Task Generalisation using Model-Free Reinforcement Learning on a Real-World Robotic System

Olivier Miller - M00676123

Supervisor: Dr Eris Chinellato

MSc Robotics Thesis – PDE4603

CONTENTS

1 INTRODUCTION

Reinforcement Learning (RL) is fast becoming a core aspect of modern robotics. As robotic systems are designed to achieve more and more complex tasks, the ability to directly code the complex functionality for these tasks and to create accurate models of the system's dynamics can become insurmountable. This is where reinforcement learning can play a key role. It allows programmers to define high-level tasks by defining only how the system should be rewarded. This is often simpler than having to program the low-level control systems of the robot [1]. With model-free learning (where no model relating states and rewards of the system to its actions is provided), this can even be achieved with no direct understanding of the specifics of how the system functions by using pure trial-and-error learning. This can enable the learning of complex behaviours that would be difficult to code manually, such as humanoid motor skills [2], dexterous manipulation [3], and complex 6-legged locomotion [4].

The use of RL on real robotic systems to learn non-trivial tasks is still in its infancy. Traditionally, RL has needed large numbers of trials/episodes to be run for any sufficient level of learning. Because of this, most RL has been entirely conducted in simulations where time could be sped up to allow for episodes to be run very rapidly (e.g. the Arcade-Learning Environment [5] and OpenAI Gym [6]). However, the use of simulations in reinforcement learning applied to robotics tends to give rise to a problem known as the *reality gap,* where the differences between simulations and the real robotic system can cause learning to be largely untransferable between the two without significant effort [7], [8]. This problem of applying reinforcement learning to real robotic systems can be addressed in many ways, including the use of model-based methods, methods that incorporate some level of prior knowledge about the task, multiple robots learning collectively, and methods for helping to bridge the reality gap and successfully allow sim-to-real training [9]–[12]. However, these methods all have serious drawbacks, and the task of finding general model-free reinforcement learning algorithms that can be successfully applied to real robotic systems is still in its infancy, though it has seen some initial success [13]–[15].

A secondary problem in modern reinforcement learning is the problem of overfitting and generalisation [16], [17], whereby the policy learnt through reinforcement learning is overly specific to the task it was trained on, and does not generalise well to similar tasks. This has started to gain traction as a problem to be investigated since the introduction of deep reinforcement learning algorithms that combine reinforcement learning with supervised learning in the form of neural networks [18]. So far, very little work has been conducted into overfitting and generalisation on real robotic systems.

This dissertation aims to begin investigations into the issues of overfitting and generalisation with regard to general model-free reinforcement learning algorithms applied to real robotic systems. To do this, the SenseAct framework and the UR5 Reacher 6D environment created by Mahmood *et al.* [13] is adapted for use with a UR10 collaborative robot manipulator. Three reinforcement learning policies are trained using different methods of resetting the arm's position between episodes. These policies are then tested for their effectiveness across the learning space on the task for which they were trained, and for their performance on similar but nominally different tasks. The results demonstrate that some level of generalisation is possible with a low level of training, while also demonstrating that the main issue with applying model-free reinforcement learning techniques to real robots – a lack of learning progression without the need for long training periods – is still prevalent when the learning task becomes more complex. The work presented here also points to many areas of further research that could prove fruitful in further refining the results obtained and

extending the knowledge base surrounding the generalisation of general model-free reinforcement learning techniques on real robotic systems.

The material will be presented in the following order:

Section 2: This section will give an overview of the academic literature related to the topic and identify the gap in the literature that this work seeks to fill.

Section 3: This section presents the methodology used for training the reinforcement learning policies.

Section 4: This section will present the tests conducted on the learnt policies and the results of both these and the training conducted.

Section 5: This section will provide a discussion of the results obtained and of future work that could be conducted to extend them.

Section 6: This section will summarise the findings and conclude the dissertation.

2 BACKGROUND

This section will provide an overview of academic literature to place the topic of this paper – generalising reinforcement learning policies on real robotic systems – in context. It will be demonstrated that there is a gap in the academic literature that the work conducted in this dissertation can begin to fill. It will cover the following topics in given order:

Section 2.1: A brief history of the evolution of reinforcement learning

Section 2.2: A high-level overview of the core concepts of reinforcement learning

Section 2.3: Previous work in using reinforcement learning on real robotic systems

Section 2.4: The problem of overfitting and generalisation in reinforcement learning

2.1 THE EVOLUTION OF REINFORCEMENT LEARNING

The psychological and biological concept of trial-and-error learning is the foundation on which reinforcement learning was built. One of the earliest clear definitions of trial-and-error learning is given by Edward Thorndike in his seminal work *Animal Intelligence*, published in 1911 [19], in which he describes two mechanisms for animal learning – the Law of Effect and the Law of Exercise. The Law of Effect states that, of the possible responses to a given situation, those that bring satisfaction to an animal will be more likely to occur in the future, and those that bring pain or discomfort will be less likely to occur. The Law of Exercise states that in any given situation, the animal is more likely to respond in a certain way if it has responded in that way in the past.

The name given to this type of learning – 'reinforcement' – is likely to have come from the English translation of Pavlov's lecture on Conditional Reflexes [20]. Pavlov uses the term to describe how certain behaviours in his experiments could be reinforced by associating them with a positive stimulus.

During the beginnings of modern thought on artificial intelligence (AI), Alan Turing suggested that trial-and-error learning could possibly be implemented on a computer using what he calls a Pleasure-Pain system. This is described in his paper titled *Intelligent Machinery*, published in 1948

[21]. In this system, a 'pleasure' or 'reward' input could be used to fix or anchor a certain behaviour of the system in some more permanent fashion. Conversely, a 'pain' or 'punishment' input could be used to disrupt the fixing of a behaviour. This paper also suggests the idea of taking random actions to begin and then using the reward or punishment received to fix these random actions to be used again in the future (if rewarded) or to discard them entirely (if punished). In this fashion, units of memory are viewed as switches which can take a state of uncertainty or be set in a specific on or off (1 or 0) position, where a reward or punishment will change the state of the switch from uncertain to 1 or 0 depending on what action was taken during its uncertainty. This way of viewing reinforcement learning also arose from systems that had implemented a basic form of reinforcement learning using actual physical switches to solve mazes. The first of these maze-solving machines that is documented is that of Thomas Ross, which he describes in the article *Machines That Think* published in the Scientific American, a popular science magazine, in April 1933 [22]. Maze-solving robots continued to evolve over the years, and a list of other examples can be found at [23], with examples where the actual maze kept track of the learnt states instead of the machine [24], and moving into the 1970s, the use of computers to learn to solve a maze (Johan de Boer's Maze Computer and Richard Browne's Maze Solving Computer, both listed at [23]).

The term 'reinforcement learning' has been applied since the 1960s (e.g. [25], [26]), starting with Minsky's paper *Steps towards artificial intelligence* published in 1961 [27]. This paper sets up one of the main problems in RL that is still studied today – that of how to assign credit to individual actions in problems where there is not a reward supplied at every step (e.g. a game of chess in which rewards are only given for winning, losing, or drawing).

For a deeper history of reinforcement learning, section 1.7 on The Early History of Reinforcement Learning in the seminal book *Reinforcement Learning: An Introduction* by Sutton and Barto [28] provides a more in-depth overview of this topic, including the sparse works conducted during the 1960s and 1970s, the evolution of the k-armed bandit problem and its relation to the foundations of reinforcement learning, the influence of economics on reinforcement learning, and the origins of Temporal Difference methods that were the basis for some of the first core reinforcement learning algorithms. Sutton and Barto themselves are pioneers of the field with their work in the 1980s [29]–[31] that helped to separate what was at the time a rather blurred distinction between supervised learning and reinforcement learning. They both continued publishing important works related to RL, including laying the foundations for Temporal Difference RL methods [32], the use of Neural Networks to generalise RL methods to continuous action and state spaces [33], Policy Gradient RL methods [34] (of which Trust Region Policy Optimisation [35], the algorithm used in this dissertation, is a member), and early applications of RL to real world problems [36]. They are often known as the 'fathers' of RL [37], [38].

2.2 What is Reinforcement Learning?

Sutton and Barto provide a succinct practical definition in *Reinforcement Learning: An Introduction* [28]:

> *Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them.*

Most other sources that define reinforcement learning do so by describing the mathematics of how it functions (e.g. [39], [40]). As Sutton and Barto state, the core function of RL is for an agent (in the case of this dissertation, a UR10 robotic arm, including its controller) to learn how to act to maximise

the *reward signal* that it receives. Initially, the agent has no concept of how to maximise these rewards, or even to receive them. It tries an *action* from its current *state*, drawn from some possible set of actions that are available in the current state, and receives a reward due to this action. The outcome of this action may have moved the agent to a different state or kept it in the same state. In the simplest examples, such as a game of noughts and crosses (tic-tac-toe in American English), every action (drawing a nought or a cross) moves the game into a new state and from then on it can never return to the previous state in that game – though it may encounter the same state in the next game. Depending on the reward structure, successive actions may lead to an immediate update of the chance that that action will be chosen from the previous state based on the reward received, or in the case of a game of chess or noughts and crosses, all probabilities of actions taken to reach the end of the game will only be updated at the end of the game, once a reward is received based on winning, losing, or drawing.

This also raises three other important concepts in RL: *value* (sometimes known as *return), policy* and *step-size parameter*. To give an intuitive understanding of value, let us use the pressing task of writing a dissertation. In the short term, the 'rewards' obtained while writing the dissertation might be negative, e.g. increased stress, decreased social life. However, in the longer term, these short-term negative rewards may lead to some higher total reward that is obtained, as completing the dissertation is likely to lead to completing a degree, which may lead to better job prospects and life satisfaction. This long-term view of rewards is quantified in reinforcement learning as the *value* of taking a particular action in a particular state. The value of a state-action pair is some combination of the immediate reward the agent can expect from the action it takes from that state, and all future rewards that are likely to occur based on taking that action from that state. This can be a simple sum of all possible future rewards multiplied with their probabilities of occurring, or it can have a discount factor applied, whereby future rewards are still included in the value calculation, but their effect on it is decreased based on how far they are in the future (e.g. a small child might want the more immediate satisfaction of knowing that they can have ice cream in the next two hours, even if they could have far more ice cream in two days). In many cases, it is not possible to definitively know what future rewards are likely to be obtained, and the value therefore has to be estimated in some way. Once the agent has a way of estimating the value of state, how much should this estimate change with new information?

The mechanism that controls action selection is known as the *policy*. The policy (along with the value function in some cases) is what is being learned in RL. While the value function gives an estimate of the values of state-action pairs, the policy uses this information to decide which action to take from the current state. This also touches on another of the core concepts of reinforcement learning: *exploration* vs *exploitation.* When the value function has found an action that is currently seen as the most profitable, should this action be *exploited* to try to gain the highest reward, or should the policy *explore* other actions to learn more and have a better estimate of their values? Again this is an ongoing area of investigation in RL, with different methods for adapting the ratio between the two being presented (e.g. [44]–[46]). In one of the simplest methods, known as the epsilon greedy method (ε-greedy), the greedy action (i.e. the action that is known to have the best current value) is exploited the majority of the time and a random action other than the greedy action is selected to be explored with some probability ε (this method is often credited to Watkins' PhD *Learning from Delayed Rewards* [47]). Often the rate of exploitation vs exploration is changed over time as the system learns in order to allow it to exploit more and explore less. This is similar to how the step-size parameter is commonly reduced to allow the policy function to converge.

The *step-size parameter* affects how much the policy and/or value estimations should be updated on each update step. This is a complex problem that many modern algorithms address in various ways. Having a learning rate that is too high can lead to instability of the system, as the value or probability of encountering a state-action pair might start to grow exponentially, whereas having a learning rate that is too low leads to learning that is much slower than it needs to be. What defines 'too high' or 'too low' can differ during the learning process, and as such many modern algorithms use adaptive methods for choosing the step-size parameter (e.g. [41]–[43]). In the simplest adaptive method, the step-size parameter is large to begin with to allow for large learning steps to start generating value estimates but is decreased over time to allow the predicted values and the policy to converge.

This section hopes to have given the reader a basic level of understanding of the main terminology used in reinforcement learning. Covering the specifics of the broad topic of reinforcement learning is well beyond the scope of this dissertation. For an overview of the various types of reinforcement learning and the related equations and proofs, the author recommends the book by Sutton and Barto mentioned earlier – *Reinforcement Learning: An Introduction* [28] – which contains a comprehensive introduction to the majority of the reinforcement learning concepts and algorithms used today.

2.3 Reinforcement Learning for Real Robotic Systems

While applications of reinforcement learning on real robotic systems have often met with success (e.g. [10], [12], [48], [49]), the problem of using model-free general reinforcement learning agents on real robotic systems has had significantly less success. This was partially due to a lack of benchmarking tasks with continuous or high-dimensional action and state spaces. While high-level robotic tasks can be easily described with discrete action and state spaces (e.g. a mobile robot that has the high-level behaviours 'patrol', 'stay still', and 'return to charging point'), all lower-level control of robotic systems is done in continuous action and state spaces (e.g. states based on continuous possible positions or velocities of joints and actions based on currents to motors). Significant progress was made on continuous control tasks with the introduction of a suite of benchmarking tests in simulation by Duan *et al.* in 2016 [50]. Some of the algorithms tested in that study have become popular due to their performance in the benchmarking tests, including Trust Region Policy Optimisation (TRPO) [35], Deep Deterministic Policy Gradients (DDPG) [51], and after the creation of the benchmarking tests, Proximal Policy Optimisation in 2017 (PPO, created by the authors of TRPO) [52]. These algorithms, along with Soft-Q Learning [53], were tested on a variety of benchmarking environments set up on real robots by Mahmood *et al.* [13], [14]. Accessible benchmarking tests such as the Arcade Learning Environment (ALE) [5] and the deep reinforcement learning benchmarking tests [50] have led to significant advancements in reinforcement learning. The benchmarking tests created by Mahmood *et al.* represent a significant advancement in the use of general reinforcement learning algorithms on real robots by making the otherwise complex task of setting up reinforcement learning tasks on real robots far more accessible. One of these benchmarking environments, *UR5 Reacher 6D*, forms the basis for the work conducted in this dissertation.

Previous work on real robotic systems has largely involved model-based rather than model-free learning [11], sim-to-real learning [10], [48], or collective learning [12], [54], [55]. Model-based learning uses (or learns) a model of the environment that can then be used for planning, such as by predicting the next state and reward caused by taking an action [28]. In contrast to this, model-free learning uses pure trial-and-error learning. This is what is meant by the term *general reinforcement learning*, as this type of learning is not tied to a specific agent as with model-based learning, and is

therefore system-independent (e.g. the term *general reinforcement learning* is used in [56], [57] to describe such reinforcement learning algorithms).

Sim-to-real learning involves some level of initial training in simulation before the policy is transferred to the real robotic system. While there have been significant improvements in this method and it has seen some recent successes – such as Rusu's *et al*. work on using a 9 Degree of Freedom (DOF) Jaco arm to reach to a visual target [10] and Tan's *et al*. work on learning gaits for quadruped robots [48] – it still has some serious shortcomings. As Tan *et al.* note, the simulation used to train the robot had to be improved significantly from its initial implementation, including adding an accurate actuator model and simulating latency found in the real system. While this method may be feasible for robotic systems that already have sophisticated simulators, for those without, general reinforcement learning algorithms applied directly to the real robotic system are likely to be more feasible, decreasing the possibility of having to create a new simulation environment from scratch or significantly updating an existing one. Creating a simulation of a robotic system inherently requires that system to be accurately modelled, and as such this method of learning requires similar information to that of model-based reinforcement learning algorithms, even if model-free algorithms are able to be applied once the simulation is created.

The final method that has seen significant success in the use of reinforcement learning on real robotic systems is collective learning. Collective learning describes a method of reinforcement learning where the learning task is conducted by multiple real robots at the same time. The episodes run by each individual robot are then used to update the policy and value function using a reinforcement algorithm that collates these episodes centrally. While this is a good solution to the problem of requiring large numbers of episodes for training policies, having the ability to access large numbers of the same robotic system is quite rare. Collective learning also requires that the task is easily distributable among many robots. This is the case for many tasks involving robotic manipulators; however, it is unlikely to be so simple for tasks that take place over a large environmental area, such as tasks with mobile robots. Some examples of successful use of collective learning include the work of Yahya *et al.* [12] and Gu *et al.* [54] on robotic manipulators, and the work of Yasuda and Ohkura on robotic swarms [55].

