Reinforcement Learning Algorithmic Adaptation to Machine Hardware Faults

by

Sheila Ann Schoepp

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

On July 20, 1969, the Apollo 11 lunar module, with Astronauts Neil Armstrong and Buzz Aldrin aboard, landed on the moon. It was a great achievement in space exploration. Most people know of this mission's success; yet, there is an untold story about this mission that many people are not aware of, and that ultimately led to its success. In the moments prior to landing on the moon, the astronauts' attention was disrupted by loud alarms, notifying them of a problem with their on-board computer systems. The moment intensified as an error, unknown to the astronauts, was identified - error 1202. After relaying the error code to mission control, the cause of the error was determined; the computer was overloaded and memory was low. Thanks to the software created by a young NASA engineer, Margaret Hamilton, the software controlling the computer system began to handle the error, re-initializing and re-assigning tasks with the highest priority, while dropping those of low priority. Margaret Hamilton, a visionary of her time, had planned for unexpected situations, and in doing so, she had created software that was able to detect, identify, and recover from errors.

Creating systems that can detect, identify, and subsequently recover from their failures are three important areas of engineering and artificial intelligence research. Respectively, these three research areas are known as fault detection, fault diagnosis, and fault tolerance. This thesis examines one of these areas - fault tolerance. With added fault tolerance, a machine recovers from a fault through either pre-engineered or artificial intelligence learning techniques; we explore the latter, empirically evaluating the effectiveness of two reinforcement learning algorithms in enabling a machine to adapt to a hardware fault. In this work, our machines are simulated robots; the faults experienced include joint damage, effector damage, and sensor damage, all of which cause the robot to be either partially immobile or to behave in an unexpected, sometimes erratic, manner. Many of the faults examined would be considered terminal, if not for algorithmic adaptation. The two reinforcement learning algorithms that we investigate are Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC). We demonstrate that algorithmic adaptation to hardware faults does indeed occur and that, for one simulated robot task, it occurs very quickly (i.e. within hours). Our results establish that reinforcement learning algorithms are successful in adding algorithmic hardware fault tolerance to simulated machines, and that added algorithmic hardware fault tolerance (with reinforcement learning) has the potential to be applied to real-world machines. This is particularly true for special cases, where a repair cannot be performed immediately, and where it is more favourable to have a machine re-learn to perform its task in the presence of a fault, than it is to terminate or proceed with reduced task performance. Example use cases include space, where specialized experts are not immediately available to make machine repairs, and disaster zones, where machines (e.g. robots) are sent to areas that are dangerous or inaccessible to humans, making their immediate repair more challenging or impossible.

Imagination is more important than knowledge. For knowledge is limited, whereas imagination embraces the entire world, stimulating progress, giving birth to evolution.

– Albert Einstein, 1924.

Acknowledgements

I wish to express my deepest gratitude to my supervisor, Dr. Osmar Zaiane, for his guidance, support, and trust in my capabilities. He has encouraged me to push beyond my self-imposed limitations; in challenging times, he has graciously provided me (and many others) with his support; and he has assisted me in the realization of my goals. All of these wonderful things make him one of the most influential people in my life.

I would also like to thank my co-supervisor, Dr. Johannes Gunther, who always has always challenged me, pushing me to think for myself and to become a better researcher. I have truly appreciated his guidance in this journey.

I would like to thank the wonderful people from Mitsubishi Electric Co., whom I have worked closely with in this project. This includes Yuto Yamanaka, Shotaro Miwa, and Yoshihiro Mitsuka. I have greatly appreciated the creative ideas they have proposed to me and the passionate interest they have shown in this project. I also offer my deepest gratitude to Mitsubishi Electric Co., for funding this research, and for providing me with the opportunity to work with closely their researchers.

Finally, I would like to give thanks to Andy Patterson, who has guided me through many challenging moments in this project. His assistance and ideas have been invaluable to this work.

Contents

List of Tables

List of Figures

List of Acronyms

3D Three Dimensional

AI Artificial Intelligence

DH Denavit-Hartenberg

DOF Degree of Freedom

GAE Generalized Advantage Estimator

GPU Graphics Processing Unit

GrBAL Gradient-Based Adaptive Learner

LED Light-Emitting Diode

MAML Model-Agnostic Meta-Learning

MB Model-Based RL

MB+DE Model-Based RL with Dynamic Evaluation

MDP Markov Decision Process

MP Megapixel

MRE Magnetic Rotary Encoders

MSE Mean Squared Error

NN Neural Network

PPO Proximal Policy Optimization

RB Replay Buffer

ReBAL Recurrence-Based Adaptive Learner

ReLu Rectified Linear Unit

RL	Reinforcement Learning
ROM	Range of Motion
ROS	Robot Operating System
SAC	Soft Actor-Critic
SDK	Software Development Kit
TRPO	Trust Region Policy Optimization
UGV	Unmanned Ground Vehicle
XML	Extensible Markup Language

Chapter 1

Introduction

The rapid advancement of technology has led to widespread automation - a movement towards *smart*, autonomous technologies that remove the need for human intervention [11]. Industry, for example, is evolving from computerized, semi-automated systems (Industry 3.0) to fully-interconnected, fully-automated, real-time data-driven, intelligent systems (Industry 4.0) [3], [25], [36], [75].

For a moment, let us ponder on what such an advanced technological evolution might look like. Imagine, for example, a modernized assembly line composed of an intricate network of advanced machines, all cooperatively working together to produce and deliver products. Often, an assembly line has interdependent segments - the output of one segment of the assembly line is input into another segment of the assembly line. Now consider that one of the machines along the line experiences a failure, degrading its ability to perform its task. (In today's systems, such an event could force the entire assembly line to shut down until a repair is completed by an expert.) In our modernized, fully-automated system, a sophisticated, fully-connected network of sensors and devices detect and diagnose this failure, sharing the failure data with the other machines in the network. A supervisory artificial intelligence (AI) algorithm responds to the detected failure, temporarily adjusting the speed of each affected segment of the assembly line; meanwhile, the machine's AI algorithm processes the failure data, enabling the machine to adapt and modify the method of performing its task, while accounting for its failure. Once adapted, the supervisory AI control system is notified and each affected assembly line segment is returned to normal speed. The intricate network of machines is able to produce and deliver in perfect synchronicity once again.

In this high-tech vision, machines are self-aware, self-learning, and adaptable. An unanticipated perturbation in a machine, such as a component failure, is immediately identified using real-time data. The AI algorithm controlling the machine extrapolates the machine's fault from this data and identifies the consequential limitations of the machine. Using this knowledge, the AI algorithm controlling the faulty machine initiates adaptation, learning a new strategy for the machine to complete its tasks in the presence of a failure. Is such an AI controlled machine a fantastical notion? A potential possibility of the future? Or has technology sufficiently advanced to make these type of machines a realization today?

1.1 Statement of the Problem

Employing techniques to identify a machine failure, extrapolate the cause of the failure from machine data, and subsequently enable a machine to adapt are all objectives of current AI research [9], [12], [51], [81]. Each of these three objectives (i.e. identification, extrapolation, and adaptation) can be considered independently. Respectively, they are widely known as fault detection, fault diagnosis, and fault tolerance [30], [35]. Although all three of these research areas are essential to complete machine autonomy, their combined focus is broad. In this thesis, we focus on one of these three areas - fault tolerance. We investigate the ability of one AI paradigm, reinforcement learning (RL), to enable adaptation in a machine experiencing a hardware failure. In this work, our machines are robots. Robots, like all machines, are susceptible to failures [5]–[7], [68]. It has been shown that, in robots, the mean time between failures ranges from 6 to 20 hours in a field environment, and from 19 to 90 hours in an indoor environment [5]–[7]; thus, their low reliability makes them an excellent platform on which to implement and assess adaptation to hardware failure using AI techniques.

By definition, a *fault* is a technological imperfection that causes a failure [38]. In robots, faults can typically be traced to either their hardware, which consists of their physical components, or their software, which is the internal programming that controls the robot (i.e. algorithms) [22], [68]. A robot's *fault tolerance* is defined as its ability to mitigate a fault, thereby preventing it from being unable to perform its task [22], [77]. Most current methods to add fault tolerance to a robot require intervention at the design stage, adding fault tolerance through hardware redundancy [22], [76], [77]. One method of redundancy involves the duplication of the robot's physical components, such as its motors and sensors [22], [50], [76], [77]. Other methods of redundancy involve the addition of extra joints per DOF, extra DOF per manipulator arm, and extra manipulator arms per robot [50], [70], [79]. In all cases, when one component fails, it can be stopped and the other component(s) can take over, ensuring task completion.

Although redundancy is an effective method for adding fault tolerance, it is a costly solution. It not only requires additional parts, but also increased space and power; in addition, it may add volume and weight to the robot, and consequently, has the potential to alter the robot's dynamics [22], [77]. In addition, for many robots that are already designed and produced, adding fault tolerance through the use of redundancy is not an available option. So how can we add hardware fault tolerance to robots without redundancy?

Within computer systems, there are several approaches to creating fault tolerant systems. These include recovery and *algorithmic reconfiguration* [22], [76], [77]. In recovery, the system is put into a safe state when a fault is present. With algorithmic reconfiguration, an algorithm modifies the computer's software and hardware usage to account for the presence of a fault within one part of the system; in computers, this may involve slowing down processes so that the remaining functional components can handle the system load [22], [76], [77]. These two methods can not only be incorporated into computers at the design and development stage, but also after. Unfortunately, these methods of adding fault tolerance are rarely used in robots [22], [77]; this is mostly due to the fact that "work in robotics is still focusing on the robot function development rather than dependability" [22, p. 48]. This is where AI techniques are relevant; the field of AI, particularly RL, has many different algorithms for learning from interaction. A robot experiencing a hardware fault can interact with its environment and, using an RL algorithm, learn a new way to complete its assigned task. AI techniques may provide a solution to the problem of adding hardware fault tolerance to new robots, without the need for redundancy, and to existing robots, where adding redundancy is not an available option.

1.2 Need & Gaps

There are many uses of robots that could be improved if robots were adaptable to their failures, thereby necessitating the need for this work. Consider, for example, the use of robots in disaster zones. Currently, autonomous robots are being designed and tested for, as well as applied to, rescue operations [7], [19], [42], [43], [56]. Carlson and Murphy [7] found that in the 2001 World Trade Center rescue operation, the unmanned ground vehicles (UGVs) needed human assistance an average of 2.8 times per minute; the noted problems included serious failures, traction slippage, and camera occlusion. In addition, the average downtime of a robot after a failure ranged from 177 to 207 hours [7]. This is a serious shortcoming in disaster zone robot technology. Robots are typically used for these tasks to avoid the risk to human life. Often, these areas are dangerous, or even inaccessible, to humans. If a robot fails in these areas, either a human is forced to enter the dangerous zone to retrieve or repair the robot, at an increased risk to their life, or the robot is deemed irretrievable, resulting in both property loss and financial loss for those who employed in it. Robots that could adapt to their failures, rather than require human intervention, would have tremendous utility in these operations. When a failing robot is situated in a location that is hazardous to humans, added fault tolerance may enable it to regain functionality, and if opted, enable it to return to a human-safe zone for repair.

Another place in which autonomously adapting robots would be useful is in space. In space, some robots are placed on planets where they autonomously collect and analyze data, never to be retrieved again [78]. Some advanced space robots independently maintain space equipment and are even capable of performing self-maintenance [1], [10]; others provide assistance to the astronauts by performing routine tasks [1]. In space, the presence of an expert to perform maintenance on a faulty robot is much less probable than on earth, and in many cases, not at all feasible. If a space robot fails, and no expert is available to perform a repair, its utility in space is significantly decreased. For example, a robot that is sent to autonomously collect data, with no future intention of its retrieval, would no longer be capable of collecting data. A space robot designed to make repairs, would no longer be able to make the required repairs; this would force the astronauts to make the repairs themselves, and would potentially require dangerous spacewalks to reach the site of a fault. A space robot that assists the astronauts with daily tasks would be disabled, no longer able to perform its duties, thereby forcing the astronauts to take over the tedious and time consuming work the robot was designed to perform. Overall, the loss of a space robot's function in the presence of a fault adds the need for autonomously adapting robots. In space, while no expert is available and there is no immediate opportunity to return the robot to earth for complete repair, added fault tolerance may enable a robot to continue to perform its tasks (i.e. collecting data, making repairs, and assisting humans). In some cases, it may even extend the robots expected end-of-life.

Overall, we believe that the utility of fault tolerant robots in society is horizonless. There are many uses of robots (and machines) today that could benefit from added algorithmic, hardware fault tolerance. This work suggests using RL algorithms (i.e. AI techniques) for adding hardware fault tolerance to a robot. A robot, experiencing a hardware fault that affects its task performance, would be expected to find a new way to complete its task; this may mean that non-faulty hardware takes on a compensatory role to allow the robot to continue its activities. Finding a new way for a robot to achieve its task in the presence of a fault can be characterized as a *RL control problem.*

It is important to note that the applications of this work may be limited to circumstances in which a human expert is not readily available to perform a robot's repair. If a human expert is available, the robot could be returned to its full performance without the need for adaptation. In addition, we may also limit this work to circumstances where the result of having a robot adapt to a failure is safe and more beneficial than continual depressed task performance or task termination. When a robot experiences a fault, it can display unusual, sometimes erratic, behaviour. Adaptation through learning requires new experiences, and so the robot may behave in an undesirable manner until it starts to learn to account for its fault.

Recent research has informed on how AI techniques can incorporate algorithmic, hardware fault tolerance into robots; despite this recent research, significant gaps are still prevalent. The research that has been conducted has used either few simulated robots [57], a single real-robot [12], or a combination of these two [9]. This leaves the generalization of their techniques and results to other robots unanswered. In addition, most research has examined a single algorithm or algorithm class, leaving the fault adaptation ability of other AI algorithms unanswered. Cully et al. [12] stated that in their experiments, raw robot sensor data, such as position sensor readings, was not sufficient to enable adaptation to faults; rather, high level behaviours had to be extracted to enable adaptation. Additional research that excludes this behaviour extraction step and uses only the robot's raw sensory data would be highly beneficial, as it would generalize adaptation across tasks.

Our intention for this work is to add to the current knowledge base for building algorithmic, hardware fault tolerance into robots, and that this later be extended to different types of machines, including real-world machines. Here, we examine the ability of two state-of-the-art, model-free RL algorithms to enable adaptation to different failures using the robot's raw sensory data as input. We test these algorithms on two simulated robots: OpenAI Gym's Ant-v2 and the SoftBank Robotics NAO, simulated in Webots. To the best of our knowledge, PPO and SAC have not been previously examined in regards to their utility in building algorithmic hardware fault tolerance into robots.

1.3 Thesis Statement

The problem of added hardware fault tolerance in robotics is a research problem that has immense value if solved. This is especially true for situations in which human assistance is delayed or non-existent. RL is a framework that may provide a solution to the problem of developing robots that can autonomously adapt to hardware failure. Having a robot autonomously adapt to a failure can be described as the robot independently (without human input) finding a new way to perform its tasks when a limiting hardware fault is present.

This thesis evaluates two state-of-the-art, model-free RL algorithms in their ability to enable adaptation to failure: Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC). Prior research has been performed on detecting malfunction in machines [29], [51] and diagnosing this malfunction [21], [30], [35], [81]; so, we assume that a failing robot is able to detect the presence of a fault and that the robot has some contextual knowledge about the fault and its specific limitations. This knowledge may alter the allowable actions of that robot.

We will answer the following questions:

- Can PPO and SAC enable a robot to adapt to a hardware fault (i.e. add fault tolerance)?

- How does adaptation to a fault with PPO compare to that with SAC, and vice versa?
- How do the differences in the faults affect a robot's adaptation?

Both PPO and SAC use neural networks; SAC also employs a replay buffer. At the onset of fault, the robot's task is changed. Although the primary goal is the same (i.e. walk fast or reach a target), it must now learn to achieve its task with new limitations. When a fault is applied to the robot, we can consider different initializations of each algorithm's neural networks as a different learning problem. In one case, we can retain the parameters learned with the normal functioning robot; in another case, we can randomly re-initialize the parameters, thereby throwing away all knowledge of the normal functioning robot and learning the new task from scratch. In addition, in the case of SAC, the initialization of the replay buffer at the onset of a fault can be considered a different learning problem. At the onset of a fault, the replay buffer contains past experiences for a normal functioning robot. The contents of the replay buffer may be inapplicable to the new task, and have the potential to temporarily impede the learning of the new task. Thus, we formulate different algorithm initializations and answer the following questions:

- How does retention (or random re-initialization) of the learned neural network parameters (i.e. models) affect a robot's adaptation to a fault?
- In the case of SAC, how does retention (or disposal) of the replay buffer contents affect a robot's adaptation to a fault?

Answering these questions will provide more information on how to incorporate fault tolerance into robots.

In summary, this research study informs on the capability of two state-of-the-art model-free RL algorithms to add algorithmic, hardware fault tolerance to new and pre-existing robots. It also informs on how to best initialize the learning process with these algorithms, once a fault has been detected. We also provide insight into whether the initialization process should be different for different faults.

1.4 Manuscript Organization

The introductory chapter has outlined the motivation and need for this research study. The remaining chapters are organized as follows: Chapter 2 provides background information on RL and presents a RL task setup for a real-robot. We discuss common robot faults and elaborate on existing approaches to the problem of building hardware fault tolerance into robots using AI techniques. Chapter 2 also includes a description of the two simulated robots models used in experiments, and briefly discusses how these models are defined within the simulation software. In Chapter 3, the design of the experiments are described in detail. In Chapter 4, the results of the experiments are presented and discussed. Chapter 5 summarizes our findings and proposes future research directions. We have additionally included an appendix. In Appendix A, we present the specifications for the SoftBank Robotics NAO that were relevant to our learning problem. In Appendix B, we present our kinematic calculations for the NAO robot. In Appendix C, we show the results of our hyperparameter search for our task with the NAO robot. In Appendix D, we present various hurdles that we encountered throughout this project, describing how each difficulty contributed to the development of this work.

Chapter 2

Background

This chapter introduces the background information for the contributions of this thesis. First, the reinforcement learning paradigm is introduced. Additionally, two state-of-the-art reinforcement learning algorithms are introduced, highlighting their main features. Furthermore, relevant previous works are recounted, and the relation of each work to the the problem of adding hardware fault tolerance is disclosed. Finally, the two simulated robots are presented, and a description is provided on how each robot model is defined.

2.1 Reinforcement Learning

Reinforcement Learning [69] is a paradigm that can be used to teach a learner, or agent, a task through interaction with its world, or environment. RL uses numerical rewards to reinforce good behaviours or to discourage bad behaviours in an agent; these reinforcers aid the agent in learning a good method to achieve its task.

2.1.1 Markov Decision Process

A *Markov Decision Process* (MDP) is framework to model a problem in which an agent, or learner, interacts with its environment over a series of discrete time steps, $t = 1, 2, 3, ...$ An finite MDP consists of:

- a finite set of states, $\mathcal{S}$,
- a finite set of actions, $\mathcal{A}$,
- a finite set of rewards, $\mathcal{R}$, and
- the environment dynamics model, p.

At each time step t, the environment provides the agent with its current state, $S_t \in \mathcal{S}$. Using the information of the environment's current state, the agent selects an action, $A_t \in \mathcal{A}(s)$. The selected action is executed and the time step is incremented. The execution of the action results in the environment updating its state, $S_{t+1} \in \mathcal{S}$, and additionally computing a numerical *reward*, $R_{t+1} \in \mathcal{R}$. The agent is provided with the state and reward resulting from its action, and the agent-environment interaction cycle repeats.

The reward and state resulting from the execution of an action in a given state is determined by the *environment dynamics model.* The environment dynamics model, p, is defined as:

$$p(s', r|s, a) \doteq Pr\{S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a\}. \tag{2.1}$$

In reinforcement learning, an MDP is assumed to have the Markov property; that is, the resultant reward and state, after taking an action, has a probability distribution that depends only on the previous state and action, and not on the entire history of all previously visited states and all previously executed actions within these states. It is assumed that the current state representation encapsulates all the historical interaction information necessary for the agent to make future action choices.

The interaction between the agent and environment results in a sequence of states, actions, and rewards defined as a trajectory:

$$S_1, A_1, R_2, S_2, A_2, R_3, S_3, ... \tag{2.2}$$

For a *continuing* task, the trajectory continues indefinitely. An *episodic* task, in contrast, does not continue indefinitely. An episodic task divides into episodes, or sequences that terminate. In an episodic task, the trajectory is terminated if the agent reaches a termination state, $S_T \in \mathcal{S}$. In some tasks, episodes are given set time limits; for these tasks, an episode is also terminated when the number of time steps in the episode has reached the set limit. After termination of an episode, the environment state is reset to a start state and a new agent-environment interaction cycle may begin.

In a reinforcement learning problem, the agent is considered to be the learner and decision-maker. The environment, in contrast, is everything that is external to the agent and outside the realm of its control. For example, consider a robot that is learning a reaching task. This robot contains joint motors, linkages, and various sensors. If the task requires learning how to control the joint positions to be successful at the reaching task, and the agent is given direct control over the motors that control the joint positions, then the robot's motors are considered to be a part of the agent. Everything else inside the robot, such as the linkages and sensors, would then be considered to be a part of the robot's environment and outside the realm of its control.

2.1.2 Unified Discounted Return

A reinforcement learning agent learns how to act through goal-directed learning. The goal of the agent is to select actions that maximize the *expected return*. The return, denoted by G_t, is defined as the sum of rewards obtained across a trajectory. For a episodic task, in which the trajectory terminates at time T, this is:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T. \tag{2.3}$$

For a continuing task, the trajectory does not terminate; rather, it continues indefinitely. The sum of rewards obtained across such a trajectory is unbounded, with the potential to grow infinitely large. For this reason, *discounting* is introduced and acts to bound the return by discounting the value of future rewards. For a continuing task, the agent aims to select actions that maximize the *discounted return*, which is defined as:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{2.4}$$

where γ is the *discount rate* and $0 \leq \gamma \leq 1$.

The discount rate parameter can be introduced to episodic tasks as well. A unified notation for both episodic and continuing tasks is formulated by modifying the description of a terminal state for an episodic task. As previously mentioned, an episodic task terminates when it reaches a terminal state. To utilize discounting and obtain a unified notation, the terminal state is changed to a special *absorbing state*, from which all transitions lead to itself and obtain a reward of 0. The unified discounted return is written as:

$$G_t \doteq \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k, \tag{2.5}$$

where $\gamma = 1$ or $T = \infty$, but not both.

2.1.3 Value Functions and Policies

A *policy* π defines an agent's behaviour in a state. A *deterministic* policy maps each state s to a single action, and is represented as:

$$\pi(s) = a. \tag{2.6}$$

A *stochastic* policy, in contrast, defines a distribution over actions for a given state s; an agent's action in state s is selected from this distribution. A stochastic policy is represented as:

$$\pi(a|s) \doteq Pr\{A = a|S = s\}, \tag{2.7}$$

where

$$\sum_{a \in \mathcal{A}(s)} \pi(a|s) = 1. \tag{2.8}$$

The *value* of a state, denoted by $v_\pi(s)$, is a measure of how good it is for an agent to in state s. Formally, it is defined as the expected return, given that an agent starts in state s and follows the policy π thereafter. In formal terms, this is written as:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right], \forall s \in \mathcal{S}. \tag{2.9}$$

The value of a state can be formulated recursively as follows:

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s], \forall s \in \mathcal{S}. \tag{2.10}$$

The value of a terminal state is 0.

The *action value* of a state-action pair, denoted by $q_\pi(s, a)$, is a measure of how good it is for an agent to start in state s, take action a, and then follow the policy thereafter. Formally, this is written as:

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t | S_t = s, A_t = a]. \tag{2.11}$$

An action value can also be formulated recursively:

$$q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a] \tag{2.12}$$

The agent and environment interaction cycle creates a sequence of experiences, or state, action, reward, and next state tuples. Values and action values are estimated from these experiences; as the agent gains more experience, these estimates converge to their true values.

Values and action values are interrelated. These relationships are defined as:

$$v(s) = \sum_a \pi(a|s) q_\pi(s, a), \tag{2.13}$$

and

$$q(s, a) = r + \gamma \sum_{s', r} p(s', r | s, a) v(s') \tag{2.14}$$

One of the tasks in a reinforcement learning problem is to find the best policy, thereby maximizing the expected return for a given task. Through an iterative process, known as generalized policy iteration, the policy is gradually improved through policy evaluation and policy improvement. A policy that is greater than or equal to all other policies is an *optimal policy*, denoted by π_*.

2.1.4 Neural Networks

An neural network (NN) is composed of several layers. The first layer is called the *input layer*. The input layer is followed by one or more *hidden layers*. The final layer is the *output layer*.

Each layer is a neural network is comprised of one or more nodes. In a fully-connected neural network, each node in a layer is connected to every node in its neighbouring layer(s). Each of these connections has an assigned *weight*, which is responsible for controlling the contribution of each input node to the output value. When an observation, x, is passed into the input layer, the output of the input layer is computed as the matrix product of the observation and the weights, plus an added bias term. The hidden layer modifies this output by passing it through an activation function. Subsequent layers go through the same process; the weighted sum of the input is computed, passed through an activation function, and subsequently output to the following layer. Activation functions are usually applied to hidden layers; however, there are some cases in which the output layer also has an activation function applied to it.

In our target algorithm SAC, the activation function applied to hidden layers was the rectified linear unit (ReLU). Given an input, the ReLU activation function will output the input value, if it is positive; otherwise, it will output 0. In the hidden layers of PPO's neural network and the output layer of the SAC policy network, we used the hyperbolic tangent activation function, or tanh. Given an input of any value, the tanh function will compress this input into the range [-1, 1]. The mathematical formulation for the tanh activation function is:

$$tanh(z) = \frac{sinh(z)}{cosh(z)} = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}. \tag{2.15}$$

Neural networks are trained using an objective function. The objective function is a measure of the error in a neural network's prediction. For example, one commonly used objective function, used in both PPO and SAC, is the mean squared error (MSE), defined by the formula:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2, \tag{2.16}$$

where Y_i is the target value and $\hat{Y}_i$ is the predicted value.

When a neural network is being trained, the weights of the network are being modified and learnt. It is common to randomly initialize the weights before training. Then the training cycle begins. First, the weights are used to compute the neural network's prediction. The cost function takes this prediction and provides a measure of the error in the neural network's prediction. This error is then backpropogated through the neural network, and the weights are adjusted. This process is repeated until training is complete, and the weights are no longer being updated.

2.1.5 Policy Gradient Methods

Policy gradient methods model a policy without the use of a value function. The policy itself is a parameterized function:

$$\pi_\theta(a|s,\theta) = Pr\{A_t = a|S_t = s, \theta\}, \tag{2.17}$$

where $\theta \in \mathbb{R}^d$ is the policy's parameters.

The policy parameters are evaluated using a performance measure, denoted by $J(\theta)$. To maximize the performance, the gradient of this performance measure, with respect to θ, is used to update the weights through gradient ascent:

$$\theta_{t+1} = \theta_t + \alpha \widehat{\nabla J(\theta_t)} \tag{2.18}$$

Although value functions are not used to model the policy, they are often used to learn the policy's parameters. Methods that learn both the policy's parameters and a value function are known as *actor-critic methods*. The policy is the actor and the value function is the critic. The addition of the critic (or baseline) stabilizes training and improves the speed of learning.

Policy gradient methods are useful in continuous action spaces. Here, the policy network learns the parameters for a probability distribution, such as a Gaussian. When an action is to be chosen, it is selected from this parameterized distribution.

2.1.6 Proximal Policy Optimization

Introduced by Schulman et al. [55], Proximal Policy Optimization (PPO) is "a new family of policy gradient methods for reinforcement learning, which alternate between sampling data through interactions with the environment, and optimizing a surrogate objective function using stochastic gradient ascent" [55, p. 1].

One of the challenges with policy gradient methods is the size of the policy update, which is controlled by the learning rate. When the learning rate is set too small, the policy is learnt very slowly. When the learning rate is set too large, there is increased variance; additionally, there may be a risk of moving to a policy that is poorly performing and where recovery is not possible. The key feature of PPO is that it restricts how much the policy can change at each update, thereby stabilizing training. This is done through the use of a clipping objective function, L^{CLIP}. This objective function is defined as:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t], \tag{2.19}$$

where

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}. \tag{2.20}$$

$r_t(\theta)$ is a probability ratio and $\hat{A}_t$ is an estimation of the advantage.

In $r_t(\theta)$, the numerator is the probability of selecting action a_t in state s_t under the new policy, parameterized by θ. The denominator is the probability of selecting action a_t in state s_t under the old policy, parameterized by θ_{old}. When $r_t(\theta) = 1$, the probability of selecting an action is unchanged in the new policy. In state s_t, when $0 < r_t(\theta) < 1$, the probability of selecting action a_t is less probable under the new policy, and when $r_t(\theta) > 1$, selecting action a_t is more probable under the new policy.

The advantage is represented as:

$$A(s, a) = Q(s, a) - V(s), \tag{2.21}$$

and can be expanded to

$$A(s, a) = \mathbb{E}_\pi[\sum_{k=t+1}^{T} \gamma^{k-t-1} R_k | S_t = s, A_t = a] - \mathbb{E}_\pi[\sum_{k=t+1}^{T} \gamma^{k-t-1} R_k | S_t = s]. \tag{2.22}$$

In the expansion of the advantage, the difference between $Q(s, a)$ and $V(s)$ is more visible. The difference between $Q(s, a)$ and $V(s)$ is that, for $Q(s, a)$, the first action taken from state s is action a, and for $V(s)$, the first action taken from state s is $\pi(a|s)$. Taking two different actions can result in a different sequence of rewards along a trajectory. Overall, the advantage is a measure of how good it is to take an action a, compared to the action that is defined by the policy π. Said another way, it measures whether it is beneficial to change the policy to action a. The advantage can be both positive or negative.

