Cronfa
Swansea University's Research Repository

Real-Time Hybrid Visual Servoing of a Redundant Manipulator via Deep Reinforcement Learning

Alexander James Williams

Submitted to Swansea University in fulfilment of
the requirements for the Degree of
Master of Engineering

Swansea University
Prifysgol Abertawe

2021

Copyright: The author, Alexander J. Williams, 2023. Released under the terms of a Creative Commons Attribution-Only (CC-BY) License. Third party content is excluded for use under the license terms.

Abstract

Fixtureless assembly may be necessary in some manufacturing tasks and environments due to various constraints but poses challenges for automation due to non-deterministic characteristics not favoured by traditional approaches to industrial automation. Visual servoing methods of robotic control could be effective for sensitive manipulation tasks where the desired end-effector pose can be ascertained via visual cues. Visual data is complex and computationally expensive to process but deep reinforcement learning has shown promise for robotic control in vision-based manipulation tasks. However, these methods are rarely used in industry due to the resources and expertise required to develop application-specific systems and prohibitive training costs. Training reinforcement learning models in simulated environments offers a number of benefits for the development of robust robotic control algorithms by reducing training time and costs, and providing repeatable benchmarks for which algorithms can be tested, developed and eventually deployed on real robotic control environments. In this work, we present a new simulated reinforcement learning environment for developing accurate robotic manipulation control systems in fixtureless environments. Our environment incorporates a contemporary collaborative industrial robot, the KUKA LBR iiwa, with the goal of positioning its end effector in a generic fixtureless environment based on a visual cue. Observational inputs are comprised of the robotic joint positions and velocities, as well as two cameras, whose positioning reflect hybrid visual servoing with one camera attached to the robotic end-effector, and another observing the workspace respectively. We propose a state-of-the-art deep reinforcement learning approach to solving the task environment and make preliminary assessments of the efficacy of this approach to hybrid visual servoing methods for the defined problem environment. We also conduct a series of experiments exploring the hyperparameter space in the proposed reinforcement learning method. Although we could not prove the efficacy of a deep reinforcement approach to solving the task environment with our initial results, we remain confident that such an approach could be feasible to solving this industrial manufacturing challenge and that our contributions in this work in terms of the novel software provide a good basis for the exploration of reinforcement learning approaches to hybrid visual servoing in accurate manufacturing contexts.

DECLARATIONS

This work has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed:

Date: 01/10/2021

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed:

Date: 01/10/2021

I hereby give consent for my thesis, if accepted, to be available for photocopying and for interlibrary loan, and for the title and summary to be made available to outside organisations.

Signed:

Date: 01/10/2021

The University's ethical procedures have been followed and, where appropriate, that ethical approval has been granted.

Signed:

Date: 01/10/2021

Contents

Acknowledgements 5

List of Figures 7

List of Tables 9

List of Algorithms 10

1 Introduction 14
- 1.1 Industry 4.0 and the Factory of the Future 14
- 1.2 A New Wave of HRC-Compatible Industrial Robots 14
- 1.3 Automation in Non-deterministic Environments 15
- 1.4 Challenges in Aerospace Manufacturing 15
- 1.5 Vision-based Control 16
- 1.6 Machine Learning for Visual Servoing 17
- 1.7 Simulating Robotic Control Tasks 17
- 1.8 Research Aims and Scope 18
- 1.9 Chapter Summary 18

2 Literature Review 19
- 2.1 Industrial Robotic Manipulators 19
- 2.2 Computer Vision and Visual Servoing 22
  - 2.2.1 Visual Servoing Methods 22
- 2.3 Artificial Neural Networks & Deep Learning 23
- 2.4 Learning Algorithms 26
  - 2.4.1 Supervised Learning 26
  - 2.4.2 Unsupervised Learning 28
  - 2.4.3 Reinforcement Learning 28
- 2.5 Deep Reinforcement Learning in Robotics 37
  - 2.5.1 Parallelised Algorithms 37
  - 2.5.2 Simulated Environments 38
  - 2.5.3 Transfer Learning and Associated Methods 39
- 2.6 Practical Considerations in Deep RL and Robotics 41
  - 2.6.1 Performance Metrics 41
  - 2.6.2 Reproducibility 41