There have also been some successes in applying algorithms that are closer to general reinforcement learning to real robotic systems, such as the work of Levine *et al.* on learning hand-eye coordination style behaviours with a robotic manipulator [15], and the work of Riedmiller *et al.* in using reinforcement learning with sparse rewards for box-stacking and table-cleaning robotic manipulator behaviours [58]. In both of these examples, however, prior information about the task or the system is still required. For Levine *et al.* this is in the form of providing the object to be interacted with at known locations during training, though they do show that this is no longer necessary during testing. Riedmiller *et al.* add an auxiliary reward function to their otherwise sparse reward structure for the tasks used in their experiments.

2.4 Generalising Reinforcement Learning Policies

The term *generalisation* as it is used in this dissertation refers to how well a policy trained for a specific task is able to perform on similar related tasks. This is distinct from how the term is sometimes used in the literature. The term *generalisation* as applied to RL has been used in the past to discuss extending reinforcement learning into new domains such as fuzzy environments [59], generalising actions and states to similar ones in order to decrease the dimensionality of the system (allowing for the use of continuous state and action spaces) [33], and learning to approximate value functions [60]. The term *general* is also sometimes used in the context of *general reinforcement*

learning, which defines reinforcement learning algorithms that can be applied to a wide range of situations, and are therefore necessarily model-free and system-independent [56], [57]. This definition is closely related to the definition of generalisation used in this dissertation.

Overfitting is a term often used in machine learning to describe the problem of learning a solution to a test set of data that works very well on that specific set of data, but does not generalise well when used to make predictions for data that is not part of the training set [61]. Overfitting can also be a problem in reinforcement learning, and specifically with deep reinforcement learning where neural networks are being taught during the learning. Zhang *et al.* conducted a study on overfitting in deep reinforcement learning and the dangers it may present [17], and concluded that even with the use of a stochastic policy and a stochastic environment, overfitting could be an issue. In spite of this, they found that stochasticity can still be effective for reducing overfitting depending on the algorithm used. In many environments where reinforcement learning is used (e.g. classic benchmarks such as learning Atari games and uses in the financial sector), the systems are largely deterministic. Real robotic systems are naturally stochastic due to noise in actuators and sensors, as well as fluctuations in the robot's environment such as temperature and humidity changes. Zhang *et al* also found that using only a small training set (ten randomly-generated maze configurations) led to far worse generalisation than larger training sets, as the tendency to overfit to a small number of data points is much stronger. As such, another method that can be useful in addressing overfitting is the generation of a larger random training set. Cobbe *et al.* conducted research into quantifying generalisation in deep reinforcement learning [62]. They note that 'among the most common benchmarks in RL, it is customary to use the same environments for both training and testing'. This can be a problem, as training and testing on the same task is unlikely to identify any overfitting. It is worth noting here that sometimes overfitting is ideal, such as when the task for training and testing is identical (e.g. for learning the best possible way of completing a single level in a video game), and the learnt policy will not be used for any other tasks. This is less than ideal for real robotic systems, however, as training times for individual tasks are often high. The policies trained for use in this dissertation were trained for 2000 episodes, corresponding to around 3.5 hours of training time, and this is a very small training set compared to those often used for learning robust behaviours. For example, AlphaGo Zero, the deep reinforcement learning method which succeeded the method that beat Go master Lee Sedol in four out of five games, was trained with 4.9 million games of self-play [63]. For this reason, learning more generic policies that can be used for similar (but not identical) tasks can save a lot of potential training time when training is unable to be sped up as it can be in simulation.

Other studies on overfitting and generalisation of reinforcement learning algorithms include Zhao's *et al*. work on creating a benchmark for testing the overfitting of deep RL algorithms [16], and the work of Bruce *et al* on using a real mobile robot to build a one-shot model of the environment it is in, and then to train offline using this model (i.e. without actually running the training episodes using the real robot) [64]. Bruce's *et al.* work is one of very few studies on real robotic systems that considers the problem of overfitting. This work does also aim to use a general reinforcement learning algorithm; however, as there is some modelling of the environment and some pre-training of visual features, this does not describe general reinforcement learning in its purest sense.

As the works outlined throughout this background section have evidenced, there is a specific gap in the literature in relation to the investigation of overfitting and generalisation of model-free general reinforcement learning policies on real robotic systems. This is the gap that this dissertation aims to begin to fill.

3 METHODOLOGY

This section will detail the methodology used to obtain the results presented in this dissertation, and is broken into the following subsections:

Section 3.1: A brief overview of the OpenAI Gym framework that forms the basis of the SenseAct framework used for running all training and tests is given.

Section 3.2: This section outlines the specifications of the UR10 collaborative robotic manipulator used in this dissertation.

Section 3.3: A detailed description of the changes made to the OpenAI Baselines implementation of Trust Region Policy Optimisation and an overview of the structure and implementation of the SenseAct framework are provided. All additions made to the SenseAct framework for use with the UR10 are also detailed in this section, as well as the method of installation for SenseAct and the OpenAI Baselines.

Section 3.4: This section gives a high-level overview of Trust Region Policy Optimisation, the algorithm used for all reinforcement learning conducted in this dissertation.

Section 3.5: This section provides the details of the task setups used to train the three policies.

The first decision to be made regarding the training and tests conducted in this dissertation was how to run reinforcement learning algorithms on a robotic manipulator. As the SenseAct framework created and used by Mahmood *et al.* [13] had already implemented an environment on a UR5 robotic arm using all 6 DOF, it was decided to use this framework. This environment is implemented as an OpenAI Gym environment [6]. OpenAI Gym provides a standardised environment format for reinforcement learning tasks in order to help the reproducibility and comparison of reinforcement learning task results. The use of a UR10 collaborative robot arm was available to the author, so it was decided to extend the framework for use with this arm.

All files in this section are either referenced in a similar way to how they would be imported from their base python package, or are files that are available at https://github.com/OlivierRobotics/MSc-Final-Project/tree/master/src.

3.1 REINFORCEMENT LEARNING AND OPENAI GYM

As discussed in section 2.2, reinforcement learning is a machine learning process whereby defined actions that a system can take are linked to observations of that system via a policy that defines how best to act in a certain state in order to maximise a balance of current and future rewards. The reward for a system is simply a number that is defined in the reinforcement learning task based on the desired outcomes of the learning.

OpenAI Gym provides a standardised Python interface for creating reinforcement learning environments that can then be used to benchmark and test different machine learning agents in an easy plug-and-play manner [6]. OpenAI Gym achieves this with four simple methods that should be defined in any derived class of the base Env class used to create reinforcement learning environments. This class is defined in gym.core.py. These methods are:

1. **step()**
   This method accepts an action for the system to take and progresses the system by one timestep. It returns the agent's observation of the environment at the end of the timestep.

The reward earned for that timestep is returned, along with a Boolean stating whether the training episode is done, and a dictionary containing any auxiliary information that may be useful.

2. **reset()**
   This method resets the environment to some initial state and returns an initial observation. It is usually called once the Boolean *done* value from step() is returned as True.
3. **render()**
   This method renders the environment in some meaningful way depending on the type of environment. This usually involves a pop-up window that displays a simulation of the agent. As the reinforcement learning tasks described here were run on a real robot, there is no need for this method; thus, it is not implemented in the environment used for the tasks in this dissertation.
4. **close()**
   This method acts as a destructor for the environment.

Any subclass inheriting from the Env class must also define action_space and observation_space attributes, and optionally a reward_range attribute defining the minimum and maximum rewards that may be obtained. The specific implementation of these methods and attributes in the environment used for the experiments that form the basis of this dissertation is described in detail in section 3.3.2. The OpenAI Baselines implementations of common Reinforcement Learning algorithms (discussed in section 3.3) expect to be able to call a step() function in the environment that is passed to them, that takes inputs and outputs as described above. Though the Baselines do not strictly require an OpenAI Gym style environment (and the environment used for all experiments in this dissertation partially overrides much of the base functionality of the OpenAI Gym environment), it is still useful to have this framework to guide the environment functionality and to help maintain a level of standardisation across different machine learning environments.

3.2 The UR10 Collaborative Robot Manipulator

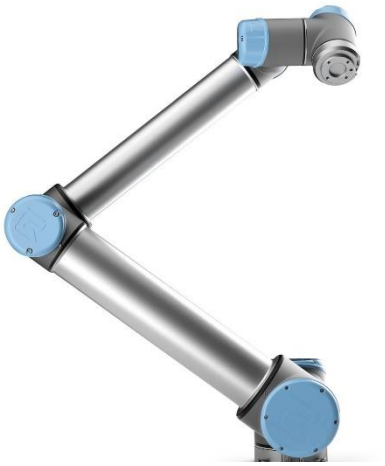

Figure 1: The UR10 robot arm

The real-world robot used for all training and testing in this dissertation is the UR10. The UR10 is a 6 DOF robotic arm sold by Universal Robots and comprised of six joints: base, shoulder, elbow, wrist 1, wrist 2, and wrist 3. It is a collaborative robot, in that it has joint force limits that are comparatively lower than those of a traditional industrial manipulator. As such, it is safe for use without a cage. The arm is also able to be freedriven (i.e. it can be put into a mode where a human operator can physically move the joints of the arm to a position they desire), making it easily programmable. The UR10 differs from other models in the series (the UR5 and the UR3) by being larger with a longer reach, and by being capable of carrying a heavier payload (10kg within a confined area – this is where it gets its name). Fully extended, the arm can reach 1.3m [65]. Figure 1 shows a public media image of the UR10 retrieved from [66].

3.3 SenseAct Framework and OpenAI Baselines

The SenseAct framework is a set of OpenAI Gym-style environments and communication protocols for benchmarking machine learning algorithms on real robots coded in Python3. The SenseAct

framework implements the TCP/IP Real-Time Data Exchange (RTDE) communication protocol used by Universal Robots to send text commands directly to the UR robot via an ethernet connection, and environments for running benchmarking tasks on a UR5. Packets are received from the UR controller detailing information such as the current joint positions and velocities, and these are used to provide the observations needed for reinforcement learning. URScript commands (the programming language used by UR manipulators) are sent directly to the RTDE socket to control the robot.

Based on the work of Mahmood *et al.* [14] it was decided to use Trust Region Policy Optimisation (TRPO) [35] as the reinforcement learning algorithm for training policies. Mahmood *et al.* found that TRPO was the most reliable algorithm they tested based on its insensitivity to hyperparameters and its consistency in improving rewards over time. The OpenAI Baselines implementation of TRPO [67] was used for learning all policies presented in this paper. The Baselines use TensorFlow [68] with NVIDIA CUDA (Compute Unified Device Architecture) [69] for their implementations of common reinforcement learning algorithms and provide easy plug-and-play functionality with OpenAI Gym environments.

3.3.1 Software Installation and Hardware Requirements

The SenseAct and OpenAI Baselines packages were installed on a computer running Ubuntu v16.04.5 LTS using the installation instructions on their respective GitHub pages. After initial testing throwing errors related to TensorFlow, it was clear that this also needed to be installed, so this was done using 'pip install tensorflow-gpu'. This installed version 1.14.0. The GPU-based version of TensorFlow requires CUDA for running tensor calculations on a NVIDIA GPU. CUDA was installed and tested via the instructions for Ubuntu installation available at [70]. Initially, CUDA v10.1 was installed; however, the pip installation of TensorFlow is only compatible with v10.0. As such, v10.1 had to be uninstalled and v10.0 was installed instead. After further testing, it was discovered that the NVIDIA CUDA Deep Neural Network library (cuDNN) [71] also needed to be installed for running TRPO. cuDNN v7.6 was installed from [72].

All training and testing was run on a desktop PC with an Intel Core i7-7700k CPU @ 4.20GHz x 8 and two CUDA-compatible GeForce GTX 1080 TI graphics cards. All code and data was stored on an SSD hard drive.

3.3.2 Reacher 6D Environment

The UR5 Reacher 6D Environment is set up to be trained in the SenseAct framework in examples.advanced.ur5_reacher_6D.py. This file formed the basis of the training environment setup for the UR10, ur10_reacher_6D.py, with changes made as described in section 3.3.3.

3.3.2.1 RTRLBaseEnv

The SenseAct framework adds a base environment class that acts in a similar manner to the OpenAI Gym environment with the addition of necessary functionality for real-time system control. This is implemented in the RTRLBaseEnv class in senseact.rtrl_base_env.py with a real-time loop that maintains a constant timestep and provides buffers for receiving observations and sending actions to the system so that messages do not get lost. RTRLBaseEnv also implements the step() and reset() functions needed for OpenAI Gym compatibility. Each of these functions calls other functions that should be implemented in any derived class. These functions are:

1. **_reset_()**
   This is called by the reset() function in RTRLBaseEnv that checks if the environment is running as a single process or in multiprocess or multithreaded mode.

2. **_check_done()**
   This method is used to check whether an episode is finished.
3. **_compute_sensation_()**
   This function implements how packets received from the system are converted into observations for use with OpenAI Baselines reinforcement learning algorithms
4. **_compute_actuation_()**
   This function implements sending packets to the system based on the action specified to be taken.

3.3.2.2 ReacherEnv

The ReacherEnv class in senseact.envs.ur.reacher_env.py provides the UR-specific reacher environment for running reinforcement learning tasks. This defines the action and observation spaces, the reward function, and the task, and implements the methods required by the RTRLBaseEnv. This was initially implemented to be similar to the OpenAI Gym Reacher task [6] using only 2 DOF of the UR5 , but was extended to have the option of using all 6 DOF [13].

Action Space

ReacherEnv implements three methods for controlling the arm – velocity, position, and acceleration control for each joint. These can be implemented as direct, first derivative, or second derivative control. These options can be chosen via the input arguments control_type and derivative_type. Position control is implemented using Servoj URScript commands. Velocity and acceleration control use SpeedJ URScript commands to control the arm. If derivative control is used, maximum values for the first and second derivatives can be specified as input arguments. The maximum and minimum values for the action space are then defined based on the maximum and minimum joint limits (for position control), the maximum speed (for velocity control), or the maximum derivative change (for first or second order derivative control). These maximum and minimum values are then used to create an OpenAI Gym Box space that defines the continuous action space as best it can using 32-bit floats. The tests run in this dissertation all use direct velocity control.

Observation Space

Observations are implemented as a concatenation of the joint positions and velocities with the cartesian distance between the end effector and the target point, and the previous action. The observation space is defined as an OpenAI Gym Box based on the maximum and minimum values for each of these. As with the action space, this is a continuous observation space approximated by 32-bit floats. The observation can include multiple joint position and velocity readings as defined by the obs_history input argument. For the tests in this dissertation, this used the default value of 1 (i.e. only the most recent joint position and velocity readings are used in each observation).

Reward Calculations

The ReacherEnv implements two methods for calculating rewards. These are 'linear' and 'precision'. The target type for the environment can also be defined to use a cartesian coordinate target ('position'), or a joint position target ('angle'). For the position target type, the linear method calculates rewards to be the negative of the Euclidean distance between the end effector position and the target position by calculating the second-order norm of the vector that specifies the difference between the two positions. This reward type will always calculate negative rewards, except when the distance between the two points is zero. The second reward type, 'precision', is calculated as $-\|td\|_2 + e^{-\|td\|_2{}^2/0.01}$, where *td* is the vector that defines the distance between the end effector and the target point. In addition, a reward can be added via the vel_penalty input argument that is the negative of the sum of the velocities of each joint in order to incentivise the system to keep lower velocities on each of the joints. These rewards are identical for the 'angle' target type, except that the 1st order norm of the vector specifying the difference in joint angles between the target and the current positions is used. The reward value is then scaled by multiplying it by the timestep divided by 0.008. The 'precision' reward type with a 'position' target and no velocity reward was used for all learning and testing in this paper. Figure 2 displays the rewards given using the 'precision' method, after scaling, up to the maximum distance between the target and the end effector. As the maximum reward obtained is 5 when the distance between the two points is 0, the maximum reward that can be obtained for any single episode is 500 (A reward of 5 on each of the 100 timesteps for the episode). This is impossible to obtain in practice, however, due to a degree of error in the measurements of the end effector position at any given time, and the fact that this would require the end effector to start the episode directly on the target point.