In the L^{CLIP} objective function (equation 2.22), the minimum is taken over two values. The first value is $r_t(\theta)\hat{A}_t$ and the second is $clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t$. If the estimated advantage, $\hat{A}_t$, is positive, then action a is better than the current policy's action. In this case, we would want to update the policy such that action a becomes more probable; so, $r_t(\theta)$ should be increased. However, the PPO objective function limits how much we can change the policy by clipping the value of $r_t(\theta)$; the maximum change in this case is $1+\epsilon$. If the estimated advantage is negative, then action a is worse than the current policy's action. Here, we would want to make this action less probable in the new policy. To do this, $r_t(\theta)$ should be decreased. Once again, the PPO objective function will limit how much $r_t(\theta)$ can be decreased by limiting its change to $1-\epsilon$. In all other cases, such as when the advantage is positive and $r_t(\theta)$ is less than 1, clipping does not apply.

When the policy network and the value network share the same parameters, the objective function is modified to:

$$L^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t[L_t^{CLIP}(\theta) - c1L_t^{VF}(\theta) + c2S[\pi_\theta](s_t)], \tag{2.23}$$

where c1 and c2 are coefficients, L_t^{VF} is the squared error loss $(v_\theta(s_t) - v_t^{targ})^2$, and S is an entropy bonus.

In our implementation of PPO, we use the $L^{CLIP+VF+S}$ objective function. Additionally, we use a generalized advantage estimator [41], where the

Algorithm PPO, Actor-Critic Style

```
for iteration=1,2,... do
    for actor=1,2,...,N do
        Run the policy $\pi_{\theta_{old}}$ in environment for $T$ time steps
        Compute advantage estimates $\hat{A}_1, ..., \hat{A}_T$
    end for
    Optimize surrogate $L$ wrt $\theta$, with $K$ epochs and minibatch size $M \leq NT$
    $\theta_{old} \leftarrow \theta$
end for
```

Figure 2.1: PPO algorithm. Figure adapted from [55].

advantage is computed as:

$$\hat{A}_t = -V(s_t) + r_t + \gamma r_{t+1} + ... + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V(s_T). \tag{2.24}$$

Here, T is a parameter that represents the time steps in a sampled trajectory. It is recommended that T be much smaller than the episode length [55].

2.1.7 Soft Actor-Critic

Haarnoja et al. [23], [24] introduced Soft Actor-Critic (SAC), an algorithm that is considered to be useful in real-world applications due to its low sample complexity and its robustness to different hyperparameter settings. Unlike PPO, which uses each experience only once for an update, SAC reuses past experiences by saving them in a replay buffer. For each network update, SAC draws a sample of the experiences from the replay buffer and uses these samples to perform an update. SAC is considered to be an *off-policy* algorithm; the samples that are retained in the replay buffer were collected under numerous different policies. (PPO, in contrast, is an *on-policy* algorithm; each experience used in an update was generated under the same policy.)

One unique feature of SAC is that it is said to be robust to model error and estimation error, due to the use of the maximum entropy framework in defining its objective [24]. This framework adds an entropy bonus to the standard RL objective, the expected return. The relative importance of the entropy is controlled by a parameter, α, known as the temperature. So, the optimal policy will maximize the sum of the expected return and the entropy bonus, given by:

$$\pi^* = \underset{\pi}{argmax} \sum_t \mathbb{E}_{(s_t,a_t)\sim\rho_\pi}[r(s_t,a_t) + \alpha\mathcal{H}(\pi(\cdot|s_t))]. \tag{2.25}$$

With this objective, if more than one policy leads to optimal performance, each policy will be assigned equal weight (i.e. multi-modal behaviour can be found); additionally, learning speed is increased [24].

Our implementation of SAC utilizes two twinned Q-networks, each of which is composed of two soft Q-networks. For one twinned Q-network, parameterized by θ, its comprising soft Q-networks are trained by minimizing the soft Bellman residual:

$$J_Q(\theta) = \mathbb{E}_{(s_t,a_t)\sim\mathcal{D}}\left[\frac{1}{2}(Q_\theta(s_t,a_t) - (r(s_t,a_t) + \gamma\mathbb{E}_{s_{t+1}\sim p}[V_{\tilde{\theta}}(s_{t+1})]))^2\right]. \tag{2.26}$$

The remaining twinned Q-network, parameterized by $\tilde{\theta}$, is an exponential moving average of the first twinned Q-network's weights. These weights are updated using Polyak averaging. The target network is used to stabilize training [24].

In addition to the twinned Q-networks, SAC utilizes a policy network. At each update, the policy is updated to the exponential of the soft Q-function, normalized by a partition function $Z^{\pi_{old}}(s_t)$. The new policy is restricted to a set of policies Π, which in our case, is the set of Gaussians. The new policy is defined by:

$$\pi_{new} = \underset{\pi' \in \Pi}{argmin} D_{KL}\left(\pi'(\cdot|s_t) \middle\| \frac{exp(\frac{1}{2}Q^{\pi_{old}}(s_t, \cdot))}{Z^{\pi_{old}}(s_t)}\right), \tag{2.27}$$

where D_{KL} is the Kullback-Leibler divergence.

The policy network, parameterized by ϕ, is trained using the objective function:

$$J_\pi(\phi) = \mathbb{E}_{s_t \sim \mathcal{D}}[\mathbb{E}_{a_t \sim \pi_\phi}[\alpha \log(\pi_\phi(a_t|s_t)) - Q_\theta(s_t, a_t)]], \tag{2.28}$$

where a_t is sampled using the re-parameterization trick:

$$a_t = f_\phi(\epsilon_s; s_t). \tag{2.29}$$

Here, ϵ is an input noise vector, sampled (in our case) from a Gaussian distribution. This changes the objective to:

$$J_\pi(\phi) = \mathbb{E}_{s_t \sim \mathcal{D}, \epsilon_t \sim \mathcal{N}}[\alpha \log(\pi_\phi(f_\phi(\epsilon_s; s_t)|s_t)) - Q_\theta(s_t, f_\phi(\epsilon_s; s_t))] \tag{2.30}$$

In the first published version of SAC [23], the temperature parameter was set to some constant value. According to Haarnoja et al., this was the only hyperparameter that needed tuning [23]. In the more recent version of SAC, Haarnoja et al. [24] removed the need to manually set the task-specific temperature parameter; rather they introduced a new method for automatically tuning the temperature. This new method restrained the entropy across all states to a predefined minimum value, thereby adding the flexibility to update the entropy for individual states based on the uncertainty of the policy in these states. For states that have been explored very little (or never), the entropy could be increased; meanwhile, for states that have already been well explored,

Algorithm Soft Actor-Critic

```
Input: $\theta_1$, $\theta_2$, $\phi$
    $\tilde{\theta}_1 \leftarrow \theta_1$, $\tilde{\theta}_2 \leftarrow \theta_2$
    $\mathcal{D} \leftarrow \emptyset$
    for each iteration do
        for each environment step do
            $a_t \sim \pi_\phi(a_t|s_t)$
            $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$
            $\mathcal{D} \leftarrow \mathcal{D} \cup (s_t, a_t, r_{t+1}, s_{t+1})$
        end for
        for each gradient step do
            $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$
            $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
            $\alpha \leftarrow \alpha - \lambda \hat{\nabla}_\alpha J(\alpha)$
            $\tilde{\theta}_i \leftarrow \tau\theta_i + (1 - \tau)\tilde{\theta}_i$ for $i \in \{1, 2\}$
        end for
    end for
Output: $\theta_1$, $\theta_2$, $\phi$
```

Figure 2.2: SAC algorithm. Figure adapted from [24].

the policy can behave more deterministically. The temperature parameter at each time step t was found by solving:

$$\alpha_t^* = \underset{\alpha_t}{argmin}\ \mathbb{E}_{a_t \sim \pi_t^*}[-\alpha_t \log \pi_t^*(a_t|s_t; \alpha_t) - \alpha_t \tilde{\mathcal{H}}]. \tag{2.31}$$

2.2 Related Work

2.2.1 Faults in the Real-World

Carlson and Murphy [7] conducted a series of studies on how unmanned ground vehicles (UGVs) fail in the real-world. Their collected data included faults experienced in one real disaster response (2001 World Trade Center disaster), faults experienced over two years in regular-use UGVs, and faults experienced in field tests of UGVs. Their studies included 24 robots from 7 different manufacturers, ranging in size from 8 lbs to 60,000 lbs. The majority of failures across all studies were found to be physical failures. The most common physical failure was effector failure (50%), followed by control system failure (33%), the power source failure (9%), and sensor failure (9%). Effector failure was found to be caused by exposure to the environment, allowing for the collection of dust and debris within the components.

Steinbauer [68] conducted a survey on the types of faults experienced in the RoboCup competitions. In these competitions, research robots are typically used to demonstrate algorithmic and hardware advancements in robotics. The 17 robots included in the study varied, including both custom-built robots (80%) and commercially available robots (20%). From the survey responses, it was determined that motor drivers and batteries are the components that experience failures most frequently. The number one cause for these faults was the connector; physical impact, wear and damage were other contributing factors. Sensors failures were found to be caused primarily by connector problems. Sensor failures led to no data or inaccurate data being read by the sensor. Finally, a robot's manipulator was found to be the most vulnerable robot component, and most important component, with 45% of faults being terminal. The primary causes of manipulator faults were physical impact and damage. Robots experiencing a fault were found to experience immobility, erratic behaviour, and reduced control capability.

Visinsky et al. [77] identified critical robot components and their failure modes. These components included sensors, joint motors, the power supply, and the control computer. This work listed various failure models for a sensor, including the failure mode *frozen*, which was defined as a sensor that failed and continuously returned its last known value.

Gao et al. [21] identifies three fault classifications: 1) actuator faults (e.g.

a blocked actuator), 2) sensor faults (e.g. a sensor reading a constant value), and 3) plant faults (e.g. the disconnection of a component).

For our work, we aimed to simulate real-world faults, drawing inspiration from these studies that examined the type and frequency of faults affecting real-world robots. From these studies, we selected a subset of commonly occurring faults that included blocked or damaged joints, effector damage, and sensor failure.

2.2.2 Meta-Reinforcement Learning

Clavera et al. [9] used model-based meta RL to enable a robot to quickly adapt to changes in the environment, such as changes in the terrain, broken or missing legs, and sensor miscalibration.

Clavera et al. [9] introduced a new algorithm that used two different types of learners - a gradient-based adaptive learner (GrBAL) and a recurrence-based adaptive learner (ReBAL). Their algorithm utilized model-agnostic meta-learning (MAML) [20] to learn an initial set of parameters for a generalized dynamics model. For GrBAL, the dynamics model was represented using a neural network and updates were performed using gradient descent; for ReBAL, the dynamics model was represented using a recurrent neural network and updates were performed using an update rule learned by the recurrent network.

To obtain the initial dynamics model using the MAML formulation, [9] had both simulated robots and a real robot learn a gait while being exposed to a variety of different environments; the resulting dynamics model was generalized and easily adaptable to new tasks. Clavera et al. [9] found that when their robots experienced new, never-seen environments, the learning agent's dynamics model adapted very quickly to the new environment. They showed that model adaptation did indeed occur, as the incidence of model error was less post-update compared to pre-update. They also showed that their methods, GrBAL and ReBAL, were more sample efficient than two traditional, model-free methods: Trust Region Policy Optimization (TRPO) and MAML-RL. [9] defined adaptation as reaching the asymptotic performance of the model-free methods. Their methods required approximately 1.5-3 hours of real-world experience to achieve adaption. The two model-free methods required approximately two days of real-world experience to achieve adaptation. In addition, for a small, predefined number of time steps equivalent to approximately 1.5-3 hours of real-world experience, their methods showed a higher return after a disturbance than the model-free TRPO, and two other model-based methods, Model-Based RL (MB) and Model-Based RL with Dynamic Evaluation (MB+DE).

The results obtained with the GrBAL and ReBAL algorithms were very favourable. Simulated experiments were conducted with OpenAI Gym's Ant and HalfCheetah. All real robot experiments were conducted with a single, six-legged robot - millirobot.

In the work by Clavera et al. [9], reinforcement learning was used to enable fast adaptation to faults (and changing environments). This is very similar to our work, with the exception of the learning method. The work by Clavera et al. [9] involved learning a generalized dynamics model for model-based RL. We do not use model-based RL; our two target algorithms, PPO and SAC, are model-free.

We have not included a comparison of the performance of [9] to our targeted algorithms, PPO and SAC, in this work. Clavera et al. [9] already compared their algorithm's performance to the performance of the model-free TRPO, showing that it performs far better (i.e. adaptation occurs significantly quicker). We expected similar results when compared to our two targeted model-free algorithms; that is, we expected that both GrBAL and ReBAL would enable quicker adaptation to faults than both PPO and SAC. Clavera et al. [9] showed extremely fast adaptation; in some cases, within time steps. Our target algorithms do not adapt this quickly.

2.2.3 Select-Test-Update

Cully et al. [12] proposed a trial and error approach to recovery from failure. In this approach, a robot simulator was used to automate the computation of a behaviour performance map for a selected robot. The behaviour performance map consisted of millions of different simulated robot behaviours (e.g. different gaits with varying amounts of time that each leg touched the ground). Each behaviour was associated with a predicted performance value and was assigned a high level of uncertainty since the behaviour was not yet experienced by the real robot. The behaviour performance map creation was considered to be a time consuming process; however, it only needed to be created once for any given robot.

Before taking an action, a real robot, with or without damage, would consult the behaviour performance map to select a behaviour that had the highest estimated performance. This behaviour would be executed in the real robot, resulting in an actual performance rating for the behaviour. The estimated performance of the behaviour and (nearby) similar behaviours within the behaviour performance map would be updated with the new performance value, and the uncertainty estimation of the specific behaviour-performance entry and nearby neighbours would be decreased. This trial-and-error search process (select-test-update) would be repeated until a high-performing behaviour was found. A high-performing behaviour was defined as "a behaviour whose measured performance is greater than 90% of the best performance predicted for any behaviour in the behaviour-performance map" [12, p. 2].

Cully et al. [12] found that by reducing the search space from a high-dimensional parameter space (e.g. all joint positions and velocities) to a low-dimensional behaviour space (e.g. gait patterns), the trial and error approach was very successful. Their robot, a hexapod, was able to perform new tasks better than their reference method, and adaptation to faults occurred on very fast timelines (e.g. within minutes). Although attempted, no solution was found when using the high-dimensional parameter space.

The work by Cully et al. [12] used AI techniques to adapt to faults through a select-test-update method. This is similar to our work, in that their algorithm adds hardware fault tolerance to a robot. However, their algorithm for adding fault tolerance is not a reinforcement learning algorithm. Additionally, Cully et al. [12] was unable to learn the adaptation task using the high-dimensional parameter space; in our work, we show that our two target algorithms can learn the adaptation task with the high-dimensional parameter space.

2.2.4 Environment Perturbations

Selfridge et al. [57] explored the ability of a reinforcement learning algorithm to adapt to disturbances in the classic pole-balancing problem.

The pole balancing environment contains a vertical pole attached to a cart on a 1 dimensional track. A force can be applied to either side of the cart, moving it along the 1 dimensional track. The problem is initiated with the pole in an upright (vertical) position and the cart centered along the 1 dimensional track. When the problem starts, gravitation forces are applied to the pole, causing it to fall to either the right or left. The learning agent's task is to keep the pole from falling by applying the appropriate force at either side of the cart at each time step. The task continues until either the pole falls to one side, the cart reaches the end of the track, or the problem time limit has been reached.

A simple evaluation feedback was used on each task - if the problem ended before 10,000 time steps, the task was considered to be a failure, and if the pole was balanced for a minimum of 10,000 time steps, it was considered to be a success.

Selfridge et al. [57] performed several experiments. In the first experiment, the pole balancing task was first learnt under normal conditions. After learning this task under normal conditions, they then applied disturbances to the problem and continued the learning process. The disturbances applied included increasing the weight of the pole, shortening the pole, adding a bias to the force that could be applied to each side of the cart, and adding a penalty when the pole was nearly vertical. In the second experiment, they did not learn the pole balancing task under normal conditions; the disturbances were immediately applied to the pole and this was considered to be the learning task.

Selfridge et al. [57] found that when the system was first trained on the pole balancing task under normal conditions, subsequent adaptation to the task with the mass and length disturbances would happen quickly, with fewer failures than having to learn the task with disturbances from scratch. When the system was exposed to a series of perturbations, such as a series of slight changes in the biases of the force, the learned algorithm parameters were found to be more generalized (thus, adaptable) to the continually changing task.

The work by Selfridge et al. [57] was a toy problem that added faults to the system, described as perturbations; their algorithm subsequently facilitated recovery, thereby showing it could add algorithmic fault tolerance. Although useful, their task was a low-dimensional toy problem with faults that may not necessarily be classified as real-world faults. Our tasks are higher-dimensional and, although simulated, are more geared towards real-world robots. OpenAI Gym's Ant-v2, although a simulated four legged ant, is similar in some ways to a hexapod robot. The SoftBank Robotics NAO is a real-world robot, but we chose to first apply our methods in simulation, before applying them to a real-world robot.

2.2.5 A Robot Reinforcement Learning Task

Mahmood et al. [37] set up a reinforcement learning task on a real-world robot, and subsequently evaluated how different components of the experimental setup affected task performance; these included components such as the medium of transmission, the action cycle time, and the action space.

In their experiments, Mahmood et al. [37] used a UR5 robot, which is an industrial robot with 6 joints. The task with this robot was a reaching task, utilizing either two or six actuated joints. For the task with six actuated joints, namely UR5 Reacher 6D, an imaginary three dimensional box was designated as the task space. This box had dimensions of 0.7 meters times 0.5 meters times 0.4 meters. At the start of an episode, the robot end effector was positioned in the middle of the imaginary box. A target, three-dimensional point was randomly selected from within the box boundaries. In each episode, the robot's end effector was tasked to reach the newly selected target point.

In this work, the observations, actions and rewards for the UR5 Reacher 6D task were derived from OpenAI Gym's Reacher problem, with slight modifications. An observation included the position and velocity of each actuated joint, the previous action, and the vector difference between the UR5 robot's end effector and the target. The reward function was defined to be the negative of the Euclidean distance between the end effector and the target.

The choice of an action was one of the experimental components examined. The two choices included velocity control, where an action was composed of target joint velocity commands, and smoothed position control, where an action included smoothed target joint position commands. Mahmood et al. [37] showed that velocity control resulted in better task performance than smoothed position control.

The work by Mahmood et al. [37] was used to formulate the RL problem setup for our NAO task. We used the same definition for states, actions and rewards; in addition, we chose to use velocity control because it was shown to have better task performance.

2.3 Robots

We will test our ideas on two simulated robots - OpenAI Gym's Ant-v2, simulated with the MuJoCo physics simulator, and the SoftBank Robotics NAO V5, simulated with Webots simulation software. This section will introduce these two robots, briefly describing how their models are built and how the model's physical characteristics are defined. We will also introduce the setup of the reinforcement learning problem for each robot.

2.3.1 OpenAI Gym Ant-v2 & MuJoCo

OpenAI Gym is an open-source collection of reinforcement learning environments. These environments are made accessible to the public so that new and existing reinforcement learning algorithms can be evaluated and compared [4]. MuJoCo is a paid subscription physics simulator that is required to simulate a sub-collection of the OpenAI Gym environments. MuJoCo was developed to provide fast physics simulation, thereby furthering research in robotics and

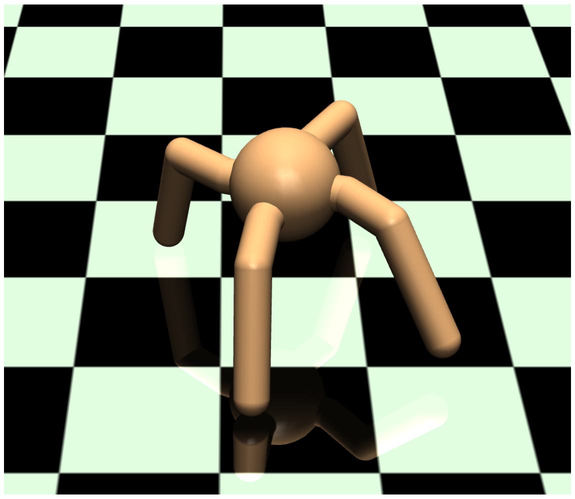


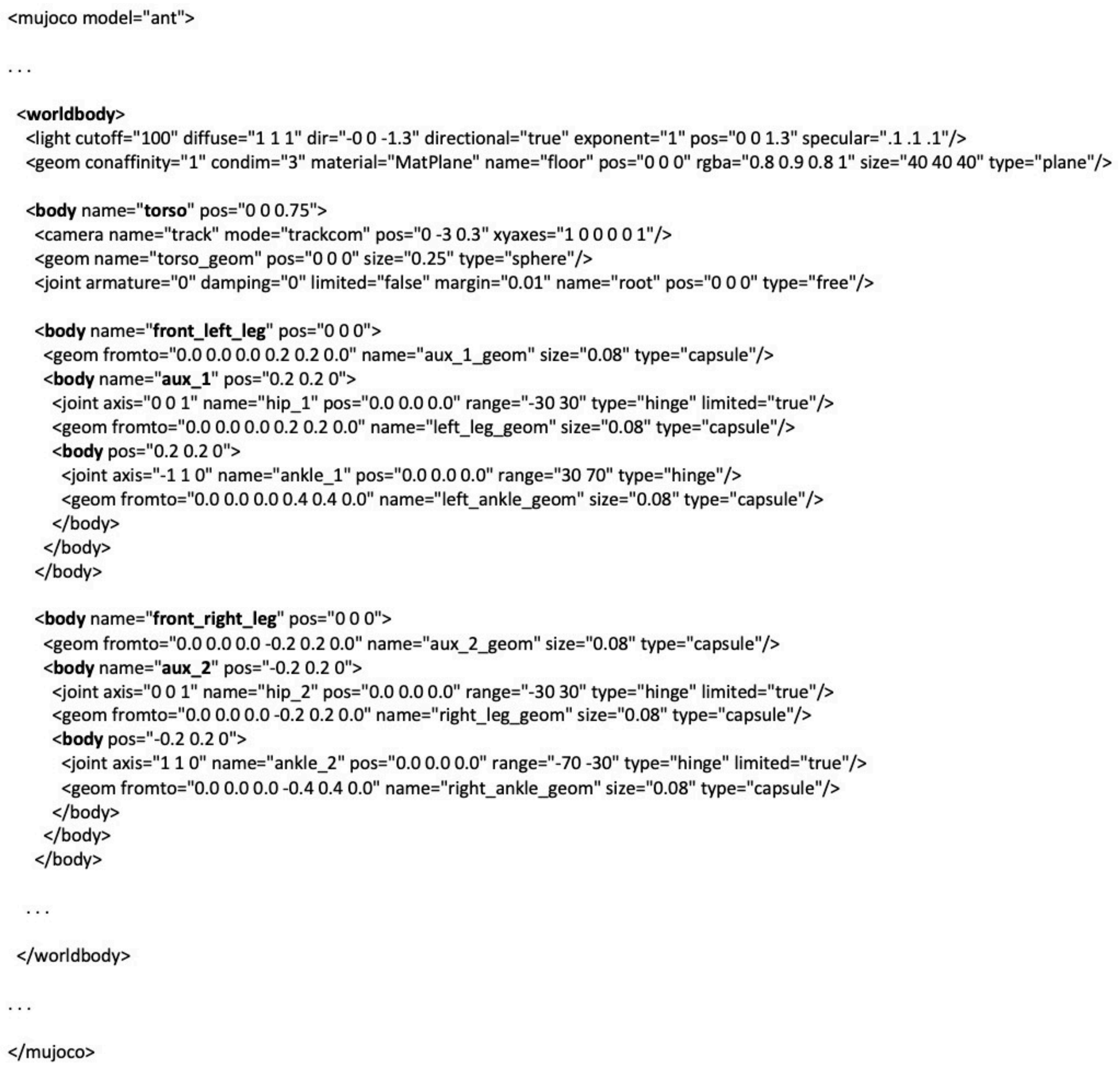
Figure 2.3: The ant model in the simulated OpenAI Gym Ant-v2 environment.

other areas [73]. Our target environment, Ant-v2, is an OpenAI Gym environment that requires MuJoCo physics simulation.

MuJoCo Ant-v2 Model

The OpenAI Gym Ant-v2 simulation [48] contains a four-legged ant model (shown in Figure 2.3). MuJoCo defines the ant model using an XML file in MJCF format. In this work, many of the malfunctions applied to the ant robot required manual modification of this file. Here we introduce important elements of the default OpenAI Gym Ant-v2 XML file; additional information can be found in [53], [54]. Figure 2.4 displays a segment of the default Ant-v2 XML file.

In an XML file, there are a hierarchy of parent-child *body* relationships. These parent-child relationships are expressed as a sequence of nested bodies. At the highest level (top-level parent) is the world body, which is considered to be the world frame. Nested within the world body is one or more children bodies. In the case of Ant-v2, there is a single child of the world body, assigned the name torso. Each of the world body children can also act as a parent and



```xml
<mujoco model="ant">

. . .

 <worldbody>
  <light cutoff="100" diffuse="1 1 1" dir="-0 0 -1.3" directional="true" exponent="1" pos="0 0 1.3" specular=".1 .1 .1"/>
  <geom conaffinity="1" condim="3" material="MatPlane" name="floor" pos="0 0 0" rgba="0.8 0.9 0.8 1" size="40 40 40" type="plane"/>

  <body name="torso" pos="0 0 0.75">
   <camera name="track" mode="trackcom" pos="0 -3 0.3" xyaxes="1 0 0 0 0 1"/>
   <geom name="torso_geom" pos="0 0 0" size="0.25" type="sphere"/>
   <joint armature="0" damping="0" limited="false" margin="0.01" name="root" pos="0 0 0" type="free"/>

   <body name="front_left_leg" pos="0 0 0">
    <geom fromto="0.0 0.0 0.0 0.2 0.2 0.0" name="aux_1_geom" size="0.08" type="capsule"/>
    <body name="aux_1" pos="0.2 0.2 0">
     <joint axis="0 0 1" name="hip_1" pos="0.0 0.0 0.0" range="-30 30" type="hinge" limited="true"/>
     <geom fromto="0.0 0.0 0.0 0.2 0.2 0.0" name="left_leg_geom" size="0.08" type="capsule"/>
     <body pos="0.2 0.2 0">
      <joint axis="-1 1 0" name="ankle_1" pos="0.0 0.0 0.0" range="30 70" type="hinge"/>
      <geom fromto="0.0 0.0 0.0 0.4 0.4 0.0" name="left_ankle_geom" size="0.08" type="capsule"/>
     </body>
    </body>
   </body>

   <body name="front_right_leg" pos="0 0 0">
    <geom fromto="0.0 0.0 0.0 -0.2 0.2 0.0" name="aux_2_geom" size="0.08" type="capsule"/>
    <body name="aux_2" pos="-0.2 0.2 0">
     <joint axis="0 0 1" name="hip_2" pos="0.0 0.0 0.0" range="-30 30" type="hinge" limited="true"/>
     <geom fromto="0.0 0.0 0.0 -0.2 0.2 0.0" name="right_leg_geom" size="0.08" type="capsule"/>
     <body pos="-0.2 0.2 0">
      <joint axis="1 1 0" name="ankle_2" pos="0.0 0.0 0.0" range="-70 -30" type="hinge" limited="true"/>
      <geom fromto="0.0 0.0 0.0 -0.4 0.4 0.0" name="right_ankle_geom" size="0.08" type="capsule"/>
     </body>
    </body>
   </body>

  . . .

 </worldbody>

. . .

</mujoco>
```

Figure 2.4: A segment of the default OpenAI Gym Ant-v2 XML file.

may have child bodies of their own. In Ant-v2, the torso body has four child bodies, two of which are shown in Figure 2.4 and are named front_left_leg and front_right_leg. These child bodies can also act as parent bodies; nested within the front_left_leg body are two children bodies, one named aux_1 and the other unnamed.

Each child of the world body defines a segment of the model. For example, the torso body defines the torso segment of the ant. The front_left_leg and front_right_leg bodies define additional segments (i.e. the front legs) of the ant.

Within all bodies, with the exception of the world body, there may be additional elements which define linkages and joints within the body, and each may contain special attributes. Linkages are defined by a *geom* element and joints are defined by a *joint* element. In Ant-v2, each geom element (or link) has attributes that include a name, a geometric shape (e.g. capsule), a size (e.g. radius of the capsule geometry), and a fromto attribute defining the link's length, position, and orientation. In Ant-v2, each joint element has attributes such as a name, a type (e.g. hinge), a position, an axis of rotation, and a range. All measurable attributes have metric units; range angles are in degrees.

Joint elements within a body are used to enable motion between the defined parent-child linkages by adding one or more degrees of freedom, dependant on the joint type. For example, a hinge joint adds one degree of freedom (DOF) of rotation between a parent link and a child link. If no joint element exists between a parent and child link, the two links are welded together. In the Ant-v2 XML, the top-level child leg bodies are welded to the parent torso body; in Figure 2.3, these welding are hidden as they are contained inside the torso. Within each leg body, are three nested bodies, each assigned a geom element, establishing that they are links; in addition, there are two joint elements, allowing for motion between the links. For example, in Figure 2.4, within the front_left_leg body, we see two joints assigned the names hip_1 and ankle_1. These are the ant's hip joint, with attachment right at the torso, and ankle joint, with attachment at the ant's knee. In the Ant-v2 model, all joints are hinge joints with a single DOF.

There are several optional, non-default attributes for both links and joints that can be added to the Ant-v2 XML elements manually by a user. For example, the limited attribute can be added to a joint element; this attribute interacts with the range attribute and, according to [54], is required to restrict the joint to the range set within the range attribute. If this attribute is not set, then the joint element's range attribute is ignored. For linkages, an optional attribute is the rgba attribute, which is used to assign a non-default colour to a link.

Modifications can be made to the XML file easily, while maintaining the stability of the simulation. Modifications include adding additional links and/or joints, and adding or altering geom element and joint element attributes. We made slight modifications to the default XML file to ensure that our added faults were functional and easily visible. For all joints within the XML file, we added the limited attribute and set it to true, ensuring that the joint limits were enabled. We also added the rgba attribute to some links to highlight their importance (e.g. a link affected by a fault). When added, these links were coloured red.

