blue	# Red	Red Policy	Rewards
Source and Target	4	4	All	Capture flag (+1) Flag captured (-1) Eliminate enemy (+0.25)
Attack	2	4	Patrol	Eliminate enemies (+1)
Scout	1	2	Patrol	Capture flag (+1)
Defend	1	4	A^*	Defend flag (+1)

Table 4.1: Different game settings and rewards used for policy creation. The "Source and Target" row defines game settings and rewards used for training overall policies used to evaluate adaptation. The "Attack", "Scout", and "Defend" rows define partial game settings and sub-rewards used for training sub-policies.

In PPO there are 2 hyperparameters that have a significant effect on training, batch size and learning rate. A full factorial of these parameters was performed with the values shown in Figure 4.3, and parameters were selected which had the fastest training time and highest win rate. For selecting hyperparameters for the HC, another smaller factorial, bounded by red in Figure 4.3, was used which focused on the best regions of the baseline testing. The c_{max} and c_δ were included in the detailed factorial for the hierarchical controller.

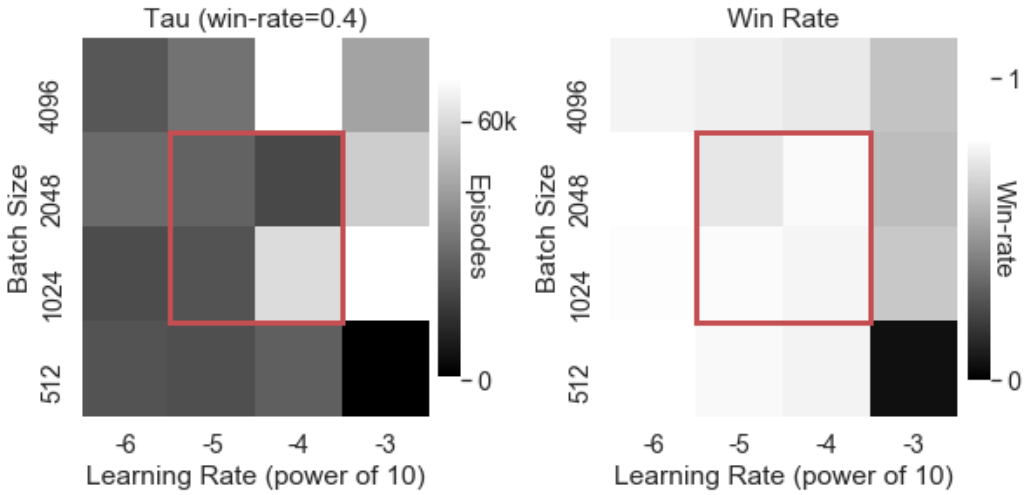


Figure 4.3: Training time and effectiveness of hyperparameters tested for PPO implementation (white signifies better performance). The parameters used for further tuning the hierarchical controller are highlighted in red.

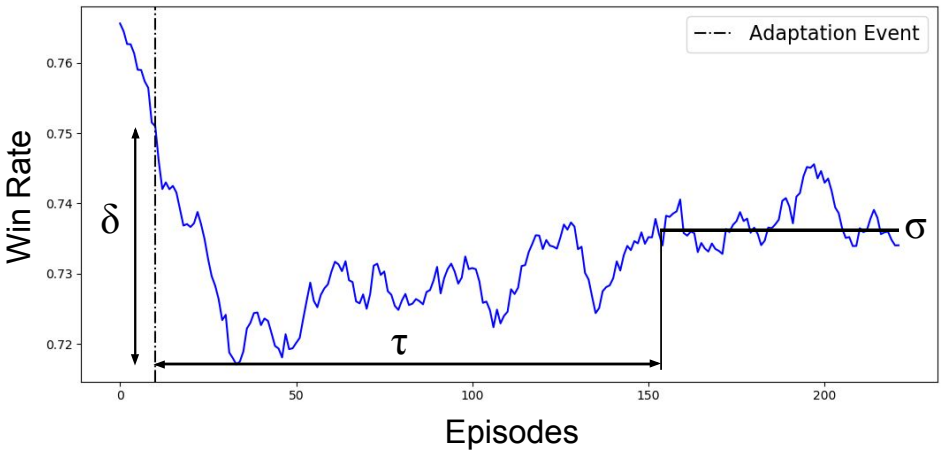


Figure 4.4: Metrics used to evaluate adaptation performance of a policy. δ is the immediate performance drop, τ is the recovery time, and σ is the steady-state performance.