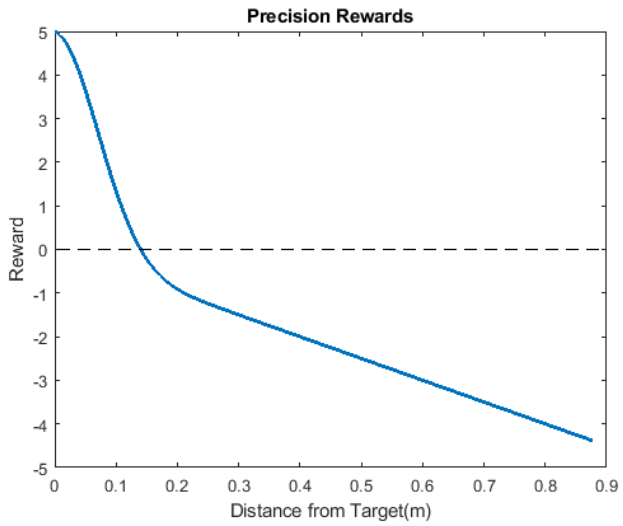

Figure 2: Rewards given using the 'precision' reward calculation

Task

The task implemented in the ReacherEnv is to move the end effector point, using solely individual joint control, to a specified target within a bounded space. As discussed in the section above, this target can be defined as cartesian coordinates, or as a set of joint positions. The environment allows for a 2 DOF or 6 DOF setup. The 2 DOF setup uses only the 2nd and 3rd (shoulder and elbow) joints of the UR arm. The 6 DOF setup uses all six joints and is used for all training and tests in this dissertation. The bounded space is defined both in terms of cartesian coordinates for the maximum and minimum end effector positions, and as joint angle limits for each of the joints. There are also boundaries put on the velocity and acceleration of the joints. These are enforced using the _handle_bounds_servoj() and _handle_bounds_speedj() functions. The joint and end effector position boundaries are handled in _handle_bounds_servoj(). Joint positions are handled such that if a joint moves outside of its boundaries, it is given a command to move back within its bounds. This is simple for the joints as each one has defined maximum and minimum positions. For the end effector position, this is more difficult. An inverse kinematics solver using the Denavit-Hartenberg (DH)

parameters of the arm is used to find a nearby point that is within the cartesian bounds to move the end effector back to using a Speedj command. In both of these cases, the arm is decelerated quickly so that it cannot stray very far outside of its bounds.

Three reset modes for resetting the arm between episodes are implemented as part of the environment. These are defined by the strings 'zero', 'random' and 'none' used as input to the environment's reset_type initialisation argument.

Reset Mode	Description
'zero'	The arm resets to a fixed position using joint position values specified in the setup dictionary (defined in senseact.devices.ur.ur_setups.py in the SenseAct framework and in ur10_setup.py for the UR10 used in this dissertation, as specified in section 3.3.3).
'random'	The arm resets to a random joint configuration within the defined joint limits and then checks whether this configuration corresponds to a valid end effector position within the bounded box using forward kinematics. If the joint configuration does not lead to a valid end effector position, the process is repeated.
'none'	This reset method is the simplest of the three. Using this method, the arm is not reset at all between episodes and instead just begins the next episode from the position that it finished the current episode.

The task is conducted using discrete timesteps, defined by the input argument to the environment named dt. The maximum length of an episode can be specified in one of two ways: by defining how long episodes should run, specified as a time (in seconds) or by the number of timesteps each episode should run. The following input arguments are used to specify episode duration:

Input argument	Definition of input value
episode_length_time	Set episode duration in time measured in seconds
episode_length_step	Set episode duration to a specific number of timesteps

The author of this dissertation modified the environment to allow for neither of these arguments to be specified, in which case the environment runs in a continuous behaviour mode with no defined episodes. The specific URScript commands used to control the UR arm are also defined as input arguments. This allows the possibility of changing how these commands are handled if a different robotic manipulator is used and increases the flexibility of the environment. An artificial observation delay can also be added. This could be useful if running the environment in simulation, as it might allow the environment to better mimic that of the real robotic system. Finally, the environment also allows for maximum speed and acceleration values to be input directly via accel_max and speed_max. If these arguments are used, they override the velocity and acceleration limits defined in the setup dictionary.

3.3.3 UR10-Specific Setup

It was necessary to adapt the SenseAct UR5 Reacher 6D Environment slightly for use with the UR10. The UR5 environment uses the DH parameters of the arm for calculating forward and inverse kinematics. These parameters, therefore, needed to be updated for the UR10. These were updated in ur10_setups.py using values obtained from [73].

Forward and Inverse kinematic functions that use these DH parameters are defined in senseact.devices.ur.ur_utils.py. The forward kinematics function is used in

senseact.envs.ur.reacher_env.py to calculate the cartesian coordinates of the end-effector based on the joint position observations received from the RTDE communication interface of the UR10 controller. These cartesian coordinates are then used to check if the end effector is within the defined bounding box at each timestep (discussed further in 3.3.3.1). The inverse kinematics solver function is also used in reacher_env.py to calculate joint configurations for positions near to the current end effector position in order to select a position to which the end effector can safely be moved back within the bounding box if it strays outside of it.

The Reacher 6D Environment defines three reset methods, as detailed in section 3.3.2.2 under the task specification: 'zero', 'random', and 'none'. The 'zero' reset method moves the arm back to a fixed reset position. For this, a position to reset the arm to within the bounding box was found by manually jogging the robotic arm into a position close to the centre of the bounding box. The joint position values from this are then used to define the reset position in ur10_setups.py.

3.3.3.1 Safety

The SenseAct framework implements three main safety features that are defined in the setup parameters for the learning environment defined in ur10_setup.py. These are:

1. *End effector maximum and minimum positions*
   These define the bounded box that the end effector point is unable to move beyond.
2. *Velocity and acceleration limits*
   These define maximum joint speeds and accelerations.
3. *Joint angle limits*
   These define angle limits for each joint.

The end effector maximum and minimum positions were selected in order to create a bounded box that did not greatly extend beyond the edges of the table that the UR10 is mounted on, so as to minimise the risk of any collisions between the UR10 and individuals working in its the vicinity. These coordinates were determined via a combination of freedriving and jogging the UR10 via the Teach Pendant.

While there are already maximum velocity and acceleration limits defined for each joint as part of the safety parameters implemented on the UR10's controller, additional limits are implemented in the UR10 Reacher 6D learning environment. This increases the safety of running tests and decreases the action space used for learning tasks, allowing the robot to learn at a faster rate. These values were kept as the defaults initially implemented for the UR5 Reacher 6D environment, as these were deemed to be acceptable based on initial observations of the learning process.

Joint angle limits were chosen via testing positions of the joints using the Teach Pendant. Reasonable limits were then selected to allow the arm to move freely in the bounded box, while limiting the chances of self-collision and the range of motion of the larger links of the arm. The bounded box created by defining the end effector maximum and minimum positions only applies to the end effector point of the robot. As such, it is still possible for the larger links of the robot to operate well outside the bounded box. Limiting these joints decreases this likelihood and also serves the purpose of decreasing the action and observation spaces for reinforcement learning tasks.

To add to the safety functionality already implemented in SenseAct, three safety boundary planes were added via the UR10's Teach Pendant. These were added to ensure that if the safety functionality implemented in SenseAct were to fail for any reason, the robot would still be unable to hit the table it is mounted on or any other objects in the vicinity of the robot's mounted position.

The general UR10 safety features were chosen during the purchase of the arm based on the environment it would be installed in – an open classroom environment with the possibility of individuals being in close proximity to the arm. These were largely implemented as speed limits on each joint, with the base and shoulder joints limited to 131°/s and the elbow and three wrist joints limited to 191°/s.

3.3.3.2 Firmware Version

The SenseAct Reacher 6D Environment was initially created for a UR5 robot running UR firmware version 3.2-3.4. The UR10 used for these experiments runs firmware version 3.8. In newer versions of the firmware, the RTDE Interface of the controller was updated. Each packet was extended from 1060 bytes to 1108 bytes via the inclusion of two additional variables – Elbow Position and Elbow Velocity. The Realtime Communications Interface is detailed in the file Client_Interface_V3.11andV5.5.xlsx, downloaded from [74].

Due to this change in communication packet size, when initially testing the UR Reacher 6D environment, errors were constantly being printed to the terminal when trying to run the environment due to a mismatch between the expected packet length and the actual packet length. To solve this, the packet length (REALTIME_COMM_PACKET_SIZE) and description (REALTIME_COMM_PACKET) were updated in senseact.devices.ur.ur_utils.py.

3.3.4 Learning a Policy

The OpenAI Baselines implementation of TRPO is run using the learn() function implemented in baselines.trpo_mpi.py. This function sets up all the necessary variables and classes and trains the policy and value function for a specified maximum number of timesteps, maximum number of iterations, or maximum number of episodes. TRPO cannot learn on every episode, as it requires a sample of state-action pair trajectories in order to decide how best to update the policy. A high-level overview of how TRPO works is provided in section 3.4. As such, each policy-update iteration requires multiple episodes. The number of timesteps per batch (i.e. the number of timesteps per batch of episodes used for an iteration) is defined as an argument to the learn() function. Each episode length is defined in the learning environment which is also passed to learn() as an argument. For the purposes of this dissertation, this is the UR10 Reacher 6D environment described in detail in section 3.3.2, in which the episode length can be directly or indirectly defined in the input arguments. The learn() function also takes a network type as an argument that then acts as the parametrized policy function to be trained for the reinforcement learning task. In this paper, a simple multi-layer perceptron (MLP) network is used with two layers and 64 nodes in each layer, where each layer is fully connected to the next. Other networks are also implemented in the OpenAI Baselines, such as Long Short-Term Memory (LSTM) [75] and different types of Convolutional Neural Networks (CNN) [76], [77]; however, these are outside the scope of this dissertation. The learn() function can also be given a path from which to load a previously trained policy, and a reference to a callback function that is called at the beginning of each training iteration and that has access to all local and global objects in the scope of the learn() function. The other arguments given to the learn() function define the various TRPO hyperparameters, and these are discussed in more detail in section 3.4.1.

Some small updates were made to the learn() function based on warning messages received while running the training environment. These were largely changes to the paths of various functions that had been moved since the implementation was last updated. These did not strictly need to be changed as it did function anyway; however, reducing the number of warning messages received while starting the training environment was a simple process that allowed more important warning

and error messages to be found. Two other small additions were also made to the learn() function – a save_path argument was added to allow the policy to be saved using the save() function defined in baselines.common.policies.py, and the if statement checking if the load_path argument exists was also modified to check whether the actual load_path exists on the operating system. It was decided to add the save functionality directly to the learn() function instead of to the callback function, as it was slightly clearer where the save path should be input as an argument, and the learn() function is also taking care of loading previously saved policies. The check on whether the load path exists on the operating system was added solely to allow a load path to be included as an input argument even if the path did not yet exist, as this meant less time was spent updating the function arguments where the learn() function is called in ur10_reacher_6D.py.

The way that the network argument for the learn() function is used has been updated in the OpenAI Baselines since the version that was used in the SenseAct framework, and some small updates needed to be made to the ur10_reacher_6D environment to make it compatible with the updated OpenAI Baselines TRPO. Previously, a policy function had to be constructed and passed to the learn() function as an argument. This was replaced by the network argument that only requires the use of a string specifying the network type; other parameters for the network can be defined by passing keyword arguments to the learn() function based on the arguments used for constructing the network. For an MLP these are the number of layers, and the number of nodes in each layer. The network builders and the arguments they can take are defined in baselines.models.py.

The callback() function used in the Reacher 6D environment was also updated in callback.py. Neither the SenseAct framework nor the OpenAI baselines implement a way to output training data. In order to be able to compare the different policies' training curves, this data needed to be stored. The callback was updated to write the episode rewards and lengths to a comma separated value (.csv) file for further processing.

3.3.5 Running a Policy

The OpenAI Baselines TRPO implementation provides no functionality for running a trained policy without learning. As such, a function for running a trained policy needed to be implemented in order to test the efficacy of each learnt policy without the continuation of learning. A function called run_policy() was implemented in run_policy.py. Due to the complexity of the learn() function, it was decided to implement run_policy() by removing all parts of the learn() function that updated the policy and otherwise leaving it untouched. The run_policy() function was then used as the basis for running the tests described in section 4.

The base ReacherEnv environment defined in senseact.envs.ur.reacher_env.py was also updated to allow for a single continuous episode to be run, with no end condition. This was initially going to be used for testing the policies but did not end up being utilised in the final tests presented here.

3.4 Trust Region Policy Optimisation (TRPO)

Trust Region Policy Optimisation (TRPO) is a Policy Gradient reinforcement learning algorithm first described by Schulman *et al.* in 2015 [35]. Where value-based reinforcement learning methods require some way to map continuous action or state spaces to discrete ones, Policy Gradient methods naturally work well with continuous state and/or action spaces. These methods are so named because they use gradient ascent to converge to at least a local maxima of the policy function. The term policy function refers to a function that returns the overall value (potential for rewards) of a specific policy. The policy is parametrized using a neural network.

Policy Gradient algorithms work by sampling a number of trajectories (chains of states, actions, and rewards) generated using the current stochastic policy and then updating the policy based on the perceived value (long-term rewards) of these trajectories. In TRPO, the values of the state-action pairs in each trajectory are estimated using Monte Carlo simulation.

Once a set of trajectories and their value estimations is obtained, TRPO constructs a policy function that provides a lower-bounds estimate of the unknown true policy function using a Minorize-Maximisation algorithm [78]. This is an optimisation algorithm that exploits the convexity of a function to generate a minorized function (i.e. $f(x) \leq g(x)$ where $f(x)$ is the minorized function and $g(x)$ is the original function). The Conjugate Gradient method (mainly developed by Hestenes and Stiefel [79]) is then applied to this minorized function to find its local maxima. Bounds are calculated for the policy change to ensure that policy updates are large enough for sufficient learning progression without endangering the current policy's efficacy. The policy's parameters are then updated to those found to be the local maxima of the minorized function or are updated as far as they can be along this function towards the maxima based on the calculated update bounds. The use of a Minorize-Maximisation algorithm guarantees that the minorized function's maxima will eventually converge with the original function's maxima.

Policy gradient methods are usually very sample inefficient, with new state-action pairs needing to be sampled after every policy update. TRPO helps to address this sample inefficiency using importance sampling, a method that allows it to use samples from a small number of previous policies as well. However, the sample complexity of this algorithm is still quite high.

TRPO is a complex and sophisticated algorithm that the author cannot hope to do full justice in explaining in the space available in this dissertation. However, for a moderately high-level explanation and some intuitions of how TRPO functions, Jonathan Hui has an excellent blogpost on the topic [80]. For the low-level explanation and all of the relevant formulas and proofs, see the original TRPO paper [35].

3.4.1 Hyperparameters

The hyperparameters (i.e. the input parameters) used for running TRPO were kept the same as those used in the UR5 Reacher 6D environment. These are listed below.

Hyperparameter	Value	Description
num_layers	2	The number of hidden layers in the neural network used
num_hidden	64	The size of the hidden layers in the neural network
timesteps_per_batch	2048	The number of timesteps that trajectories are sampled for between policy updates
max_kl	0.05	A bound on the Kullback-Leibler divergence which is used to implement the bounds on each policy update
cg_iters	10	The number of iterations for each use of the conjugate gradient algorithm
cg_damping	0.1	A damping value used in the conjugate gradient algorithm
vf_iters	5	The number of iterations used for the value estimation algorithm
vf_stepsize	0.001	The learning rate of the value estimation algorithm
gamma	0.995	Variables used as part of the importance sampling
lam	0.995	

As TRPO was shown by Mahmood *et al.* to be the most insensitive to hyperparameters of the algorithms they tested, it was decided that tuning these values was not necessary.

3.5 Trained Policies

Three policies were trained corresponding to each of the three possible reset modes – 'zero', 'random', and 'none'. These three tasks were chosen due to the likelihood of differences in how well they might be able to generalise to other tasks. It was hypothesised that the 'random' and 'none' reset methods would generalise better than the 'zero' reset method, as there would be more robustness in their policies due to not starting in the same position on every episode.

Each policy was trained with a maximum of 200,000 timesteps using 4-second episodes with a timestep of 0.04 seconds (100 timesteps per episode). This timestep value was found to be a good intermediate value by the creators of SenseAct in [13]. The reset time between episodes for the 'zero' and 'random' reset modes was 2 seconds. Episodes were batched for processing by TRPO every 2048 timesteps with approximately 20 episodes per batch (some batches had 21 episodes due to the batch length not being an exact multiple of the episode length). These settings mirror those used by Mahmood *et al.* [13], [14]. 200,000 timesteps corresponds to around 2000 episodes. Each episode takes 4 seconds to run, plus 2 seconds of reset time for the 'zero' and 'random' reset methods. This corresponds to a training time of around 3 hours 20 minutes, or around 2 hours 13 minutes for the 'none' reset method. Due to the location of the UR10, it was not possible to leave it running unsupervised; therefore, all training was broken into separate sessions of 50,000 or 100,000 timesteps. Because the end condition for the learning of a policy is only checked at the end of a batch, the total runtime of these tests was 200,704 timesteps (the nearest multiple of 2048 that is greater than 200,000). Therefore, 2007 full episodes were conducted.