2.3.2 SoftBank Robotics NAO & Webots

The SoftBank Robotics NAO is a humanoid robot that was developed for robotics research. The NAO is fully programmable and autonomous [59]. The

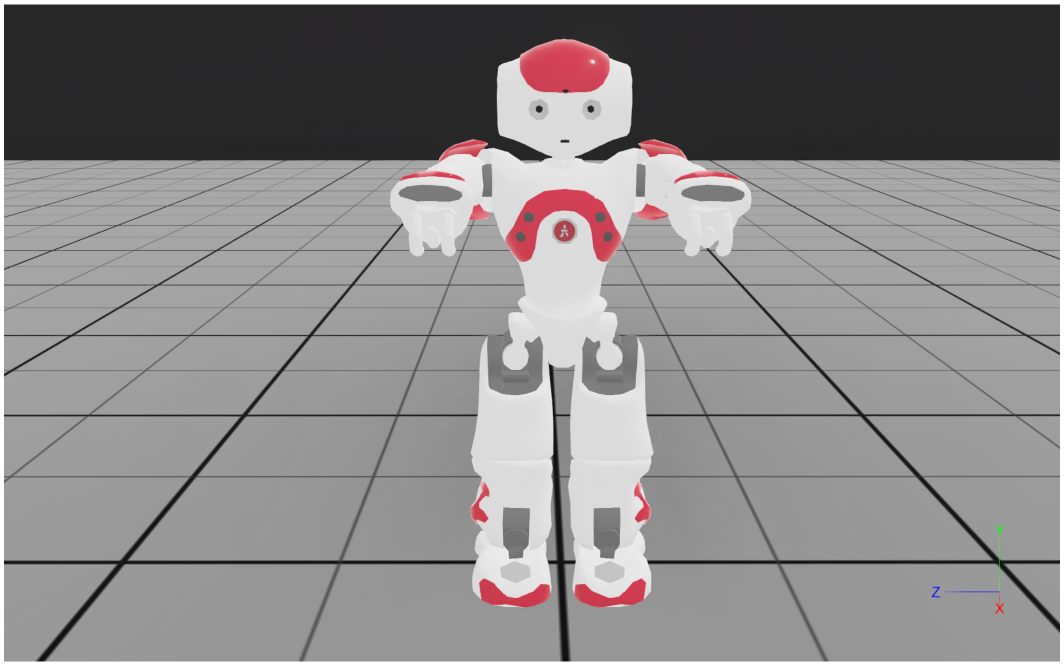


Figure 2.5: The NAO V5 in the Webots simulator. The Webots coordinate system is displayed in the lower right.

standard NAO V6 model has 25 degrees of freedom, including two degrees of freedom in the head, five degrees of freedom in each arm, one DOF in the hip, and five degrees of freedom in each leg. There is an additional DOF in each hand, allowing for the opening and closing of the fingers.

Webots is an open-source robot simulator that provides users the ability to develop, program and simulate robots [17], [40]; Webots simulation software has been used for projects in both academia and industry [13]. The current Webots distribution includes the NAO V5 robot and provides a well-documented API for controlling the simulated NAO robot.

Webots NAO Model

Webots defines the NAO V5 model using a PROTO file [16]. Figure 2.6 shows a segment of the default NAO PROTO file included in the Webots software. The first section of the PROTO file, enclosed in square brackets, contains modifiable fields for the NAO. These include fields such as the robot's translation and rotation within the Webots world frame; the robot's name, version and degrees of freedom; control levels, such as setting the robot to a supervisor mode, where it is capable of retrieving environment data not normally accessible to a real NAO; physics attributes, such as the enabling or disabling of self-collision; and sensor properties. The second part of the PROTO file, contained within curly brackets, is the PROTO body. This section contains the robot definition, which defines the robot's constituent parts, including sensors, linkages and joints, and describes the physical properties of these parts.

In a PROTO file, the highest-level definition, aside from the robot definition, is a *joint type* definition. A joint type definition has one or more child *devices*, dependent on the joint type. In Figure 2.6, definition (DEF) RShoulderPitch is shown to be a Hinge2Joint. A Hinge2Joint, shown in 2.7, has two degrees of freedom. Accordingly, DEF RShoulderPitch Hinge2Joint has two child devices, one named RShoulderPitch and the other named RShoulderRoll. Each child device is a joint and contains both a rotational motor and a position sensor. For each child device (joint) in the PROTO file, device attributes are specified - these include the maximum angular velocity, the minimum and maximum joint positions, and the maximum torque. If a device attribute is not specified, then this attribute takes on a default value of zero. Although not shown in Figure 2.6, a PROTO file also contains links and link physical attributes. A link's physical attributes include density, mass, centre of mass, and an inertia matrix. Within the PROTO file, all measurable attributes use metric units and angle attributes use radian units.

Simple attributes within the PROTO file, such as the aforementioned child device attributes, can be manually modified by a user and will be properly expressed in the NAO simulation. Modifications to complex link attributes

```
PROTO nao [
 field SFVec3f                        translation                      0 0.334 0
 field SFRotation                     rotation                         1 0 0 -1.5708
 field SFString                       name                             "nao"
 field SFString{"3.3","4.0", "5.0"}   version                          "5.0"
 field SFInt32{21, 25}                degreeOfFreedom                  25
 field SFColor                        color                            -1 -1 -1
 field SFString                       controller                       "nao_demo"
 field MFString                       controllerArgs                   []
 field SFString                       customData                       ""
 field SFBool                         supervisor                       FALSE
 field SFBool                         synchronization                  TRUE
 field SFBool                         selfCollision                    FALSE
 field SFFloat                        gpsAccuracy                      0.0
 field SFInt32                        cameraWidth                      160
 field SFInt32                        cameraHeight                     120
 field SFBool                         cameraAntiAliasing               FALSE
 field SFFloat                        cameraAmbientOcclusionRadius     0
 field SFFloat                        cameraBloomThreshold             -1
 field SFNode                         cameraFocus                      NULL
 . . .
]
{
 . . .
 Robot {
  . . .
   DEF RShoulderPitch Hinge2Joint {
    device [
     RotationalMotor {
      name "RShoulderPitch"
      maxVelocity 8.26797
      minPosition -2.08567
      maxPosition 2.08567
      maxTorque 4
     }
     PositionSensor {
      name "RShoulderPitchS"
     }
    ]
    device2 [
     RotationalMotor {
      name "RShoulderRoll"
      maxVelocity 7.19407
      minPosition -1.32645
      maxPosition 0.314159
      maxTorque 5
     }
     PositionSensor {
      name "RShoulderRollS"
     }
    ]
    jointParameters HingeJointParameters {
     axis 0 1 0
     anchor 0 -0.098 0.1
    }
    jointParameters2 JointParameters {
     axis 0 0 1
    }
    . . .
   }
  }
 }
}
```

Figure 2.6: A segment of the default PROTO file for the NAO.

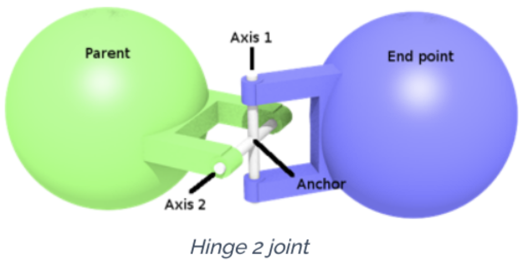


Figure 2.7: A Webots Hinge2Joint with two degrees of freedom. Figure from [15].

(e.g. mass) can cause instabilities in the simulation software.

There are some restrictions to the values that attributes can have within a PROTO file. For device attributes, all minimum values must be less than or equal to zero, and all maximum values must be greater than or equal to 0. This restriction is important for two reasons: 1) for a single NAO joint, LElbowRoll, with a small negative maximum joint angle, this restriction forces the maximum joint angle within the PROTO file to be set to 0 (and thus, deviates from the published value for the real NAO robot), and 2) for the application of malfunction to the robot's joints, this restriction limits the available malfunction choices and their severity.

The NAO PROTO in Webots has additional deviation from the real world NAO. The NAO PROTO file defines three separate motors to actuate the three joints within each finger. In the real NAO, these joints are actuated with a single motor.

Aside from the minor modification to the LElbowRoll joint's range (and matching modification to the RElbowRoll's joint range), and the number of motors controlling the NAO finger, all the devices, links, and attributes listed within the Webots NAO V5 PROTO file are equivalent to the SoftBank Robotics' published data for the NAO V6 in [64]. (Few changes have been made to the NAO structure across versions.)

2.3.3 NAO Kinematics

Kinematics is the study of motion in objects, without consideration of the forces and mass causing the motion [39]. Kinematics is a useful tool when working with robots. Robot kinematics can be classified into two related, but distinct subcategories: forward kinematics and inverse kinematics. Forward kinematics can inform on the position and orientation of a robot's end effector, given the positions of the robots joints, particularly those that contribute to the end effector's position and orientation. For example, one may want to know the three dimensional position and orientation of a robot's finger. To calculate this using forward kinematics, one would input the current positions of all the arm joints and finger joints into a forward kinematics solver. Inverse kinematics, in contrast, can take a target position and orientation for a robot's end effector in three dimensional space and attempt to find a solution for the required joint positions, such that the robot's end effector attains the target position and orientation. For the scope of this work, we are primarily focused on forward kinematics and we use the forward kinematics to obtain the position of the robot's end effector. We are not concerned with the end effector's orientation.

Kofinas et al. [34] presented a solution for the problem of forward and inverse kinematics for the NAO robot. The forward kinematics for both the right and left arms of the NAO robot are included in their work. The forward kinematics are computed to the NAO hand; the kinematics for the NAO fingers are not included. [34] originally presented the kinematics for the NAO V3; however, subsequent additions to their work have added the kinematics for the NAO V4 and are available in [44]. Although their most recent kinematics calculations are for the NAO V4 model, they remain valid for the NAO V6 model as there are minimal changes to the NAO's structure across these models.

Iberahim et al. [27] solved the forward kinematics of a virtual, four-joint finger. They presented their step-by-step process for computing the forward kinematics. These steps were utilized to extend the forward kinematics work done by [34], [44] and compute the kinematics for the NAO to the inner fingertip.

For this work, we extended the kinematic calculations done by Kofinas et al. [34] by computing the kinematics to the NAO inner fingertip. We have included these calculations in Appendix B.

Chapter 3

Experimental Setup

In this chapter, we introduce our experimental setup. We start by describing the hardware on which our experiments were performed and the software we used. We have chosen a subset of real-world faults to add to our robots. We present each selected fault and describe how we applied the fault to our simulated robots. We also introduce some new terminology, to be used throughout the remainder of this work. We define a normal robot to be a robot with no fault and a faulty robot as a robot with a fault. Finally, we describe important elements of the two RL algorithms we use in this work, including added algorithm features and the best performing hyperparameter settings.

3.1 Hardware and Software

Hardware. Our experiments were performed on Linux servers, running Ubuntu 20.04 and CentOS 7.8.2003. Each server was equipped with Tesla V100 GPUs.

Simulation software. For simulation of OpenAI Gym's Ant-v2, we used MuJoCo Pro version 1.50. For simulation of the SoftBank Robotics' NAO, we used the Webots R2020b revision 1 distribution. The Webots software requires a machine with a display to run a simulation. To use the software headless, on a remote server, we had to complete a few steps. Immediately after installation of the Webots software on the server, we used Xpra [80], an open-source remote display server and client, to forward the display to a local machine. This step was required to initialize the software by closing the in-built tutorial. (Webots could not be used headless without this step being completed.) After this, we no longer needed to forward the display to a local machine. To run our experiments, we used Xvfb [18], a display server, to run the software headless through the terminal interface.

Environments. We conducted two distinct sets of experiments. For each set of experiments, we created a unique virtual environment. These environments were created using Anaconda 3 for the x86_64 architecture, version 4.9.1. All virtual environments were setup with Python 3. The Python version and added libraries for each environment are shown in Tables 3.1 and 3.2.

Table 3.1: Python version & added libraries for Stable Baselines implementations.

Ant-v2 Virtual Environment	
Python	3.7.6
gym	0.16.0
stable-baselines	2.9.0
tensorflow-gpu	1.14.0

Algorithms. Stable Baselines [26] is a library that includes implementations of popular RL algorithms. In our preliminary OpenAI Gym Ant-v2 experiments, we used the Stable Baselines implementations of PPO and SAC. In our later experiments, for the NAO, we used our own implementation of these algorithms. Our implementations included additional features not included in

Table 3.2: Python version & added libraries for our implementations.

NAO Virtual Environment	
Python	3.7.9
matplotlib	3.3.2
numpy	1.19.4
pandas	1.1.4
termcolor	1.1.0
torch	1.7.0+cu110

the Stable Baselines library.

3.2 Faults

The faults that were applied to our robots were intended to closely mimic real-world faults. The faults that we applied fell into three categories: 1) blocked or damaged joints, 2) effector damage, where an effector is any device on a robot that interacts with its environment (e.g. a robot's arms, legs, and fingers) and 3) sensor failure. To demonstrate successful adaptation, we aimed to apply faults that would have a considerable effect on the robot's abilities; if not for adaptation, the robot would not be able to perform its task. Only a single fault was applied to a robot at any given time.

OpenAI Gym Ant-v2 & MuJoCo

In OpenAI Gym's Ant-v2, the ant is tasked to learn a gait that propels it forward as fast as possible. (Specific details are provided in Section 3.3.) For this reason, we apply our faults to the ant's legs. We decided to target a single leg on the ant. To choose the target leg, we opted to view the gait learned by the ant with our two algorithms, PPO and SAC. We observed that the learned gait primarily used two legs for forward motion, while the remaining two legs were used for stability. To maximize damage, and severely reduce the ant's ability to perform its task (i.e. propel forward), we chose to target one of the two legs primarily responsible for forward motion. Within the Ant-v2 XML file, our target leg is named right_back_leg. The target leg is shown in Figure 3.1.

Blocked or damaged joint. When foreign material enters a joint, it can cause the joint's range of motion (ROM) to become restricted. Alternatively, a joint can become damaged by some other means, and the normal ROM reduced. Our first and second faults in Ant-v2 were blocked or damaged joints. To simulate this type of fault, we reduced the ROM for a joint in the ant's target leg. Our first fault was applied to the hip_4 joint within the XML file. For this joint, we changed the range attribute from $[-30, 30]$ degrees to $[-5, 5]$ degrees. In a separate XML file, our second fault was applied to the ankle_4 joint. For this joint, we changed the range attribute from $[30, 70]$ degrees to $[65, 70]$ degrees. Figure 3.1 shows the two joints within the target leg.

Broken, severed effector. In robots, it is common for end effectors to become damaged. For this reason, the third fault we examined was a broken, severed end effector. To simulate this fault within Ant-v2, we reduced the length of the lower link within the target leg (i.e. the link in contact with the floor). In the Ant-v2 XML file, this link (or geom element) is named fourth_ankle_geom; we modified its fromto attribute, changing it from "0.0 0.0 0.0 0.4 -0.4 0.0" to "0.0 0.0 0.0 0.2 -0.2 0.0". Figure 3.2 shows the ant with a broken, severed ankle link.

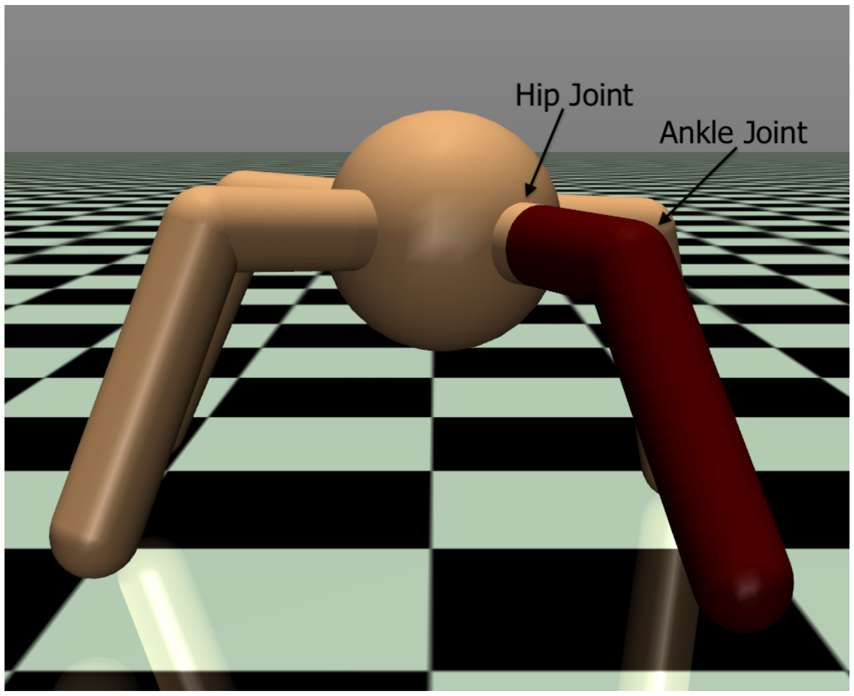


Figure 3.1: Hip and ankle joints within ant's target leg. Target leg is coloured red.

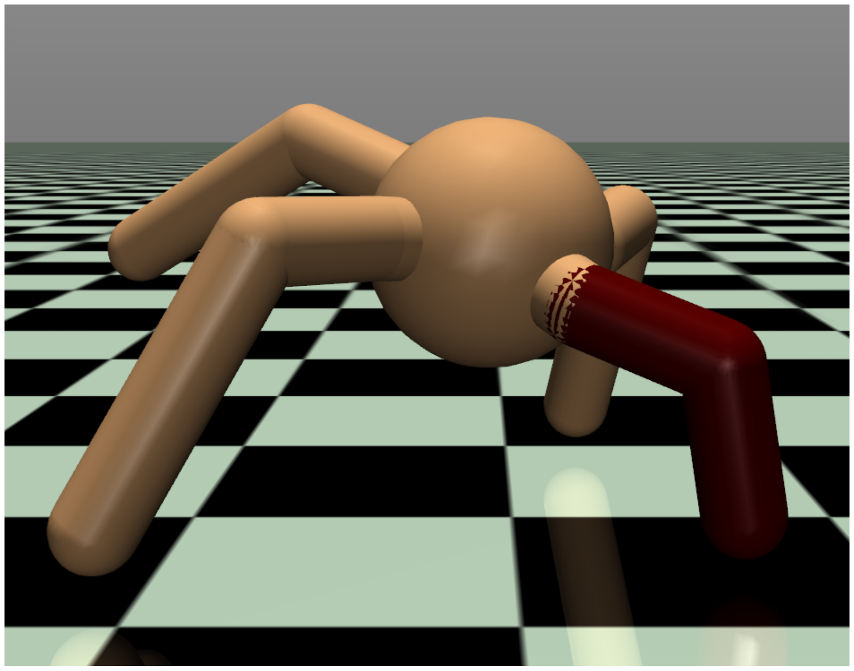


Figure 3.2: Ant with broken ankle in target leg. The target leg is coloured red.

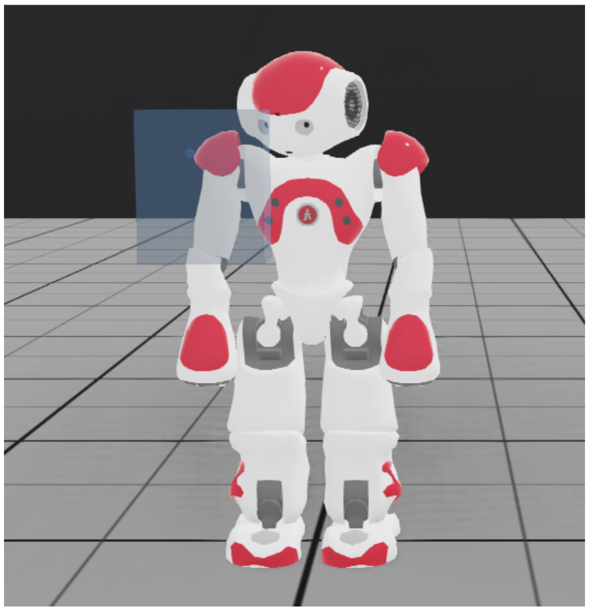
SoftBank Robotics NAO & Webots

In Webots, the NAO's task is an arm reaching task (shown in Figure 3.3). (Specific details on this task are provided in Section 3.3). This task actuates the NAO's arm joints and, in some cases, its finger joints. For this reason, we applied our faults to the NAO's actuated joints and their position sensors.

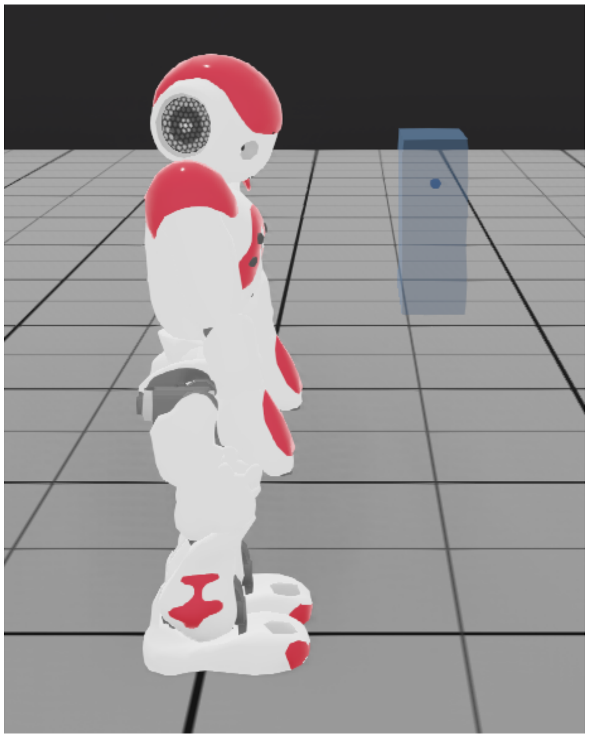
Blocked or damaged joint. A robot joint (or joint motor) can experience a terminal fault. A terminal fault would stop all motion within the joint, leaving the joint fixed in some position. (ROM is unchanged for all other arm joints.) There are many possible causes for this type of behaviour. For example, this could occur if the joint's motor driver is faulty. Our first NAO fault is a terminal fault in its ShoulderRoll joint. To simulate this type of fault in the NAO's right arm, we restricted all motion within the RShoulderRoll joint, changing its range within the NAO PROTO file from $[-1.32645, 0.314159]$ to $[-0.00, 0.00]$. This type of fault restricts all shoulder adduction and abduction. The restricted movements are shown in Figure 3.4.

Sensor failure. A robot's sensors are susceptible to damage. The most common cause is faults in the connectors. A sensor fault can cause a inaccurate reading (i.e. a random, biased or constant value), or no reading. Our second fault is a frozen sensor; the sensor reads a constant value - its last recorded position. We chose to have the ShoulderPitch sensor reading record a constant value of -2.0 radians. This type of fault did not require modification of the NAO's PROTO file. Rather, within our code, we overwrote the position sensor's reading with this constant value.

Broken effector. A robot's manipulator is the most sensitive part of the robot. According to [68], damage to a robot's manipulators results in terminal



(a) Front view.



(b) Side view.

Figure 3.3: NAO reaching task. An imaginary 3D box is in front of the NAO's body. At the start of each episode, a 3D point is randomly generated from within the imaginary box's boundaries. The NAO is tasked to touch the point with its fingertip.

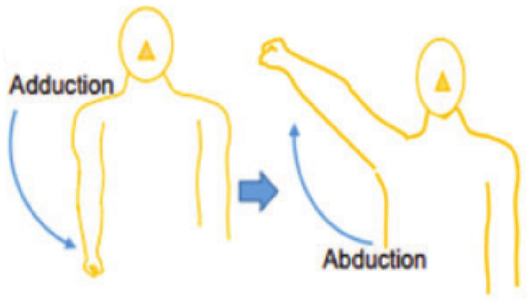


Figure 3.4: Shoulder abduction and adduction. Figure from [2].

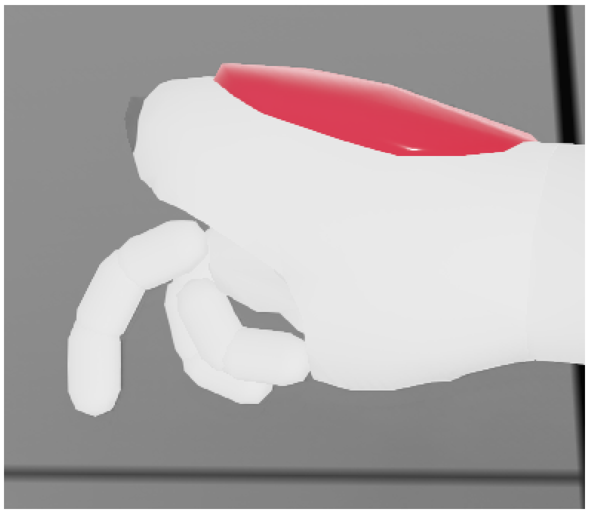
failure of a robot 45% of the time; the robot is no longer able to perform its task. Our third and final fault is damage to the NAO's finger joints, resulting in its finger (i.e. manipulator) being unable to extend normally. This is shown in Figure 3.5. Again, this fault did not require modification to the NAO PROTO. Rather, we altered the code by overwriting action commands for the three finger joints. In the NAO RL problem, actions are velocity commands. We set the target velocity and target position for the three finger joints to 0.0 meters/second and 0.0 radians, respectively. (Specific details on how we implemented velocity control in the NAO are provided in Section 3.6.)

3.3 Reinforcement Learning Problem

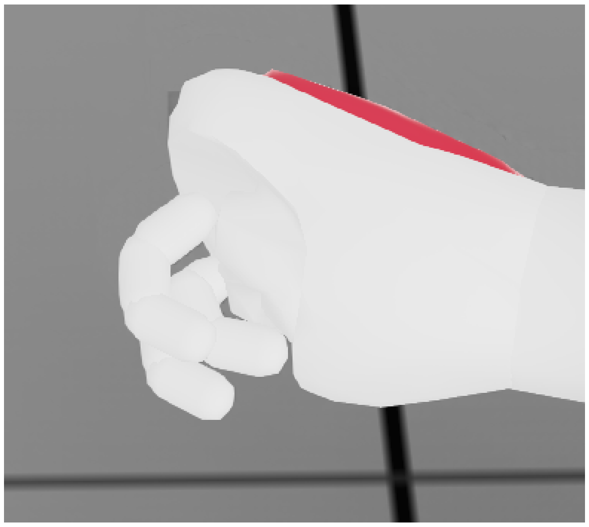
OpenAI Gym Ant-v2

The task to be learned in OpenAI Gym's Ant-v2 is to move the ant model forward as quickly as possible through the actuation of its joints, while minimizing a control cost and a contact cost. In the Ant-v2 model, there are a total of 8 controllable joints, including a hip joint and an ankle joint for each of the four legs.

The Ant-v2 task can be formulated as a finite MDP. In the Ant-v2 environment, observations, actions and rewards are predefined; however, they



(a) Normal finger. All three finger joints are free to move within their normal range.



(b) Faulty finger. All three finger joints are locked at 0.0 radians.

Figure 3.5: The NAO manipulator.

can be manually edited by a user. For this work, we mostly used the default observations, actions, and rewards, with one exception, where we modified the structure of an observation. We now present the default Ant-v2 setup; later we will outline the changes we made to an observation for one special case.

According to [47], a default Ant-v2 observation is expressed as an 111-element array, consisting of:

- the z position of the torso (height),
- the orientation of the torso (quaternion x, y, z, w),
- 8 joint angles,
- the x, y, z velocity of the torso,
- the x, y, z angular velocity of the torso,
- 8 joint velocities, and
- the 84 external forces (*cfrc_ext*) applied to the links - this includes the x, y, z forces and the x, y, z torques for each of the 14 links.

Within this array, the joint positions and joint velocities are ordered according to their vertical order within the complete XML, which can be found in [49]. Using names similar to those found in the XML file, this ordering is as follows: front left leg hip, front left leg ankle, front right leg hip, front right leg ankle, back leg hip, back leg ankle, right back leg hip and right back leg ankle.

A default action, a, in OpenAI Gym's Ant-v2 is an 8-element array containing a torque command for each of the 8 joints. The ordering of the joint torque commands contained within an action follow the same ordering as the vertical ordering of joints within the complete XML file.

The reward function for Ant-v2 is the sum of the following:

- forward reward $= v$, where v is the velocity of torso (i.e. the change is distance divided by the length of a time step),
- control cost $= -1 * 0.5 * \sum a^2$,
- contact cost $= -1 * 0.5 * 1e^{-3} * \sum clip((cfrc\_ext), -1, 1)^2$, and
- survive reward $= 1.0$

In Ant-v2, there are two primary episode termination conditions; if either occurs, an episode is terminated and the agent-environment interaction cycle can be reinitiated. The first termination condition is in the episode length. By default, episode lengths are limited to a maximum of 1000 time steps. The second termination condition restricts the height of the ant model's torso; when the centre of mass of the ant torso falls below 0.2 meters or rises above 1.0 meters, the episode is terminated.

NAO Reinforcement Learning Problem

The task to be learned with the NAO robot is an arm-reaching task. Using a single arm, the NAO is tasked to position its fingertip on a randomly generated target goal in 3D space. In each NAO arm, there are a total of 5 actuated arm joints that contribute to the position of its finger. Within the finger, there are a total of 3 joints. For this work, we mostly left the finger joints unactuated and set their position to 0.999 radians (or 57.29 degrees); this resulted in the finger being fully extended. For a single experiment, we actuated the finger joints, allowing them to take on any value within their normal range. We first present the RL problem setup for the experiments with unactuated finger joints; then we present the setup for the special case where we actuated the finger joints.

The NAO reaching task can be formulated as a finite MDP. We base our choice of the structure of states and actions, as well as the formulation of our reward function, on the work done by [37]. In addition, we choose velocity control, as [37] showed a higher average return with velocity control than position control.

For the NAO reaching task, we defined a state to be an 18-element array, composed of:

- 5 joint angles,
- 5 target joint velocities,
- the previous action (i.e. the previous step's 5 target joint velocities), and
- the vector difference between the fingertip's current 3D position and the target 3D position.

Within this array, the joint angles and joint velocities are ordered according to their their order in the arm kinematic chain: ShoulderPitch, ShoulderRoll, ElbowYaw, ElbowRoll, and WristYaw.

An action in the arm reaching task is a 5-element array, composed of a velocity command for each of the five arm joints. The ordering of the velocity commands within this array followed the ordering of the arm kinematic chain.

The reward function for the NAO reaching task is defined as: $R_t = -d_t$, where d_t is the Euclidean distance between the fingertip's current 3D position and the target 3D position.