4.3.4 Evaluating Adaptation

A common metric for evaluating adaptation performance is the steady-state performance after n samples from the target environment. This metric captures an important aspect of adaptation, but does not fully consider the transient dynamics of the adaptation process [29]. Instead, we propose a set of three metrics to more thoroughly evaluation the adaptation capabilities of a given control policy. We define δ as the immediate performance drop after a change to the target environment, which captures the initial robustness of the policy. We define σ as the steady-state performance after adapting to the target environment, which captures the long-term ability of the policy to adapt. Finally, we define τ as recovery time, calculated as the number of training episodes required to bring the win rate to within 2% of the σ, which captures the ability for the policy to quickly adapt. These metrics are visualized in Figure 4.4.

4.4 Results

Table 4.2 shows mean values for our adaptation metrics in our eight target environments. Overall, we see that our confidence HC method generally shows better immediate performance drop results than the fixed HC and

Target Environment -Enemy Capabilities-		Confidence HC (ours)			Fixed HC			Baseline PPO		
Strength	Speed	δ	τ	σ	δ	τ	σ	δ	τ	σ
Weak	Fast	**0.045**	**5.0**	0.752	0.116	11.0	0.664	0.134	58.0	**0.775**
-	Fast	**0.078**	46.0	0.681	0.135	42.0	0.685	0.157	**30.0**	**0.698**
Strong	Fast	**0.255**	**1.0**	**0.580**	0.283	N/A	0.453	0.324	46.0	0.568
Weak	-	0.061	N/A	0.838	-0.026	**17.0**	0.823	**-0.055**	27.0	**0.858**
Strong	-	-0.003	**29.0**	0.837	**-0.084**	N/A	0.809	0.117	102.0	**0.924**
Weak	Slow	0.121	**56.0**	**0.891**	**0.079**	N/A	0.617	0.161	83.0	0.777
-	Slow	**-0.047**	92.0	0.809	0.020	N/A	0.754	0.030	**90.0**	**0.879**
Strong	Slow	**-0.017**	N/A	0.780	0.078	N/A	0.704	0.182	**106.5**	**0.852**

Table 4.2: Comparison of our adaptation metrics for the eight target environments. A τ of "N/A" indicates that the method did not reach a steady-state performance above the initial performance drop. The best performing method is shown in bold.

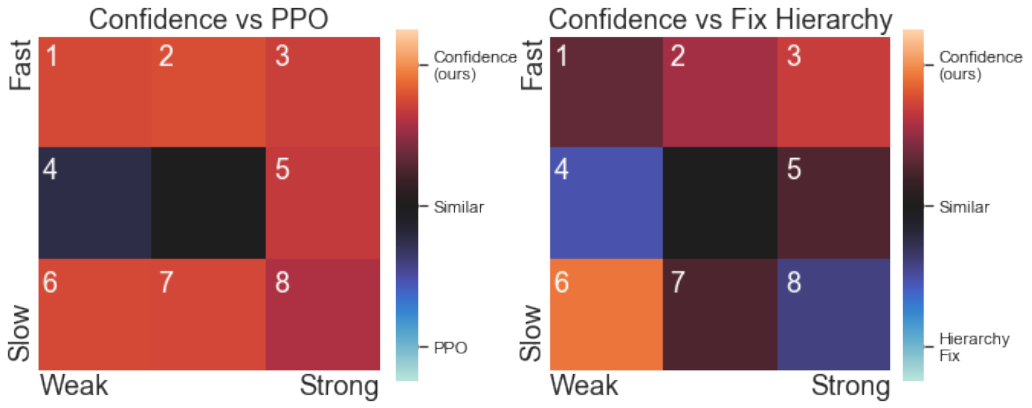


Figure 4.5: Heat map comparison of immediate performance drop between methods for the eight target environments. The x-axis defines the strength of enemy agents, while the y-axis defines their speed. Colors show how much better one method was than the other for the target environment, with respect to δ.