4 Tests and Results

Four different tests were implemented using the run_policy() function described in section 3.3.5. The implementation of these tests is in ur10_tests.py. All tests except the Reset Test required implementing new classes that override certain functions in the base ReacherEnv environment class. These are explained in detail in the sections below.

All results were obtained using data that was written to .csv files during the testing and learning of policies. The figures presented in this section were created in Matlab by importing the data from the .csv files and plotting it using either a 2D line or bar plot, or a 3D scatter plot in the case of the Grid Test data.

Section 4.1: This section will present the rewards obtained during the training of the three models.

Section 4.2: This section will present results detailing the efficacy of the three policies across the area of the bounded box and how this test was implemented.

Section 4.3: The results of two different moving point tests conducted on the 'zero' reset model are presented as well as the implementation-specific details of these tests.

Section 4.4: The results of running each trained policy using all three reset methods are detailed in this section.

4.1 TRAINED POLICIES

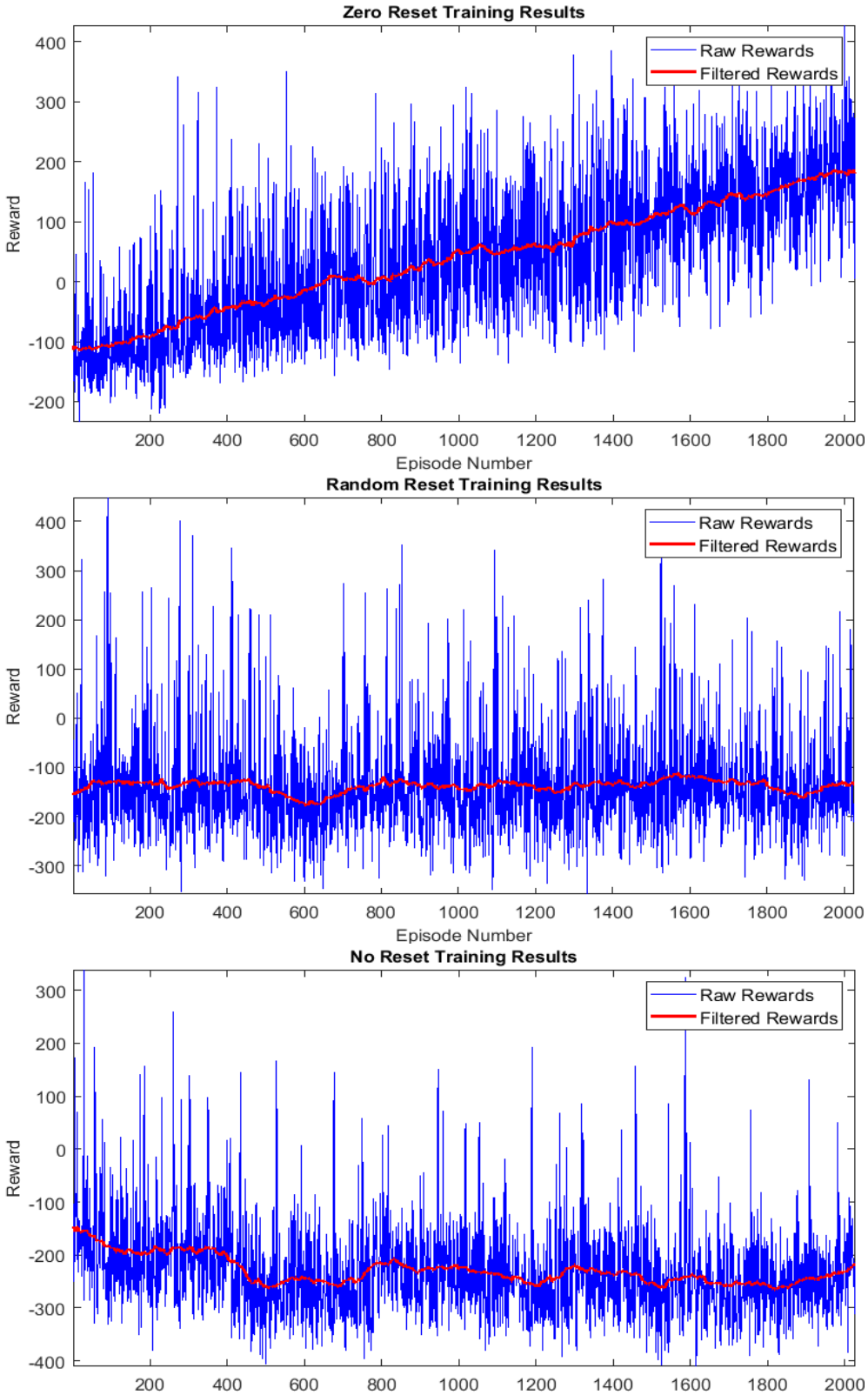

Figure 3: Rewards each episode for each of the models over the course of the training

Figure 3 shows the rewards obtained per episode over the 200,000 timesteps for which each policy was trained. The raw rewards lines correspond to the total reward that was obtained for each individual episode. There is a large variance in these due the random target generation for each episode and the stochastic nature of the policy. Because of this, filtered rewards lines have been added to better demonstrate the overall trend of the rewards during training. These were added using a moving average filter with a window size of 100 episodes centred on the current and previous episodes (since 100 is an even number and therefore does not have an integer median). Where there are not 100 episodes to draw from at each end of the training, the moving average draws as many samples as it can – e.g. for the first episode the filtered rewards value is obtained by averaging the first 50 episodes, and for the final episode the filtered rewards value is obtained by averaging the last 51 episodes.

The 'zero' reset policy consistently improved over the course of the training. The 'random' reset policy did not see any significant improvements during the training time. The 'none' reset policy received decreasing rewards across the first few episodes, and then evened out to moderately consistent rewards, though these more consistent reward values were worse than that of the 'random' policy.

4.2 GRID TEST

This test is designed to split the bounded area into a grid of points and to test how the policy functions for each of these points multiple times. The rewards for each test on a point are then averaged. This allows an overview of how the policy functions across the bounded area.

This test is implemented in the GridTestEnv class in ur10_tests.py. The class takes four extra input arguments as well as the same input arguments as ReacherEnv. These are a number of x, y, and z points to split the bounded box into and the number of tests to run on each point. The GridTestEnv class overrides the reset() function of the ReacherEnv. Instead of generating a new random target point at each reset, it yields a new point from a generator function target_generator(). The target_generator() generates lists of the x, y, and z coordinates to be yielded and loops across these. These points are generated to be evenly spaced within the bounding box without generating points on the edge of the box. Points on the edge of the bounding box would not give an accurate representation of the policy due to the way in which the environment deals with the end effector point moving outside of the box (as detailed in section 3.3.2 and expanded upon later in section 5). These would potentially generate higher reward values as the end effector can only move in half the space (or a quarter of the space for points on an edge and an eighth of the space for corners) than it can around other points.

The total number of episodes to be run is calculated by multiplying the x, y, and z point numbers and the number of episodes per point. This very quickly leads to a large number of tests needing to be run. Based on this, the tests used on the trained policies split the bounding box into a 5 x 5 x 5 grid and tested each point five times, leading to a total of 625 episodes (5 x 5 x 5 x 5). The call to run_policy() is set up so that each iteration includes the same number of episodes as the number of episodes per point. This allows each call to callback() within run_policy() to publish one point's worth of tests to the .csv file.

A separate callback() function was created for use with the GridTestEnv. This is similar to the callback function used during training, but instead of saving only the episode returns and lengths, it also saves the coordinates of the target point for that iteration.

4.2.1 Results

Figure 4, Figure 5 and Figure 6 show the results of running the Grid Test using each of the three trained policies. Each point on these graphs represents a point in space that was used as the target for five consecutive tests using the 'zero' reset method. The rewards value for each is the mean of the values obtained from these five tests. The box on the graph displays the bounded box formed by the upper and lower end effector position limits in which all training and tests took place. The red star represents the point defined as the 'zero' reset position. This is the point to which the arm was reset between every episode. Note that the colour scale for each graph corresponds to different reward values based on the maximum and minimum rewards obtained for that test. Note also that due to a minor error in the implementation of the grid test, the final point's mean reward value was not saved, and therefore it is not included in these graphs. For context, the base of the UR10 is mounted at (0, 0, 0) on the XY plane. In this case, y is the direction in which the arm is extending further from itself as it is increased, x is to the left and right of this, and z is the height of the end effector.

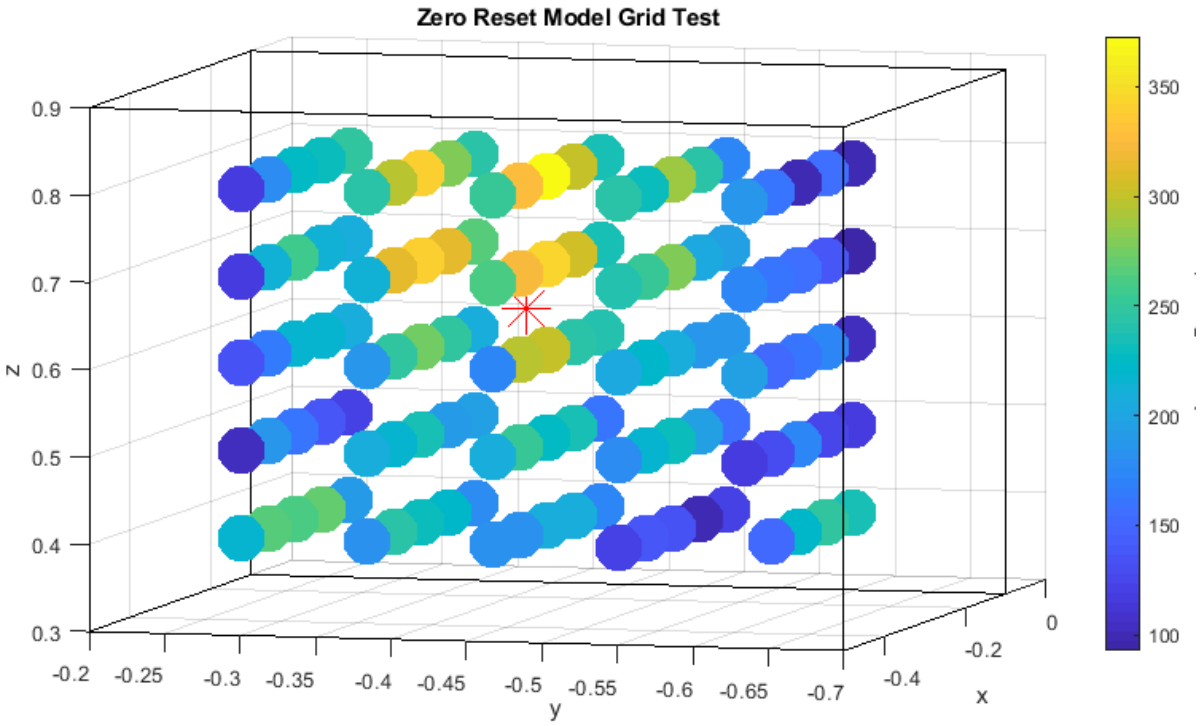

Figure 4: Grid Test results for the model trained using the 'zero' reset mode

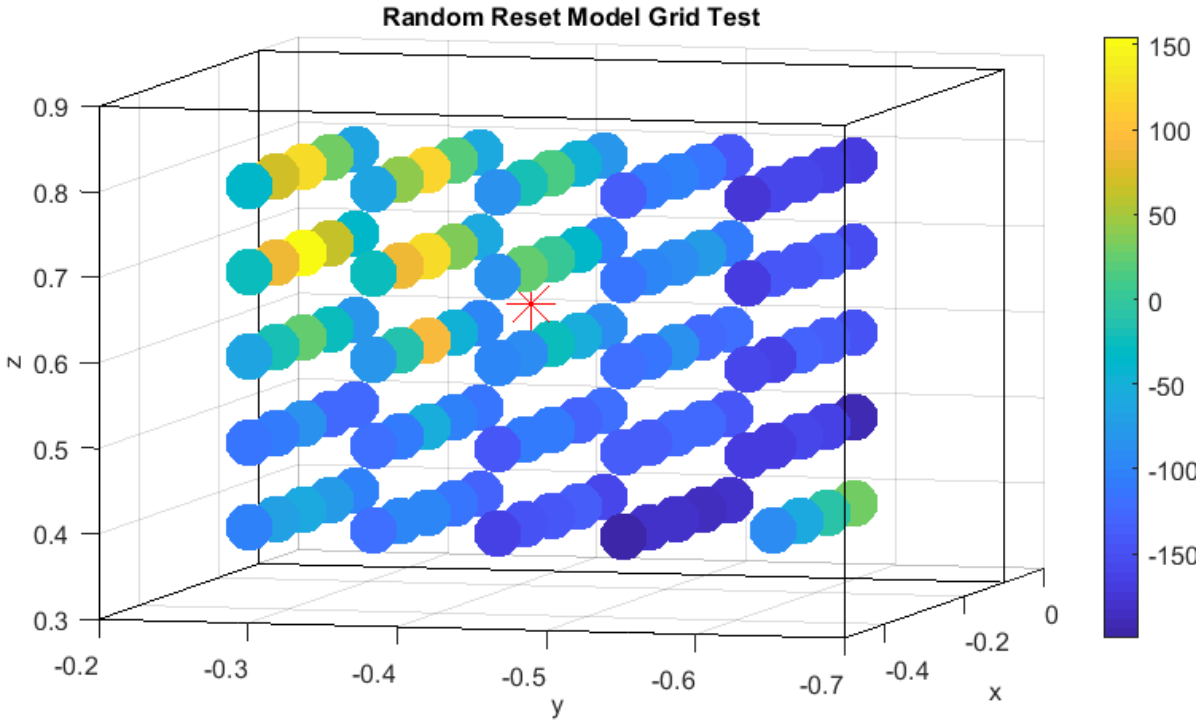

Figure 5: Grid Test results for the model trained using the 'random' reset mode

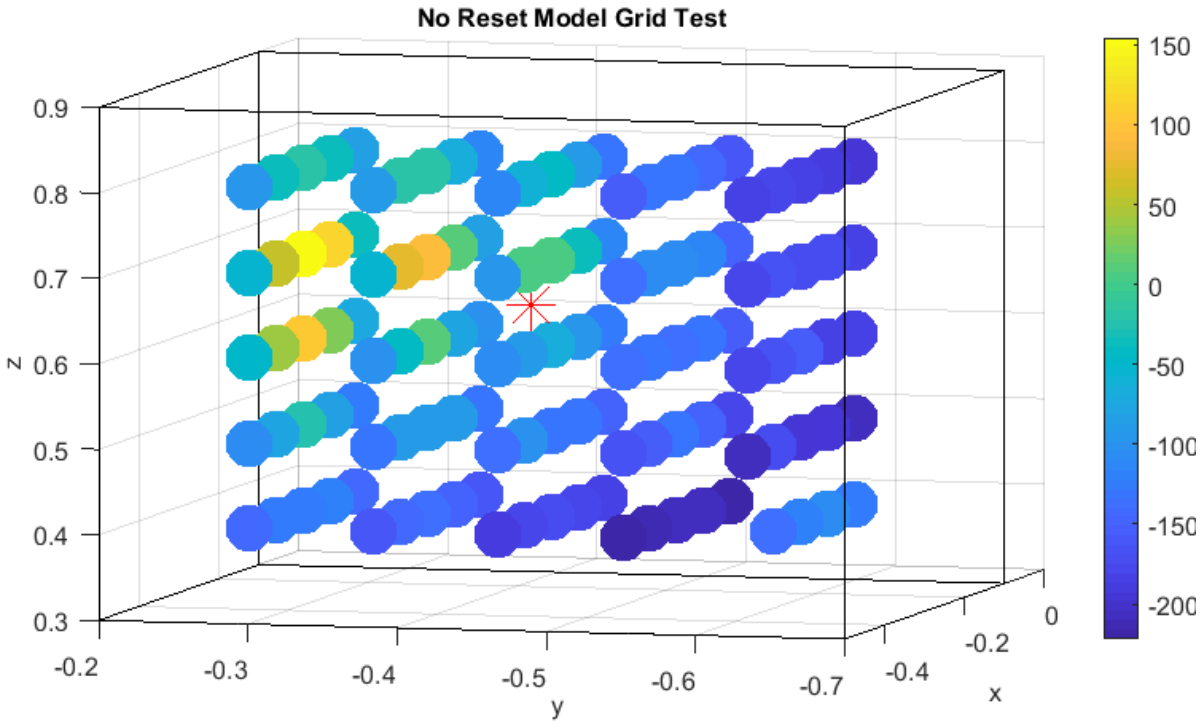

Figure 6: Grid Test results for the policy trained using the 'random' reset mode

These tests display some small biases in the directions that the trained policies direct the end effector. Overall, they show that none of the trained policies are as isomorphic as might be expected. For the 'random' and 'none' reset policies, this bias is towards the positive y direction. For the 'zero' policy it is towards the positive z direction.