There were two termination conditions in the NAO reaching task. The first termination condition was the maximum episode length, which we set to 1000 time steps. The second termination condition checked if the target position had been reached. If the Euclidean distance between the NAO inner fingertip's 3D position and the target 3D position was less than 0.001 meters (1 mm), we considered the target to have been reached, and the episode would be terminated. After episode termination, the environment would be reset and the agent-environment interaction cycle could be re-initiated.

Special Case. The third NAO malfunction of a broken effector, or damaged manipulator, required us to actuate the NAO's finger joints. In the normal NAO, these joints had a full range of motion and could take on any value within their normal range. In the faulty NAO, these joints no longer worked as expected, and were locked at 0.0 radians. The addition of the three actuated finger joints to our RL problem resulted in the action vector being assigned an additional three elements, changing its size from 5-elements to 8-elements. Our state vector was also affected by this change. Each joint is assigned two elements within the state array - one for its velocity and one for its position. The addition of three joints to the RL problem resulted in the addition of three velocity elements and three position elements to the state vector. In addition, because the action vector size had now increased by three elements, three additional elements were required in the state vector for the previous action component. Our new state vector was defined to an 27-element array, composed of:

- 8 joint angles,
- 8 target joint velocities,
- the previous action (i.e. the previous step's 8 target joint velocities), and
- the vector difference between the fingertip's current 3D position and the target 3D position.

The ordering of the joints angles and velocities within the state array, as well as the ordering of the target velocities within the action array, followed the ordering of the arm kinematic chain, with the addition of the three inner finger joints, starting at the knuckle and ending at the finger tip. The reward function was unchanged.

3.4 Algorithms

Added features. Our implementations of PPO and SAC included additional features not included in the Stable Baselines implementations. For SAC, we added the option to automatically tune the entropy [24]. For PPO, we added added options to linearly decay the learning rate, to clip the value function, and to use a generalized advantage estimator (GAE) [41]. In our experiments, we used some of these optional additional features, while others we did not use. For SAC, we opted to automatically tune the entropy. For PPO, we opted to decay the learning rate linearly and to use a generalized advantage estimator; however, we decided to not use the added feature of value function clipping as it was shown to lead to lower rewards [28].

Neural networks. Both PPO and SAC use neural networks for learning.

In our implementation of PPO, we used an actor critic network that was composed of two networks - a policy network and a value network. The policy network had a differentiable parameter - the policy distribution's log standard deviation. Bot the policy network and the value network were feed-forward networks, each consisting of an input layer, two hidden layers with 64 nodes each, and one output layer. We added two code-level optimizations which were shown to result in greater rewards with PPO; we used hyperbolic tan activations, and orthogonal initialization of the networks [28]. To select an action, we sampled from a multivariate normal distribution, using the policy network's output as the distribution mean, and a diagonal matrix, with the distribution variance along the diagonal, as the co-variance.

For SAC, we used two twinned Q-networks, each of which was composed of two Q-networks. All Q-networks were feed-forward networks with the same architecture; each had an input layer, two hidden layers with 256 nodes each, and one output layer. Actions were sampled from a Gaussian policy network, using the outputs of the policy network as the distribution mean and distribution standard deviation.

For both algorithms, we used the Adam optimizer [33].

Hyperparameters. The experiments with Stable Baselines were intended to be preliminary, giving us a rough idea of the performance of PPO and SAC in a faulty robot. For these experiments, we used the published hyperparameters for PPO and SAC in [55] and [23], respectively. For later experiments, we performed a hyperparameter search to find the best performing hyperparameters. We assumed that machines are primarily optimized for use under normal conditions, and not under faulty conditions. For this reason, a hyperparameter search was conducted for each algorithm using the normal robot only. We then used the same set of best-performing hyperparameters for experiments with our faulty robot. The selected hyperparameters are shown in Tables 3.3-3.4. Our hyperparameter plots are shown in Appendix C.

3.5 Experiments

Instead of having a single, continuous learning process, we structured some of our experiments as a two-part process to minimize the time to complete our experiments. By having each algorithm first learn with the normal robot,

Table 3.3: Hyperparameters used in Ant with Stable Baselines implementations of PPO and SAC. Parameters names match those used by the Stable Baselines library in [26].

PPO		**SAC**	
gamma	0.99	gamma	0.99
n steps	128	learning rate	0.0003
learning rate	0.00025	buffer size	1000000
ent coef	0.01	train freq	1
vf coef	0.5	batch size	256
max grad norm	0.5	tau	0.005
lam	0.95	target update interval	1
nminibatches	4	gradient steps	1
nooptepochs	4		
cliprange	0.2		

then saving the learned data, we were able to use this saved data to continue our training for many different faults (without having to repeat the learning with the normal robot more than once). We saved all data, such that the only difference between running the algorithm continuously versus running it as a two-part process, was the early termination of a single episode (i.e. the final episode in the first half of the learning process). The data that was saved and loaded is shown in Table 3.5. Webots, unfortunately, is unable to provide the current seed state, so we were unable to save and load this. For continuation of learning within a Webots experiment, we were forced to reset the seed state to the experiment seed.

Proximal Policy Optimization (PPO)

PPO experiment 1. The learning algorithm was applied to the task with the normal robot first, stopping the learning after a set number of time steps

Table 3.4: Best performing hyperparameters in NAO task for our implementation of PPO and SAC.

PPO		**SAC**	
learning rate	0.00025	gamma	0.96
linear lr decay	True	tau	0.005
gamma	0.94	alpha	0.2
number of samples	256	learning rate	0.00045
mini-batch size	32	hidden layers	2
epochs	4	hidden dim	256
epsilon	0.1	replay buffer size	1000000
vf loss coef (c1)	0.5	batch size	256
policy entropy coef (c2)	0.01	model updates per step	1
clipped value fn	False	target update interval	1
max grad norm	0.5	auto-tune entropy	True
use gae	True		
gae lambda	0.95		
hidden layers	2		
hidden dim	64		
log std	0.0		

Table 3.5: Data saved and loaded for two-part learning process.

Saved/Loaded Data
experiment parameters
evaluation and loss data
seed states: random, numpy, torch, and torch cuda
(Ant-v2 only) Gym env seed states: seed and action space seed
algorithm models, optimizers, and parameters (if any)
number of updates
(SAC only) log alpha, replay buffer contents
(PPO only) memory contents

(potentially terminating the final episode early). We applied a fault to the robot. If policy evaluation was performed, we first evaluated the policy in the task with the faulty robot. Then, we resumed training with the faulty robot, retaining all the algorithm's knowledge. This knowledge included all the neural network data (i.e. model weights and parameter values) and all the experiences stored in memory, if any. We stopped learning with the faulty robot after a set number of time steps.

PPO collects a small number of experiences under the current policy and saves these experiences in memory. Once the memory is full, the experiences contained within it are used to update the model (i.e. update the policy). After, they are discarded, and a new batch of experiences are collected under the updated policy. We chose to not perform an experiment where we cleared the memory for PPO before initiating the learning with a faulty robot. We believed that retaining the experiences in memory would have a very short-lived impact on the task performance immediately after the onset of a fault. Mnih et al. [41] proposed a policy gradient implementation in which a finite set of experiences are collected before each model update; the number of samples collected should be fixed at a value that is much less than the episode length. In our parameter search, described in Appendix C, when searching for the parameter number of samples (i.e. memory size), we found that the best memory size for the NAO task was 256. This is relatively small. At the start of learning with the faulty robot, if any old, non-representative experiences remained in memory (or, in the worst case, if the memory was full of old, non-representative samples), the samples would be used in only one network update to improve the model, then they would be discarded permanently. They would be entirely replaced by new samples that are representative of the changed task.

For example, in our NAO task, complete replacement would occur within a maximum of 256 time steps. So, these few retained old experiences in memory would not impact performance for a long period of time. Experiments could have be performed in which the memory was cleared at the onset of a fault, but we did not believe that the results would have been very different from those obtained when the memory was retained. We felt that this was a trivial choice.

PPO experiment 2. We immediately applied a fault to the robot. If policy evaluation was performed, we evaluated the randomly initialized policy in the task with the faulty robot. Then we ran the learning algorithm with the faulty robot, stopping learning after a set number of time steps. There was no prior learning with the normal robot.

Soft Actor-Critic (SAC)

One of the primary features of SAC is its use of a replay buffer. The replay buffer used in our experiments with SAC contained 1 million experiences. We initially chose this size because it was shown to do well in OpenAI Gym tasks [23]. Additionally, Haarnoja et al. [23] stated that the only SAC hyperparameter that needed tuning was the reward scale, which was responsible for controlling the policy's entropy. Ideally, if time and resources were unlimited, we would have tested this statement and tuned all our hyperparameters; however, we had to select a subset of parameters to tune. We were not certain of the impact that a large replay buffer would have on our problem, so we decided to put it to the test.

We believed that for some cases, retaining a large replay buffer may aid fault adaptation, particularly if the robot's dynamics were relatively unchanged after the application of a fault. If we retained the replay buffer in this case, its contents could act like a model for the changed dynamics.

For other cases, we believed that a large replay buffer might have hindered adaptation, particularly if a fault caused the robot's dynamics to change dramatically. In this case, at the onset of a fault, we would have a replay buffer containing 1 million experiences from the robot's old dynamics; many, if not all, experiences would no longer be representative of the changed task. At each time step, SAC would draw samples from the non-representative replay buffer and uses these samples to update the networks, which includes the policy network. In addition, only one entry of the replay buffer would be replaced with a new experience (generated from a biased policy). Overall, it would take 1 million time steps to completely replace the contents of the replay buffer. The effects of the prolonged network updates using the non-representative replay buffer contents could last longer than 1 million time steps. So, we believed that retaining the replay buffer could have have a detrimental effect on learning the task with the faulty robot (i.e. adapting) for some faults.

To examine the effects of the large SAC replay buffer on the problem of adding fault tolerance to a robot, we added added experiments 1 and 2.

SAC experiment 1. The learning algorithm was applied to the task with the normal robot first, stopping the learning after a set number of time steps. We then applied a fault to the robot. If policy evaluation was performed, we immediately evaluated the policy with the faulty robot. Then, we resumed the learning algorithm with the faulty robot, retaining all the algorithm's knowledge from its experience with the normal robot. This knowledge includes the algorithm's replay buffer contents and neural network data (i.e. model weights). We stopping learning with the faulty robot after a set number of time steps.

SAC experiment 2. We did the same procedure as in experiment 1 but instead of retaining all the algorithm's knowledge, we retained only some of its knowledge. We discarded the replay buffer contents but retained the neural network data.

SAC experiment 3. We applied a fault to the robot. If policy evaluation was performed, we evaluated a randomly initialized policy in the task with the faulty robot. We then ran the learning algorithm with the faulty robot, stopping the learning after a set number of time steps. There was no prior learning with the normal robot.

Time steps & policy evaluation. Each algorithm was run for a different number of time steps. Similarly, if policy evaluation was performed, each algorithm was evaluated at different intervals. These settings are shown in Table 3.6. For the NAO task, learning and evaluation could not be performed concurrently due to limitations within the Webots software (i.e. only one robot could be created per controller program and the seed state could not be saved or loaded). To rectify this, we saved the current model at each evaluation point during the learning process. After learning was complete, we began evaluating all the saved models. At the start of each evaluation, we set the Webots seed to the experiment seed, then we evaluated the deterministic policy for 10 episodes, calculating the average return across these episodes.

Table 3.6: Task learning time for each robot and policy evaluation frequency.

	Time Steps	**Eval Frequency (Time Steps)**
Stable Baselines implementations		
PPO	10000000	No evaluation
SAC	3000000	No evaluation
Our implementations for NAO		
PPO	6000000	10000
SAC	1500000	2000

3.6 Additional Considerations

Runs. Our experiments with Ant-v2 and Stable Baselines were intended to be preliminary. For this reason, we only performed 5 independent runs for these experiments. These experiments are included in this work as they were part of the process that led us to the decision to implement our own versions of the two algorithms. For all experiments with our own implementations, we performed 30 independent runs.

Robot in Webots. When we initially attempted the learning task with the NAO robot in Webots, the robot would rotate quite severely due to all movement being limited to a single arm. This rotation made the learning task very difficult, and potentially impossible. (We did not see improvement in the NAO's capability to perform the task). To resolve this, we added a physics plugin within the Webots software that attached the NAO frame to the world frame, thereby preventing any rotation.

Imaginary box. The dimensions of our imaginary box were 0.055 meters x 0.15 meters x 0.125 meters (or x=5.5 cm, y=15 cm, and z=12.5 cm). We refer to this box as imaginary because it was not defined as an object that

Table 3.7: Translation of imaginary box in Webots (meters).

arm	**x**	**y**	**z**
right	0.2125	0.43	0.0875
left	0.2125	0.43	-0.0875

could collide with the NAO and it was not assigned any physics properties. Although it is part of our simulation, it was only added visually to ensure that the task was being performed correctly. It is essential to know its positioning, however, because all target positions were randomly generated from within its boundaries. The coordinates for the centre of the imaginary box, for both the left and right arms, are shown in Table 3.7.

Our final box was significantly smaller than we originally intended. We found that with a larger box, many of the randomly generated points were unreachable by the NAO (with a rigid finger). Thus, we limited ourselves to a box in which all randomly generated points were reachable, which resulted in a box with highly restricted dimensions. Later, after actuating the finger joints, we found that the reachable area was increased and that the box dimensions could have been slightly expanded for this case. We did not test to see how much the box could be expanded, nor did we expand the box for this work, as our problem was already formulated and other experiments had been completed.

Velocity control. Although the Webots software does allow for velocity control, when velocity control is implemented as instructed in [14], restrictions on the joint angle ranges are no longer maintained; rather, joints can rotate freely with 360 degrees of motion. This behaviour was not acceptable for this work, so we had to find an alternative method for velocity control. We decided to use a modified version of position control, that we believe is equivalent to pure velocity control. To implement our control method, we had to make two changes. First, within our code, we changed the velocity range for an actuated joint from $[0, maxVelocity]$ to $[-maxVelocity, maxVelocity]$. Webots does not accept negative velocity commands, so the second change we made was to convert our positive/negative velocity to a (Webots acceptable) position and velocity command. If our target velocity was zero, we sent a command to Webots to maintain the joint's current position and to set the joint's (angular) velocity to 0.0 radians/second. If our target velocity was less than zero, we sent a command to Webots to move to the joint's minimum position and we set the joint's velocity to the absolute value of the target velocity. If our target velocity was greater than zero, we sent a command to Webots to move to the joint's maximum position and we set the joint's velocity to be the same as the target velocity.

Target positions and velocities. Commands sent to a robot are target positions or target velocities. For the NAO task, we observed that target positions and velocities were never reached; rather, the (noisy) sensors would read positions that were close to our target, but not exact. For this reason, we had to define target ranges rather than exact target positions or velocities. For example, if our target position was 0.2 radians for some joint, then a position sensor reading in the range $[0.2 - 0.00005, 0.2 + 0.00005]$ radians would be sufficient to indicate that our target position was reached. To simplify this expression, we identified the position sensor readings to have a *limit* of $1e^{-5}$, or 0.00005. Velocity measurements, which relied on position sensor readings, were assigned a similar limit of $1e^{-5}$ radians/second. A limit also had to be assigned for the computation checking to see whether the goal state had been reached; the goal was considered reached if the Euclidean distance between the NAO's inner finger and the goal's 3D position was less than or equal to $1e^{-3}$ meters (1 mm). The accuracy of each position sensor in the NAO is 0.1 degrees; for all 5 arm joints combined, this was equivalent to approximately 1 mm of accuracy. This justified our decision to use the limit of 1 mm for the goal computation.

Webots with Xvfb. In Section 3.1, we included a description of how we used the Webots software headless on a remote server using Xvfb [18]. When running Webots headless, we found that for some seeds, we would see an unusual, severe degradation of performance. Our code was entirely reproducible on our local machine (i.e. when we started experiments with the same seed, we see the same trajectory of states, actions and rewards throughout the entire experiment). However, when we would repeat experiments on the remote server for a seed which displayed this degraded performance, we would obtain different results; the repeated experiments would behave as expected, with no severe degradation. This behaviour was unexpected and unexplainable, and was not reproducible on our local machine. We believe it may be linked to one commonly observed error. Many remote server experiments signaled a non-terminal error, reported as "AL lib: (EE) ALCpulsePlayback_streamStateCallback: Received stream failure!". This error was present for most remote experiments and did not always lead to degraded performance. We can only hypothesize that this error may have led to some non-terminal problem occurring within Webots, which affected the learning process for some seeds. We observed this problem in experiments for a total of 23 seeds, drawn from all our experiments with the NAO in Webots; ultimately, we opted to repeat the experiments for these seeds, as they greatly influenced the learning curve. For this reason, we believe that for future work it would be advisable to not run Webots headless on a remote server with no display, if resources are available. For us, due to limitations in our available local hardware, it was absolutely necessary to run experiments remotely.

Chapter 4

Results and Discussion

In this chapter, we present our results for our targeted RL algorithms, SAC and PPO, examining each algorithm's ability to enable adaptation to faults in the Ant-v2 walking task and the NAO reaching task. For the first time, we show that adaptation to faults does indeed occur, and that performance is regained after a period of learning. We also discuss the effect of different *algorithm initializations* on task performance after the onset of a fault. After learning the task with the normal robot, each algorithm has knowledge of the task. Here, by algorithm initialization, we mean the knowledge that is retained at the start of the learning task with the faulty robot. The possible combinations of retained knowledge are: 1) (for SAC only) retaining the replay buffer contents and model(s) learned in the normal robot task, 2) retaining the model(s) learned in the normal robot task, and 3) discarding all learned knowledge from the normal robot task.

4.1 OpenAI Gym Ant-v2

4.1.1 Results

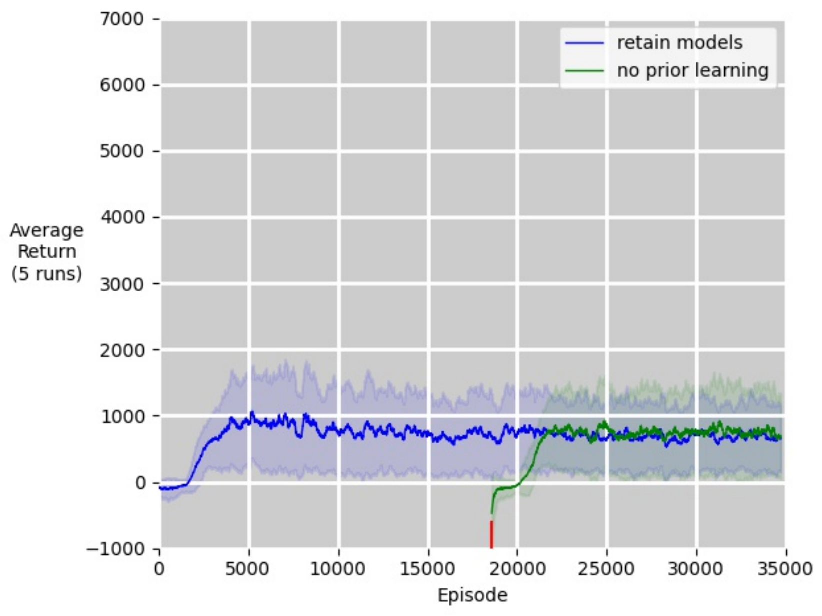
We now present our preliminary experiments with Ant-v2, in which we used the Stable Baselines [26] implementations of our two target algorithms, PPO and SAC. These experiments were intended to be preliminary only; however, they are included here because the results that we obtained with the Stable Baselines implementations led us to re-evaluate our choice to use these implementations, and consequently, implement our own versions of these two algorithms. These experiments were also responsible for us changing the format for future experiments. The preliminary Ant-v2 experiments did not include policy evaluation; rather we recorded the return for each episode. Our goal was to compare adaptation for each algorithm after learning for a fixed number of time steps; running learning for a fixed number of time steps often resulted in seeds with different episode lengths. For these experiments, we ran each experiment for a fixed number of time steps, however, we were forced to truncate our plots to the seed with the minimum episodes so that we could average our data across runs. These preliminary experiments were our first experiments that showed adaptation to faults with a reinforcement learning algorithm. Unfortunately, the number of runs was minimal (five) and as such, the 95% confidence intervals, computed with a t-distribution (ideal for small samples), are visibly very large. This prevents us from drawing definitive conclusions about the differences in performance across each algorithm initialization and each fault.

The results of the preliminary Ant-v2 experiments are shown in Figures 4.1-4.3. In these plots, we shifted the learning curves for PPO experiment 2 and SAC experiment 3 to the right. These experiments were performed with no prior learning in the normal robot task, so the onset of a fault occurred at episode 0. We considered this learning task to be identical to a learning task where, after learning with the normal robot, the replay buffer contents were discarded (for SAC only) and the model weights were randomly re-initialized. For comparison purposes, we have shifted this curve such that its origin is inline with the onset of a fault for the other experiments. For PPO, this was episode 18586. For SAC, this was episode 3359.

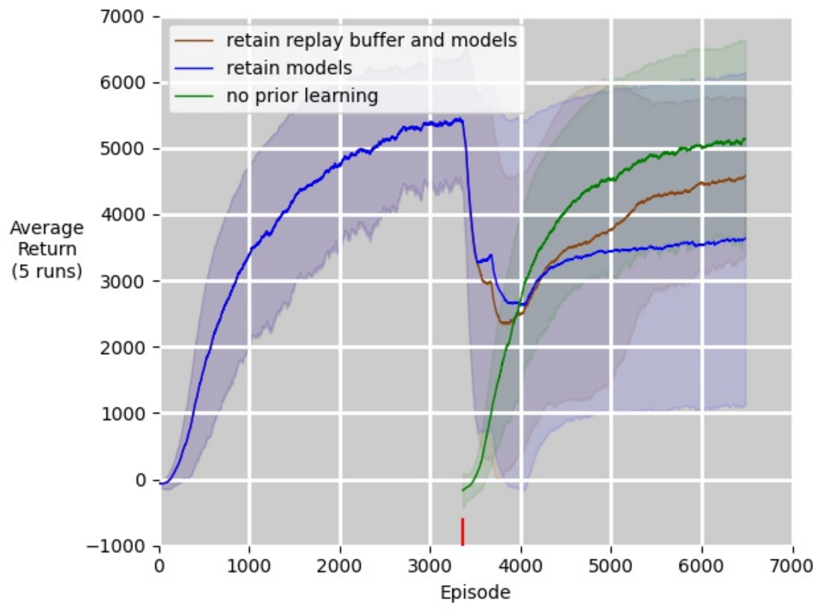
In each plot, the red marker indicates the onset of a fault. The shaded area represents a 95% confidence interval. Each algorithm had different initializations, which are indicated by the colour of the lines in each plot. Each PPO plot shows two different algorithm initializations: the retention or disposal of the learned models. Each SAC plot shows three different algorithm initializations: the retention of the learned models and the replay buffer contents, the retention of the learned models and disposal of the replay buffer contents, and the disposal of all learned knowledge (i.e. learning from scratch). For the discussion of our results, we will refer to *performance* on an algorithm often. Here, we define the performance to be the average return in an episode. A higher performance is equivalent to a higher average return.

4.1.2 Discussion

The results we obtained with the Ant-v2 task were surprising; this was particularly true for the observed contrast in performance between PPO and SAC. When learning the task with the normal ant, SAC did significantly better, achieving an average reward per episode of 5300. In contrast, PPO did not do well at the task with the normal robot. The average reward attained by PPO was, at its highest point, approximately 1000. When observing the learned gait for each algorithm (by rendering the OpenAI Gym task), we observed



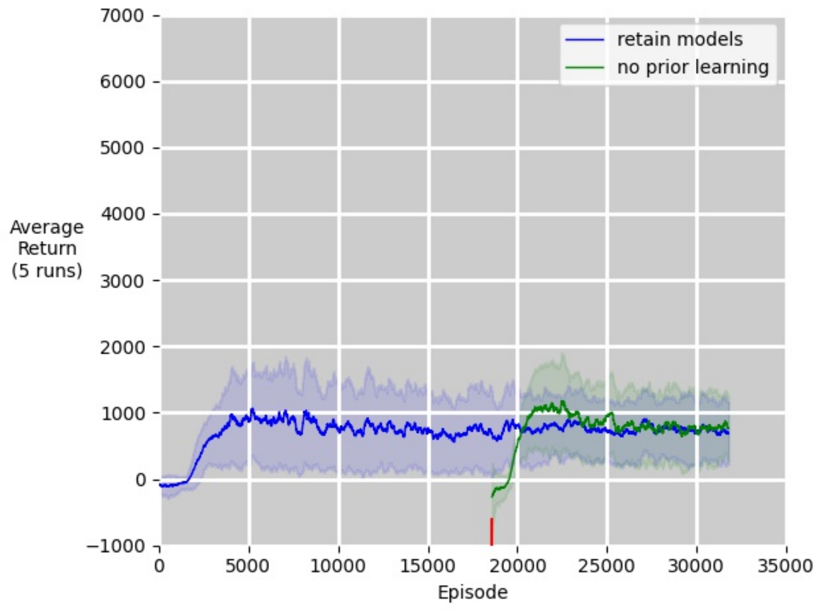
(a) PPO.



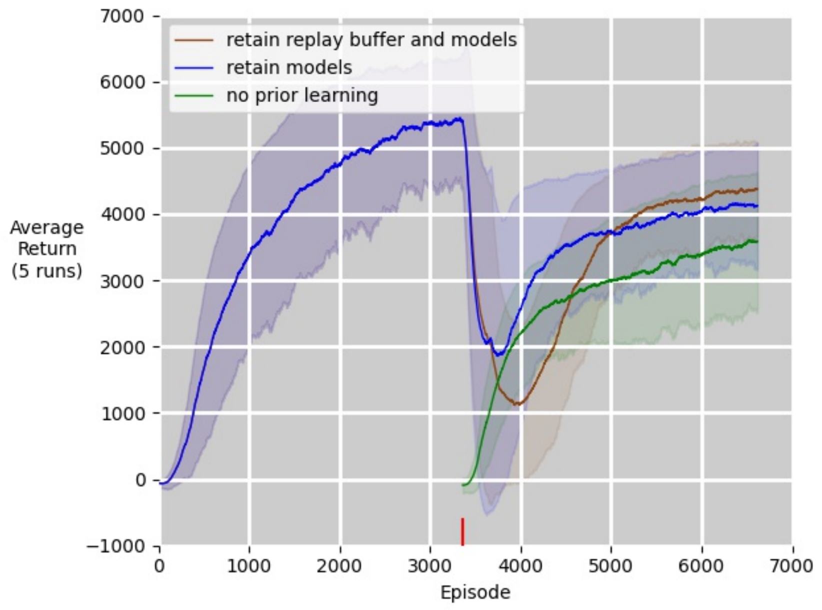
(b) SAC.

Figure 4.1: Hip range of motion decreased (from [-30, 30] to [-5, 5]). The red marker indicates the onset of a fault. Labels indicate the data retained after learning the task with the normal ant, if applicable. The shaded areas correspond to a 95% confidence interval.

that the SAC gait was highly specialized. Two of the four legs were primarily used for forward movement, while the remaining two legs were used to balance the ant. The gait learned by SAC propelled the ant forward very quickly. In



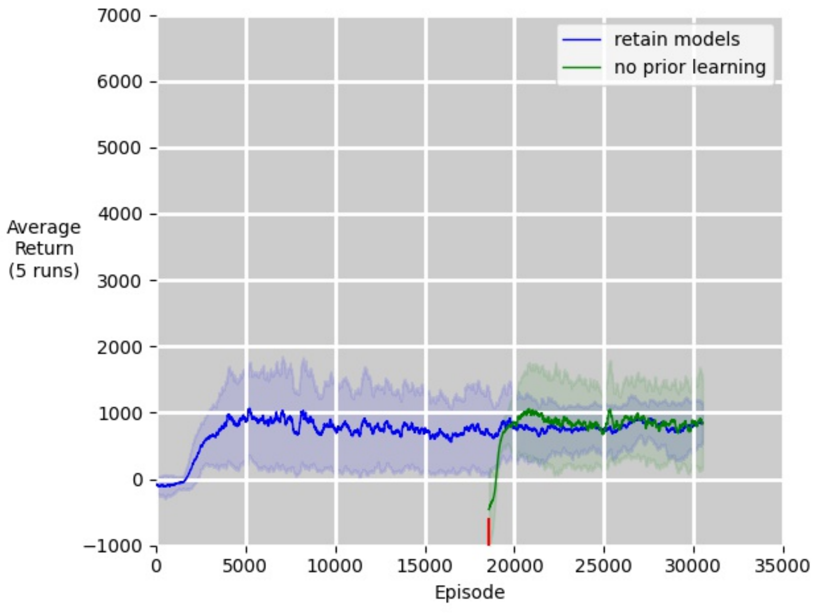
(a) PPO.



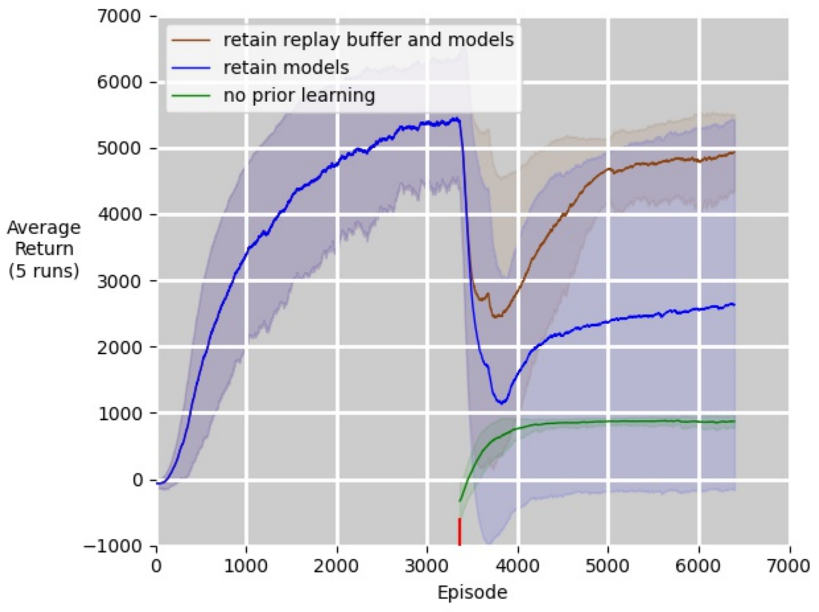
(b) SAC.