2.6.3 Ablation Studies . . . 41
2.6.4 Hyperparameter Optimisation . . . 42
2.6.5 Software Implementation . . . 43
2.7 Research Gap Discussion . . . 44
2.8 Novelty . . . 46
2.9 Chapter Summary . . . 47

3 Methodology **48**
3.1 Methodology Overview . . . 48
3.2 Literature Review . . . 50
3.3 Robot Specification . . . 50
3.4 Deep Reinforcement Learning . . . 52
3.5 Simulation . . . 52
3.6 Software Libraries . . . 53
3.6.1 Machine Learning Library . . . 53
3.6.2 Simulation Library . . . 54
3.6.3 Experiment Management Library . . . 54
3.6.4 Programming Language . . . 55
3.7 Experimental Approach . . . 55
3.8 Chapter Summary . . . 56

4 System Design **57**
4.1 Task Description . . . 57
4.2 Environment Modelling - Markov Decision Process . . . 59
4.2.1 States . . . 60
4.2.2 Goals . . . 60
4.2.3 Observations . . . 60
4.2.4 Actions . . . 62
4.2.5 Initial State-Goal Distribution . . . 62
4.2.6 Rewards . . . 62
4.3 Reinforcement Learning Algorithm . . . 66
4.3.1 Learning Algorithm . . . 66
4.3.2 Training Procedure . . . 67
4.3.3 Exploration Policy . . . 67
4.4 Neural Network Architecture . . . 67
4.4.1 Network Description . . . 68
4.4.2 Network Discussion . . . 70
4.5 Chapter Summary . . . 72

5 Software Implementation **73**
5.1 3D Modelling . . . 73
5.2 PyBullet Environment . . . 74
5.2.1 Constructor . . . 76
5.2.2 Constants . . . 77
5.2.3 Observation Spec . . . 78

5.2.4 Action Spec . . . 79
5.2.5 Observation Function . . . 79
5.2.6 Reset Function . . . 79
5.2.7 Step Function . . . 80
5.2.8 Reward Function . . . 81
5.2.9 Goal Generation . . . 82
5.2.10 Termination Function . . . 83
5.3 Chapter Summary . . . 83

6 Experiments and Results **85**
6.1 Experimental Work . . . 85
6.2 Data Collection and Evaluation Metrics . . . 86
6.3 List of Experiments . . . 86
6.4 Results . . . 90
6.4.1 Summary of all experiments . . . 91
6.4.2 Seeding . . . 93
6.5 Discussion . . . 94
6.6 Further Experiments and Ablation Studies . . . 95
6.7 Chapter Summary . . . 96

7 Project Challenges and Discussion **97**
7.1 Environment . . . 97
7.2 Network . . . 99
7.3 Algorithmic Choices . . . 99
7.4 Software Implementation . . . 100
7.5 3D Render . . . 101
7.6 Experiments . . . 101
7.7 Physical Deployment . . . 103
7.8 Chapter Summary . . . 103

8 Conclusions and Future Work **104**
8.1 Conclusion . . . 104
8.2 Future Work . . . 105
8.3 Dissemination . . . 107
8.4 Chapter Summary . . . 107

Appendices **109**

A SAC Algorithm **110**

B Code Repository **112**

C Experiment Data **113**

D Proposed Experiments and Ablation Studies for Future Work 115
D.1 Observation Modularity . . . 115
D.2 Camera Resolution . . . 116
D.3 Sim-2-Real . . . 116
D.4 Reward Sparsity . . . 116
D.5 Initial State-Goal Distribution . . . 117
D.6 Curriculum Learning . . . 118

Acknowledgements

I would first like to thank my supervisors, Dr. Christian Griffiths and Dr. Ian Cameron, for their support, guidance and understanding in writing this thesis and for allowing me the freedom and time to explore this topic in depth. I would also like to thank my colleagues in ASTUTE and at Swansea University for the opportunities to contribute to fantastic research and for all of their support in my academic career to date, as well as being all-round brilliant people to work with. Finally, I would like to thank all of my wonderful family and friends for their love, support and company while I have been writing this thesis, and I would particularly like to thank my parents David and Ceri, without whom I wouldn't have been able to get through this thesis without short-circuiting.