4.3 Moving Point Tests

These tests are designed to replace the static target point with a moving point in order to test how well the trained policies are able to generalise to other similar tasks. Two tests are included in the MovingPointEnv class: a test that moves the target point back and forth along a line and a test that moves the point clockwise around a circle. These can be selected via the move_shape input argument. Due to the fact that the RTRLBaseEnv environment that ReacherEnv inherits from is set up to use multithreading or multiprocessing (depending on an input argument) to run its main real-time loop, updating class attributes does not update them in the thread or process running the loop. As such, implementing an update to the target point on every timestep required overriding not just the reset() function as with the GridTestEnv environment, but also the start(), step() and run_loop() functions from the RTRLBaseEnv.

The start() function in RTRLBaseEnv initialises the separate thread or process that runs the real-time loop. For all training and tests, multiprocessing was used. The function was modified to add a queue to the process in order to be able to pass the updated target values into the process as part of each loop. The run_loop() also had to be updated to get the most recent value put into the queue and update the target point with it. The actual update of the target point on each timestep is done in the overridden step() function that is called by the TRPO learn() function to progress the environment by one timestep. The step() function calls either the circle or line generator function as appropriate. These generators are described in detail in sections 4.3.1 and 4.3.2.

As with the GridTestEnv, the reset() function from ReacherEnv is overridden to change the method of resetting the target point. The appropriate generator function is reinitialised in order to reset the target point to the centre of the line or beginning point of the circle, rather than just allowing it to continue from where the previous test finished. The first point is then yielded from this function as the reset target point.

A separate callback() function was created for use with this class that writes the parameters of the test to the .csv file alongside the rewards.

4.3.1 Line Test

For the line test, a generator is implemented that uses input arguments to the environment to generate a target that moves back and forth along a straight line. The input arguments related to this test are the velocity that the point should travel in m/s, the midpoint that the line should pass through, the axis that the line should travel in the direction of, and the length of the line in metres. The length of the line defines how far the moving point should travel along the specified axis from the midpoint before changing direction and going back along the line. The length defines the full length of the line and as such, the point will travel half of this distance in one direction before changing direction and travelling back through the midpoint to half of this distance in the other direction. The point is moved by a distance specified by the velocity multiplied by the timestep of the environment in the direction of travel. When the point reaches the end of the line, the direction of travel is reversed.

For testing the trained policies, this test is run along each of the axes with a set length (0.30m) and increasing velocity – 0.025m/s, 0.05m/s, and 0.1m/s – in order to see how the velocity of the moving point affects the rewards obtained. The results are averaged across 20 episodes. Since the ‘random’ and ‘none’ reset method policies largely did not improve across their training, it was decided that these tests would only be run for the ‘zero’ reset mode trained policy, as no information would be gained by running them on the others.

4.3.1.1 Results

The results for the test conducted on the 'zero' reset policy are presented below. The *Raw Reward* column displays the mean of the rewards obtained across the 20 episodes for each test. Each of the 20 episodes was run for a number of timesteps corresponding to how long it would take the moving point to get from the centre of the line it is tracing all the way to the left, to the right, and then back to the centre (i.e. twice the *Line Length*). This is displayed in the *Timesteps Per Episode* column. The *Normalised Reward* column is then calculated from the raw reward by dividing the raw reward by the number of timesteps per episode for that test and multiplying it by 100 – the number of timesteps for all other episodes in this paper. This allows for comparison of the normalised reward values to those obtained in other tests and during training. All raw and normalised reward values are rounded to two decimal places. The unrounded raw values were used to calculate the normalised rewards.

Table 1: Line Test results for the 'zero' reset method policy

Velocity (m/s)	Line Length (m)	Plane	Raw Reward	Timesteps Per Episode	Normalised Reward
0.025	0.3	XY	564.95	600	94.16
0.05	0.3	XY	604.76	300	201.59
0.1	0.3	XY	358.04	150	238.70
0.025	0.3	YZ	797.23	600	132.87
0.05	0.3	YZ	527.87	300	175.96
0.1	0.3	YZ	297.99	150	198.66
0.025	0.3	XZ	1087.02	600	181.17
0.05	0.3	XZ	732.03	300	244.01
0.01	0.3	XZ	267.72	150	178.48

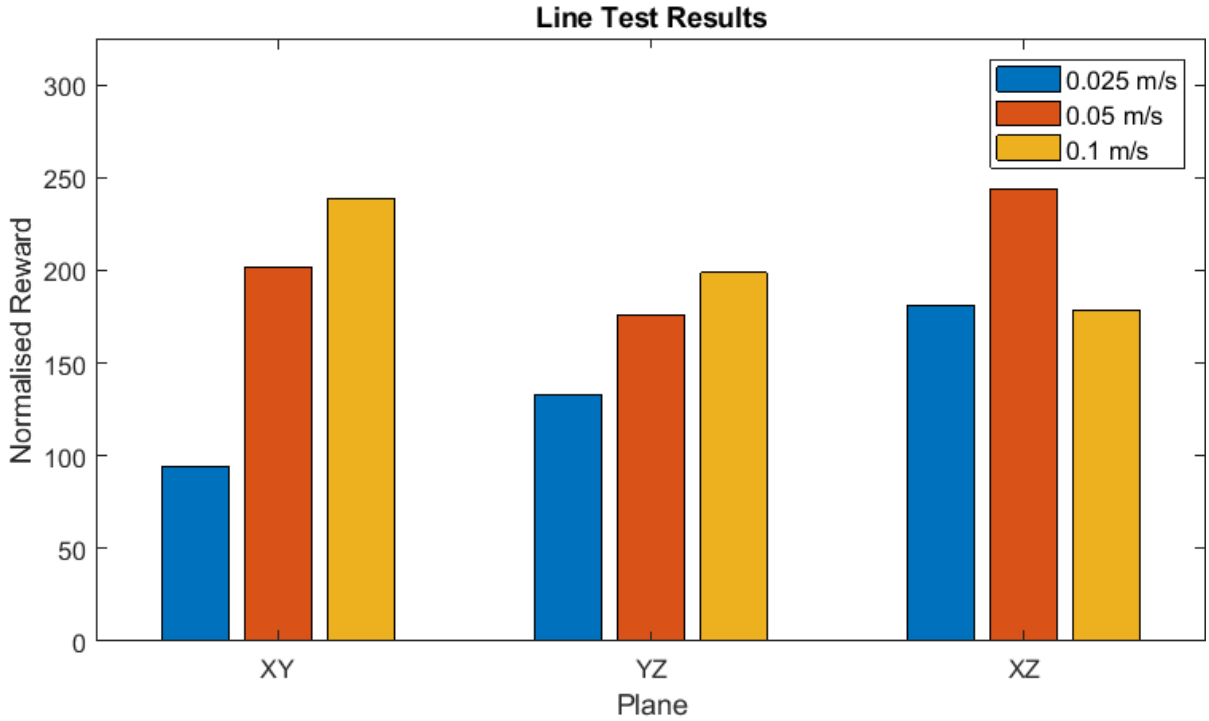

Figure 7: Line test rewards obtained for each of the three different point velocities

Figure 7 shows the normalised mean rewards obtained for the line tests on the 3 planes. The three bars represent the 3 different point velocities that were used for the tests. The planes represent the plane through which the line that the point was travelling runs perpendicular to – i.e. for the XY plane tests the point was only moving in the Z-direction.

These results somewhat counterintuitively show higher rewards for the faster moving point, except in the case of the XZ plane in which the medium velocity point gave the best rewards.

4.3.2 Circle Test

The circle test point generator is implemented in a similar manner to the line generator. The input arguments related to this test are velocity in rad/s, radius in metres, and the plane that the circle should lie on (xy, yz, or xz). The circle is centred around the centre point of the bounded box. At each timestep, the angle around the circle – theta – is increased by the velocity multiplied by the length of the timestep. This theta value is then used to calculate the appropriate coordinates for the point using basic trigonometry.

For testing the trained policies, this test is run on each of the planes with a constant angular velocity of 0.5 rad/s and an increasing radius – 0.10m, 0.15m, and 0.20m. Increasing the radius with a constant angular velocity also increases the linear velocity of the moving point. The results are averaged across 20 episodes. Since the 'random' and 'none' reset method policies largely did not improve across their training, it was decided that these tests would only be run for the 'zero' reset mode policy, as no information would be gained by running them on the others.

4.3.2.1 Results

Table 2 displays the data obtained from the circle tests on the 'zero' reset mode policy. As with the line test, the episode lengths for these tests were set to be the number of timesteps required for the moving point to finish one complete circumference of the circle. The normalised rewards are calculated by dividing the raw reward by the timesteps per episode and multiplying this by 100 (the number of timesteps used per episode in all other policy training and testing). All raw and normalised rewards are rounded to two decimal places. The unrounded raw values were used to calculate the normalised rewards. The planes mentioned define the coordinate plane of the listed axes, translated to pass through the centre point of the bounded box – i.e. the XY plane defines the XY coordinate plane, translated along the Z axis by 0.6m (0.6 being the centre point of the minimum and maximum Z bounds of the box – 0.3m and 0.9m respectively).

Table 2: Circle Test results for the 'zero' reset policy

Radius (m)	Velocity (rad/s)	Plane	Raw Reward	Timesteps Per Episode	Normalised Reward
0.1	0.5	XY	462.9673	314	147.44
0.1	0.5	YZ	334.121	314	106.41
0.1	0.5	XZ	587.8081	314	187.20
0.15	0.5	XY	236.7523	314	75.40
0.15	0.5	YZ	155.2297	314	49.44
0.15	0.5	XZ	366.6981	314	116.78
0.2	0.5	XY	-0.4975	314	-0.16
0.2	0.5	YZ	-8.26881	314	-2.63
0.2	0.5	XZ	91.27394	314	29.07

Figure 8 below displays the information from Table 3 in a bar graph. Note: as the reward value for the test conducted on the XY plane with a 0.2m circle radius is very close to 0, it does not show up on the graph.

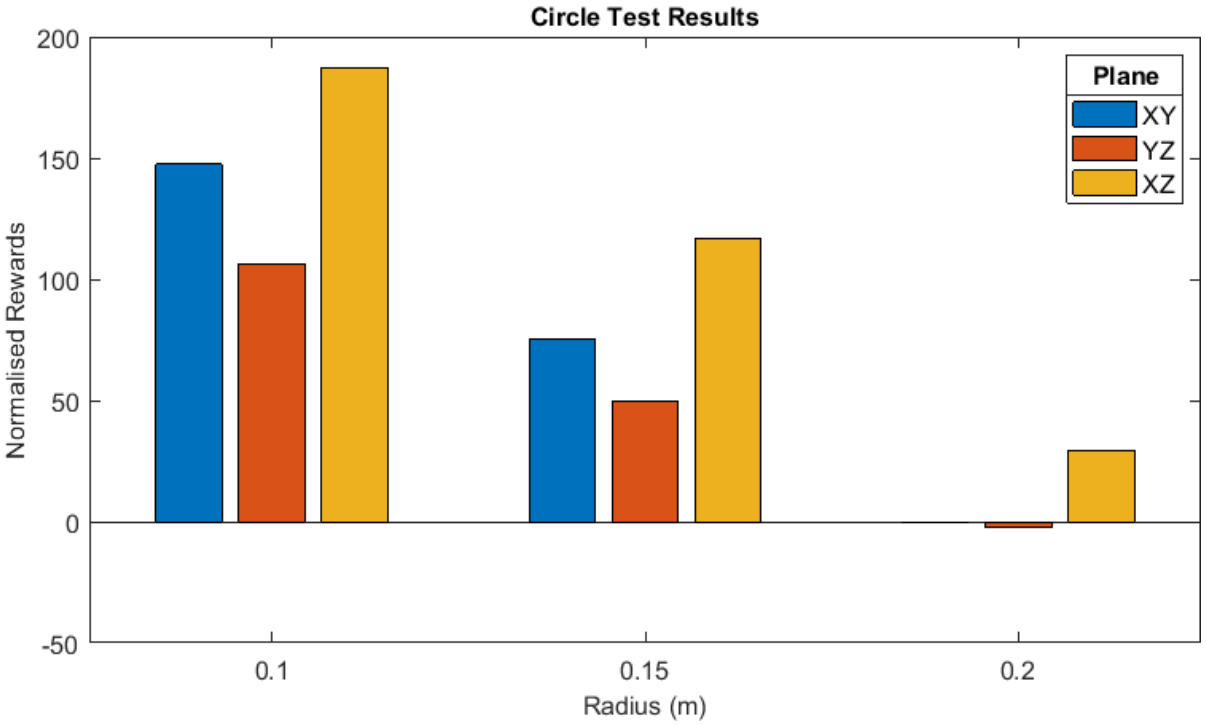

Figure 8: Circle Test results for each of the different velocities tested on each of the 3 planes

Overall the policy was best at dealing with a circle on the XZ plane, and worst when dealing with one on the YZ plane. As the radius of the circle grows (and therefore the linear velocity of the point increases) the rewards drop significantly.

4.4 Reset Test

Each trained policy was tested by running it for 10,000 timesteps using each of the three reset modes – 'zero', 'random' and 'none'. This was done using a 4-second episode time with a timestep of 0.04 seconds, meaning each episode lasted for 100 timesteps. The timesteps per batch was set to 2048. This mimics the task setup used during training. Due to the fact that the end condition for the learning or running of a policy is only checked at the end of a batch, the total runtime of these tests was 10,240 timesteps (the nearest multiple of 2048 that is greater than 10,000). As such, 102 full episodes were conducted.

This test aims to compare how well the policies trained using different reset methods can then generalise to using the other reset methods. For example, it is expected that the policy trained using the 'zero' reset method would perform best when running it using the 'zero' reset method, but would perform worse when using the 'random' and 'none' reset methods than the policies that were trained using those reset methods.

4.4.1 Results

Figure 9, Figure 10 and Figure 11 display the results from running the trained policies using each of the three reset methods ('zero', 'random, and 'none') – i.e. Figure 9 shows the rewards obtained by running the policies trained using the 'zero', 'random' and 'none' reset modes on a test using the

'zero' reset mode. A filtered rewards line was added using the same method as detailed in section 4.1, with a window size of 10 episodes for ease of readability. As the policies were not learning during these tests, the variance in rewards is due only to the random target locations and the stochastic nature of the policies.

Table 3: Mean rewards obtained across each of the reset tests

Test Reset Mode	Policy Reset Mode	Mean Reward	Standard Deviation (σ)
Zero	Zero	165.11	62.30
	Random	-81.16	98.93
	None	-112.56	87.59
Random	Zero	-45.80	123.34
	Random	-118.09	121.62
	None	-150.06	104.64
None	Zero	-168.60	92.35
	Random	-197.57	120.93
	None	-186.27	108.06

Table 3 displays the mean and standard deviation for the rewards obtained across each of the nine reset tests. *Test Reset Mode* specifies the reset mode that was used during the test. *Policy Reset Mode* specifies which trained policy was used for that test. All mean rewards and standard deviations are rounded to two decimal places.