Figure 4.2: Ankle range of motion decreased (from [30, 70] to [65, 70]). The red marker indicates the onset of a fault. Labels indicate the data retained after learning the task with the normal ant, if applicable. The shaded areas correspond to a 95% confidence interval.

contrast, the gait learned with PPO appeared to have random, less-specialized motion. All four legs were being used with the PPO gait, but there was no apparent pattern and the learned gait was not as successful at propelling the



(a) PPO.



(b) SAC.

Figure 4.3: Lower limb severed. The red marker indicates the onset of a fault. Labels indicate the data retained after learning the task with the normal ant, if applicable. The shaded areas correspond to a 95% confidence interval.

ant forward; in fact, there was very little forward movement. One possible (very likely) cause for this poor performance by PPO in the task with the normal ant could be the fact that no hyper-parameter search was conducted for this task. The hyperparameters used in the preliminary experiments were those presented in each algorithm's introductory paper [23], [55] for MuJoCo tasks. For PPO, the performance for many MuJoCo tasks were included in the introductory paper; however, the performance of PPO in the Ant-v2 task was excluded.

When we examine the performance of each algorithm in the task with the faulty ant, we once again see surprising results with the performance of PPO. It appears that PPO was unaffected by the presence of a fault, as there was no obvious degradation in performance. We believe that the gait learned by the normal ant was too random and non-specialized to be affected. The performance was already poor. In contrast, we see that the performance of SAC was highly affected by each fault. We see a big drop in performance after fault onset. We believe that, because the observed gait of SAC was highly specialized, the application of a fault that targeted a leg responsible for forward motion, caused the gait pattern to be significantly disrupted. This led to a big drop in initial performance as the ant could no longer propel itself forward as efficiently. We confirmed our supposition by rending the OpenAI Gym task. We saw that immediately after the application of a fault, the ant that learned with SAC could not move forward as effectively, and in the initial stages of learning, appeared to be staying near its initial (x, y) position within the world frame. The fault severely disrupted its motion. In contrast, immediately after the onset of a fault, the ant that learned with PPO appeared to have no change in its seemingly random pattern of movements.

The 95% confidence intervals in these experiments are large, and as a result we cannot definitively draw conclusions about the statistical significance between different algorithm initializations. However, we will still try to distinguish a pattern from the results for SAC. Unfortunately, for PPO, the performances after the onset of a fault are visually indistinguishable, so we will refrain from looking for a pattern within these plots.

When comparing the results for each SAC initialization, across all three faults, the results are somewhat inconclusive. For example, in the ankle range of motion decrease fault (Figure 4.2b) and the lower limb severed fault (Figure 4.3b), it appears that learning from scratch leads to poorer final performance than the other two algorithm initializations (within the allotted time frame). In contrast, for the hip range of motion decrease fault, learning the task with the faulty ant from scratch results in the best final performance. While our results are not statistically significant, the cause of this observation may be that the hip range of motion decrease fault causes the task to be dramatically changed, and that all prior knowledge of the old task hinders the ant's performance. If this were certainly true, we would expect that the more knowledge we have of the old task, the more severe the degradation in performance; however, this is not what we see. Retaining the replay buffer contents and the models leads to better performance that only retaining the models. So, the replay buffer contents, immediately after the onset of a fault, helped the ant adapt to the fault. We may be able to understand what is happening here if we consider the contrary case, and think about how emptying the replay buffer contents may impact performance. Immediately after the replay buffer contents are cleared, the replay buffer begins to fill up with new experiences once again. Initially, these new experiences are limited and are only representative of a small portion of the overall task (i.e. relatively few states have been visited and few actions have been taken). Since the ant task is high-dimensional (and complex), this may hinder its overall performance, when these limited samples are being used to update the policy. It is possible that by clearing the replay buffer contents, actions are taken and updates are made, that ultimately lead to a poorly performing policy. So why would a full replay buffer be better? Although the task is changed, it is possible that some of the experiences contained within the replay buffer are still valid. This would lead to better network updates and a better performing policy than the case where we empty the replay buffer contents, and use only a small sample of experiences to perform updates.

In the other two faults, ankle range of motion decrease fault and lower limb severed fault, we once again see slightly contradictory results when we examine the two algorithm initializations where some (or all) knowledge is retained. For the ankle range of motion decrease fault, retaining the replay buffer and models led to a larger drop in performance immediately after the onset of a fault than only retaining the models. In contrast, for the lower limb severed fault, retaining the replay buffer contents and models led to a higher performance immediately after the onset of a fault than retaining only the models. There may be an explanation for this observation. When we rendered each of these faulty ants in OpenAI Gym, we observed that the initial movement pattern of the ant with the lower limb fault was the least different from the normal ant's movement pattern. If we think about how this fault affected the ant, it makes sense. All the joints in the ant still had a full range of motion. The only effect caused by the fault was a shorter leg. In contrast, the ankle range of motion decrease fault caused the ankle to be angled in an usual manner (at one end of the normal range of motion) and this caused the movement pattern to be quite different. Since the normal ant task is similar to the lower limb severed task, we could reasonably assume that the replay buffer contents would be representative of this task. It contains state, action, reward, next state transitions that are unchanged; in other words, the underlying dynamics of the problem are unchanged. As a result, the replay buffer contents are still representative of this problem. By discarding the replay buffer contents, we lose valuable information for this fault, thus we see a reduced performance when only the models are retained. For the ankle range of motion decreased fault, the underlying environment dynamics changed. Since we had restricted the range of motion, actions that would have normally led to joint positions outside of the restricted range, now led to joint positions within the restricted range. So, some of the experiences contained in the replay buffer were inaccurate. Using these inaccurate, non-representative samples to make updates could have degraded the policy learned with the fault present.

There is one more observation that further justifies how the replay buffer contents and models can aid adaption to a task where the underlying dynamics are unchanged. We have already pointed out that for the lower limb severed fault (Figure 4.3b), discarding all learned knowledge led to very poor performance. Here, we see that by discarding this highly beneficial knowledge, the performance attained after the onset of a fault is very poor. This shows that the learning task with the fault present is more difficult and may not even be completely learnable without prior knowledge. The performance attained after discarding all learned knowledge is similar to that attained with PPO.

4.2 SoftBank Robotics NAO in Webots

4.2.1 Results

We now present our results for the experiments with the SoftBank Robotics NAO, simulated in Webots. Here, we use our own implementations of the two target algorithms, PPO and SAC. We perform all experiments using the NAO's right arm. The arm used in the arm reaching task was a trivial choice; our work allowed for either arm to be used.

In PPO experiment 2 and SAC experiment 3, there was no prior learning in the task with the normal robot, so the onset of a fault occurred at time step 0. To make the data from these experiments comparable to the data obtained from the other experiments, we have once again shifted the learning curves for these two experiments such that their origin lies at the fault marker, positioned at time step 6.0 million for PPO and time step 1.5 million for SAC.

Figures 4.4, 4.5, and 4.6 show the learning curves for PPO and SAC for each NAO fault. In each plot, the red marker indicates the onset of a fault. The shaded area represents a 95% confidence interval, which we computed with a t-distribution. Each algorithm has different initializations, which are indicated by the different line colours. Within these plots, we measure performance as the *median return* in a policy evaluation. A higher performance is equivalent to a higher (more positive) median return.

We also present our statistical results, in tabular form. We performed a series of paired t-tests. T-tests are used to determine if there is a significant difference between two variables for the same subject. For each t-test, our subject was the robot. Here, we defined the robot's task performance as the *average return* computed in a single policy evaluation. We considered two variables for our robot. The first variable was the robot's average task performance *before* a fault was applied. The second variable was the robot's task performance at some point *after* a fault was applied. For each algorithm initialization and fault, we identified three critical periods after a fault was applied, using each one as a second variable in a separate t-test. These included:

- the average performance in the 10 episodes after the onset of a fault;
- the average performance after partial adaption to a fault, where partial adaptation for PPO was defined as 2 million time steps of learning with

the fault present, and partial adaptation for SAC was defined as 300,000 time steps of learning with the fault present; and

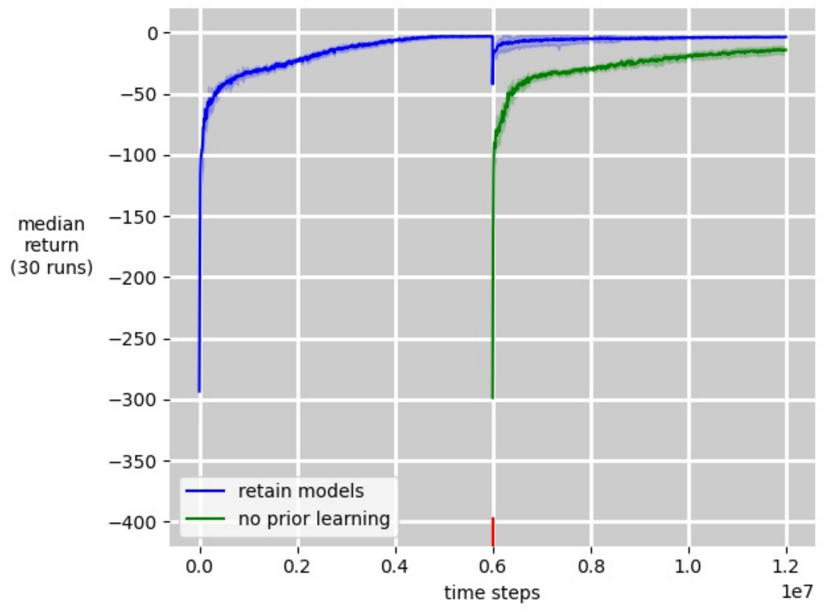
- the average performance after full adaptation to a fault, where full adaptation for PPO was defined as 5.9 million time steps of learning with the fault present and full adaptation for SAC was defined as 1.498 million time steps of learning with the fault present.

For each run, we computed the difference in performance between these two variables. In each table we present the average difference across 30 independent runs. We additionally provide the standard error, as well as the upper bound and lower bound for a 95% confidence interval.

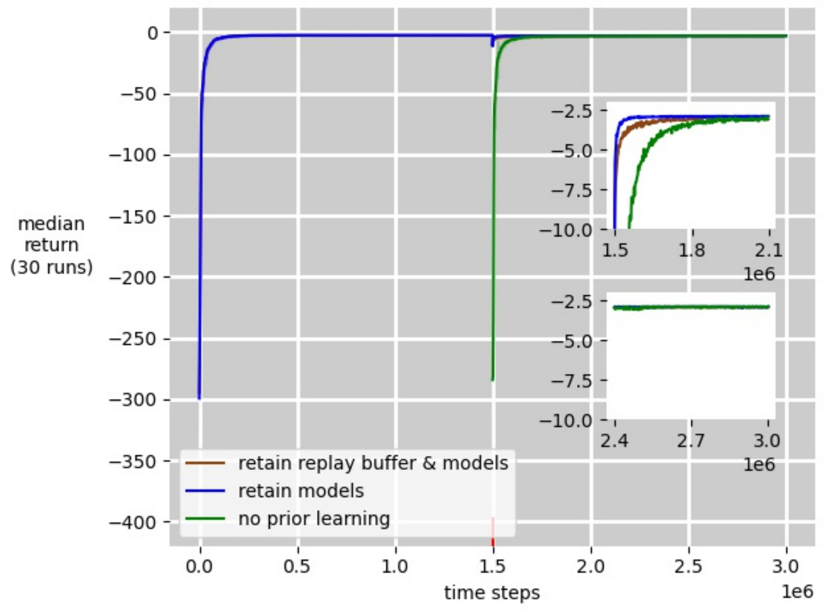
In many experiments, we had the normal NAO learn the arm reaching task for some number of time steps before applying the fault; then, learning was continued with the fault present. For each algorithm (and seed), the learning of the normal NAO task was performed only once and all important data was saved. Thus, for all t-tests, the performance prior to the application of a fault is the same. This is assumed to be true for PPO experiment 2 and SAC experiment 3, in which there was no prior learning (i.e. all learned knowledge was discarded).

Two of the critical periods that we have defined for our t-tests define a set number of time steps. The number of time steps chosen for each definition of partial adaptation was intentional and was selected to highlight the difference among the different initializations for an algorithm. The number of time steps chosen for each definition of full adaptation was based on the need for 10 policy evaluations to occur *after* full adaptation; thus, for each algorithm, we obtained the number of time steps for full adaptation by subtracting $10 \times policy\ evaluation\ frequency$ from the total number of time steps allowed for learning after the application of a fault.

The statistical results for PPO are shown in Tables 4.1, 4.2, and 4.3. The statistical results for SAC are shown in Tables 4.4, 4.5, and 4.6.

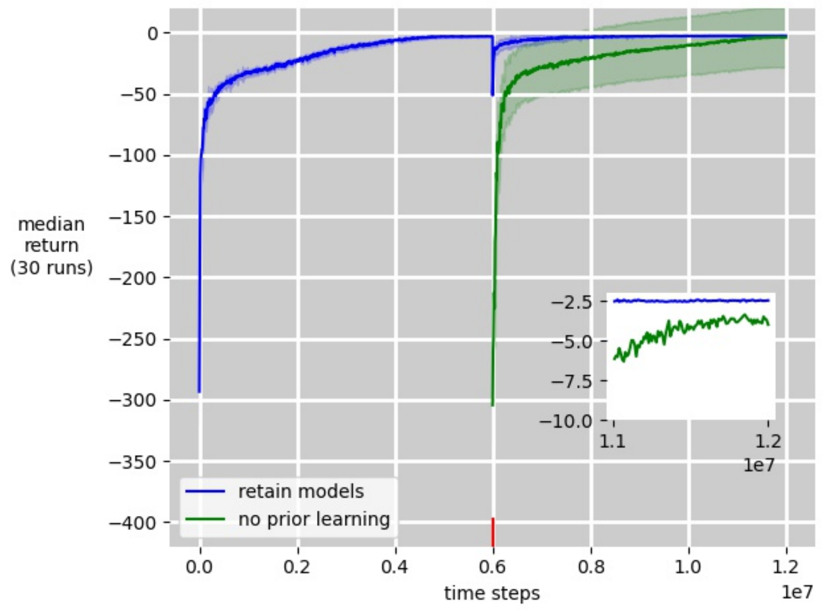


(a) PPO.

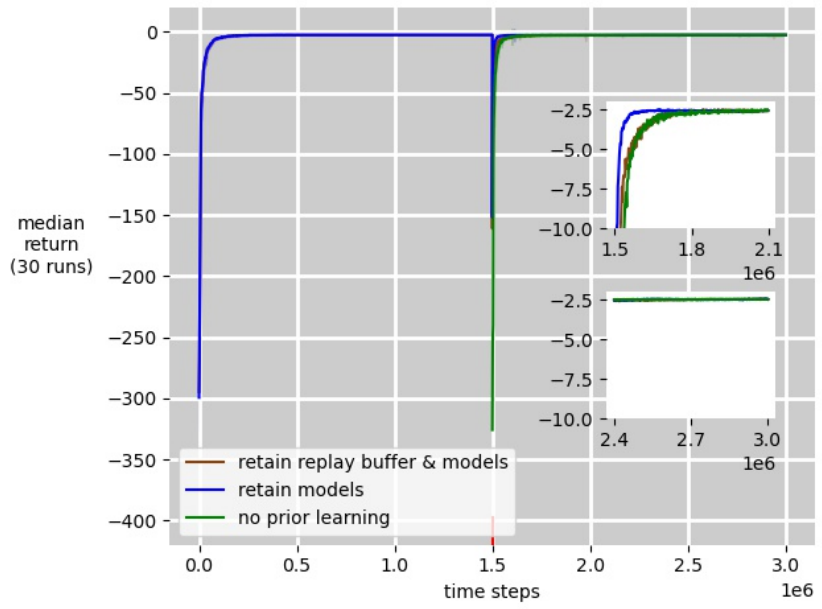


(b) SAC.

Figure 4.4: RShoulderRoll range of motion decrease (range decreased to [0.0, 0.0] radians). The red marker indicates the onset of a fault. Labels indicate the data retained after learning the task with the normal NAO, if applicable. In (b), the brown and blue lines overlap most of the time. Later, the green line overlaps them both. The shaded areas correspond to a 95% confidence interval. Zoomed plots do not show confidence intervals.

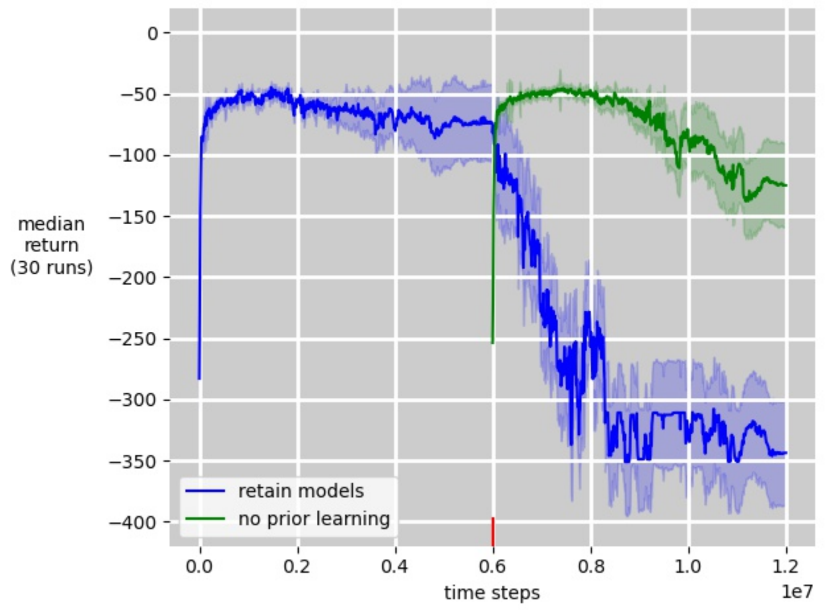


(a) PPO.

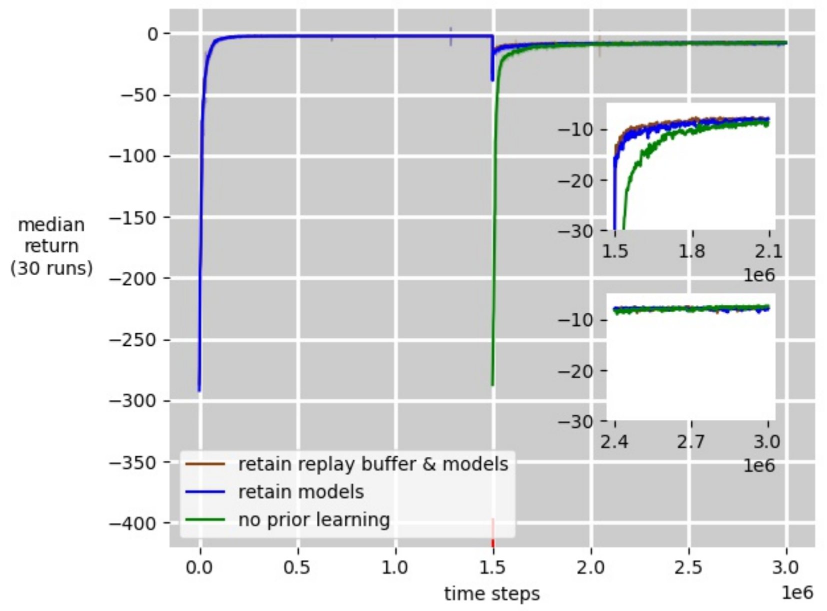


(b) SAC.

Figure 4.5: RShoulderPitch frozen position sensor (always reading -2.0 radians). The red line marker indicates the onset of a fault. Labels indicate the data retained after learning the task with the normal NAO, if applicable. In (b), the brown and blue lines overlap in the first half of the experiment. After the fault is applied, the green line mostly overlaps the brown line. Eventually, all lines converge. The shaded areas correspond to a 95% confidence interval. Zoomed plots do not show confidence intervals.



(a) PPO.



(b) SAC.

Figure 4.6: Broken inner finger (finger joints stuck at 0.0 radians despite commands to move them). The red marker indicates the onset of a fault. Labels indicate the data retained after learning the task with the normal NAO, if applicable. In (b), the brown and blue lines overlap for most of the experiment. Eventually, all lines converge. The shaded areas correspond to a 95% confidence interval. Zoomed plots do not show confidence intervals.

Table 4.1: PPO. Performance immediately after the onset of a fault. For each run, the severity of the drop in performance is measured as the difference between the mean performance in the 10 evaluations before a fault was applied and the mean performance in the 10 evaluations after a fault was applied. The mean is the average performance drop across 30 runs. Lower bound (LB) and upper bound (UB) 95% confidence intervals are shown.

	Performance Drop After Fault Onset			
PPO	**Mean**	**Standard Error**	**LB**	**UB**
RShoulderRoll range of motion decrease (range decreased to [0.0, 0.0] radians)				
retain models	-21.2772	2.6677	-26.7327	-15.8217
no prior learning	-125.5292	4.8511	-135.4497	-115.6087
RShoulderPitch frozen position sensor (always reading -2.0 radians)				
retain models	-20.5986	2.5223	-25.7567	-15.4405
no prior learning	-204.1549	8.7509	-222.0505	-186.2593
Broken inner finger (finger joints stuck at 0.0 radians despite commands to move them)				
retain models	-29.3569	10.2907	-50.0477	-8.6661
no prior learning	-24.3201	16.7431	-57.9842	9.344

Table 4.2: PPO. Performance after partial adaptation to a fault. For each run, partial fault adaptation is measured as the difference between the mean performance in the 10 evaluations before a fault was applied and the mean performance in the 10 evaluations after learning for 2.0 million time steps with the fault present. The mean is the average change in performance across 30 runs. Lower bound (LB) and upper bound (UB) 95% confidence intervals are shown.

	Performance Drop After Partial Adaptation			
PPO	**Mean**	**Standard Error**	**LB**	**UB**
RShoulderRoll range of motion decrease (range decreased to [0.0, 0.0] radians)				
retain models	-4.2813	1.3481	-7.0381	-1.5245
no prior learning	-26.0191	0.7662	-27.586	-24.4522
RShoulderPitch frozen position sensor (always reading -2.0 radians)				
retain models	-0.988	0.4364	-1.8805	-0.0955
no prior learning	-28.8328	11.1168	-51.5666	-6.099
Broken inner finger (finger joints stuck at 0.0 radians despite commands to move them)				
retain models	-146.8022	18.9805	-184.9649	-108.6395
no prior learning	49.9383	14.8802	20.0197	79.8569

Table 4.3: PPO. Performance after full adaptation to a fault, where full adaptation is defined as 5.9 million time steps of learning with the fault present. For each run, full fault adaptation is measured as the difference between the mean performance in the 10 evaluations before a fault was applied and the mean performance in the final 10 evaluations conducted with the fault present. The mean is the average change in performance across 30 runs. Lower bound (LB) and upper bound (UB) 95% confidence intervals are shown.

	Performance Drop After Full Adaptation			
PPO	**Mean**	**Standard Error**	**LB**	**UB**
RShoulderRoll range of motion decrease (range decreased to [0.0, 0.0] radians)				
retain models	-0.3554	0.2499	-0.8663	0.1555
no prior learning	-11.8544	1.2724	-14.4566	-9.2522
RShoulderPitch frozen position sensor (always reading -2.0 radians)				
retain models	0.6154	0.2414	0.1218	1.109
no prior learning	-12.8651	11.6288	-36.6459	10.9157
Broken inner finger (finger joints stuck at 0.0 radians despite commands to move them)				
retain models	-167.3417	20.3302	-208.2183	-126.4651
no prior learning	-44.9794	25.6397	-96.5312	6.5724

Table 4.4: SAC. Performance immediately after the onset of a fault. For each run, the severity of the drop in performance is measured as the difference between the mean performance in the 10 evaluations before a fault was applied and the mean performance in the 10 evaluations after a fault was applied. The mean is the average performance drop across 30 runs. Lower bound (LB) and upper bound (UB) 95% confidence intervals are shown.

	Performance Drop After Fault Onset			
SAC	**Mean**	**Standard Error**	**LB**	**UB**
RShoulderRoll range of motion decrease (range decreased to [0.0, 0.0] radians)				
retain replay buffer & models	-5.6075	0.5398	-6.6929	-4.5221
discard replay buffer & retain models	-3.4839	0.4058	-4.3138	-2.654
no prior learning	-128.4207	4.2046	-137.0192	-119.8222
RShoulderPitch frozen position sensor (always reading -2.0 radians)				
retain replay buffer & models	-49.7094	3.719	-57.3148	-42.104
discard replay buffer & retain models	-32.9239	2.6429	-38.3286	-27.5192
no prior learning	-120.2054	2.8671	-126.0687	-114.3421
Broken inner finger (finger joints stuck at 0.0 radians despite commands to move them)				
retain replay buffer & models	-18.8985	1.6901	-22.2375	-15.5595
discard replay buffer & retain models	-18.3593	1.0748	-20.4444	-16.2742
no prior learning	-181.8428	4.3499	-190.2819	-173.4037

Table 4.5: SAC. Performance after partial adaptation to a fault. For each run, partial fault adaptation is measured as the difference between the mean performance in the 10 evaluations before a fault was applied and the mean performance in the 10 evaluations after learning for 300,000 time steps with the fault present. The mean is the average change in performance across 30 runs. Lower bound (LB) and upper bound (UB) 95% confidence intervals are shown.

	Performance Drop After Partial Adaptation			
SAC	**Mean**	**Standard Error**	**LB**	**UB**
RShoulderRoll range of motion decrease (range decreased to [0.0, 0.0] radians)				
retain replay buffer & models	-1.013	0.1004	-1.2149	-0.8111
discard replay buffer & retain models	-0.5818	0.0624	-0.7094	-0.4542
no prior learning	-0.9843	0.0766	-1.1409	-0.8277
RShoulderPitch frozen position sensor (always reading -2.0 radians)				
retain replay buffer & models	-1.5361	0.167	-1.8775	-1.1947
discard replay buffer & retain models	-0.7949	0.5138	-1.8457	0.2559
no prior learning	-0.3884	0.0425	-0.4753	-0.3015
Broken inner finger (finger joints stuck at 0.0 radians despite commands to move them)				
retain replay buffer & models	-12.1577	3.8352	-19.7347	-4.5807
discard replay buffer & retain models	-8.7368	0.6231	-9.9456	-7.528
no prior learning	-8.1909	0.5495	-9.2569	-7.1249

Table 4.6: SAC. Performance after full adaptation to a fault, where full adaptation is defined as 1.498 million time steps of learning with the fault present. For each run, full fault adaptation is measured as the difference between the mean performance in the 10 evaluations before a fault was applied and the mean performance in the final 10 evaluations conducted with the fault present. The mean is the average change in performance across 30 runs. Lower bound (LB) and upper bound (UB) 95% confidence intervals are shown.

	Performance Drop After Full Adaptation			
SAC	**Mean**	**Standard Error**	**LB**	**UB**
RShoulderRoll range of motion decrease (range decreased to [0.0, 0.0] radians)				
retain replay buffer & models	-0.5235	0.0584	-0.641	-0.406
discard replay buffer & retain models	-0.5126	0.0571	-0.6294	-0.3958
no prior learning	-0.5274	0.0577	-0.6455	-0.4093
RShoulderPitch frozen position sensor (always reading -2.0 radians)				
retain replay buffer & models	-0.1125	0.0348	-0.1837	-0.0413
discard replay buffer & retain models	-0.07	0.0067	-0.0838	-0.0562
no prior learning	-0.0716	0.0073	-0.0865	-0.0567
Broken inner finger (finger joints stuck at 0.0 radians despite commands to move them)				
retain replay buffer & models	-15.3037	6.3905	-27.9291	-2.6783
discard replay buffer & retain models	-6.3313	0.5018	-7.3047	-5.3579
no prior learning	-5.7807	0.4743	-6.7009	-4.8605

4.2.2 Discussion

It would be ideal for an algorithm to be robust to a dynamic environment, and to see only small perturbations in performance after a fault occurs. Large changes in performance could easily equate to irregular behaviour. Irregular behaviour, even that which is temporary, could lead to further damage to a robot and potentially to surrounding objects or persons. To evaluate each algorithm's ability to enable adaptation to faults, we considered three evaluation conditions:

- its performance immediately after the onset of a fault,
- its performance after partial adaptation to a fault, and
- its performance after full adaptation to a fault.

We will first focus on two faults: RShoulderRoll range of motion decrease and RShoulderPitch frozen position sensor. The remaining fault, broken inner finger, is an outlier. We reserve our discussion of this fault for last.

Visually, our results (for the two faults) show that the retention of learned knowledge generally leads to less degradation in performance in the initial stages of adaptation that disposal of the learned knowledge. For PPO, the learned knowledge is the model weights, while for SAC, the learned knowledge is both the model weights and the replay buffer contents.

Retained models are beneficial in the early stages of learning because, from the experience of the learning task with the normal robot, the models have learned to define good behaviours in the robot's different states and have well-formed estimates of the values of these states (or state-action pairs). When our faults are applied, the dynamics may change; if the dynamics change, they only change to some degree, while much of the dynamics remains the same.

Table 4.7: Summary of the performance immediately after the onset of a fault. 95% confidence intervals are shown. Original data in Tables 4.1 and 4.4.

	Performance Drop After Fault Onset	
	PPO	**SAC**
RShoulderRoll range of motion decrease		
retain models	[-26.7327, -15.8217]	[-4.3138, -2.654]
no prior learning	[-135.4497, -115.6087]	[-137.0192, -119.8222]
RShoulderPitch frozen position sensor		
retain models	[-25.7567, -15.4405]	[-38.3286, -27.5192]
no prior learning	[-222.0505, -186.2593]	[-126.0687, -114.3421]

Although the policy and value estimates may now be incorrect for the new task, it is likely that they are incorrect to a smaller degree than when we throw away all knowledge. When we throw away all knowledge, we start with randomly initialized networks; so, our policy initially behaves randomly and our value estimates are highly inaccurate. It takes time to re-establish a good policy and good value estimates, so in the initial stages of learning, we would expect poor performance. The statistical analysis of our data (summarized in Table 4.7) supports the visual results. Immediately after the onset of a fault, retention of the models is better than no prior learning for both PPO and SAC.