baseline PPO methods. Regarding recovery time, the confidence HC and baseline PPO methods show the best results for a similar number of target environments. However, despite this similarity, we also see that when the confidence HC shows a shorter recovery time than the baseline PPO, it tends to be significantly shorter. Alternatively, when the baseline PPO shows the shortest recovery time, the confidence HC result is only slightly worse. Finally, we see that the steady-state performance of the baseline PPO is generally higher than both HRL methods, though these differences are relatively small compared to differences seen in the other adaptation metrics. This result is likely due to the fact that we did not update our sub-policies during adaptation and indicates that our sub-policies were not optimally defined for these target environments. Enabling sub-policy adaptation may address these differences by enabling the HC to reach a globally optimal policy for these target environments; however, enabling sub-policy adaptation may also lead to sub-policy convergence and limit adaptation performance to future environment changes (i.e., it may lead to over-fitting for to the target environment).

Figure 4.5 provides a more direct comparison of the immediate performance drop results for our confidence HC method relative to the baseline PPO and fixed HC methods. Here, we clearly see that the confidence HC method provides a smaller or similar immediate performance drop compared to the

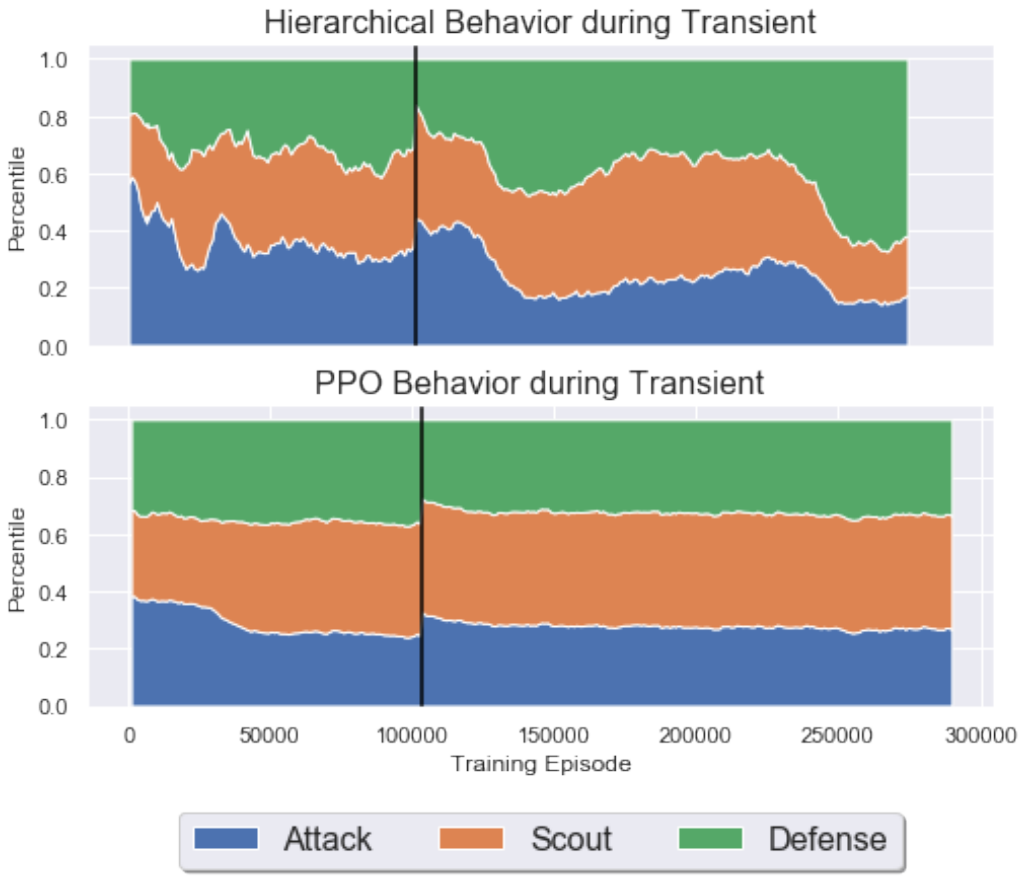


Figure 4.6: The transient win-rate of the confidence and fixed HCs replicates in the weak-slow and slow environments. Confidence HC shows more stable convergence than fixed HC.

baseline PPO in most target environments. Relative to the fixed HC method, the confidence HC only shows worse performance in immediate performance drop in scenarios where the environment got easier (i.e., the enemy got slower, weaker, or some combination of the two).