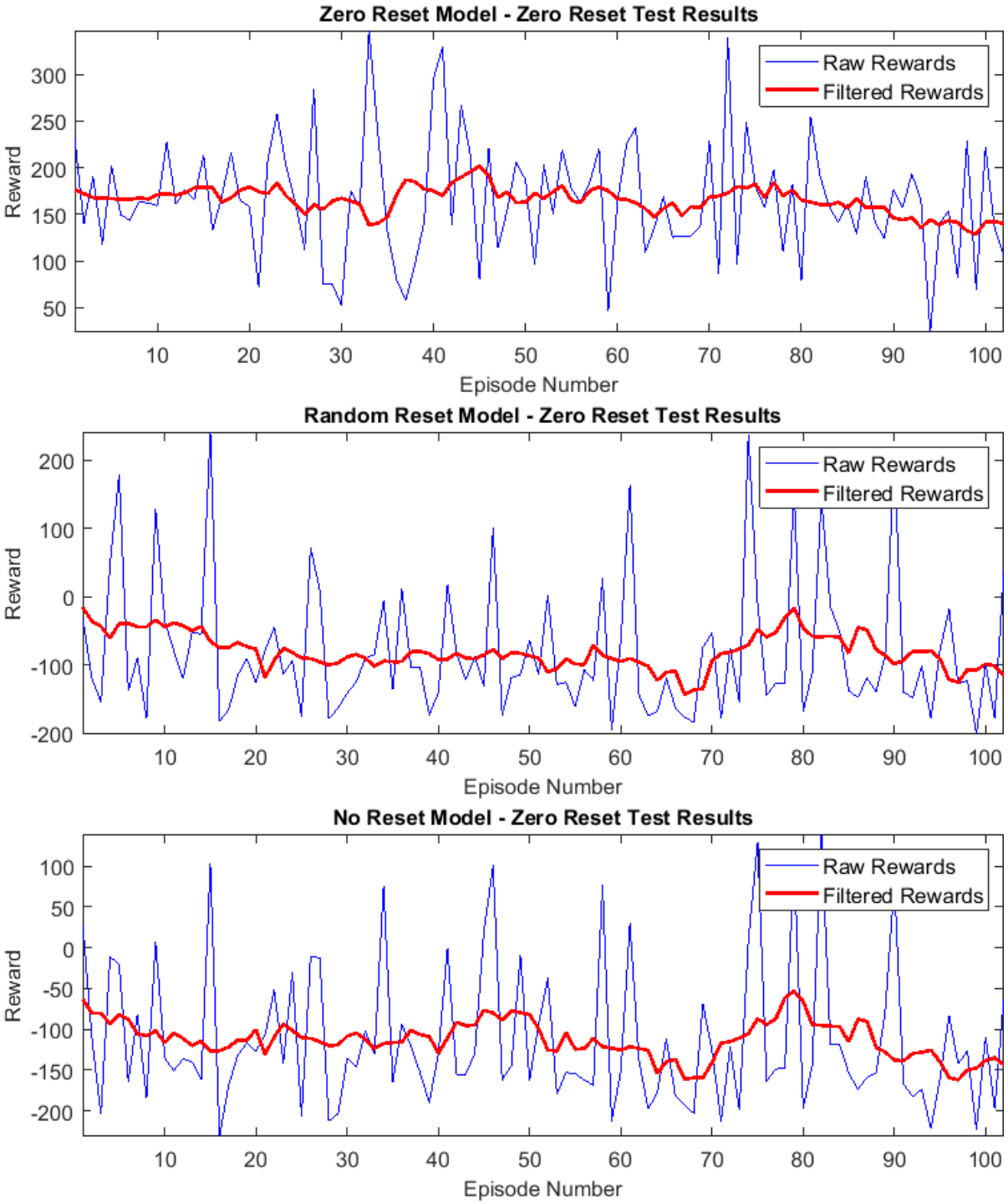

Figure 9: 'Zero' reset mode tests for each trained policy

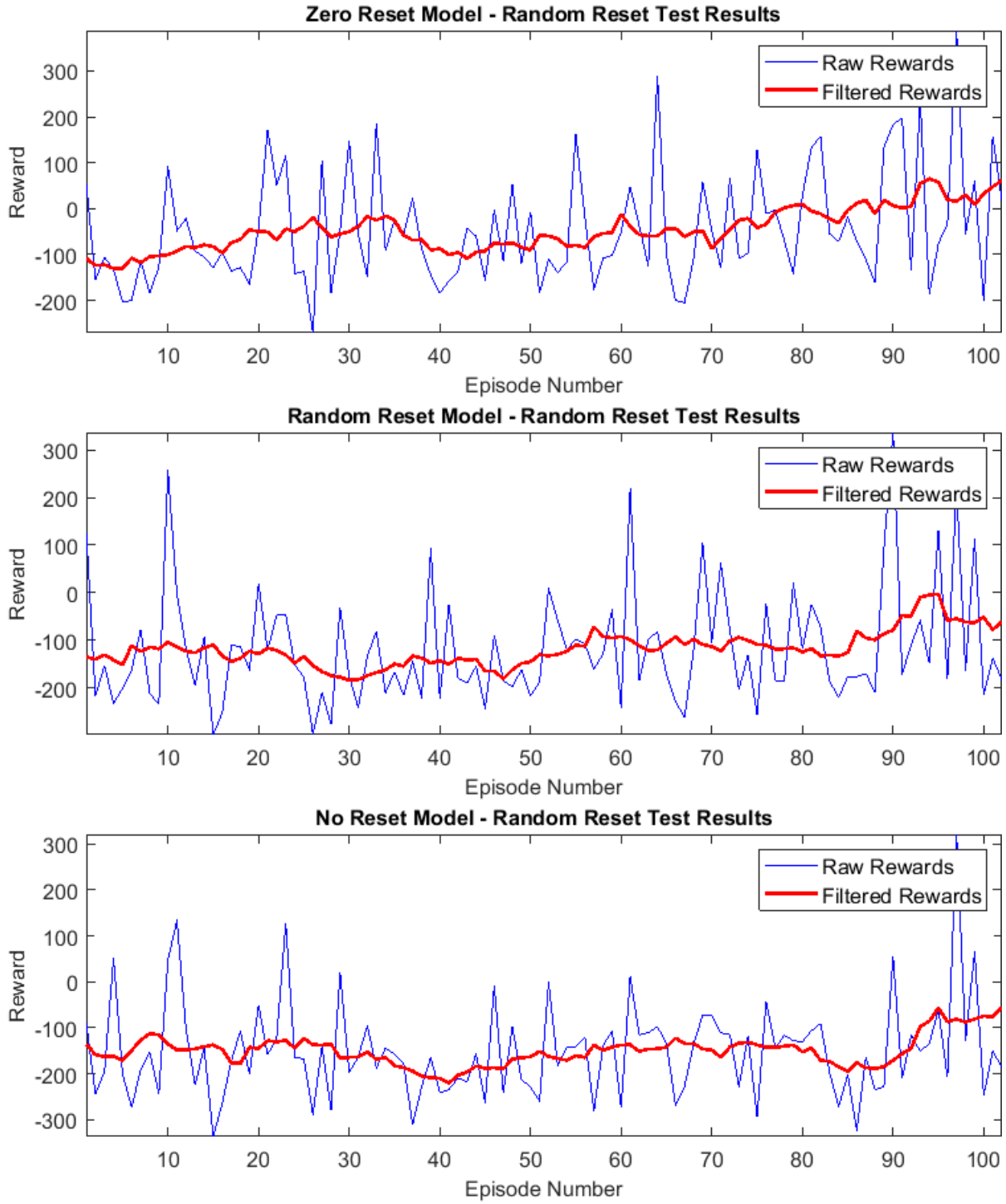

Figure 10: 'Random' reset mode tests for each trained policy

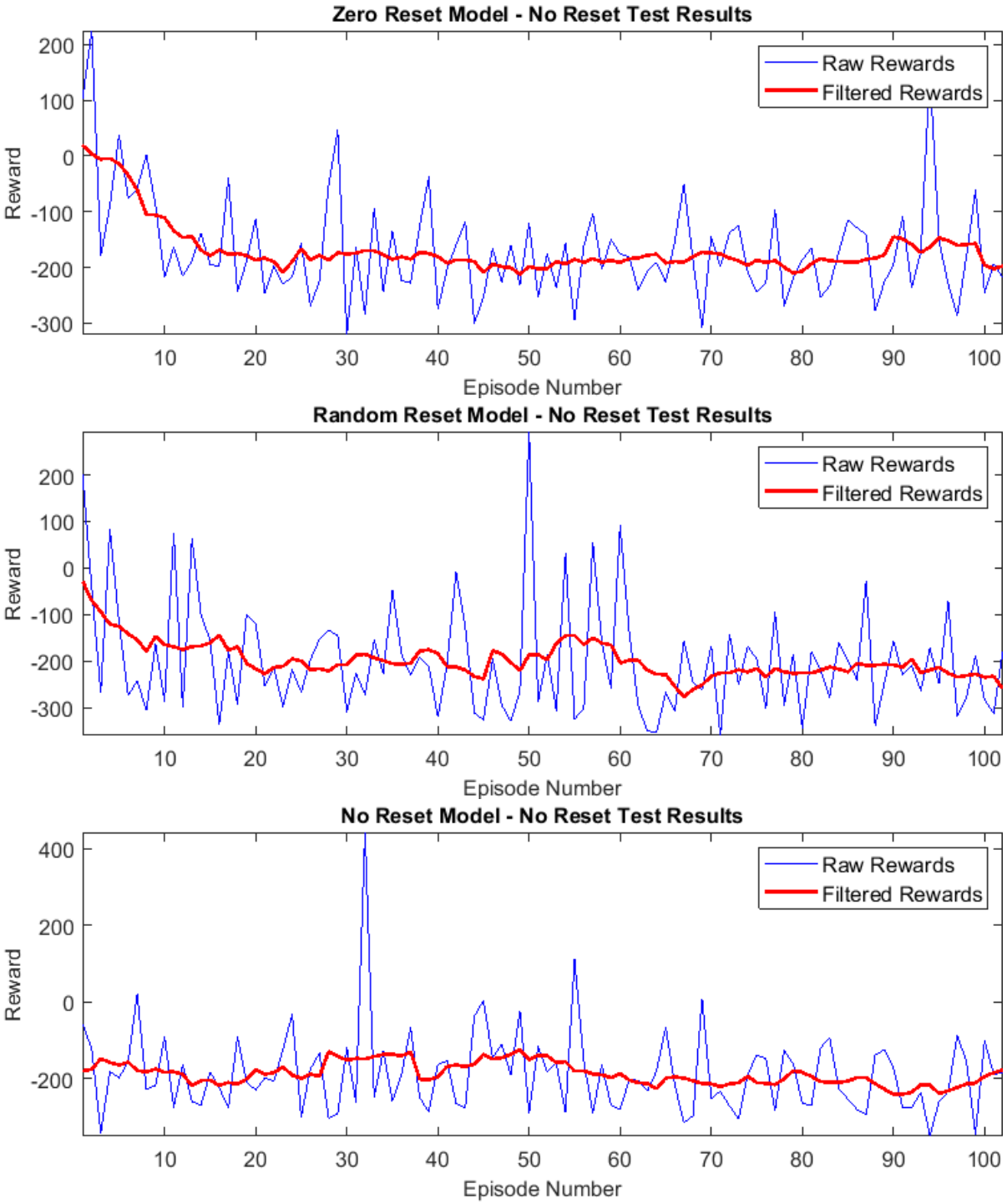

Figure 11: 'None' reset mode tests for each trained policy

The most interesting information to be drawn from these graphs is demonstrated in Figure 11. The 'zero' and 'random' policies start off performing better than the 'none' reset policy, but very quickly drop to receiving rewards in a similar range to the 'none' reset policy.

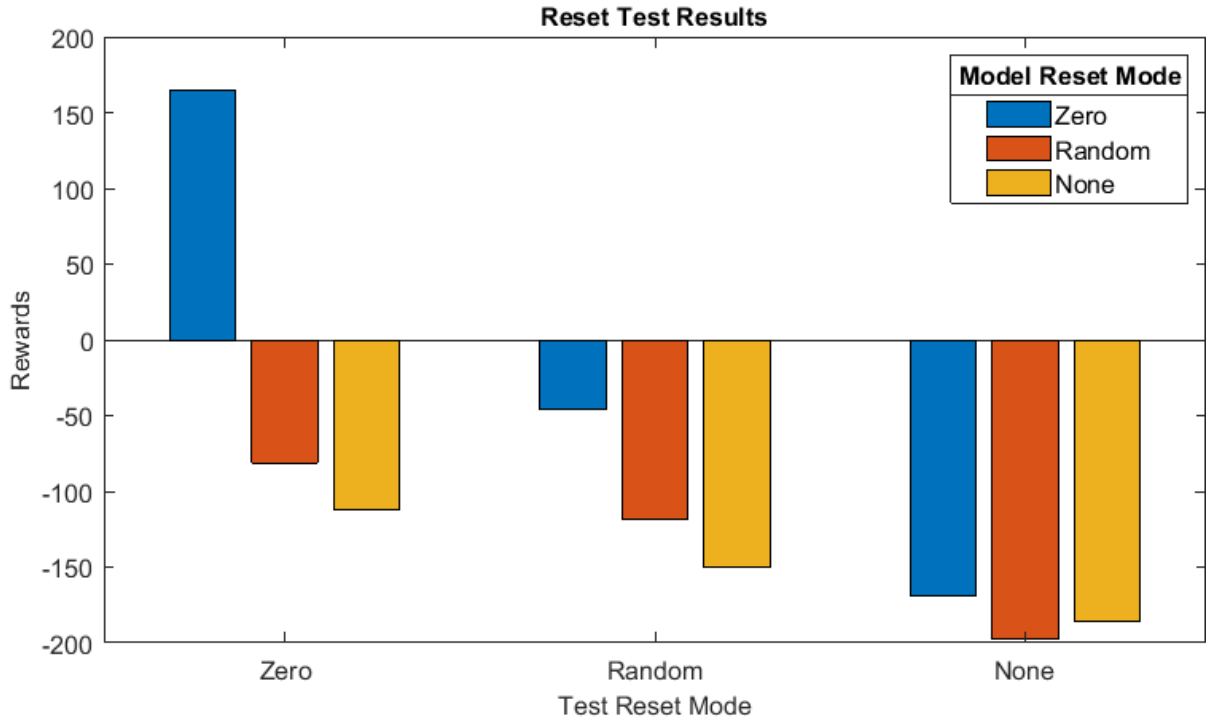

Figure 12: Summary of the mean rewards obtained using each of the three trained models for each of the reset tests

Figure 12 displays the data from Table 1 as a bar graph by test reset mode. Each of the three trained policies are presented for each of the tests.

Table 3 and Figure 12 show that the policy trained using the 'zero' reset method performs better when run using the other two reset methods than the policies that were trained using those reset methods. The 'random' reset policy also performs better than the 'none' reset policy, except in the case of the 'none' reset test where the 'none' reset policy slightly outperforms it, though all policies receive very poor rewards on this test.

5 DISCUSSION

This section will discuss the results presented in section 4 and provide avenues for future research on this topic based on gaps in the information gained from the results. The following information will be discussed in the order given below:

Section 5.1: The results for each of the three trained policies will be discussed.

Section 5.2: The results for the tests performed using all three reset methods on each trained policy will be discussed.

Section 5.3: The efficacy and biases of the policies demonstrated in the grid tests will be discussed.

Section 5.4: This section will discuss the results of the 'zero' reset policy on the moving point line tests.

Section 5.5: This section will discuss the results of the 'zero' reset policy on the moving point circle tests.

Section 5.6: Some avenues for other possible research and testing are outlined here. Only those not already covered in the previous discussion sections are included.

5.1 Trained Policies

The three trained policies detailed in Figure 3 all performed quite differently. The 'zero' reset policy improved consistently over the training time, with no signs of slowing down. This implies that if the policy were to be trained longer it would continue to improve, and this would be worth exploring in future research. The task with the 'zero' reset is the simplest of the three, as there is at least one constant between each episode – the set reset position. Therefore, it is not surprising that this policy learnt the fastest of the three that were trained.

The 'random' reset method policy did not improve in any significant way over the course of the training. This is likely due to the fact that the system did not regularly encounter states that it was able to relate to states previously encountered, as for each episode there was a new random starting configuration of the joints as well as a new random target point. It is possible that a functional policy could be learnt for this training task with a much larger training time, and that if it did learn a policy, the policy would be less overfit and more generalisable than that of the 'zero' reset method. However, this was not able to be confirmed from the experimental results obtained and would need to be explored further to test this hypothesis.

The 'none' reset policy has the most interesting training graph of the three. There is an initial dip in the rewards gained each episode, followed by a further dip in rewards around episode 400. The 'none' reset policy began the first episode in the 'zero' reset position near the centre of the bounded box. From here, observation of the learning showed that over the next few episodes it drifted to one of the walls of the box. As the arm was not reset between each episode, it ended up getting somewhat 'stuck' against this wall (or wedged against an edge of the box), with a large portion of each episode taken up with moving the arm back inside the bounded box, using the boundary condition handling detailed in section 0. This leaves very little time in each episode for the actions dictated by the current policy to be taken. The second drop in rewards was observed to be due to the end effector getting stuck in one of the corners of the bounded box. This position leads to the most negative rewards on each episode, as the random target position will be further away on average than if the end effector is stuck against an edge, and much worse than if the end effector is towards the centre of the box. During the training, for certain joints, the arm would also regularly get stuck near its joint limits. This, combined with the end effector position correction, decreases the amount of time it can be actively testing the results of actions.