Visually, our results also show that, for SAC, disposal of the replay buffer contents generally leads to better performance in the initial stages of learning than retention of the replay buffer contents. At the start of learning the task with the faulty robot, the experiences contained within the replay buffer are non-representative of the changed task. With a large replay buffer, it takes a long time to replace the contents with new, representative samples; thus, at each time step, the samples being used to update the policy and value functions are based on old experiences. This can hinder adaptation in the initial stages.

Table 4.8: Summary of the performance immediately after the onset of a fault. 95% confidence intervals are shown. Here, discard replay buffer refers to the algorithm initialization condition where the replay buffer is discarded and the models are retained. Original data in Table 4.4.

	Performance Drop After Fault Onset
	SAC
RShoulderRoll range of motion decrease	
retain replay buffer	[-6.6929, -4.5221]
discard replay buffer	[-4.3138, -2.654]
RShoulderPitch frozen position sensor	
retain replay buffer	[-57.3148, -42.104]
discard replay buffer	[-38.3286, -27.5192]

In Table 4.8, we present our statistical results that support this claim; there is a statistically significant difference in the initial performance between retention and disposal of the replay buffer for both faults, and among the two faults, the most severely affected is the RShoulderPitch frozen position sensor fault. The large initial degradation of the RShoulderPitch frozen position sensor fault may be indicative that, initially, this fault is more severe than the RShoulderRoll range of motion fault.

After learning the task with a faulty robot for a predefined amount of time (i.e. 2.0 million time steps for PPO and 300,000 time steps for SAC), partial adaptation was achieved. Similar to the initial adaptation stage, we would want to ideally see a high performance at this stage. Visually, for PPO, retention of the learned models shows obvious higher partial adaptation performance than discarding the learned models. Our statistical results for PPO, summarized in Table 4.9, support the visual results. For SAC, the different algorithm initializations start to converge at 1.8 million time steps, so it is hard to differentiate their performances visually. Our statistical analysis

Table 4.9: Summary of the performance after partial adaptation to a fault. Replay buffer is indicated by RB. 95% confidence intervals are shown. Original data in Tables 4.2 and 4.5.

	Performance Drop After Partial Adaptation	
	PPO	**SAC**
RShoulderRoll range of motion decrease		
retain RB & models	N/A	[-1.2149, -0.8111]
retain models	[-7.0381, -1.5245]	[-0.7094, -0.4542]
no prior learning	[-27.586, -24.4522]	[-1.1409, -0.8277]
RShoulderPitch frozen position sensor		
retain RB & models	N/A	[-1.8775, -1.1947]
retain models	[-1.8805, -0.0955]	[-1.8457, 0.2559]
no prior learning	[-51.5666, -6.099]	[-0.4753, -0.3015]

shows that for the RShoulderRoll range of motion decrease fault, discarding the replay buffer contents and retaining the models is better than all other SAC initializations; the two remaining algorithm initializations are not statistically different and cannot be distinguished. It could be the case that, after starting from scratch, SAC has learned enough such that the performance of this algorithm initialization matches the slightly hindered performance of the retention of a full replay buffer (and models). For the RShoulderPitch frozen position sensor fault, our statistical analysis shows a significant difference between the no prior learning initialization and the retention of the replay buffer and models initialization; all other algorithm initialization pairs are not statistically different. By this point, the robot has significantly adapted to the fault and only minor subsequent adaptation occurs. Calculations on the percentage of adaptation at the 1.8 million time step marker for this fault indicate that over 96% of adaptation has occurred for all three algorithm initializations. The fast adaptation to the RShoulderPitch frozen position sensor fault with

Table 4.10: Summary of the performance after full adaptation to a fault. Replay buffer is indicated by RB. 95% confidence intervals are shown. Original data in Tables 4.3 and 4.6.

	Performance Drop After Full Adaptation	
	PPO	**SAC**
RShoulderRoll range of motion decrease		
retain RB & models	N/A	[-0.641, -0.406]
retain models	[-0.8663, 0.1555]	[-0.6294, -0.3958]
no prior learning	[-14.4566, -9.2522]	[-0.6455, -0.4093]
RShoulderPitch frozen position sensor		
retain RB & models	N/A	[-0.1837, -0.0413]
retain models	[0.1218, 1.109]	[-0.0838, -0.0562]
no prior learning	[-36.6459, 10.9157]	[-0.0865, -0.0567]

SAC is our first evidence that the *overall* learning task with this fault may be easier than the learning task with the RShoulderRoll range of motion decrease fault.

After learning the task with the faulty robot for an extended time (i.e. 5.9 million time steps for PPO and 1.498 time steps for SAC), full adaptation is reached. Visually, for the RShoulderRoll range of motion decrease fault, it is clear that for PPO, the algorithm initialization in which the models were retained has a better performance than discarding the models. For the RShoulderPitch frozen position sensor fault, the performances of the two algorithm initializations converge. Table 4.10 summarizes our statistical data for the performance of each algorithm initialization after full adaptation. Here, we can see that the difference in performance of these two PPO initializations for the RShoulderPitch frozen position sensor fault are not statistically significant; the confidence intervals overlap. The confidence interval for the disposal of the models algorithm initialization is large. As a result, it would be beneficial to perform more runs with this algorithm initialization to better distinguish its performance.

For both PPO initializations with the RShoulderPitch frozen position sensor fault, the upper bound 95% confidence intervals extended into positive values, indicating the possibility that the true mean performance of the task with the faulty NAO may be higher than that with the normal NAO. For the retention of the models algorithm initialization, the lower bound of the 95% confidence interval was also positive. There are a few possible explanations for these observations. First, although the PPO learning curve with the normal NAO appears to stabilize for a short time, it is possible that learning was incomplete, and that a better policy could have been found with an extended learning period; the extended number of time steps to learn with the faulty robot may have helped in learning a better policy. Second, it is possible that the learning task with this fault is easier for PPO to learn. (We have already seen that, for this fault, SAC achieved over 96% adaptation at the partial adaptation marker of 1.8 million time steps.) By overwriting the position sensor reading, the models may begin to learn that this input is not useful; this may result in the agent learning the reduced-dimension task slightly better. For SAC, the differences in the full adaptation performance for each algorithm initialization are visually indistinguishable and are not statistically significant.

We expect that a robot experiencing a hardware fault would have a slightly degraded task performance, even after full adaptation. In most cases, the performance drop after full adaptation was negative, verifying that performance was slightly degraded. In Table 4.11, we summarize our statistical data in the form of t-values. For SAC, these differences in performance were all statistically significant, as their t-values were all greater than the critical value of 2.045. For PPO, two of the four combinations of algorithm initializations and

Table 4.11: Summary of the performance after full adaptation to a fault. Replay buffer is indicated by RB. Computed t-values are shown. For our sample size (30), a t-value greater than the critical value of 2.045 indicates that the difference between our two variables is significant. Original data in Tables 4.3 and 4.6.

	Performance Drop After Full Adaptation	
	PPO	**SAC**
RShoulderRoll range of motion decrease		
retain RB & models	N/A	8.9640
retain models	1.4222	8.9772
no prior learning	9.3166	9.1404
RShoulderPitch frozen position sensor		
retain RB & models	N/A	3.2328
retain models	2.5493	10.4478
no prior learning	1.1063	9.8082

faults resulted in statistically significant differences, while the remaining two were not statistically significant. That is, for two combinations there was no significant difference between pre-fault and post-adaptation performance, indicating highly successful adaptation to a fault. These two combinations are: 1) RShoulderRoll range of motion decrease, with the retention of the models and 2) RShoulderPitch frozen position sensor, with no prior learning.

So, from our observations so far, we conclude that both PPO and SAC are generally able to add hardware fault tolerance to our robot, NAO. Both algorithms were very successful at this task, reaching a full adaptation performance close to that attained without the fault present. Additionally, in two cases, we have seen PPO reach a full adaptation performance with the fault present that is not statistically different from that attained without the fault present. When we additionally consider the performance immediately after the onset of a fault, we have mixed results that do not support that one algorithm is better than the other. What we can suggest is that retaining the models and, for SAC, discarding the replay buffer contents leads to better initial performance. When we consider the performance after partial adaption, for PPO, retention of the models consistently performs better. For SAC, discarding the replay buffer contents and retaining the models performs better for one fault, while for the other fault, adaptation for all algorithm initializations has already been mostly achieved. If fast adaptation to a fault is required, then we conclude that SAC may be a better algorithm choice. In this case, to achieve the best initial performance after the onset of a fault, we recommend that the replay buffer contents are discarded and that the models are retained. However, if complete adaptation to a fault is required, we would recommend the use of PPO. We have seen half of our combinations of algorithm initializations and faults result in a performance that is just as good as the task performance with no fault present. In addition, to achieve the best initial performance with PPO, we recommend that the model is retained.

Now we consider the final fault, a broken inner finger. We consider this fault to be an outlier in our results for both algorithms. For PPO, the performance severely degrades overtime after the onset of a fault. In fact, we see that performance starts to degrade even before the onset of a fault. This is unexpected. However, we believe that several factors (or a combination of them) may contribute to this observation. First, one of the requirements for PPO to learn a good policy is a good sample size (i.e. memory size) and the required sample size increases with task complexity [24], [28]. We performed a hyperparameter search for our original task, in which the finger joints were not actuated; actuating the finger joints was a decision made towards the end of our work. The task with actuated finger joints is much more complex as we increased the number of actuated joints from five to eight. So, it may be neces-

Table 4.12: Summary of the performance during adaptation to a fault. Replay buffer is indicated by RB. 95% confidence intervals are shown. Original data in Tables 4.1-4.6.

	Broken Inner Finger Fault	
	PPO	**SAC**
Performance drop immediately after the onset of a fault		
retain RB & models	N/A	[-22.2375, -15.5595]
retain models	[-50.0477,-8.6661]	[-20.4444, -16.2742]
no prior learning	[-57.9842, 9.344]	[-190.2819, -173.4037]
Performance after partial adaptation to a fault		
retain RB & models	N/A	[-19.7347, -4.5807]
retain models	[-184.9649, -108.6395]	[-9.9456, -7.528]
no prior learning	[20.0197, 79.8569]	[-9.2569, -7.1249]
Performance after full adaptation to a fault		
retain RB & models	N/A	[-27.9291, -2.6783]
retain models	[-208.2183, -126.4651]	[-7.3047, -5.3579]
no prior learning	[-96.5312, 6.5724]	[-6.7009, -4.8605]

sary to increase the memory size to learn a good policy with PPO in this task. The performance degradation prior to the onset of a fault, and with random initialization of the networks after the onset of a fault, both provide evidence to support this claim. Second, in our work with PPO, we commonly observed that the entropy of the policy distribution would temporarily increase after the onset of a fault. After a short time, it would once again start to decrease, thereby taking less random actions; however, in this particular experiment, the entropy of the policy distribution continually increased. This led to more and more random action choices, drawn from a normal distribution with an increasing standard deviation. Continually taking increasingly random actions, rather than greedy actions, is likely to result in degraded performance. Finally, we believe that with this particular fault, the environment dynamics change the most drastically; commands to move the three finger joints result in no change to the position and velocity of these joints. The resulting next state, which in part contains the position and velocity of each actuated joint, is very different from that when the finger was not broken. It is possible that because of this drastic change in the environment dynamics, the retained model (which is now incorrect to the changed task) causes PPO to select poor actions and adapt its policy such that it can never recover, even through increased exploration. For SAC, the deviation with the broken inner finger fault is that, visually, the algorithm initialization where the replay buffer contents and the models are retained appears to enable slightly faster adaptation than the algorithm initialization where the replay buffer contents are disposed and the models are retained. Throughout the entire learning process, including immediately after the onset of a fault, after partial adaptation, and after full adaptation, our statistical analysis (summarized in Table 4.12) cannot differentiate the performance between these two algorithm initializations; their performance was the same. Again, this could be an indication that our hyperparameters were not properly tuned to this task. Or, for both PPO and SAC, this may be our first evidence that these two algorithms behave differently for different faults in the NAO arm reaching task.

Our overall findings indicate that the three NAO faults can be ordered in terms of their severity, where a fault's severity is defined to be the performance after full adaptation to the fault; a fault with a lower performance after full adaptation would be considered more severe. Listed in order of descending severity, the faults are: 1) broken inner finger, 2) RShoulderRoll joint range of motion decrease, and 3) RShoulderPitch frozen position sensor. The two most severe faults affect the robot's motion, and consequently affect the underlying environment dynamics. We believe that the broken inner finger fault led to the most severe change in environment dynamics. Three (of eight) joints were affected with this fault. This is confirmed by the performance of both PPO and SAC in the presence of this fault. PPO's performance severely degrades when the models are retained and slowly degrades when the networks are randomly re-initialized. SAC's performance after full adaptation is visibly lower than that obtained prior to the onset of a fault; a degraded performance this severe was not visible in our other SAC learning curves. With the RShoulder-Roll joint range of motion decrease, only a single joint (of five) was affected; thus, the environment dynamics were less changed than with the broken inner finger fault. We do not believe that the RShoulderPitch frozen position sensor fault changed the underlying environment dynamics. Rather, we believe that this fault only affected the model; the networks learned that the RShoulderRoll position sensor reading was not useful, and they began to ignore its contribution.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This thesis set out to answer several questions:

Can PPO and SAC enable a robot to adapt to a hardware fault?

How does adaptation to a fault with PPO compare to that with SAC, and vice versa?

How do the differences in the faults affect a robot's adaptation?

How does retention (or random re-initialization) of the learned models affect a robot's adaptation to a fault?

In the case of SAC, how does retention (or disposal) of the replay buffer contents affect a robot's adaptation to a fault?

Overall, we have seen that both PPO and SAC can add hardware fault tolerance to a robot; that is, both algorithms have shown that after a fault is applied, task performance can be mostly regained after a period of adaptation. SAC appears to be the fastest algorithm to adapt and has shown to be robust to its hyperparameters. PPO, in contrast, appears to lead to better overall adaption.

Based on our results, we would advise that when a fault occurs, the best option would be to use SAC to recover, discarding the replay buffer contents, and retaining the models. This is the algorithm initialization for SAC that led to the fastest recovery in our experiments. We believe that for real-world tasks, where repair is possible after some short time delay, fast adaptation is most important. In cases where repair is either not possible or there is a longer time delay for repair by an expert (e.g. in space), PPO may be a better option. In many cases, PPO was able to achieve a performance with the fault present that was indistinguishable from the performance without the fault present. However, adaptation with PPO did take longer, so if it were used, one could expect undesirable machine behaviour for a longer period of time than with SAC. In real-time, PPO took approximately 4-6 hours to achieve full adaption to a fault when the models were retained. In contrast, SAC took approximately 1.6-3.2 hours to achieve full task adaptation when the replay buffer contents were discarded and the models were retained, which strongly highlights its advantage over PPO in real-world applications.

We have observed that the performance for each algorithm can vary for different faults. For example, the broken inner finger fault was an outlier. However, we contribute this observation to the lack of a hyperparameter search for this modified problem. It would be advisable to run a hyperparameter search for this task, before validating that it is truly an outlier. If we only consider the remaining two faults, we do see a pattern that is apparent and could be useful in applications of this work and in future research. The effect that the retention of the replay buffer has on the subsequent adaptation has a common pattern; retention of the replay buffer leads to hindered initial to mid training adaptation. The replay buffer is full of experiences that are representative of the old task, and as such, network updates that use these experiences are biased. In almost all cases, retention of the learned models led to better overall adaptation; the added knowledge aided adaption, even when the policy and value estimates were inaccurate after the onset of a fault. Learning from scratch was a slower process that resulted in poorer initial performance, and in some cases (i.e. for PPO), poorer final performance. Poor performance in real-world applications results in unplanned robot (and machine) behaviour.

In this work, we have examined the ability of two RL policy gradient algorithms to add hardware fault tolerance to a robot. With RL algorithms, a robot learns to perform its task through interaction with its environment. An added hardware fault changes the learning problem by adding restrictions, such as a more restrictive joint range. Provided that the added restrictions do not prevent a robot from performing its task, we believe that adaptation to a hardware fault can occur through the interactive learning process. For this reason, we believe that our work can be extended to other RL policy gradient algorithms, and can likely be extended to value-based RL algorithms. Similarly, we believe that we would see similar adaptation to hardware faults in other robots. In summary, we believe that our work has added information to the problem of adding algorithmic hardware fault tolerance to machines through reinforcement learning and that our work is generalizable to other RL algorithms, other robots (including real-world robots), and other hardware faults.

5.2 Future Work

Re-adaptation to normal robot task. One of the limitations of our work is that after adaptation to a fault occurs, an agent's knowledge about how to behave optimally in the normal environment is lost. RL research that has examined adaptation to hardware faults [9], [12], [57] has not considered the post-adaptation required to restore the system to a normal state after a repair is complete. Although the post-adaptation is expected to be similar to adaptation to faults, many questions do remain. Assume that prior to a hardware fault, a machine is performing optimally. After its adaptation to a fault, and subsequent post-adaptation once a fault is repaired, is it possible to regain its optimal performance through learning? Or does post-adaptation lead to sub-optimal performance? Do we need to save knowledge of the task under normal conditions in memory prior to adapting to a hardware fault? What knowledge needs to be saved?

Gradual faults. In this work, we examined sudden, severe hardware faults. In the real-world, many faults occur gradually, and slowly increase in severity. It would be useful to examine adaptation to gradual hardware faults and determine how adaptation to gradual faults differs from adaptation to sudden, severe hardware faults. It would also be of interest to know if better adaptation can be attained with gradual faults, that slowly reach some pre-selected severity, than that attained with sudden, severe faults of equal severity.

Real-world applications. Additionally, in our work, our robots were simulated and were not real-world robots. It is our intention to extend our work into real-world robots. However, we believe that faster adaption is needed for the real-world. Real robots are costly and easily broken; training for many hours on a real-robot increases the likelihood of degradation or damage. The two algorithms examined in this work, PPO and SAC, required hundreds of thousands of experiences to adapt to a fault. This may not be sufficient for real-world applications. Clavera et al. [9] showed exceptional progress with their

MAML-based algorithm, achieving very fast adaptation; the use of MAML [20] on our problem is an avenue to be explored in future work.

Bibliography

[1] R. O. Ambrose, H. Aldridge, R. S. Askew, R. R. Burridge, W. Bluethmann, M. Diftler, C. Lovchik, D. Magruder, and F. Rehnmark, "Robonaut: Nasa's space humanoid," *IEEE Intelligent Systems and their Applications*, vol. 15, no. 4, pp. 57–63, 2000. DOI: 10.1109/5254.867913.

[2] M. Assad-Uz-Zaman, M. R. Islam, S. Miah, and M. Rahman, "Nao robot for cooperative rehabilitation training," *Journal of Rehabilitation and Assistive Technologies Engineering*, vol. 6, Aug. 2019. DOI: 10.1177/2055668319862151.

[3] M. Bahrin, F. Othman, N. Azli, and M. Talib, "Industry 4.0: A review on industrial automation and robotic," *Jurnal Teknologi*, vol. 78, Jun. 2016. DOI: 10.11113/jt.v78.9285.

[4] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *CoRR*, vol. abs/1606.01540, 2016. arXiv: 1606.01540. [Online]. Available: http://arxiv.org/abs/1606.01540.

[5] J. Carlson and R. R. Murphy, "Reliability analysis of mobile robots," in *2003 IEEE International Conference on Robotics and Automation (Cat. No.03CH37422)*, vol. 1, 2003, 274–281 vol.1. DOI: 10.1109/ROBOT.2003.1241608.

[6] J. Carlson, R. R. Murphy, and A. Nelson, "Follow-up analysis of mobile robot failures," in *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004*, vol. 5, 2004, 4987–4994 Vol.5. DOI: 10.1109/ROBOT.2004.1302508.

[7] J. Carlson and R. Murphy, "How ugvs physically fail in the field," *IEEE Transactions on Robotics*, vol. 21, pp. 423–437, 2005.

[8] Carnegie Mellon University, *Tekkotsu*, http://www.tekkotsu.org/.

[9] I. Clavera, A. Nagabandi, R. S. Fearing, P. Abbeel, S. Levine, and C. Finn, "Learning to adapt: Meta-learning for model-based control," *CoRR*, vol. abs/1803.11347, 2018. arXiv: 1803.11347. [Online]. Available: http://arxiv.org/abs/1803.11347.

[10] E. Coleshill, L. Oshinowo, R. Rembala, B. Bina, D. Rey, and S. Sindelar, "Dextre: Improving maintenance operations on the international space station," *Acta Astronautica*, vol. 64, no. 9, pp. 869–874, 2009, ISSN: 0094-5765. DOI: `https://doi.org/10.1016/j.actaastro.2008.11.011`. [Online]. Available: `http://www.sciencedirect.com/science/article/pii/S0094576508003627`.

[11] F. Cugurullo, "Urban artificial intelligence: From automation to autonomy in the smart city," in *Frontiers in Sustainable Cities*, 2020.

[12] A. Cully, J. Clune, and J.-B. Mouret, "Robots that can adapt like natural animals," *CoRR*, vol. abs/1407.3501, 2014. arXiv: `1407.3501`. [Online]. Available: `http://arxiv.org/abs/1407.3501`.

[13] Cyberbotics, *Portfolio*, `https://cyberbotics.com/#cyberbotics`.

[14] ——, *Velocity control*, `https://cyberbotics.com/doc/reference/motor#velocity-control`.

[15] ——, *Webots reference manual: Hinge2joint*, `https://www.cyberbotics.com/doc/reference/hinge2joint`.

[16] ——, *Webots reference manual: Proto*, `https://cyberbotics.com/doc/reference/proto`.

[17] ——, *Webots: Robot simulator*, `https://cyberbotics.com/`.

[18] David Wiggins and The Open Group Inc., *Xvfb*, `https://www.x.org/releases/X11R7.6/doc/man/man1/Xvfb.1.xhtml`.

[19] A. Davids, "Urban search and rescue robots: From tragedy to technology," *IEEE Intelligent Systems*, vol. 17, no. 2, pp. 81–83, 2002. DOI: `10.1109/MIS.2002.999224`.

[20] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," *CoRR*, vol. abs/1703.03400, 2017. arXiv: `1703.03400`. [Online]. Available: `http://arxiv.org/abs/1703.03400`.

[21] Z. Gao, C. Cecati, and S. X. Ding, "A survey of fault diagnosis and fault-tolerant techniques—part i: Fault diagnosis with model-based and signal-based approaches," *IEEE Transactions on Industrial Electronics*, vol. 62, no. 6, pp. 3757–3767, 2015. DOI: `10.1109/TIE.2015.2417501`.

[22] J. Guiochet, M. Machin, and H. Waeselynck, "Safety-critical advanced robots: A survey," *Robotics and Autonomous Systems*, vol. 94, pp. 43–52, 2017, ISSN: 0921-8890. DOI: `https://doi.org/10.1016/j.robot.2017.04.004`. [Online]. Available: `http://www.sciencedirect.com/science/article/pii/S0921889016300768`.

[23] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," *CoRR*, vol. abs/1801.01290, 2018. arXiv: `1801.01290`. [Online]. Available: `http://arxiv.org/abs/1801.01290`.

[24] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine, "Soft actor-critic algorithms and applications," *CoRR*, vol. abs/1812.05905, 2018. arXiv: 1812.05905. [Online]. Available: http://arxiv.org/abs/1812.05905.

[25] M. Hermann, T. Pentek, and B. Otto, "Design principles for industrie 4.0 scenarios," in *2016 49th Hawaii International Conference on System Sciences (HICSS)*, 2016, pp. 3928–3937. DOI: 10.1109/HICSS.2016.488.

[26] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, *Stable baselines*, https://github.com/hill-a/stable-baselines, 2018.

[27] M. A. I. Iberahim, S. N. Shamsuddin, M. Makhtar, M. Rahman, and N. Simbak, "Time-based simplified denavit-heartenberg translation (ts-dh) for capturing finger kinematic data," *International Journal of Engineering and Technology(UAE)*, vol. 7, pp. 20–23, Aug. 2018. DOI: 10.14419/ijet.v7i3.28.20958.

[28] Ilya Kostrikov, *Github: Pytorch-a2c-ppo-acktr-gail*, https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail.

[29] R. Isermann, "Supervision, fault-detection and fault-diagnosis methods — an introduction," *Control Engineering Practice*, vol. 5, no. 5, pp. 639–652, 1997, ISSN: 0967-0661. DOI: https://doi.org/10.1016/S0967-0661(97)00046-4. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0967066197000464.

[30] R. Isermann, "Fault diagnosis systems an introduction from fault detection to fault tolerance," *SERBIULA (sistema Librum 2.0)*, Jan. 2006.

[31] Jens Fischer, *Gitlab: Simspark*, https://gitlab.com/robocup-sim/SimSpark.

[32] Jon Watte, *Github repository: Tekkotsu*, https://github.com/jwatte/Tekkotsu/tree/master/Tekkotsu.

[33] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017. arXiv: 1412.6980 [cs.LG].

[34] N. Kofinas, E. Orfanoudakis, and M. G. Lagoudakis, "Complete analytical inverse kinematics for nao," in *2013 13th International Conference on Autonomous Robot Systems*, 2013, pp. 1–6. DOI: 10.1109/Robotica.2013.6623524.

[35] J. Korbicz, J. Kościelny, Z. Kowalczuk, and W. Cholewa, *Fault diagnosis. Models, artificial intelligence, applications*. Jan. 2004. DOI: 10.1007/978-3-642-18615-8.

[36] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld, and M. Hoffmann, "Industry 4.0," *Business & Information Systems Engineering*, vol. 6, pp. 239–242, Aug. 2014. DOI: 10.1007/s12599-014-0334-4.

[37] A. R. Mahmood, D. Korenkevych, B. J. Komer, and J. Bergstra, "Setting up a reinforcement learning task with a real-world robot," *CoRR*, vol. abs/1803.07067, 2018. arXiv: 1803.07067. [Online]. Available: http://arxiv.org/abs/1803.07067.

[38] Merriam-Webster, *Fault*, in *Merriam-Webster.com dictionary*. [Online]. Available: https://www.merriam-webster.com/dictionary/fault.

[39] ——, *Kinematics*, in *Merriam-Webster.com dictionary*. [Online]. Available: https://www.merriam-webster.com/dictionary/kinematics.

[40] O. Michel, "Cyberbotics ltd. webots™: Professional mobile robot simulation," *International Journal of Advanced Robotic Systems*, vol. 1, no. 1, p. 5, 2004. DOI: 10.5772/5618. eprint: https://doi.org/10.5772/5618. [Online]. Available: https://doi.org/10.5772/5618.

[41] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *CoRR*, vol. abs/1602.01783, 2016. arXiv: 1602.01783. [Online]. Available: http://arxiv.org/abs/1602.01783.

[42] R. R. Murphy, "Trial by fire [rescue robots]," *IEEE Robotics Automation Magazine*, vol. 11, no. 3, pp. 50–61, 2004. DOI: 10.1109/MRA.2004.1337826.

[43] R. R. Murphy, S. Tadokoro, and A. Kleiner, "Disaster robotics," in *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds. Cham: Springer International Publishing, 2016, pp. 1577–1604, ISBN: 978-3-319-32552-1. DOI: 10.1007/978-3-319-32552-1_60. [Online]. Available: https://doi.org/10.1007/978-3-319-32552-1_60.

[44] N. Kofinas, *Naokinematics*, https://github.com/kouretes/NAOKinematics.

[45] Open Robotics, *Ros*, https://www.ros.org/.

[46] Open Source Robotics Foundation, *Gazebo*, http://gazebosim.org/.

[47] OpenAI, *Descriptions of action spaces & observation spaces #585*, https://github.com/openai/gym/issues/585, 2017.

[48] ——, *Ant-v2*, https://gym.openai.com/envs/Ant-v2/.

[49] ——, *Ant.xml*, https://github.com/openai/gym/blob/master/gym/envs/mujoco/assets/ant.xml.

[50] C. J. J. Paredis and P. K. Khosla, "Designing fault-tolerant manipulators: How many degrees of freedom?" *The International Journal of Robotics Research*, vol. 15, no. 6, pp. 611–628, 1996. DOI: 10.1177/027836499601500606. eprint: https://doi.org/10.1177/027836499601500606. [Online]. Available: https://doi.org/10.1177/027836499601500606.

[51] M. Riazi, O. Zaiane, T. Takeuchi, A. Maltais, J. Günther, and M. Lipsett, "Detecting the onset of machine failure using anomaly detection methods," in *Big Data Analytics and Knowledge Discovery*, C. Ordonez, I.-Y. Song, G. Anderst-Kotsis, A. M. Tjoa, and I. Khalil, Eds., Cham: Springer International Publishing, 2019, pp. 3–12, ISBN: 978-3-030-27520-4.

[52] Ricardo A. Téllez, *Generation of dynamic gaits in real aibo using distributed neural networks*, http://www.ouroboros.org/evo_gaits.html.

[53] Roboti LLC, *Chapter 3: Modeling*, http://mujoco.org/book/modeling.html, 2018.

[54] ——, *Mujoco xml reference*, http://mujoco.org/book/XMLreference.html, 2018.

[55] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017. arXiv: 1707.06347. [Online]. Available: http://arxiv.org/abs/1707.06347.

[56] M. Schwarz, T. Rodehutskors, D. Droeschel, M. Beul, M. Schreiber, N. Araslanov, I. Ivanov, C. Lenz, J. Razlaw, S. Schüller, D. Schwarz, A. Topalidou-Kyniazopoulou, and S. Behnke, "Nimbro rescue: Solving disaster-response tasks through mobile manipulation robot momaro," *CoRR*, vol. abs/1810.01345, 2018. arXiv: 1810.01345. [Online]. Available: http://arxiv.org/abs/1810.01345.

[57] O. Selfridge, R. Sutton, and A. Barto, "Training and tracking in robotics.," in *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 1*, ser. IJCAI'85, Los Angeles, California, Jan. 1985, pp. 670–672.

[58] SoftBank Robotics, *Choregraphe suite*, http://doc.aldebaran.com/2-8/software/choregraphe/index.html.

[59] ——, *Nao*^{6}, https://www.softbankrobotics.com/emea/en/nao.

[60] ——, *Softbank robotics documentation: Actuator & sensor list*, http://doc.aldebaran.com/2-8/family/nao_technical/lola/actuator_sensor_names.html.