List of Figures

2.1 Architecture of a Feedforward Neural Network with three input units, two output units, and two hidden layers [39] 23
2.2 Structure of a Convolutional Neural Network with 4 hidden layers [41] 25
2.3 Agent-environment interaction in a Markov Decision Process [21, p. 38] 29
2.4 Actor Critic Algorithmic Structure [79] 33

3.1 KUKA LBR iiwa R800 [29] 50

4.1 Task setup. 58
4.2 Goal attainment. **Left**: profile view; **Right**: end-effector view 59
4.3 KUKA LBR iiwa working envelope, side view [142] 65
4.4 KUKA LBR iiwa working envelope, top view [142] 65
4.5 General actor network architecture. 69
4.6 General critic network architecture. 70

5.1 Render of simulated environment in PyBullet GUI showing a typical placement of the floor, target and robot, with windowed eye-in-hand and eye-to-hand camera views in the left panel and joint angles A1:7 in degrees in the right panel. 74

6.1 Specific actor network used for experiments. Key for layer types is shown. Unless stated, the default parameters are used for each layer type as they appear in [171] and [172] respectively. 89
6.2 Specific critic network used for experiments. Key and other parameters as defined in Figure 6.3. 90
6.3 Average return across episodes for all experiments 91
6.4 Average episode length across episodes for all experiments 91
6.5 Loss across episodes for experiments where actor / critic / alpha learning rate = 0.0003 92
6.6 Loss across episodes for all experiments 92
6.7 Average return for experiments where all parameters fixed except for seed 93
6.8 Loss for experiments where all parameters fixed except for seed 93
6.9 Average episode length for experiments where all parameters fixed except for seed 94

A.1 SAC algorithm pseudocode as it appears in [179] 111
B.1 Github repository. 112
C.1 Comet.ML repository with composite graphs and hyperlink access to each experiment. 113
C.2 Location of recorded data in comet interface for an individual experiment 114

List of Tables

2.1 Industrial collaborative robot manipulators and their features. Summarised from [30]. 21

3.1 Maximum range and velocities of motion for KUKA LBR iiwa R800 joints [142]. 51

5.1 Environment customisation options available in class constructor. . . 77
5.2 Environment class constants 77
5.3 Observation Spec Components. 78
5.4 Action Spec 79

6.1 Non-default hyperparameter settings for each experiment. 87
6.2 (Non-exhaustive) list of hyperparameters including training, reinforcement learning, and environment hyperparameters with their function and the default value used for experiments. 88

D.1 Possible valid observation permutations. 116
D.2 Reward Function Combinations. 117
D.3 Initial State-Goal Distribution Function Combinations. 118

List of Algorithms

1 Observation Spec Function 78
2 Observation Function 79
3 Environment Reset Function 80
4 Step Function 80
5 Reward Function 82
6 Random Goal Generation in generateGoal() 83
7 Termination Function 83

Nomenclature

Abreviations

AI Artificial Intelligence

ANN Artificial Neural Network

CNN Convolutional

CPU Central Processing Unit

DoF Degrees of Freedom

EIH Eye-In-Hand (Visual Servoing)

ELU Exponential Linear Unit

ETH Eye-to-Hand (Visual Servoing)