Unlike the 'random' policy, if the 'none' policy does get stuck in similar positions for a long period of time, it will at least have a chance to learn a decent policy for those positions. Once the policy is strong enough to move it away from that point, however, it will then be starting each episode in a new position that it cannot easily relate to the previous set of positions for which it learnt the policy. As with the 'random' reset policy, it is possible that with a much longer training time, the 'none' reset policy would be able to learn some level of functionality; however, this would likely require a very large number of episodes. The one advantage of the 'none' reset mode over the other two is that, as there is no reset time between episodes, training times for the same number of episodes are significantly shorter (33% shorter with 4-second episodes used for all training and 2-second reset times used for the 'zero' and 'random' reset tasks). The reason for attempting to train a policy in this way is that, as with the 'random' reset policy, it was hypothesised that this method of training would lead to a more robust policy across the entire state space, which would generalise better than the 'zero' reset policy to similar tasks.

5.2 Reset Tests

These tests aim to demonstrate how well each of the three trained policies can generalise to the somewhat different tasks of using the reset methods that the policy was not trained with, and to compare this to the policy that was trained using that method.

Overall, from Figure 12 it can be seen that the 'zero' reset method performs best on all three tasks. This is likely due to the fact that during training, this policy was able to learn some level of functionality, and did succeed in improving its rewards over time. While it still fails to obtain a positive overall reward during the 'random' reset tests, it does perform significantly better than the other two policies, demonstrating that this policy does have some capacity for generalisation to this task.

All three policies perform very poorly when using the 'none' reset mode. From observations of the experiments, this is likely due to the same issues that occurred in training the 'none' reset policy, where each of the policies begins the first episode towards the centre of the bounded box and then ends up drifting into a wall, edge, or corner of the box. The graphs in Figure 11 show that for the policies trained using the 'zero' and 'random' reset methods, there is a decrease in rewards over the first few episodes as this pattern occurs. Even the 'zero' reset method – which should be able to perform better than just taking random actions – still got very stuck and received very poor rewards in the 'none' reset tests.

The performance of the policies on this test suggests that instead of training a policy using the 'random' reset method from the outset, it might be better to train it for some amount of time using the 'zero' reset method, and then to switch to training it using the 'random' method in order to incrementally grow its skills. This could also be true for the 'none' reset method, as the 'zero' reset policy does perform marginally better on this task as well. However, it is likely that addressing the issues with how the environment handles the joint and end effector position limits would be more fruitful in the first instance.

Two possible options to address these boundary issues are, adding negative rewards based on distance outside of the bounded box or proximity to joint limits, or adding a large negative reward and ending the episode if these bounds are breached. With the first option, it would still be possible (and in fact fairly likely during the early training stages) for the arm to self-collide or to end up large distances outside of the bounded box. With the second option, this would not be a problem. The issue with both options, however, is that teaching the robot to avoid the joint and end effector limits specified necessarily diminishes the ability of the policies to generalise. If the actual policy learns not to approach the limits, then it would not be possible to use the policy in other situations, such as with different joint and end effector limits, without retraining it. Another possible option to address this would be to add some form of collision detection to the environment, such as using MoveIt with ROS for self-collision detection and prevention [81]. This would do away with the need for the joint limits; instead episodes could be ended if within some padded area for detecting self-collision, and a significant negative reward given. In this way, the arm could learn how to avoid self-collision in a way that is still generalisable.

Training the arm to a certain degree in simulation before transferring the policy to be trained further on the real robotic arm (sim-to-real training) could also be another option for addressing these issues. This was shown to be a potentially promising method by Rusu *et al.* [10]. If points are only generated within the bounded box and the arm has learned some level of functional policy, it should not need to stray far outside the bounded box, and therefore the end effector limits could be

omitted altogether. In this way, the potentially unsafe period of the training could be skipped on the real robot and done in the safe environment of a simulation.

5.3 GRID TESTS

The grid test results presented in Figure 4, Figure 5 and Figure 6 show results that are expected, and some that are more surprising. It was expected that for these tests, the rewards would be somewhat higher around the reset position (marked by the red star), and would tend to be lower as the points get further from this. This is because the initial reward obtained for points further from the reset position is lower, and even if the arm is successful in moving closer to those points, the overall reward for the episode will be lower because of this. This is true for all three policies to some extent; however, there are also clear biases in the three policies, and certain points (such as those nearest the edge defining the minimum Z and Y values of the box) actually show slightly higher rewards than the points around them. The learning is not isomorphic across the cartesian bounded box, even in the case of the 'zero' reset policy, which did improve itself during training.

There are a few possible causes for these discrepancies. For the points nearest the edge that define the minimum Y and Z limits of the box, it is possible that the end effector got stuck in that corner when overshooting these points (as described in the two previous sections of this discussion in relation to the 'none' reset method), and that this would then lead to higher rewards for these points, due to the end effector being limited in multiple directions as to how far it was able to stray from the target point. It is also possible that this position, with the end effector fairly low and the arm outstretched, is more likely to occur based on the joint configuration of the UR10.

While with an optimal policy it would be expected to be generally isomorphic, in fact the robot has each joint controlled individually. As with all industrial robotic manipulators, moving in joint space is simpler and faster than moving in cartesian space, and moving linearly in cartesian space does not equate to evenly distributed work between the joints – i.e. increasing the velocity of any of the first three joints – hip, shoulder and elbow – by a certain amount leads to much larger movements in cartesian space than increasing the velocity of the final three joints by a similar amount. At the beginning of the training, the policy has no particular bias for any of the six joints and will attempt to use all of them equally. This might help to explain the very similar biases in the 'random' and 'none' reset policies towards the positive Y-direction. This corresponds to the end effector moving close to the Z-axis, which runs through the base of the arm. It is likely that this is a more 'comfortable' (e.g. more probable) configuration of the arm in joint space, and as these two policies still have not progressed far beyond random action selection, there is a bias towards points in these directions. In future work, it would be interesting to quantify how 'comfortable' the robot is in various positions and to see if this has any significant correlation with the results obtained in these tests. This could be done using a sophisticated inverse kinematics solver to assess the number of possible joint configurations for the set of points tested. This would require a definition of what constitutes different joint figurations. However, in the continuous space that is the real world, there is a technically infinite number of joint configurations for any cartesian point that the UR10 can reach.

The second potential contributing factor to biases is that of the random target generation. With the relatively small number of episodes on which the policies were trained (slightly more than 2000 episodes), there is no guarantee of a particularly even distribution of the random target points across the cartesian space. It is possible that some of the biases these policies learnt are due to this uneven distribution. For example, with the 'zero' policy, it may have had a disproportionate number of targets generated that were in the positive Z-direction from the reset point, and this may account for the bias of the policy to the test points in this direction. In general, the direction bias of the 'zero'

policy is less pronounced than that of the ‘random’ and ‘none’ policies, with the target points that earned the highest rewards mostly being closer to the reset point. It should also be noted that the reward scales for each graph are somewhat different, with the ‘zero’ reset policy generally achieving far higher rewards for all points than those of the ‘random’ and ‘none’ policies. In future research, when training the policies, the targets for each episode could be recorded in order to plot their distribution across the space and to see how this correlates with the biases found in the grid tests.

A third possible contributing factor to any single-point outliers on the grid test is simply the small number of samples that were averaged for each target point. Each point was tested five times, and the rewards of these episodes were averaged to give the values presented on the graphs. With the small sample size of only five episodes and the stochastic nature of the policy, it is possible that individual points recorded higher or lower rewards than their true value. This could obviously be mitigated by testing each point a larger number of times. Having a higher density of points would also provide more information; however, due to the nature of the test, increasing the number of points and the number of episodes per point greatly increases the total number of episodes that need to be run – e.g. running it with 10 points along each axis with 10 trials each instead of five points on each axis and five tests per point, as was done for these results, would lead to a total of 10 x 10 x 10 x 10 episodes – 10,000 episodes – being required, which would take almost 17 hours. As the UR10 was run in a work environment where it required supervision, the times needed for these tests were limited. It would be worthwhile to pursue this in future research.

5.4 Line Tests

The line tests shown in Figure 7 demonstrate some level of generalisation ability for the ‘zero’ reset policy. The rewards received for each test, while slightly worse than those received for the static points tested in the grid test around the same areas (around 200 – 300), are still significantly better than the rewards received by the fairly untrained ‘random’ and ‘none’ reset policies for the static points in a similar area in the grid tests. The midpoint of the lines tested was at the centre of the bounded box, and therefore was quite close to the reset position, meaning that quite high rewards were likely to be obtained overall.

From the results presented, rewards are clearly better for the tests where the moving point had a higher velocity. There are two potential reasons for this – the policy is more comfortable making larger movements, and the duration of the episodes with higher velocity points were shorter and therefore closer to the duration of the training episodes.

As the training for the ‘zero’ reset policy was still quite short, it is likely that it developed a preference for controlling the joints of the robot that create the largest movements in cartesian space, as having some control of these would show the most immediate increase in rewards. This may mean the joints that control smaller links of the arm are still not particularly well trained, and overall the policy has a preference for large movements over smaller, more subtle ones. With this in mind, over the longer duration of the tests with the lower velocity point, it is possible that the policy overshoots the current location of the point more than when the point is moving faster. Further testing would be required to confirm whether this is the case.

The second potential reason – that the duration of the tests affected the accuracy of the policy for the task – could be more easily tested in future work. This would involve simply running the policy on the same task used to train it but varying the episode lengths. This would give a sense as to whether the policy learnt during training does generalise to shorter and longer episode lengths, and

whether this is the likely cause of the discrepancies in the rewards based on the velocity of the moving point.

The other point worth mentioning from these results is the slight variance in rewards due to the direction of movement of the point. Overall, the rewards obtained for the three velocities of the point on the XZ plane (i.e. with the point moving back and forth in the Y-direction) were slightly higher than for the other two planes. The Y-direction in these tests defines the linear reach of the arm from its base. It is likely that this bias is due to the UR10 more easily controlling its movement in this direction, as only two joints (the shoulder and elbow) are really required to move within this plane. The simpler SenseAct environment, which uses only 2 DOF, uses these two joints to move the end effector across the YZ plane, and the results of Mahmood *et al.* [14] confirm that this is clearly a simpler task than the 6 DOF version. This does point to the likelihood of a bias towards movements in this direction being simpler for the UR10. It is also worth noting, however, that though the moving point is travelling along the Y-direction, it is also translated in the X and Z directions from the base. Therefore, it is not possible for the UR10 to follow this line accurately using only the shoulder and elbow joints.

In relation to policy training times, there is some potential that a policy which is trained to a level closer to the optimal policy would overfit the training task to a greater extent and would generalise less effectively than the policy used in this work that was trained for a shorter number of episodes. This would also be worth investigating in future work by training multiple policies using a varying number of timesteps and comparing their results on these line tests.

5.5 Circle Tests

The results of the circle tests displayed in Figure 8 show a marked decrease in rewards as the radius of the circle grows. With the smallest radius tested, 0.1m, the 'zero' policy that was used for these tests performs fairly well, receiving rewards that are only slightly worse than those received for static points. This points towards some ability to generalise to this task. However, as the radius of the circles increases, the rewards drop significantly. There are two potential reasons for this decrease – the increase in linear velocity as the radius grows bigger (due to the constant angular velocity), and the distance of the moving point from the reset position of the arm in this task.

The linear velocities for each of the three radii are 0.05 m/s, 0.075 m/s, and 0.1 m/s respectively (0.5 rad/s * radius). Interestingly, the increase in linear velocity has the opposite effect to the velocities in the results discussed for the Line Test in section 5.4 above. This could be due to the difference between the line tests – where the point's velocity vector is always along the same line – and the circle tests, where the direction of the velocity vector is always changing. It is also strictly speaking a more complex task to generalise from a static point to a point that is moving in a two-dimensional space, as opposed to the line tests where the point is only moving in a one-dimensional space. As the line test has the point moving across the line and back, all points except the end points are covered twice, meaning that even if the end effector does not manage to keep up with the moving point, it is likely still to receive a moderate reward – a broken clock is still right twice a day. With the circle test, the only points in close proximity across the test are the beginning point of the circle, which the moving point reaches again as the test ends. This means the moving point is covering a much larger area of the bounded box; therefore, if the end effector lags behind the moving point or gets stuck in a specific area, the rewards it receives will decrease significantly. In future research, testing a circle of constant radius with a variety of angular velocities (and therefore a variety of linear velocities) would help to test how much of an effect the velocity of the point is having on the

rewards obtained, and how much of the effect can be contributed to the second potential issue – the distance of the moving point from the reset position.

The centre point of each of the circles is the centre of the bounded box; therefore, as the radius of the circles grow, the moving point necessarily moves nearer to the edges of the box. As has been shown with the training of the 'none' reset policy, as well as the reset tests using the 'none' reset method, the policies are not very good at moving the end effector away once it gets stuck on a boundary. This may not be a significant problem when the end effector is reset between tests and the static point it was aiming for is close to a boundary anyway. However, when the point is moving, this tendency to stick to the walls of the bounded box, combined with the point made in the preceding paragraph regarding the area of the bounded box that is covered by the moving point, could significantly decrease the rewards that the agent is likely to receive. This may explain the drop in rewards as the circle's radius increases. To test this further, policies trained with varying box sizes could be used to see if these still find circles with radiuses towards the edge of their bounded boxes problematic. Using a policy that has been successfully trained for a longer period of time and is less likely to overshoot the target and get stuck on the walls of the bounded box would also be interesting, though as with any policy trained to a high level of optimality on a specific task, the potential for overfitting does increase.

There are also marked differences in the rewards received based on which plane the circle followed by the moving point lies on. These are quite consistent across all three radii tested, with circles lying on the XZ plane receiving the greatest rewards, and circles lying on the YZ plane receiving the least rewards. The sizes of these planes that define cross-sections of the bounded box through its centre vary, as the box is not a cube. The XY plane is the smallest, measuring 0.4m x 0.5m. The XZ plane measures 0.4m x 0.6m, and the YZ plane measures 0.5m x 0.6m. Interestingly, this does not exactly correspond with the rewards received for the tests conducted on that plane. It might be expected that with the smallest plane, the end effector has the least room to overshoot the moving point before reaching the boundary of the box, and receives higher rewards because of this. Conversely, it could be argued that due to the boundary handling issues, the end effector would be more likely to get stuck on the boundaries for the smaller planes, and would receive lower rewards. Neither of these options seem to be dominant contributing factors though, as the worst performing plane (YZ), is actually the largest, and the best performing plane (XZ) is the medium-sized plane. Overall, this may imply that the end effector is more comfortable moving in the X-direction (with this being the common axis between the best and middle performing planes), and least comfortable moving in the Y-direction (with this being the common axis between the worst and middle performing planes). This does seem to contradict the results obtained from the line tests. However, as mentioned in the discussion surrounding them, the planes do not intersect the origin; therefore, the robot requires the use of multiple joints to manoeuvre across them. It is more likely that the complex kinematics of the arm are simply more comfortable moving near the XZ plane as it is defined for these tests, than near the YZ plane. As suggested for the grid tests, the UR10's level of 'comfort' with various points across the bounded box could be mapped in the future to ascertain whether this hypothesis has validity.

5.6 Other Areas for Further Research

There are many other tests that could be devised to assess the generalisation ability of the trained policies. These include testing points outside the bounded box used for training, testing moving points that move across all three dimensions of the bounded box, and testing moving points that do not follow a specific path, but instead change direction somewhat randomly. Of these, testing points

outside the bounded box in which the policy was trained may be the most interesting, as this could give some insight into the effects that the boundary conditions, and how they are handled, have on the learnt policy.

Another method of testing how well the policy generalises could be to rotate the position of the bounded box with respect to the base of the UR10. For example, would the policy still perform well if the box were rotated to be on the other side of the arm's base? This would reverse the directions that velocities would need to be applied to some of the joints to reach the target position – e.g. applying a velocity in a specific direction to the shoulder joint corresponds to lifting the arm on one side of the base, whereas applying the same velocity on the other side of the base would lower the arm. It is likely that the policy would not generalise well in this case, and a broader training task might be needed, such as using a bounded torus around the Z-axis of the UR10's base as the learning space.

Another potential improvement to the training tasks beyond those recommended in the other discussion sections would be to draw the target points randomly from a pool of target points that are evenly distributed across the bounded box, and to run a number of episodes of training equal to the number of target points. This would ensure that variations caused by the completely random nature of the target selection in the current task would be mitigated and may lead to the agent learning a more isomorphic policy across the space.