[61] ——, *Softbank robotics documentation: Effector & chain definitions*, http://doc.aldebaran.com/2-8/family/nao_technical/bodyparts_naov6.html.

[62] ——, *Softbank robotics documentation: Joint position sensors*, http://doc.aldebaran.com/2-8/family/nao_technical/mre_naov6.html.

[63] ——, *Softbank robotics documentation: Joints*, http://doc.aldebaran.com/2-8/family/nao_technical/joints_naov6.html.

[64] ——, *Softbank robotics documentation: Kinematics data*, http://doc.aldebaran.com/2-8/family/nao_technical/kinematics_naov6.html.

[65] ——, *Softbank robotics documentation: Links*, http://doc.aldebaran.com/2-8/family/nao_technical/links_naov6.html.

[66] ——, *Softbank robotics documentation: Motors*, http://doc.aldebaran.com/2-8/family/nao_technical/motors_naov6.html.

[67] ——, *Softbank robotics documentation: Technical overview*, http://doc.aldebaran.com/2-8/family/nao_technical/index_dev_naov6.html.

[68] G. Steinbauer, "A survey about faults of robots used in robocup," in *RoboCup 2012: Robot Soccer World Cup XVI*, X. Chen, P. Stone, L. E. Sucar, and T. van der Zant, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 344–355, ISBN: 978-3-642-39250-4.

[69] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018, ISBN: 0262039249.

[70] D. Tesar, D. Sreevijayan, and C. Price, "Four-level fault tolerance in manipulator design for space operations," Jun. 1990.

[71] The Learning Agents Research Group, *Github: Utaustinvilla3d*, https://github.com/LARG/utaustinvilla3d.

[72] E. Tira-Thompson, N. S. Halelamien, J. J. Wales, and D. S. Touretzky, "Tekkotsu: Cognitive robotics on the sony aibo," in *Proceedings of the Sixth International Conference on Cognitive Modelling*, Jul. 2004, pp. 390–391.

[73] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012, pp. 5026–5033. DOI: 10.1109/IROS.2012.6386109.

[74] D. Touretzky and E. Tira-Thompson, "Tekkotsu: A framework for aibo cognitive robotics.," Jan. 2005, pp. 1741–1742.

[75] S. Vaidya, P. Ambad, and S. Bhosle, "Industry 4.0 – a glimpse," *Procedia Manufacturing*, vol. 20, pp. 233–238, 2018, 2nd International Conference on Materials, Manufacturing and Design Engineering (iCMMD2017), 11-12 December 2017, MIT Aurangabad, Maharashtra, INDIA, ISSN: 2351-9789. DOI: https://doi.org/10.1016/j.promfg.2018.02.034. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2351978918300672.

[76] M. Visinsky, J. Cavallaro, and I. Walker, "Robotic fault detection and fault tolerance: A survey," *Reliability Engineering & System Safety*, vol. 46, no. 2, pp. 139–158, 1994, ISSN: 0951-8320. DOI: https://doi.org/10.1016/0951-8320(94)90132-5. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0951832094901325.

[77] M. Visinsky, I. Walker, and J. Cavallaro, "Fault detection and fault tolerance in robotics," pp. 262–271, Jan. 1992.

[78] K. H. Williford, K. A. Farley, K. M. Stack, A. C. Allwood, D. Beaty, L. W. Beegle, R. Bhartia, A. J. Brown, M. de la Torre Juarez, S.-E. Hamran, M. H. Hecht, J. A. Hurowitz, J. A. Rodriguez-Manfredi, S. Maurice, S. Milkovich, and R. C. Wiens, "Chapter 11 - the nasa mars 2020 rover mission and the search for extraterrestrial life," in *From Habitability to Life on Mars*, N. A. Cabrol and E. A. Grin, Eds., Elsevier, 2018, pp. 275–308, ISBN: 978-0-12-809935-3. DOI: https://doi.org/10.1016/B978-0-12-809935-3.00010-4. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780128099353000104.

[79] E. Wu, J. Hwang, and J. Chladek, "A failure tolerant joint design for the space shuttle remote manipulator system: Analysis and experiment," in *[Proceedings 1992] The First IEEE Conference on Control Applications*, 1992, 330–335 vol.1. DOI: 10.1109/CCA.1992.269854.

[80] Xpra, *Multi-platform screen and application forwarding system*, https://xpra.org/.

[81] W. Zhang, G. Peng, C. Li, Y. Chen, and Z. Zhang, "A new deep learning model for fault diagnosis with good anti-noise and domain adaptation ability on raw vibration signals," *Sensors*, vol. 17, no. 2, p. 425, Feb. 2017, ISSN: 1424-8220. DOI: 10.3390/s17020425. [Online]. Available: http://dx.doi.org/10.3390/s17020425.

Appendices

Appendix A

NAO V6

This appendix chapter summarizes the NAO V6 specifications. We include kinematics data that was published by SoftBank Robotics [64]. This data includes a summary of all links and joints within the NAO V6 that were relevant to our arm-reaching task. This information is included because it was required to compute the forward kinematics to the NAO's fingertip; these computations are described in detail in Appendix B. We additionally define the default NAO pose, zero pose, and briefly describe how we assign rotation angles within the NAO frame.

A.1 Specifications

The standard NAO V6 weights 5.48 kilograms. It is 0.574 meters tall, 0.275 meters wide and has a depth of 0.311 meters [67]. The NAO has five kinematic chains, which include the head, the left arm, the right arm, the left leg, and the right leg. Each kinematic chain is composed of the joints necessary for movement in that segment of the NAO body. The joints listed within a kinematic chain are ordered. Within the ordered list, two adjacent joints are either 1) directly connected, positioned alongside each other in the NAO body, or 2) connected to each other through a rigid link. For this work, we focus solely on the arm kinematic chains. The kinematic chains for the NAO's left arm and right arm are shown in Table A.1. [61] provides a complete description of all the NAO kinematic chains. The location of the joints within each kinematic chain are shown in Figure A.1.

The NAO has multiple sensory devices. These include two loudspeakers; four omnidirectional microphones; two 5-MP cameras; multiple LEDs; eight force sensitive resistors; an inertial unit, consisting of a 3-axis gyrometer and a 3-axis accelerometer; four sonars; joint position sensors; and tactile sensors. In this work, we utilized the joint position sensors within the NAO arms. The remaining sensory devices were not required for our reaching task.

Each NAO joint contains a motor and a position sensor. The standard NAO V6 has five different types of motors, each with a different speed limitation. The motors that are used in each of the arm joints are listed in Table A.2. [66] provides a complete list of the motor types for all the NAO joints. The NAO joint position sensors are 36 to 2 MRE (Magnetic Rotary Encoders) with 12 bit precision; each joint position sensor has approximately 0.1° precision [62].

The NAO coordinate system is shown in Figure A.2. In this figure, the NAO is in *zero pose*, with its arms fully extended in front, parallel with the ground, and with its hands in a closed position. In zero pose, all arm and finger joint positions are 0.0 radians. Figure A.3 shows the rotation axis of each joint in zero pose.

In zero pose, the NAO's joint names can be misleading. Each joint is named according to its rotation axis when the arm is placed downward, at the side of the NAO frame. Respectively, roll, pitch, and yaw are rotation about the x, y, and z axes. What is considered to be rotation about the z-axis with the arm placed at the side of the NAO frame (i.e. ElbowYaw) is rotation about the x-axis in zero pose (i.e. roll).

The NAO has a pre-defined point on its body, which all other points are relative to, assigned the name Torso. This point, located at the base of the NAO torso, is 0.33309 meters from the ground and is horizontally centred within the NAO frame. SoftBank Robotics has published data that includes the x, y, z offsets of all joints in the NAO robot. The x, y, z offsets for the first link within a kinematic chain are always defined relative to the Torso point; subsequent offsets of joints within a kinematic chain are relative to the previous joint within the chain. Table A.3 shows the offsets for links contained within the left arm, while in zero pose. Offsets for the right arm are unpublished; however, assuming that the NAO frame is perfectly symmetric, the offsets for the right arm would be identical to those for the left arm, with the exception of the Y offsets, which require a change in sign. (The Y-axis is positive to the left of the NAO Torso point, and negative to the right of the NAO Torso point.) Figure A.4 and Table A.4 show additional pre-defined link offsets and their values.

Figure A.5 shows the NAO finger joints for both the left and right hand. Table A.5 includes the offsets for each finger joint. In each finger, the first joint's x, y, z offsets are relative to the WristYaw joint; subsequent finger joint offsets are relative to the previous joint within the finger. The values in Table A.5 are consistent for both the left and right hand due to the numbering of the fingers and finger joints (i.e. there is no need for a change in sign). On the left hand, the outer (leftmost) finger contain the LFinger11, LFinger12, and LFinger13 joints and the inner (rightmost) finger contains the LFinger21, LFinger22, and LFinger23 joints. On the right hand, the outer (rightmost) finger contains the RFinger21, RFinger22, and RFinger23 joints and the inner (leftmost) finger contains the RFinger11, RFinger12, and RFinger13 joints.

[65] provides additional information on the joints contained within the thumb. For this work, we focus solely on the inner finger joints of either hand.

Figures A.6 and A.7 show the rotation trajectory of each joint in the NAO arm, as well as the range for each joint in degrees. Signs of the angles follow the right-hand rule convention - the right thumb points along the rotation axis, and rotation in the direction of the hand's curled fingers is positive rotation, while that in the reverse direction is negative rotation. Tables A.6 and A.7 summarize this information in tabular format, and additionally provide the range of each joint in radians. Figure A.8 shows the rotation trajectory and the range of each joint in a NAO finger. Table A.8 indicates which finger joints have rotations, and informs on their degree of rotation and their rotation axis.

Table A.1: NAO V6 arm kinematic chains.

Kinematic Chain	**Joints**
Left Arm	LShoulderPitch LShoulderRoll LElbowYaw LElbowRoll LWristYaw
Right Arm	RShoulderPitch RShoulderRoll RElbowYaw RElbowRoll RWristYaw

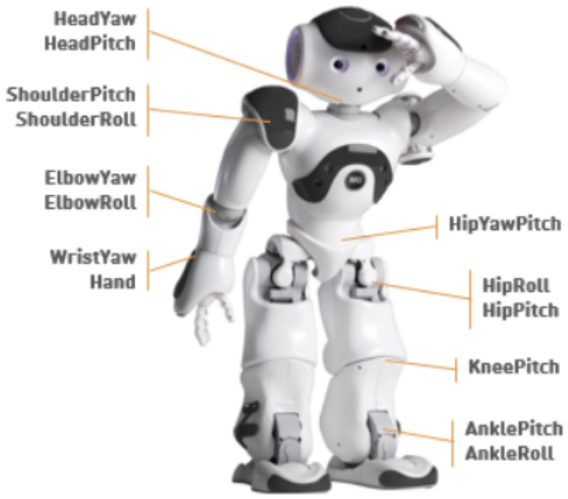


Figure A.1: Joint locations within the NAO V6 body. Figure from [66].

Table A.2: NAO V6 motor types. Table adapted from [66].

	Motor Type 2	**Motor Type 3**	**Motor Type 4**
Model	17N	16GT	DCX 16S
No load speed	8400 rpm ± 12%	10700 rpm ± 10%	11400 rpm ± 10%
Stall torque	9.4 mNm ± 8%	14.3 mNm ± 8%	22.4 mNm ± 10%
Nominal torque	4.9 mNm max	6.2 mNm max	2.6 mNm max
	Speed Reduction Ratio A		
	50.61	150.27	150.27
	WristYaw	ElbowYaw	ShoulderPitch
	Speed Reduction Ratio B		
	36.24	173.22	
	Hand	ShoulderRoll, ElbowRoll	

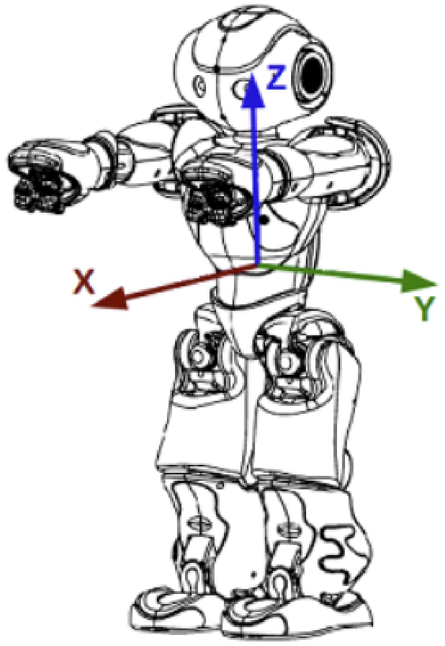


Figure A.2: NAO V6 coordinate system. Figure from [65].

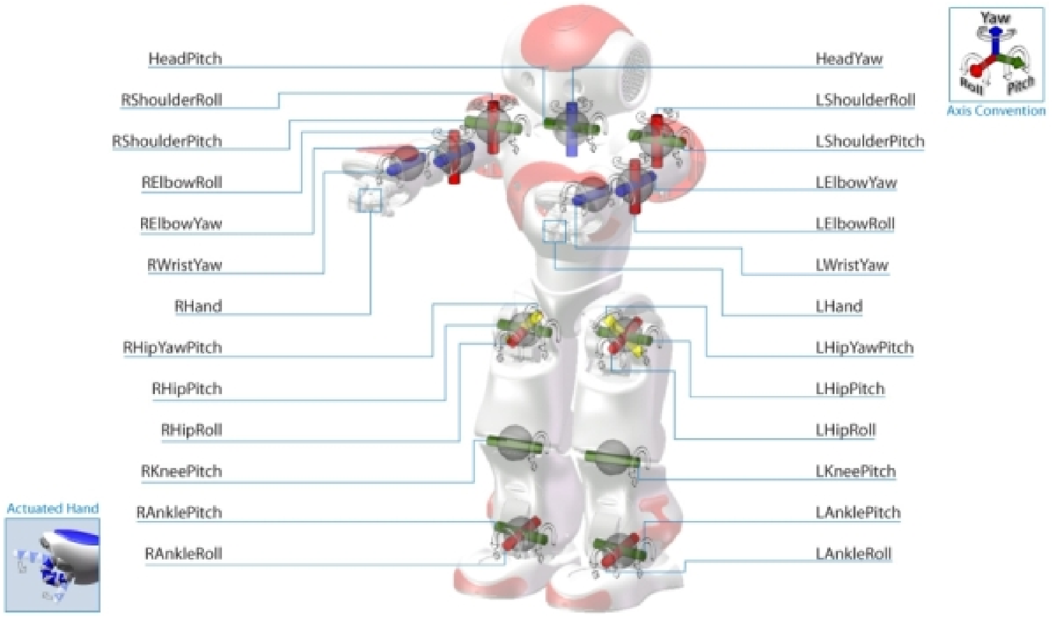


Figure A.3: Rotation axis of joints for NAO V6 in zero pose. Figure from [60].

Table A.3: NAO V6 left arm links. Table from [65].

From ...	**To ...**	**X (mm)**	**Y (mm)**	**Z (mm)**
Torso	LShoulderPitch	0.00	98.00	100.00
LShoulderPitch	LShoulderRoll	0.00	0.00	0.00
LShoulderRoll	LElbowYaw	105.00	15.00	0.00
LElbowYaw	LElbowRoll	0.00	0.00	0.00
LElbowRoll	LWristYaw	55.95	0.00	0.00

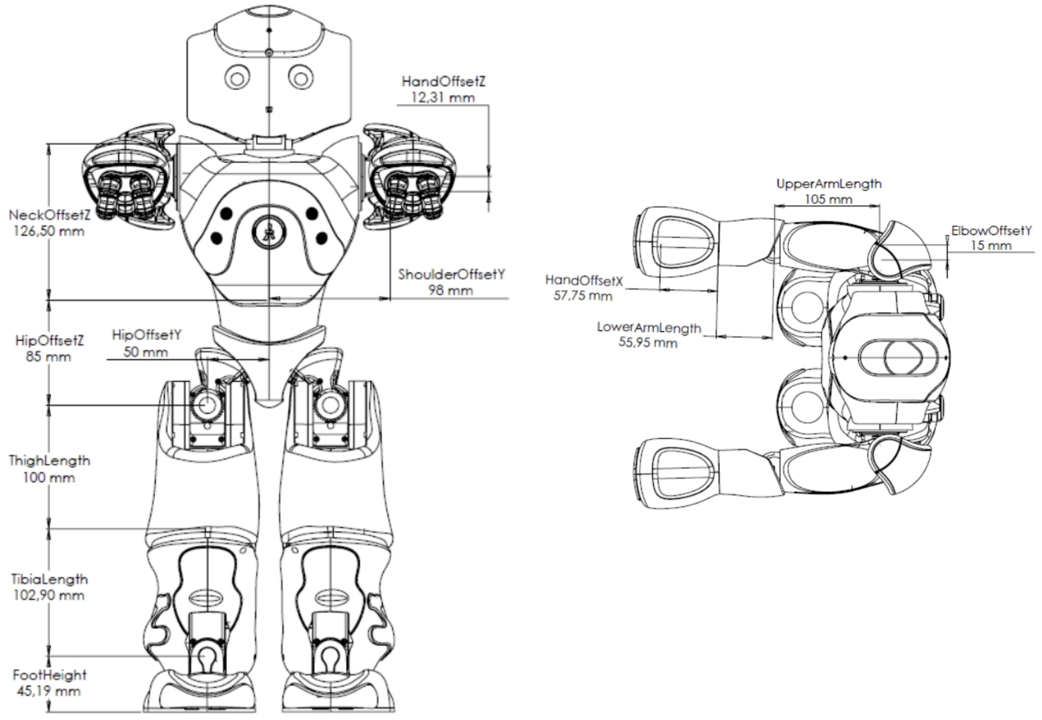


Figure A.4: NAO V6 pre-defined offsets. Figure from [65].

Table A.4: NAO V6 pre-defined offsets. Table from [65].

Main length (mm)	
ShoulderOffsetY	98.00
ElbowOffsetY	15.00
UpperArmLength	105.00
LowerArmLength	55.95
ShoulderOffsetZ	100.00
HandOffsetX	57.75
HandOffsetZ	12.31

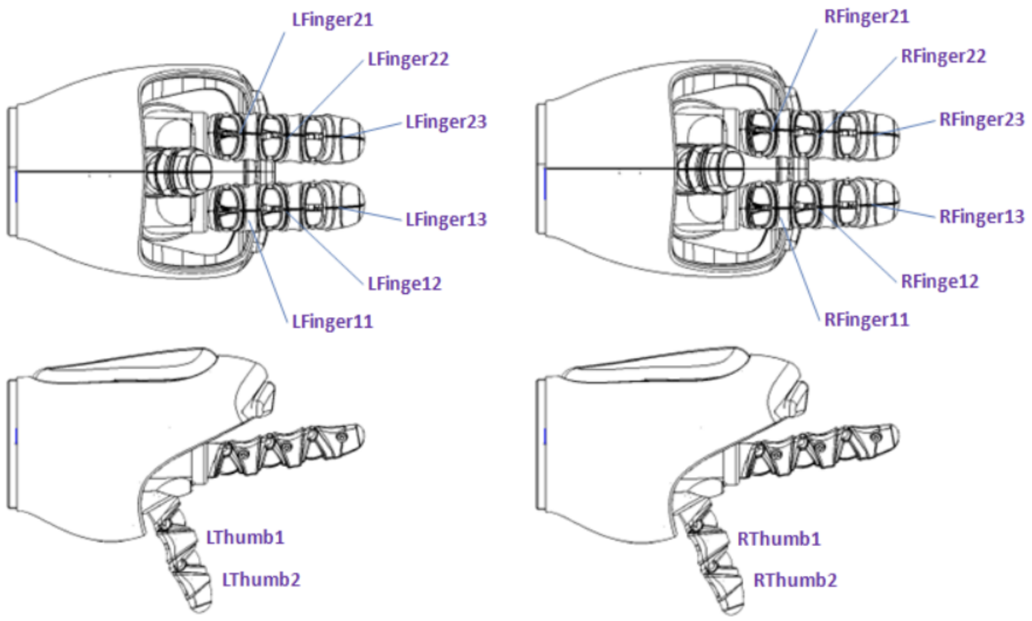


Figure A.5: NAO V6 finger links. Figures from [65].

Table A.5: NAO V6 finger joint offsets. Table adapted from [65].

From ...	**To ...**	**X (mm)**	**Y (mm)**	**Z (mm)**
WristYaw	Finger11	69.07	11.57	-3.04
Finger11	Finger12	14.36	0.00	0.00
Finger12	Finger13	14.36	0.00	0.00
WristYaw	Finger21	69.07	-11.57	-3.04
Finger21	Finger22	14.36	0.00	0.00
Finger22	Finger23	14.36	0.00	0.00
WristYaw	Thumb1	48.95	0.00	-26.38
Thumb1	Thumb2	14.36	0.00	0.00

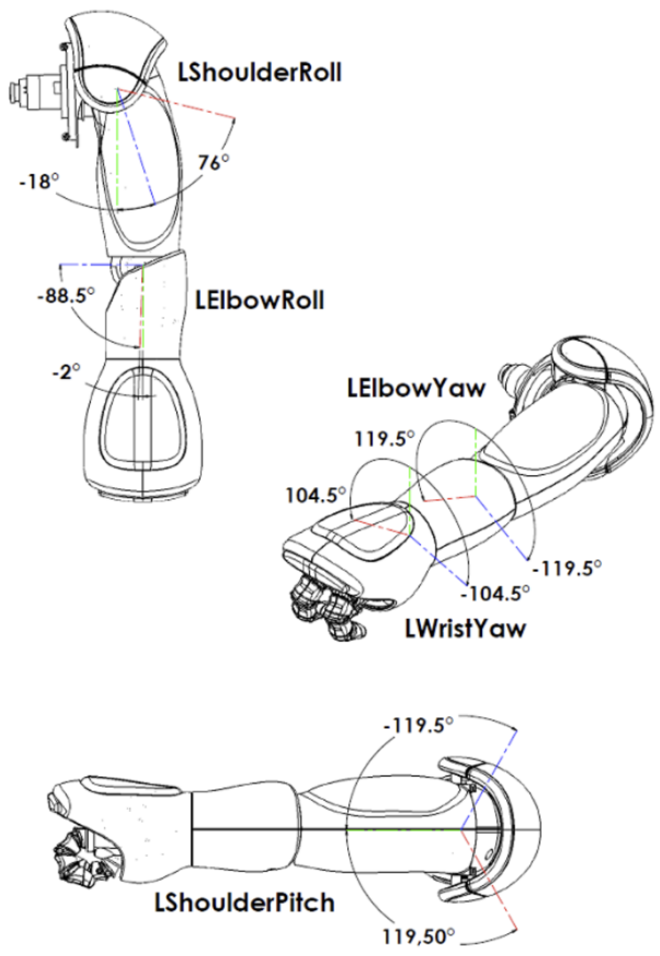


Figure A.6: NAO V6 left arm joint offsets. Figure from [63].

Table A.6: Range of NAO V6 left arm joints. Table from [63].

Joint name	**Range (degrees)**	**Range (radians)**
LShoulderPitch	-119.5 to 119.5	-2.0857 to 2.0857
LShoulderRoll	-18 to 76	-0.3142 to 1.3265
LElbowYaw	-119.5 to 119.5	-2.0857 to 2.0857
LElbowRoll	-88.5 to -2	-1.5446 to -0.0349
LWristYaw	-104.5 to 104.5	-1.8238 to 1.8238
LHand	Open and close	Open and close

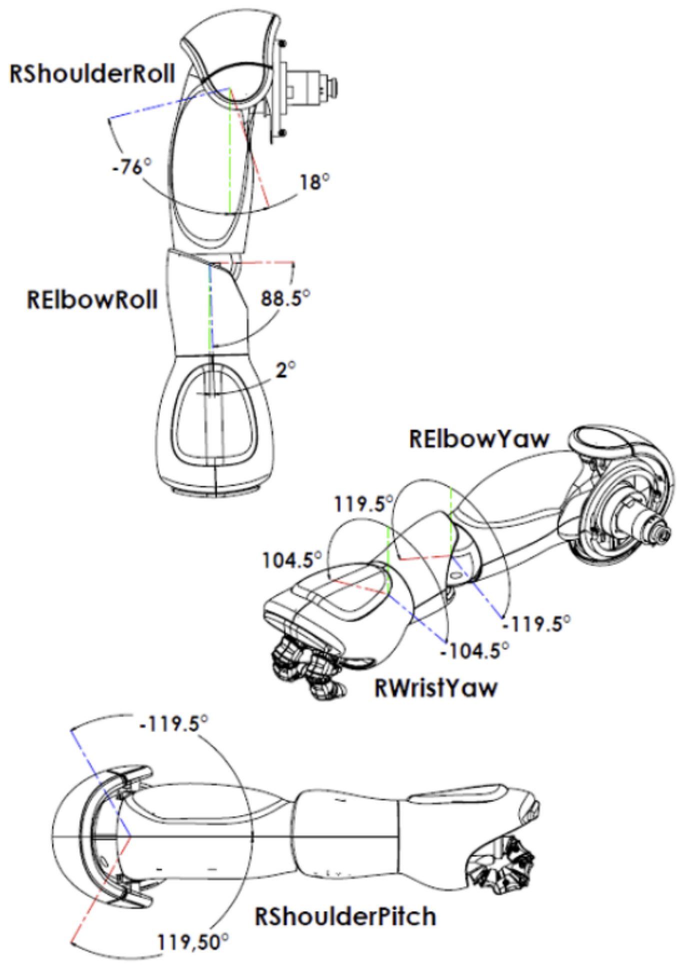


Figure A.7: NAO V6 right arm joints. Figure from [63].

Table A.7: Range of NAO V6 right arm joints. Table from [63].

Joint name	**Range (degrees)**	**Range (radians)**
RShoulderPitch	-119.5 to 119.5	-2.0857 to 2.0857
RShoulderRoll	-76 to 18	-1.3265 to 0.3142
RElbowYaw	-119.5 to 119.5	-2.0857 to 2.0857
RElbowRoll	2 to 88.5	0.0349 to 1.5446
RWristYaw	-104.5 to 104.5	-1.8238 to 1.8238
RHand	Open and close	Open and close

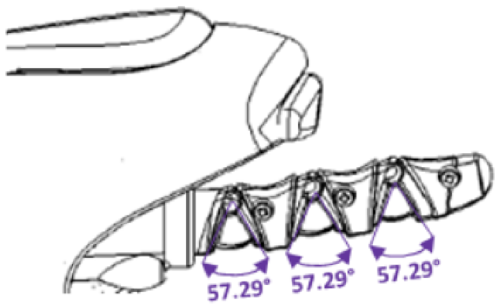


Figure A.8: NAO V6 finger joints. Figure from [65].

Table A.8: NAO V6 finger rotations [65].

Left	
LFinger11	RotX(10.0)
LFinger21	RotX(-10.0)
LThumb1	RotX(180.0)*RotY(-60.0)
Right	
RFinger11	RotX(10.0)
RFinger21	RotX(-10.0)
RThumb1	RotX(180.0)*RotY(-60.0)

Appendix B

NAO Kinematics

In this appendix chapter, we introduce the NAO kinematics calculations. First, we provide a brief summary of the background material required to understand the kinematics calculations. This includes a basic understanding of transformation and rotation matrices. Next, we introduce the Denavit-Hartenberg (DH) method, which is a technique for computing the kinematics of a robot. We briefly describe prior work that has computed the forward kinematics to the NAO hand. Lastly, we provide a detailed explanation on how we modified and extended the prior work to compute the forward kinematics to the NAO inner fingertip.

B.1 Transformation Matrices

A point p in three dimensional space has the coordinates (p_x, p_y, p_z). This point can be represented as the vector $\vec{p}$, where:

$$\vec{p} = \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

An *translation matrix* translates a point in three dimensional space by a fixed distance. The translation distance along the x, y, and z axes are represented by d_x, d_y, and d_z, respectively. A translation matrix has the form:

$$A = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A three dimensional point, p, can be translated by multiplying the translation matrix by the vector $\vec{p}$:

$$\vec{p'} = \begin{bmatrix} p'_x \\ p'_y \\ p'_z \\ 1 \end{bmatrix} = A\vec{p} = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

The new point has coordinates (p'_x, p'_y, p'_z). We represent a translation matrix as $A(d_x, d_y, d_z)$.

An *rotation matrix* rotates a point in three dimensional space. Rotation around the x, y, and z axes by γ, β, and α degrees (or radians) is represented by $R_x(\gamma)$, $R_y(\beta)$, and $R_z(\alpha)$, respectively. $R_x(\gamma)$, $R_y(\beta)$, and $R_z(\alpha)$ are defined as:

$$R_x(\gamma) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\gamma) & -sin(\gamma) & 0 \\ 0 & sin(\gamma) & cos(\gamma) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y(\beta) = \begin{bmatrix} cos(\beta) & 0 & sin(\beta) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(\beta) & 0 & cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z(\alpha) = \begin{bmatrix} cos(\alpha) & -sin(\alpha) & 0 & 0 \\ sin(\alpha) & cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A three dimensional point, p, can be rotated about the x, y, or z axes by multiplying $R_x(\gamma)$, $R_y(\beta)$, or $R_z(\alpha)$ with the vector $\vec{p}$, respectively. For example, to rotate a point p about the x axis, we would perform the following matrix multiplication:

$$\vec{p'} = \begin{bmatrix} p'_x \\ p'_y \\ p'_z \\ 1 \end{bmatrix} = R_x(\gamma)\vec{p} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\gamma) & -sin(\gamma) & 0 \\ 0 & sin(\gamma) & cos(\gamma) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

To rotate a point first about the y axis, then about the z axis, the following matrix multiplication would be performed:

$$\vec{p'} = R_z(\alpha)R_y(\beta)\vec{p}$$

B.2 Denavit–Hartenberg Method

The Denavit-Hartenberg (DH) method is a standardized technique for computing the kinematics of a robot. Forward kinematics involves computing the position and orientation of a robot end effector, given the position of the robot's actuated joints within the end effector's kinematic chain.