We also compare the convergence performance of our confidence HC method to the fixed HC method, as shown in Figure 4.6. As previously stated, the fixed HC method requires that the fixed step update parameter is temporally aligned with changes in the game, which is difficult to ensure. These results show that using our confidence-based training method notably reduces the likelihood of failed convergence when training HRL architectures, relative to fixed step training.

Finally, we explore the transient behaviors of our confidence HC method and the baseline PPO to understand how these networks before and during

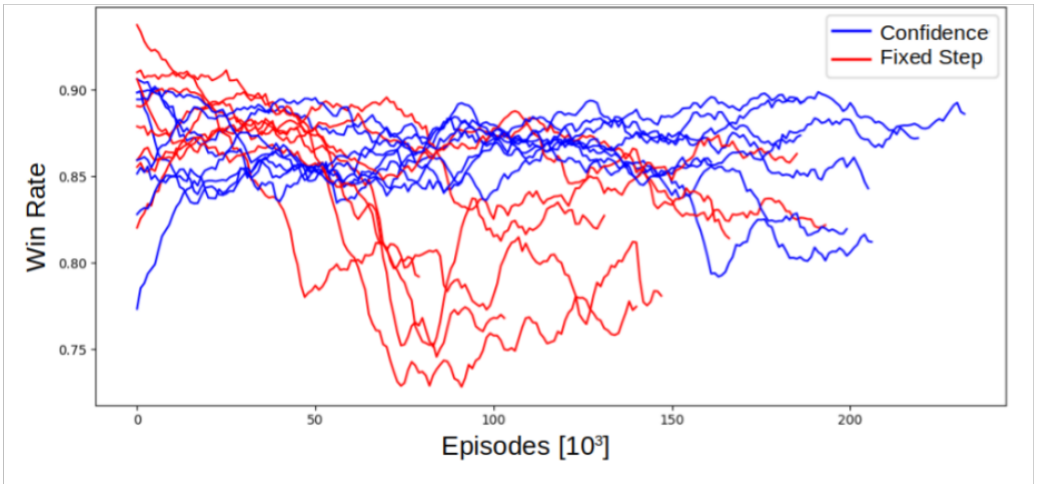


Figure 4.7: A comparison of the transient performance of the confidence HC and baseline PPO methods with an adaptation occurring at the black vertical line. The confidence HC behavior (top) shows sub-policy usage. The baseline PPO behavior (bottom) is estimated by calculating the sub-rewards that would have been received from the environment, and determining the fraction corresponding to each sub-policy.

adaptation. Figure 4.7 shows how often each sub-policy was selected by a confidence HC policy before and during adaptation, and compares this with the distribution of sub-rewards received from actions taken by the baseline PPO. While this is not an direct comparison, we see that the confidence HC method creates significant shifts in the selected sub-policy distribution during adaptation, whereas the baseline PPO maintains near constant behaviors. These differences in transient behaviors may explain how our confidence HC method was often able to outperform the baseline PPO in adaptation, as they suggest the confidence HC policies were more flexible in their ability to change sub-policy usage to better suit target environments.