With the current task, the 6 DOF of the UR10 are not fully utilised, as the target point is only defined in cartesian space, and does not include any specifications for roll, pitch, and yaw. As such, the smaller links of the arm play a somewhat insignificant roll in the efficacy of the policy. In future research, it would be worth investigating the possibility of limiting the control to the three lower joints of the arm that provide full coverage of the cartesian space on their own. This might allow faster initial learning of policies that could then be extended to include the final three joints of the arm. Another related possibility would be to change the target definition to include rotational orientation as well, though this would make the learning task and the reward structure significantly more complex, and may impede the progress of the learning.

6 CONCLUSION

This dissertation investigated the potential for generalising RL policies learnt for a specific task on a real robotic system to other similar tasks. Three policies were trained on a simple reacher task using all 6 DOF of a UR10 robotic manipulator. This was implemented using the SenseAct framework created by Mahmood *et al.* for benchmarking reinforcement learning algorithms on real robots [13], [14], with various updates for the UR10 robotic manipulator that was used.

The results obtained from training three policies using the Trust Region Policy Optimisation algorithm demonstrate that the issue of training times for RL tasks on real robotic systems is still a significant barrier, even with a modern algorithm that has been shown to be effective at learning policies in a reasonable time frame for certain tasks. This was true specifically for the model trained using random reset positions for each episode, and for the model trained with no reset between episodes, both of which did not improve their rewards during training. On the other hand, for the simpler of the three tasks – the model trained with a fixed reset position – the algorithm was highly effective at improving the policy over the training time, and the learning showed no signs of slowing. If the policy were trained for longer on this task, it is likely that it would continue to improve.

A suite of tests was developed to investigate both the efficacy of the learning and the potential for generalisation. The results of the tests conducted show that there is some potential for generalisation to similar tasks if the agent did improve its rewards during training. The most successful of the three trained policies generally received rewards that were only marginally worse than its final performance on the training task when subjected to a task where the target point was moving instead of stationary, and when the reset mode between tests was different to the fixed position used during training.

Specific issues were encountered with the task setup involving the handling of boundary conditions, and this certainly impacted some of the results – especially those related to the no reset policy and tests. Potential solutions for this were discussed. These included the possibility of using sim-to-real learning transfer to reduce the need for such strict boundary conditions on the end effector position, as safety is always a concern when running learning tasks on a real robot rather than a simulation.

Overall, this work provides a basis for future research along various avenues. These include the investigation of how models trained for different lengths of time compare in terms of their generalisation, testing correlations between the joint configuration of the robotic manipulator used and the distribution of rewards obtained across the learning space, and testing a wider range of similar tasks to which the policy may be able to generalise.

7 REFERENCES

[1] W. D. Smart and L. P. Kaelbling, “Effective reinforcement learning for mobile robots,” in *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No.02CH37292)*, 2002, vol. 4, pp. 3404–3410.

[2] F. Stulp, J. Buchli, E. Theodorou, and S. Schaal, “Reinforcement learning of full-body humanoid motor skills,” in *2010 10th IEEE-RAS International Conference on Humanoid Robots*, 2010, pp. 405–410.

[3] A. Rajeswaran *et al.*, “Learning complex dexterous manipulation with deep reinforcement learning and demonstrations,” *arXiv Prepr. arXiv1709.10087*, 2017.

[4] F. Kirchner, “Q-learning of complex behaviors on a six-legged walking machine,” *Rob. Auton. Syst.*, vol. 25, no. 3–4, pp. 253–262, Nov. 1998.

[5] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, “The arcade learning environment: An evaluation platform for general agents,” *J. Artif. Intell. Res.*, vol. 47, pp. 253–279, 2013.

[6] G. Brockman *et al.*, “OpenAI Gym,” Jun. 2016.

[7] S. Koos, J.-B. Mouret, and S. Doncieux, “Crossing the reality gap in evolutionary robotics by promoting transferable controllers,” in *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, 2010, pp. 119–126.

[8] A. Boeing and T. Bräunl, “Leveraging multiple simulators for crossing the reality gap,” in *2012 12th International Conference on Control Automation Robotics & Vision (ICARCV)*, 2012, pp. 1113–1119.

[9] J. Kober, J. A. Bagnell, and J. Peters, “Reinforcement learning in robotics: A survey,” *Int. J. Rob. Res.*, vol. 32, no. 11, pp. 1238–1274, Aug. 2013.

[10] A. A. Rusu, M. Vecerik, T. Rothörl, N. Heess, R. Pascanu, and R. Hadsell, “Sim-to-Real Robot Learning from Pixels with Progressive Nets,” Oct. 2016.

[11] A. S. Polydoros and L. Nalpantidis, “Survey of Model-Based Reinforcement Learning: Applications on Robotics,” *J. Intell. Robot. Syst.*, vol. 86, no. 2, pp. 153–173, 2017.

[12] A. Yahya, A. Li, M. Kalakrishnan, Y. Chebotar, and S. Levine, “Collective robot reinforcement learning with distributed asynchronous guided policy search,” in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017, pp. 79–86.

[13] A. Rupam Mahmood, D. Korenkevych, B. J. Komer, and J. Bergstra, “Setting up a Reinforcement Learning Task with a Real-World Robot,” in *IEEE International Conference on Intelligent Robots and Systems*, 2018, pp. 4635–4640.

[14] A. R. Mahmood, D. Korenkevych, G. Vasan, W. Ma, and J. Bergstra, “Benchmarking reinforcement learning algorithms on real-world robots,” *Proc. 2nd Annu. Conf. Robot Learn.*, 2018.

[15] S. Levine, C. Finn, T. Darrell, and P. Abbeel, “End-to-end training of deep visuomotor policies,” *Journal of Machine Learning Research*, vol. 17. pp. 1–40, 2016.

[16] C. Zhao, O. Siguad, F. Stulp, and T. M. Hospedales, “Investigating Generalisation in Continuous Deep Reinforcement Learning,” *arXiv Prepr. arXiv1902.07015*, 2019.

[17] C. Zhang, O. Vinyals, R. Munos, and S. Bengio, “A study on overfitting in deep reinforcement

learning," *arXiv Prepr. arXiv1804.06893*, 2018.

[18] V. Mnih *et al.*, "Playing atari with deep reinforcement learning," *arXiv Prepr. arXiv1312.5602*, 2013.

[19] E. L. Thorndike, *Animal intelligence : experimental studies*. New York: The Macmillan Company, 1911.

[20] I. P. Pavlov, *Conditioned reflexes: an investigation of the physiological activity of the cerebral cortex*. Translated and edited by Anrep, GV (Oxford University Press, London, 1927), 1927.

[21] A. M. Turing, "Intelligent machinery." .

[22] T. Ross, "Machines That Think," *Scientific American*, New York, pp. 206–208, Apr-1933.

[23] R. Hoggett, "Early Maze Solving Machines." [Online]. Available: http://cyberneticzoo.com/early-maze-solvers/. [Accessed: 11-Oct-2019].

[24] R. J. Curran, "Maze with a mechanical memory," 3087732, 1963.

[25] J. M. Mendel and R. W. McLaren, "Reinforcement-Learning Control and Pattern Recognition Systems," in *Mathematics in Science and Engineering*, vol. 66, Elsevier, 1970, pp. 287–318.

[26] M. Waltz and K. Fu, "A heuristic approach to reinforcement learning control systems," *IEEE Trans. Automat. Contr.*, vol. 10, no. 4, pp. 390–398, 1965.

[27] M. Minsky, "Steps toward artificial intelligence," *Proc. IRE*, vol. 49, no. 1, pp. 8–30, 1961.

[28] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[29] A. G. Barto, R. S. Sutton, and P. S. Brouwer, "Associative search network: A reinforcement learning associative memory," *Biol. Cybern.*, vol. 40, no. 3, pp. 201–211, 1981.

[30] A. G. Barto and R. S. Sutton, "Landmark learning: An illustration of associative search," *Biol. Cybern.*, vol. 42, no. 1, pp. 1–8, 1981.

[31] A. G. Barto and P. Anandan, "Pattern-recognizing stochastic learning automata," *IEEE Trans. Syst. Man. Cybern.*, no. 3, pp. 360–375, 1985.

[32] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Mach. Learn.*, vol. 3, no. 1, pp. 9–44, 1988.

[33] R. S. Sutton, "Generalization in reinforcement learning: Successful examples using sparse coarse coding," in *Advances in neural information processing systems*, 1996, pp. 1038–1044.

[34] 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.

[35] J. Schulman, S. Levine, P. Moritz, M. Jordan, and P. Abbeel, "Trust region policy optimization," in *32nd International Conference on Machine Learning, ICML 2015*, 2015, vol. 3, pp. 1889–1897.

[36] R. H. Crites and A. G. Barto, "Improving elevator performance using reinforcement learning," in *Advances in neural information processing systems*, 1996, pp. 1017–1023.

[37] G. Piatetsky, "Exclusive: Interview with Rich Sutton, the Father of Reinforcement Learning," 2017. [Online]. Available: https://www.kdnuggets.com/2017/12/interview-rich-sutton-reinforcement-learning.html. [Accessed: 12-Oct-2019].

[38] C. Szepesvári, “Algorithms for reinforcement learning,” *Synth. Lect. Artif. Intell. Mach. Learn.*, vol. 4, no. 1, pp. 1–103, 2010.

[39] L. P. Kaelbling, M. L. Littman, and A. W. Moore, “Reinforcement learning: A survey,” *J. Artif. Intell. Res.*, vol. 4, pp. 237–285, 1996.

[40] J. Hu and M. P. Wellman, “Multiagent reinforcement learning: theoretical framework and an algorithm.,” in *ICML*, 1998, vol. 98, pp. 242–250.

[41] W. C. Dabney, “Adaptive step-sizes for reinforcement learning,” 2014.

[42] W. Dabney and A. G. Barto, “Adaptive step-size for online temporal difference learning,” in *Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[43] K. Young, B. Wang, and M. E. Taylor, “Metatrace Actor-Critic: Online Step-Size Tuning by Meta-gradient Descent for Reinforcement Learning Control,” 2019, pp. 4185–4191.

[44] M. Castronovo, F. Maes, R. Fonteneau, and D. Ernst, “Learning exploration/exploitation strategies for single trajectory reinforcement learning,” in *Proceedings of the 10th European Workshop on Reinforcement Learning (EWRL 2012)*, 2012, pp. 1–9.

[45] M. Tokic, “Adaptive ε-Greedy Exploration in Reinforcement Learning Based on Value Differences BT - KI 2010: Advances in Artificial Intelligence,” 2010, pp. 203–210.

[46] R. Fruit, M. Pirotta, A. Lazaric, and R. Ortner, “Efficient Bias-Span-Constrained Exploration-Exploitation in Reinforcement Learning,” Feb. 2018.

[47] C. J. C. H. Watkins, “Learning from delayed rewards,” 1989.

[48] J. Tan *et al.*, “Sim-to-Real: Learning Agile Locomotion For Quadruped Robots,” 2018.

[49] S. Gu, E. Holly, T. Lillicrap, and S. Levine, “Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates,” in *Proceedings - IEEE International Conference on Robotics and Automation*, 2017, pp. 3389–3396.

[50] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, “Benchmarking deep reinforcement learning for continuous control,” in *International Conference on Machine Learning*, 2016, pp. 1329–1338.

[51] T. P. Lillicrap *et al.*, “Continuous control with deep reinforcement learning,” *arXiv Prepr. arXiv1509.02971*, 2015.

[52] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” *arXiv Prepr. arXiv1707.06347*, 2017.

[53] T. Haarnoja, H. Tang, P. Abbeel, and S. Levine, “Reinforcement learning with deep energy-based policies,” in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, 2017, pp. 1352–1361.

[54] S. Gu, E. Holly, T. Lillicrap, and S. Levine, “Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates,” in *2017 IEEE international conference on robotics and automation (ICRA)*, 2017, pp. 3389–3396.

[55] T. Yasuda and K. Ohkura, “Collective Behavior Acquisition of Real Robotic Swarms Using Deep Reinforcement Learning,” in *2018 Second IEEE International Conference on Robotic Computing (IRC)*, 2018, pp. 179–180.

[56] L. C. Baird III and A. W. Moore, “Gradient descent for general reinforcement learning,” in *Advances in neural information processing systems*, 1999, pp. 968–974.

[57] D. Silver *et al.*, “Mastering chess and shogi by self-play with a general reinforcement learning algorithm,” *arXiv Prepr. arXiv1712.01815*, 2017.

[58] M. Riedmiller *et al.*, “Learning by playing - Solving sparse reward tasks from scratch,” in *35th International Conference on Machine Learning, ICML 2018*, 2018, vol. 10, pp. 6910–6919.

[59] H. R. Berenji, “Fuzzy Q-learning for generalization of reinforcement learning,” in *Proceedings of IEEE 5th International Fuzzy Systems*, 1996, vol. 3, pp. 2208–2214 vol.3.

[60] J. A. Boyan and A. W. Moore, “Generalization in reinforcement learning: Safely approximating the value function,” in *Advances in neural information processing systems*, 1995, pp. 369–376.

[61] S. Lawrence and C. L. Giles, “Overfitting and neural networks: conjugate gradient and backpropagation,” in *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, 2000, vol. 1, pp. 114–119.

[62] K. Cobbe, O. Klimov, C. Hesse, T. Kim, and J. Schulman, “Quantifying generalization in reinforcement learning,” *arXiv Prepr. arXiv1812.02341*, 2018.

[63] D. Silver *et al.*, “Mastering the game of go without human knowledge,” *Nature*, vol. 550, no. 7676, p. 354, 2017.

[64] J. Bruce, N. Sünderhauf, P. Mirowski, R. Hadsell, and M. Milford, “One-shot reinforcement learning for robot navigation with interactive replay,” *arXiv Prepr. arXiv1711.10137*, 2017.

[65] Universal Robots, “Technical Specifications UR10,” 2014. [Online]. Available: https://www.universal-robots.com/media/50880/ur10_bz.pdf. [Accessed: 14-Oct-2019].

[66] Universal Robots, “UR10 str,” 2017. [Online]. Available: https://www.skyfish.com/p/universal-robots/all-files/24593780. [Accessed: 14-Oct-2019].

[67] P. Dhariwal *et al.*, “OpenAI Baselines,” *GitHub repository*. GitHub, 2017.

[68] M. Abadi *et al.*, “TensorFlow: Large-scale machine learning on heterogeneous systems,” 2015.

[69] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable parallel programming with CUDA,” *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008.

[70] NVIDIA, “NVIDIA CUDA Installation Guide for Linux,” 2019. [Online]. Available: https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html. [Accessed: 30-Sep-2019].

[71] S. Chetlur *et al.*, “cuDNN: Efficient Primitives for Deep Learning,” Oct. 2014.

[72] NVIDIA, “NVIDIA cuDNN,” 2019. [Online]. Available: https://developer.nvidia.com/cudnn. [Accessed: 30-Sep-2019].

[73] Universal Robots, “Parameters for calculations of kinematics and dynamics,” 2019. [Online]. Available: https://www.universal-robots.com/how-tos-and-faqs/faq/ur-faq/parameters-for-calculations-of-kinematics-and-dynamics-45257/. [Accessed: 30-Sep-2019].

[74] Universal Robots, “Remote Control via TCP/IP,” 2019. [Online]. Available: https://www.universal-robots.com/how-tos-and-faqs/how-to/ur-how-tos/remote-control-via-tcpip-16496/. [Accessed: 27-Sep-2019].

[75] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Comput.*, vol. 9, no. 8,

pp. 1735–1780, Nov. 1997.

[76] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, p. 436, May 2015.

[77] L. Espeholt *et al.*, “IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures,” in *35th International Conference on Machine Learning, ICML 2018*, 2018, vol. 4, pp. 2263–2284.

[78] D. R. Hunter and K. Lange, “Quantile regression via an MM algorithm,” *J. Comput. Graph. Stat.*, vol. 9, no. 1, pp. 60–77, 2000.

[79] M. R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*, vol. 49, no. 1. NBS Washington, DC, 1952.

[80] J. Hui, “RL — Trust Region Policy Optimization (TRPO) Explained,” *Medium*, 2018. [Online]. Available: https://medium.com/@jonathan_hui/rl-trust-region-policy-optimization-trpo-explained-a6ee04eeeee9. [Accessed: 18-Oct-2019].

[81] D. Coleman, I. Sucan, S. Chitta, and N. Correll, “Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study,” Apr. 2014.