To compute the forward kinematics for a robot kinematic chain, each link i in the chain is sequentially assigned a reference frame, F_i. Four parameters, namely the DH parameters, are extracted by comparing each link's reference frame, F_i, to that of the preceding link, F_{i-1}. The reference frame of the first link in a chain, F_1, is compared to a fixed link (base) frame, F_0. The four DH parameters extracted are a_i, α_i, d_i, and θ_i. Once the DH parameters have been extracted, they are input into a transformation matrix, A_i, where:

$$A_i = \begin{bmatrix} cos(\theta_i) & -sin(\theta_i) \cdot cos(\alpha_i) & sin(\theta_i) \cdot sin(\alpha_i) & a_i \cdot cos(\theta_i) \\ sin(\theta_i) & cos(\theta_i) \cdot cos(\alpha_i) & -cos(\theta_i) \cdot sin(\alpha_i) & a_i \cdot sin(\theta_i) \\ 0 & sin(\alpha_i) & cos(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The final representation of the robot's end effector is found multiplying each sequential transformation matrix:

$$T_n^0 = A_1 ... A_n \tag{B.1}$$

In the Denavit-Hartenberg method, reference frames for each link are assigned according to a set of rules:

1. The z-axis is assigned to be the joint's axis of rotation.

2. The x-axis, or the common normal, is perpendicular to both z_i and z_{i-1}.

3. The y-axis is assigned using the right-hand rule.

The four DH parameters for a link i are defined as follows:

$\boldsymbol{a_i}$ is the length of the common normal, or the distance between z_{i-1} and z_i,

$\boldsymbol{\alpha_i}$ is the angle between z_{i-1} and z_i, around the common normal x_i,

$\boldsymbol{d_i}$ is the distance between x_{i-1} and x_i, along z_{i-1}, and

$\boldsymbol{\theta_i}$: the angle around z_i between x_{i-1} and x_i.

[34], [44] provided the DH parameters from the NAO base frame to the hand end effector. These are summarized in Tables B.1 and B.2.

Table B.1: DH parameters for the NAO V6 left arm kinematic chain. Table adapted from [34], [44].

Frame (Joint)	$\boldsymbol{a}$	$\boldsymbol{\alpha}$	$\boldsymbol{d}$	$\boldsymbol{\theta}$
Base	A(0, ShoulderOffsetY, ShoulderOffsetZ)			
LShoulderPitch	0	$-\frac{\pi}{2}$	0	θ_1
LShoulderRoll	0	$\frac{\pi}{2}$	0	$\theta_2 + \frac{\pi}{2}$
LElbowYaw	ElbowOffsetY	$\frac{\pi}{2}$	UpperArmLength	θ_3
LElbowRoll	0	$-\frac{\pi}{2}$	0	θ_4
Rotation	$R_x(-\frac{\pi}{2}), R_z(-\frac{\pi}{2})$			
End Effector	A(LowerArmLength+HandOffsetX, 0, -HandOffsetZ)			

Table B.2: DH parameters for the NAO V6 right arm kinematic chain. Table adapted from [34], [44].

Frame (Joint)	$\boldsymbol{a}$	$\boldsymbol{\alpha}$	$\boldsymbol{d}$	$\boldsymbol{\theta}$
Base	A(0, -ShoulderOffsetY, ShoulderOffsetZ)			
RShoulderPitch	0	$-\frac{\pi}{2}$	0	θ_1
RShoulderRoll	0	$\frac{\pi}{2}$	0	$\theta_2 + \frac{\pi}{2}$
RElbowYaw	-ElbowOffsetY	$\frac{\pi}{2}$	UpperArmLength	θ_3
RElbowRoll	0	$-\frac{\pi}{2}$	0	θ_4
RElbowYaw	0	$\frac{\pi}{2}$	0	θ_5
Rotation	$R_x(-\frac{\pi}{2}), R_z(-\frac{\pi}{2})$			
End Effector	A(LowerArmLength+HandOffsetX, 0, -HandOffsetZ)			

B.3 New Kinematic Calculations

SoftBank robotics' published kinematic data includes offsets for the finger joints, as shown in Table A.5. The WristYaw joint is used as the initial reference point. For this reason, we had to reverse the translation to the hand that was done by [34], [44]. To do this, we removed HandOffsetX and HandOffsetZ from the end effector translation matrix, replacing:

A(LowerArmLength+HandOffsetX, 0, -HandOffsetZ)

with

A(LowerArmLength, 0, 0).

The output matrix for the work done by [34], [44], with this one minor modification, was then used as a base matrix for our kinematics computations. We label this matrix as A_{base}.

We determined that the first step to be taken was to perform a translation from the WristYaw joint to the first finger joint (i.e. knuckle joint). For the left and right arms, this translation matrix was defined as:

$$A_0^{left} = \begin{bmatrix} 1 & 0 & 0 & X_{WristYaw\_to\_LFinger21} \\ 0 & 1 & 0 & Y_{WristYaw\_to\_LFinger21} \\ 0 & 0 & 1 & Z_{WristYaw\_to\_LFinger21} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_0^{right} = \begin{bmatrix} 1 & 0 & 0 & X_{WristYaw\_to\_RFinger11} \\ 0 & 1 & 0 & Y_{WristYaw\_to\_RFinger11} \\ 0 & 0 & 1 & Z_{WristYaw\_to\_RFinger11} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

,

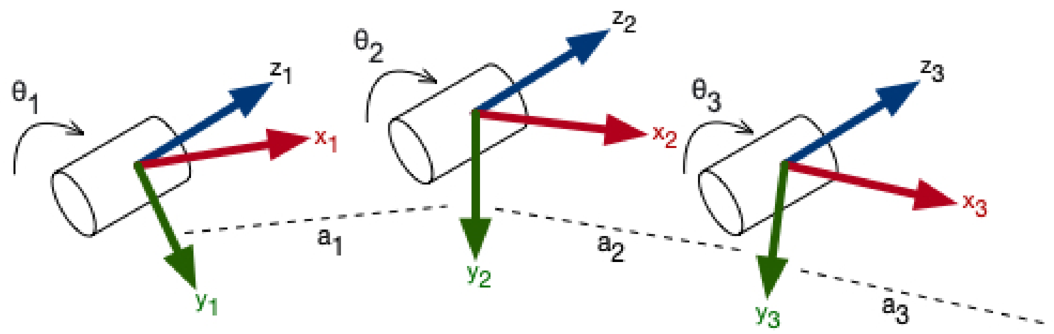


Figure B.1: NAO V6 finger joint reference frames and DH parameters. Figure adapted from [27].

[27] presented steps for computing the kinematics for a virtual, four-jointed finger. This work included a visual image of the reference frame required for their finger kinematics computation. The final reference frame for the WristYaw joint, computed by [34], [44], had the same orientation as the NAO coordinate frame, as shown in Figure A.2. To match the orientation presented by [27], we had to modify the output reference frame; a rotation of $-\frac{\pi}{2}$ radians about the x-axis was required. This rotation is indicated by $R_x(-\frac{\pi}{2})$.

Next, the DH parameters for each of the three finger links were found. Figure B.1 shows the reference frame for each link within the NAO inner finger. The orientation of the reference frame for the base link (not shown) is equivalent to the orientation of the reference frames for these three links. So, the z-axes of all four reference frames are parallel. With parallel z-axes, one DH parameter is given an automatic assignment. Each d_i can take on any value because parallel axes have an infinite number of common normals (we set d_i to zero). Each finger link has the same length, therefore, each a_i is the same, and is given a value of 0.01436 (meters). The DH parameter α_i represents the rotation around each x_i, or link twist. As shown in Table A.8, RFinger11 has a slight rotation: RotX(10.0). Therefore, α_1 for the first right finger link (RFinger11) is radians(10.0), where radians() indicates a conversion from degree to radian units. The remaining links in the right inner finger have no twist, thus α_2 and α_3 are both zero. Finally, we consider the value of θ_i for each link. θ_i represents the rotation around axis z_i required to align the previous x-axis, x_{i-1}, with the current x-axis, x_i. From the figure, it may appear that θ_i is equal to θ_1, θ_2 and θ_3 for the first, second, and third link, respectively. However, the NAO position sensor does not read the angle as shown in Figure B.1. In Figure A.8, it can be seen that when the finger joint sensor reading is at its maximum value of 57.29 degrees, the finger is fully extended and the value for each θ_i is 0. When the finger joint sensor reading is at its minimum value of 0.0 degrees, the finger is fully contracted and the value of each θ_i is at its maximum value of 57.29 degrees. Therefore, each θ_i must be defined as radians(57.29) - θ_i. The DH parameters are used to formulate three transformation matrices: A_1, A_2, and A_3.

The final reference frame was not oriented with the NAO coordinate frame. To change the orientation of this frame to match the NAO coordinate frame, a rotation of $\frac{\pi}{2}$ about the x-axis was performed, indicated by $R_x(\frac{\pi}{2})$.

The final step was matrix multiplication:

$$T = A_{base}A_0^{right/left}R_x(-\tfrac{\pi}{2})A_1A_2A_3R_x(\tfrac{\pi}{2})$$

The position of the end effector (fingertip) was extracted from the matrix T. For this work, we only required the position of the end effector and not the orientation. Therefore, no additional calculations to compute the end effector's orientation were performed.

The DH parameters, and all required rotations and translations, are summarized in Tables B.3 and B.4.

Table B.3: New DH parameters for the NAO V6 left kinematic chain to inner fingertip.

Frame (Joint)	$\boldsymbol{a}$	$\boldsymbol{\alpha}$	$\boldsymbol{d}$	$\boldsymbol{\theta}$
Base	A_{base}			
Knuckle	A_0^{left}			
Rotation	$R_x(-\frac{\pi}{2})$			
LFinger21	0.01436	radians(-10.0)	0	radians(57.29) - θ_6
LFinger22	0.01436	0	0	radians(57.29) - θ_7
LFinger23	0.01436	0	0	radians(57.29) - θ_8
Rotation	$R_x(\frac{\pi}{2})$			

Table B.4: New DH parameters for the NAO V6 right kinematic chain to inner fingertip.

Frame (Joint)	$\boldsymbol{a}$	$\boldsymbol{\alpha}$	$\boldsymbol{d}$	$\boldsymbol{\theta}$
Base	A_{base}			
Knuckle	A_0^{right}			
Rotation	$R_x(-\frac{\pi}{2})$			
RFinger11	0.01436	radians(10.0)	0	radians(57.29) - θ_6
RFinger12	0.01436	0	0	radians(57.29) - θ_7
RFinger13	0.01436	0	0	radians(57.29) - θ_8
Rotation	$R_x(\frac{\pi}{2})$			

Appendix C

Hyper-Parameter Search

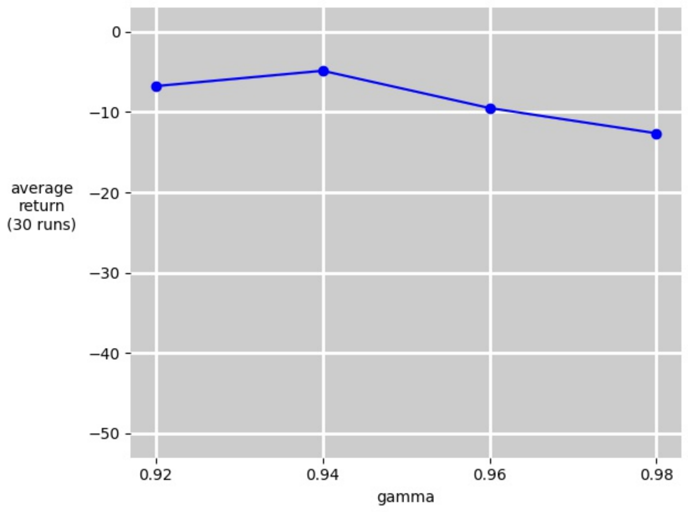
Both PPO and SAC have a large number of hyperparameters. We chose to be selective in our hyperparameter search and target parameters that we believed would have the most significant impact on learning. If we felt that a hyperparameter setting was good, then we would not necessarily target that hyperparameter within our search. We started our hyperparameter search with values that were already known to do well within other people's work [28], or with published values [23], [55].

To compare hyperparameter settings, we computed the average return across the last 100 evaluations in a run, then we averaged the computed average return (for a single run) across 30 runs.

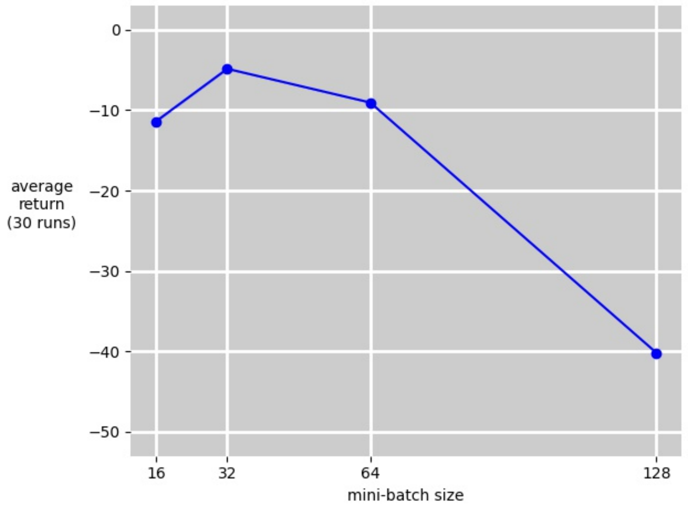
For PPO, we first performed a hyperparameter search for gamma, then once we found a good value for gamma, we performed a search for the parameters number of samples and mini-batch size. We chose to target the number of samples parameter because Minh et al. [41] stated that, for their GAE estimator, the number of samples should be significantly less than the episode length. We wanted to see how learning would be affected by this setting, especially considering that our episode lengths shortened as learning progressed. We did not search for a clipping hyperparameter; we believed that our clipping hyperparameter (set at 0.1) was at a good value as the number of clippings dropped exponentially, to near zero, quite quickly. This reflected that our learning algorithm was not making large updates to the policy. Our hyperparameter search for the NAO task is shown in Figure C.1.

For SAC, we first performed a hyperparameter search for the target smoothing coefficient, tau, and gamma (combined). In SAC, this value is used to update the target network using Polyak averaging. Haarnoja et al. [23] presented the performance for different values of tau, showing that large values of tau could cause lead to very poor performance (i.e. instabilities), while small values of tau could slow the learning speed. After searching for the best value of tau and gamma for our problem, we additionally searched for a value the learning rate.

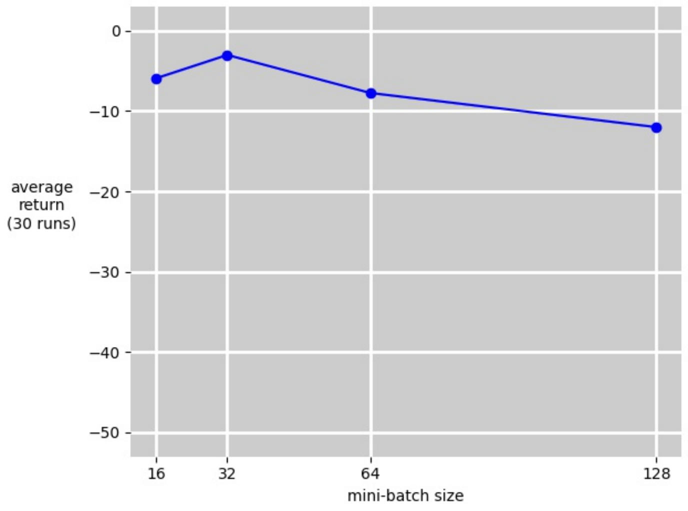
Haarnoja et al. [23] claimed that SAC was robust to all hyperparameter settings, with the exception of the reward scale. The reward scale is responsible for controlling the stochasticity of the policy. Since we added the feature to automatically tuning the entropy [24], the reward scale no longer need manual tuning. Our hyperparameter search for the NAO task is shown in Figures C.2 and C.3. Based on our results for the NAO task hyperparameter search, we can confirm that SAC is indeed robust to the hyperparameters that we examined.



(a) Varying gamma. Number of samples is 128 and mini-batch size is 32.

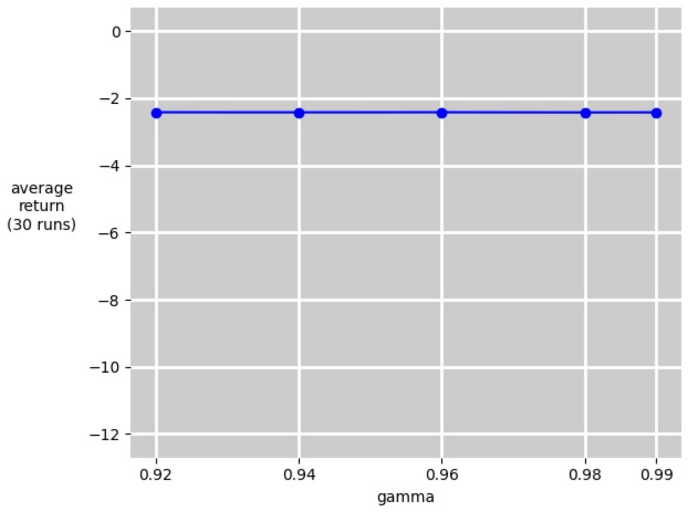


(b) Varying mini-batch size when number of samples is 128. Gamma is 0.94 (best from (a)).

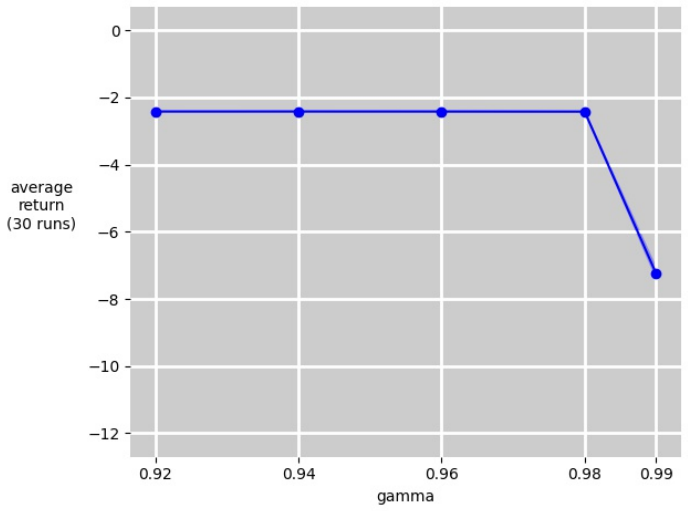


(c) Varying mini-batch size when number of samples is 256, Gamma is 0.94 (best from (a)).

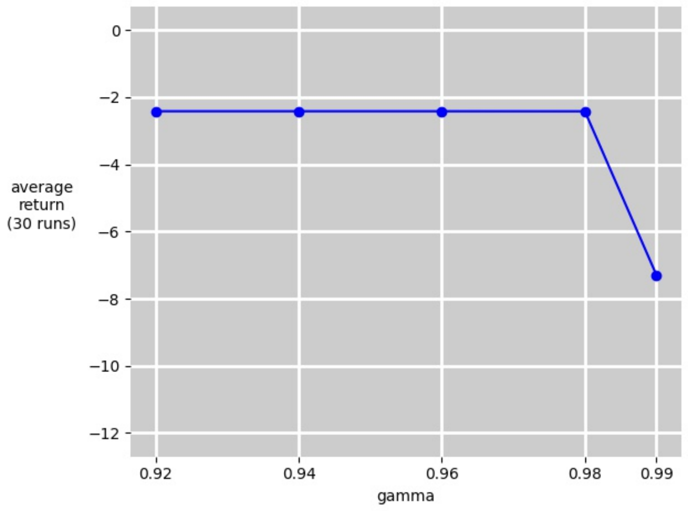
Figure C.1: PPO hyperparameter search for NAO task: gamma, number of samples, and mini-batch size. Best found parameters are gamma = 0.94, number of samples = 256 and mini-batch size = 32.



(a) Varying gamma. Target smoothing coefficient is 0.005.



(b) Varying gamma. Target smoothing coefficient is 0.010.



(c) Varying gamma. Target smoothing coefficient is 0.015.

Figure C.2: SAC hyperparameter search for NAO task: target smoothing coefficient (tau) and gamma. Best found found parameters (so far) are gamma = 0.96 and tau = 0.005.

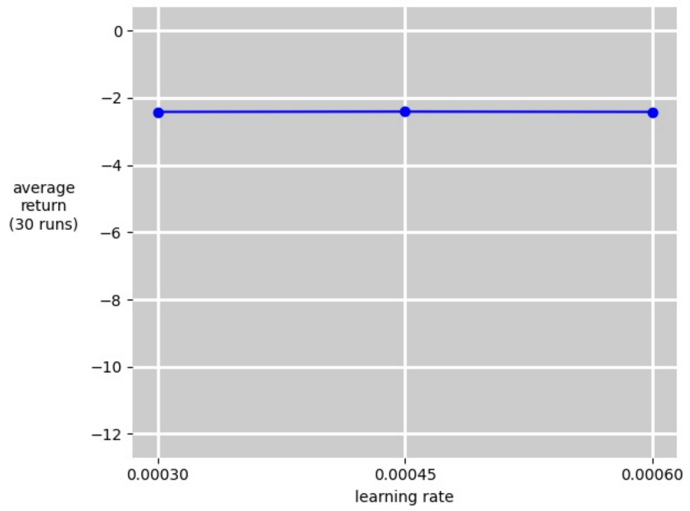


Figure C.3: SAC hyperparameter search for NAO task: learning rate. Gamma is 0.96 and target smoothing coefficient is 0.005 (best from Figure C.2). Best found learning rate is 0.00045, but differences across the learning rates are negligible.

Appendix D

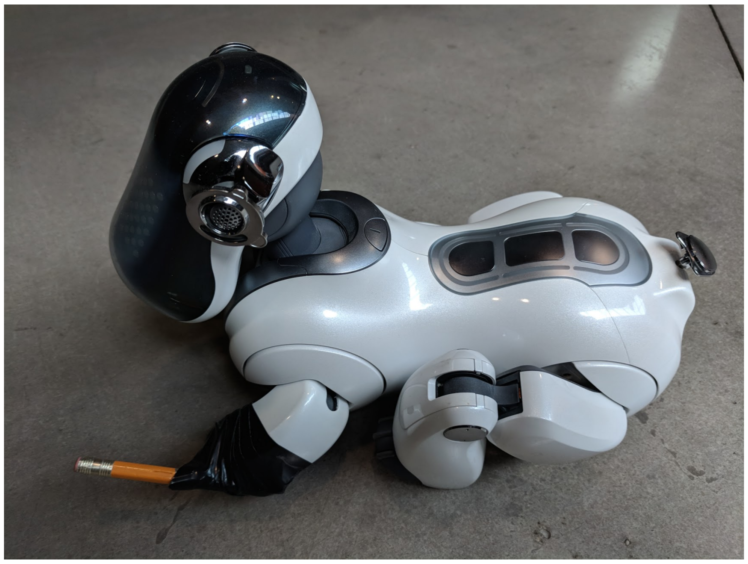
Challenges

This appendix chapter describes challenges that were faced at the start of this research project. We briefly describe our original goal and report on why this goal was not attained. We justify the changes that had to be made to the research project, thereby evolving our work into the form it is in today.

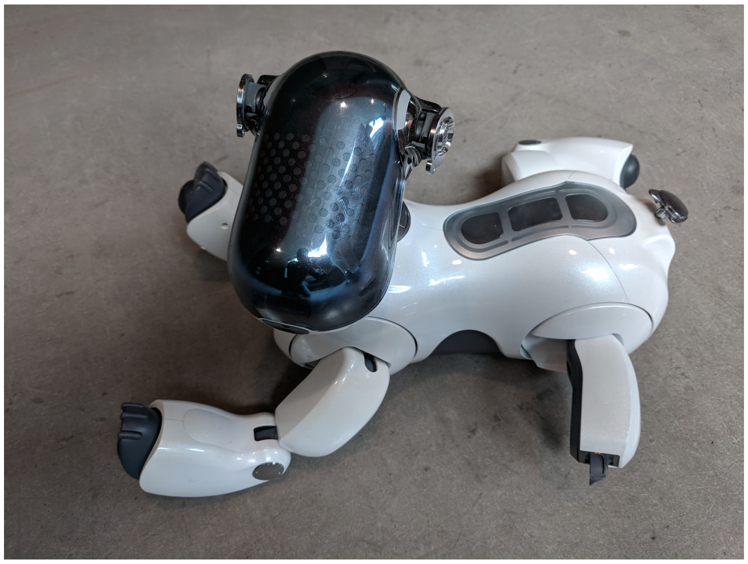
D.1 Sony AIBO ERS-7

At the start of our research, our objective was to use the Sony AIBO ERS-7 robot as a platform to test our ideas. The University of Alberta owned several of these robots, including two that were faulty (shown in Figure D.1). Each of the faulty robots had a single broken leg, where the lower limb was severed and only the upper limb remained attached. We replaced the lower limb of one robot with a half-pencil, firmly affixing it to the robot using electrical tape, and thereby creating a makeshift leg replacement.

AIBO task. Our original objective was to use a simulated Sony AIBO to learn a gait using our chosen RL algorithms. We would then transfer this learned gait to a normal Sony AIBO, and we would allow the real robot to



(a) AIBO with makeshift limb.



(b) AIBO with broken, severed limb.

Figure D.1: Sony AIBO ERS-7 robot.

continue this learning. Once a good gait had been learned in the normal AIBO, we intended to transfer the learned gait to one of the two faulty AIBOs. We would then allow this faulty AIBO to continue training, thereby learning a new gait that accounted for its faulty leg (i.e. adapting).

ROS and Gazebo. Before working with the real robot, it was our intention to find an accurate simulator for the Sony AIBO. Our first attempt at robot control and simulation was with Robot Operating System (ROS) and Gazebo. ROS is an open-source framework for writing control software for a robot [45]. Gazebo is a simulation tool that is compatible with ROS [46]. ROS packages were publicly available for the Sony AIBO; however, they were very old, dating back to 2002. These packages were intended to be used with ROS Kinetic. We made numerous attempts to setup the AIBO within ROS using these packages, however we were always unsuccessful. There were incompatibility errors that we could not resolve.

Tekkotsu. After our failed attempt at simulation with ROS, we looked for an alternative simulation option. This led us to Tekkotsu, an open-source framework for the control and simulation of robots, developed at Carnegie Mellon University [8], [74]. We chose Tekkotsu because of its in-built forward and inverse kinematics solvers for the Sony AIBO robot [72]. The setup of Tekkotsu was by far our most challenging task. First, we were unable to download the most recent version of the Tekkotsu software; a missing file error prevented the software from downloading from the CVS server. We found an older version, Tekkotsu 5.0.4 (2016), in [32]. We attempted to install this software version but were unsuccessful after repeated attempts. The most significant problems were incompatible libraries; we had to downgrade many libraries to make headway in the installation process. Ultimately, persistent problems with the libxml2 library, and the fact that the Tekkotsu software had not been maintained since 2010, led us to the decision to find an alternative option.

Sony OPEN-R SDK. We decided to stop our attempts to find simulation software for the Sony AIBO and instead use the real robot. Sony had released the OPEN-R software development kit (SDK) for the AIBO, written in C++. This SDK gave developers complete control over the AIBO robot. The OPEN-R SDK came with sample programs for the AIBO; these included a ball-tracking program, a sensor observer program and various other sample programs. We were able to run all the sample programs on the normal AIBO successfully. We then opted to find a pre-existing learned walk for the Sony AIBO robot, which we found in [52]. We tested this program on a normal AIBO and found that it was able to walk with this program. Next, we decided to test this walk on the faulty AIBO; this is where we discovered, for the first time, that the faulty AIBOs would not run software. Immediately after turning on the faulty AIBO, it would begin its initialization process, extending its legs and standing up. It would start walking, but then a few seconds into the process, it would signal an error by emitting a loud beep, then collapse to the ground, and subsequently shutdown. We found that the code controlling this safety shutdown mechanism was not accessible to the public, potentially being proprietary to Sony; therefore, we were not able to modify this behaviour. After this realization, we decided to no longer use the Sony AIBO for our work.

D.2 SoftBank Robotics NAO V6

A new robot was purchased for our work, with the intention of testing our ideas on a real-world robot. Again, we wanted to find accurate simulation software for our robot. Prior to purchasing the NAO, we investigated the simulation options for the NAO and discovered that several different simulation options existed. These included Choregraphe [58], ROS and Gazebo [45], [46], a RoboCup 3D soccer simulator named SimSpark [31], and Webots [17]. Choregraphe, which was released by SoftBank Robotics, did not provide physics simulation, and therefore was not an good simulator for our work.

ROS and Gazebo. We first chose to use ROS and Gazebo as our control and simulation platform, given that we had prior experience working with it (with the Sony AIBO). There were no available ROS packages for the NAO, so we had to import the NAO model into the simulation software using SoftBank Robotics' published URDF file. We were able to successfully import the NAO model into the Gazebo simulator and, without gravity enabled, control its joints with ROS; to our disappointment, once gravity was enabled, the robot model would break into its component parts and collapse to the ground. When trying to resolve this issue, it was found that this was a persistent problem with the SoftBank Robotics models (i.e. NAO and Pepper) in Gazebo. No solutions to this problem were found, so we decided to move to another simulator. We had two remaining options that we were certain worked, verified by others who had used them.

SimSpark vs Webots. Before deciding which simulator to use, SimSpark or Webots, we installed them both and examined them. [71] provided an open-source implementation of a NAO agent, in C++, for SimSpark. We setup the SimSpark simulator and ran the open-source code. Everything worked as expected. The simulation software appeared aged, with the visuals being less refined than modern simulators. The open-source implementation gave us three options to setup our simulation. These included the option to add a team of NAOs, a goalie NAO, or a third option that was not relevant to our project. It was clear that this simulation software was targeted for RoboCup. The Webots simulation software, in contrast, was modern and not RoboCup specific. We found that Webots had been supported by SoftBank Robotics up to and including the NAO V5 model. (The V6 model was the first model that was not supported.) Webots was found to have many advantages, which included its capability to run headless in the terminal interface, its ability to simulate a wide range of robots (for future work), and its flexibility in programming languages, allowing users to use Python, C++, or Java. Ultimately, we decided that Webots would be the simulation software that we would use.

NAO task. Initially, we wanted our NAO task to be a gait task; however, after careful consideration, we decided that we should first attempt a simpler task, involving less complicated kinematics. We decided upon an arm-reaching task, leaving our gait task for future work.