CHAPTER 5

CONCLUSIONS

In this thesis we have introduced two key contributions which address the applicability of learned decision-making models towards real-world robotics implementations. The first is the construction of AI development pipeline which allows for progressive testing and integration of the RL decision-making and machine vision algorithms into real robotics. This pipeline accomplishes this goal by starting with a computationally simple environment which can be used for rapid development of new algorithms. These algorithms can then be transitioned to increasingly realistic scenarios to fine-tune and adapt to minor differences. The second is the introduction of the RL techniques which improve the adaptability of RL to novel situations, which can be leveraged in a deployment to real-world scenarios. This was done, by implementing two native structures that support adaptation, in the form of hierarchical adaptation and a confidence-based training metric. These leverage the developed testing architecture to display improvements in the adaptability of RL algorithms. Combined these two contributions take a step towards the deployment of RL based decision-making systems to real-world robotics.

There are two primary ways this work towards adaptation can be improved upon, improving the composition of sub-policies and replicating biological mechanisms that humans use for adaptation The easiest way is to address the composition of policies that would lead to the best adaptation. This can be addressed by decomposing the environment into implicit representations and features and then defining sub-policies around those features. This would allow policies to be learned without explicitly defining the rewards, which may limit the diversity and impact of the sub-policies. The second way that can be improved upon is a more biologically inspired HC, which learns quicker and creates high level associations with sub-policies. Mechanisms like this are what allow humans to generalize information between tasks and can serve as the basis of future adaptive learning schemes.

REFERENCES

[1] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, J. Quan, S. Gaffney, S. Petersen, K. Simonyan, T. Schaul, H. van Hasselt, D. Silver, T. Lillicrap, K. Calderone, P. Keet, A. Brunasso, D. Lawrence, A. Ekermo, J. Repp, and R. Tsing, "StarCraft II: A New Challenge for Reinforcement Learning," 2017. [Online]. Available: http://arxiv.org/abs/1708.04782

[2] OpenAI, :, C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. de Oliveira Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang, "Dota 2 with large scale deep reinforcement learning," 2019.

[3] A. A. Rusu, M. Vecerik, T. Rothörl, N. Heess, R. Pascanu, and R. Hadsell, "Sim-to-Real Robot Learning from Pixels with Progressive Nets," no. CoRL, pp. 1–9, 2016. [Online]. Available: http://arxiv.org/abs/1610.04286

[4] T. Osa, J. Peters, and G. Neumann, "Experiments with Hierarchical Reinforcement Learning of Multiple Grasping Policies," pp. 160–172, 2017.

[5] A. T. Miller and P. K. Allen, "Graspit! a versatile simulator for robotic grasping," *IEEE Robotics & Automation Magazine*, vol. 11, no. 4, pp. 110–122, 2004.

[6] J. A. Joergensen, L.-P. Ellekilde, and H. G. Petersen, "Robworksim-an open simulator for sensor based grasping," in *ISR 2010 (41st International Symposium on Robotics) and ROBOTIK 2010 (6th German Conference on Robotics)*. VDE, 2010, pp. 1–8.

[7] T. Haarnoja, S. Ha, A. Zhou, J. Tan, G. Tucker, and S. Levine, "Learning to walk via deep reinforcement learning," *arXiv preprint arXiv:1812.11103*, 2018.

[8] T. Hester, M. Vecerik, O. Pietquin, M. Lanctot, T. Schaul, B. Piot, D. Horgan, J. Quan, A. Sendonaris, I. Osband et al., "Deep q-learning from demonstrations," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[9] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems*, 2000, pp. 1057–1063.

[10] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," in *International Conference on Learning Representations (ICLR)*, 2016. [Online]. Available: http://arxiv.org/abs/1509.02971

[11] V. R. Konda and J. N. Tsitsiklis, "Actor-critic algorithms," in *Advances in neural information processing systems*, 2000, pp. 1008–1014.

[12] Z. Lu, H. Pu, F. Wang, Z. Hu, and L. Wang, "The expressive power of neural networks: A view from the width," in *Advances in neural information processing systems*, 2017, pp. 6231–6239.