FC Fully Connected

GAN Generative Adversarial Network

GPU Graphics Processing Unit

HRC Human-Robot Collaboration

IDE Integrated Development Environment

MAE Mean Absolute Error

MDP Markov Decision Process

MLP Multi-Layer Perceptron

ML Machine Learning

MSE Mean Squared Error

NN Neural Network

POMDP Partially Observable Markov Decision Process

$ReLU$ Rectified Linear Unit

RL Reinforcement Learning

ROS Robot Operating System

$URDF$ Unified Robot Description Format

VS Visual Servoing

Markov Decision Processes

α Learning rate

γ Discount rate

$\mathcal{A}$ Set of all actions

$\mathcal{G}$ Set of all goals

$\mathcal{R}$ Set of all possible rewards, a finite subset of $\mathbb{R}$

$\mathcal{S}$ Set of all states

$\mathcal{S}_1$ Set of initial states

$\mathcal{T}$ Set of terminal states

π Policy

$\pi*$ Optimal Policy

π_θ Policy with parameters θ

τ Trajectory. e.g. $\tau = (s_0, a_0, s_1, a_1, \ldots, s_n, a_n)$

a Action

$A(s, a)$ Advantage function

E Environment

G Return (cumulative discounted reward) following time t

g Goal

o Observation

$Q(s, a)$ Action-value (Q) function

r Reward

r^* Optimal Reward

s, s' States

s_0 Initial state

T Final time step of an episode

t Discrete time step

$V(s)$ State-value function

Mathematical Notation

$(a_1, \ldots, a_n)$ n-dimensional vector

$<$ Less Than

$>$ Greater Than

$\approx$ Approximately equal

$\geq$ Greater Than Or Equal

$\in$ Is an element of e.g $s \in \mathcal{S}$

$\wedge$ Logical And

$\leq$ Less Than or Equal

$\leftarrow$ Assign to e.g $x \leftarrow 1$

$\vee$ Logical Or

$\mathbb{R}$ Set of real numbers

$\mathbb{R}^n$ n-dimensional real space

$\odot$ Hadamard Product

$\subset$ Subset of e.g $\mathcal{R} \subset \mathbb{R}$

$P(x)$ Probability of x

$\text{argmax}_a f(a)$ A value of a at which $f(a)$ takes its maximal value

Chapter 1

Introduction

1.1 Industry 4.0 and the Factory of the Future

We are in the midst of the fourth industrial revolution, Industry 4.0, where advanced digitalisation within factories is being brought about by a combination of Internet technologies and future-oriented technologies such as 'smart' machines and products which promise to fundamentally shift modern industrial production towards a future where manufacturing is ubiquitously equipped with sensors, actuators, and autonomous systems [1].

This paradigm shift is facilitating progress in industrial practices to transition from mass production towards mass user-driven customisation in manufacturing, where buyers more commonly define the conditions of trade leading to increased individualisation of products, and even individual products. These requirement changes in manufacturing adaptability make flexibility in modern production methods a necessity [1], [2].

Reconfigurable automation technologies such as robots are required to meet this rise in customisation whilst also producing products within a reasonable time-frame and they are expected to be one of the main enablers of the transition to the transformable factory of the future. However, current industrial robotics systems are notoriously difficult to program, which can lead to high changeover times when new products are introduced. In order to remain competitive, the factory of the future needs complete production lines that can be effortlessly reconfigured or repurposed when the need arises and robots that are easily reprogrammable by non-robotics experts for new tasks [2].

1.2 A New Wave of HRC-Compatible Industrial Robots

Robot manipulators, which consist of a series of rigid links connected by motor-actuated joints that extend from a fixed base to an end-effector, are widely used in manufacturing [3] due to their simplicity, flexibility, versatility, general applicability

and reconfigurability.

Traditionally, industrial robotics systems needed to be segregated from human colleagues using extensive safety equipment (for example, physical cages around their working area) in order to comply with safety standards such as ISO 10218-1:2011 [4] and ensure human operator protection. However, there are a growing number of Human-Robot Collaboration (HRC) compatible, collaborative robots, or 'cobots', that have been developed for industrial applications.

The typical features of HRC-compatible robots such as joint torque sensors, kinematically redundant joint structure, reduced velocity, high accuracy, low weight, and a variety of hard-and-software-based safety features [5] bring forth a number of advantages over their non-collaborative industrial counterparts. The most obvious is their ability to work alongside and in the presence of humans without the need for extensive / intrusive safety procedures [6]. This capability admits greater opportunities for semi-automation, where easily-automated task segments may be fulfilled by a cobot, whilst more onerous tasks can be fulfilled by humans, thus allowing production lines to flexibly evolve over time whilst making incremental efficiency savings.

Furthermore, a general advantage of newer robots and typically associated with HRC-compatible robots is that they are programmable in more sophisticated and modern programming languages compared with many traditional manipulators. The ability to program more complex behaviour with ease becomes necessary in the domain of human-interactive environments and to automate complex and variable tasks.

1.3 Automation in Non-deterministic Environments

Robots perform best at highly repetitive tasks and as such automation is typically implemented in deterministic environments. This is achieved by the application of mechanical jigs and fixtures that precisely position each workpiece that the robot is expected to interact with, but jigs are expensive, time-consuming to manufacture and arrange, occupy space, and thus hindering to flexible or reconfigurable automation lines [7, p. 455],[8].

In contrast, fixtureless assembly environments are uncertain for robots to operate in but are often necessary due to spatial constraints imposed by a plant, or by irregular or sizeable work piece geometries rendering the installation of specialised tools and fixtures to be infeasible or expensive. This makes the implementation of automation systems in such environments challenging.