[13] T. D. Kulkarni, K. R. Narasimhan, A. Saeedi, and J. B. Tenenbaum, "Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation," *Advances in Neural Information Processing Systems*, no. Nips, pp. 3682–3690, 2016.

[14] X. B. Peng, G. Berseth, K. Yin, and M. Van De Panne, "DeepLoco: Dynamic locomotion skills using hierarchical deep reinforcement learning," *ACM Transactions on Graphics*, vol. 36, no. 4, 2017.

[15] B. Marthi, S. Russell, D. Latham, and C. Guestrin, "Concurrent hierarchical reinforcement learning," in *IJCAI International Joint Conference on Artificial Intelligence*, 2005, pp. 779–785.

[16] M. Ghavamzadeh, S. Mahadevan, and R. Makar, "Hierarchical multi-agent reinforcement learning," *Autonomous Agents and Multi-Agent Systems*, vol. 13, no. 2, pp. 197–229, 2006.

[17] Y. Cai, S. X. Yang, and X. Xu, "A combined hierarchical reinforcement learning based approach for multi-robot cooperative target searching in complex unknown environments," *IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning, ADPRL*, pp. 52–59, 2013.

[18] R. S. Sutton, D. Precup, and S. Singh, "Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in RL," *Artificial Intelligence*, vol. 112, pp. 181–211, 1999.

[19] P. L. Bacon, J. Harb, and D. Precup, "The option-critic architecture," *31st AAAI Conference on Artificial Intelligence, AAAI 2017*, pp. 1726–1734, 2017.

[20] H. B. Ammar, E. Eaton, P. Ruvolo, and M. E. Taylor, "Unsupervised cross-domain transfer in policy gradient reinforcement learning via manifold alignment," *Proceedings of the National Conference on Artificial Intelligence*, vol. 4, pp. 2504–2510, 2015.

[21] G. Joshi and G. Chowdhary, "Cross-Domain Transfer in Reinforcement Learning Using Target Apprentice," *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 7525–7532, 2018.

[22] O. Vinyals, C. Blundell, T. Lillicrap, K. Kavukcuoglu, and D. Wierstra, "Matching networks for one shot learning," *Advances in Neural Information Processing Systems*, pp. 3637–3645, 2016.

[23] A. Santoro, S. Bartunov, M. Botvinick, D. Wierstra, and T. Lillicrap, "Meta-Learning with Memory-Augmented Neural Networks," *33rd International Conference on Machine Learning, ICML 2016*, vol. 4, pp. 2740–2751, 2016.

[24] D. J. Rezende, S. Mohamed, I. Danihelka, K. Gregor, and D. Wierstra, "One-Shot generalization in deep generative models," *33rd International Conference on Machine Learning, ICML 2016*, vol. 4, pp. 2274–2282, 2016.

[25] S. Ravi and H. Larochelle, "Optimization as a Model for Few-Shot Learning," *ICLR*, pp. 1–11, 2017.

[26] J. Mahler, M. Matl, V. Satish, M. Danielczuk, B. DeRose, S. McKinley, and K. Goldberg, "Learning ambidextrous robot grasping policies," *Science Robotics*, vol. 4, no. 26, 2019.

[27] L. Wang and J. T. Wu, "Characterizing the dynamics underlying global spread of epidemics," *Nature Communications*, vol. 9, p. 218, 2018. [Online]. Available: http://www.nature.com/articles/s41467-017-02344-z

[28] J. Schulman, S. Levine, P. Moritz, M. Jordan, and P. Abbeel, "Trust Region Policy Optimization," in *Proceedings of the 31st International Conference on Machine Learning*, Lille, France, 2015.

[29] D. D. Woods, "Four concepts for resilience and the implications for the future of resilience engineering," *Reliability Engineering and System Safety*, vol. 141, pp. 5–9, 2015.

APPENDIX A: CODE AVAILABILITY

Select code for the Thesis is publicly available.

Code for the 2D Abstracted Simulation is avaliable at `https://github.com/raide-project/ctf_public`

Code for the adaptation research is available at `https://github.com/vanstrn/RL`