An example of such a manufacturing environment can be found in the aerospace manufacturing industry where the enormous workpieces, limited assembly space, and the relatively high accuracy and precision needed for assembly [9] impose significant barriers to any potential automation solutions.

1.4 Challenges in Aerospace Manufacturing

The aerospace industry is under pressure to automate its manufacturing processes to induce higher productivity, improve aircraft quality, and circumvent ergonomic

challenges for the workers that construct them. Due to recent developments in their rigidity and accuracy, manufacturing robots have become the lower-cost tools of choice for many aerospace manufacturing operations including drilling, fastening, sealing and painting due to their repeatability and ability to outperform human workers at these tasks [10]. According to Brumson, drilling holes into components was the largest use of robotics in the aerospace industry in 2012. Aircraft need thousands of holes drilled precisely with sub-mm accuracies of +/-0.25 mm [9]. Drilling these holes manually quickly becomes infeasible as the process requires a skilled and experienced worker [11]. Furthermore, repetitive tasks such as this can cause ergonomic challenges for employees, causing injuries such as RSI [10].

Aircraft manufacturing lines can have a lifetime of several decades, so robotic solutions that are to be integrated into existing production environments should have little or no dedicated physical infrastructure due to space and weight restrictions typical within factories [12]. In 2016, Airbus issued the ”Shopfloor Challenge” with the aim of finding a robot capable of drilling holes in aircraft wings at the required accuracy with suitably fast speed with minimal setup cost and downtime. The issuing of this industrial challenge clearly indicates the need for solutions to this problem [12]. While the results of the competition were promising, no solution was declared by Airbus to be unilaterally successful or satisfactory [13] and this is still considered an unsolved problem in the industry.

1.5 Vision-based Control

One potential solution to produce flexible closed loop control in fixtureless assembly environments is vision-based control, or visual servoing (VS), whereby information from one or more cameras or other visual sensors [14] can be used to guide a robot in order to achieve a task, resulting in a dynamic adaptive control system [7], [15], [16]. Greater control accuracy and environmental awareness can be achieved by employing a hybrid visual servoing approach whereby two vision sensors provide input to a control system, with one sensor attached to the robot's end-effector for accurate positioning and another observing the robot in its work environment for greater positional awareness.

The maturity of computer vision (CV) applications, which has achieved great strides in recent years, means that suitably processed vision data can yield complex environmental insight way beyond the basic positional awareness and immediate touch of robot manipulators equipped with joint-torque sensors [16]. Vision applications for industrial manufacturing are wide-ranging. From interpretation of work surface and component geometry [17], to locating relevant points of interest [18], and detecting external persons or items within the workspace [19].

Therefore, incorporating vision sensors into robots with kinematic redundancy could be key to allowing operation in unstructured or dynamically changing environments that characterise advanced industrial applications [20] such as aerospace manufacturing, and it is possible that a new wave of HRC-compatible redundant manipulators such as the KUKA LBR iiwa can be successfully applied to automation in

uncertain fixtureless environments if equipped with visual servoing technologies.

One of the main caveats of using vision sensing is in the complexity and difficulty of interpreting image data using computational methods and the size of data obtained from vision sensors which makes methods computationally expensive and can delay robotic responses. Artificial Neural Networks (ANNs) trained with Machine Learning (ML) methods are able to provide solutions with approximations nearing more accurate (and computationally expensive) mathematical models [16], providing computation to complex problems in a fraction of the time.

1.6 Machine Learning for Visual Servoing

ML and ANNs have been successfully applied to many complex image processing and robotic control tasks. In particular, the paradigm of Reinforcement Learning (RL) - which frames problems as interactions between an agent and its environment in which it receives feedback in order to achieve a goal, has found particular success in robotics due to its natural encapsulation of control problems via Markov Decision Processes [21, p. 55].

Still, applying AI to robotics systems is challenging for several reasons including the high dimensionality of control problems, high performance requirements (particularly in safety-critical systems), and factors associated with physical embodiment that bring real-world unpredictability and physical and timing constraints. Adequate performance in the physical world, however, is essential for any robotics controller so that it is robust to the full set of dynamics and possible disturbances [22].

1.7 Simulating Robotic Control Tasks

Simulators are widely used in the robotics community and in RL as they allow for real world systems to be quickly and cheaply prototyped without the need for physical access to hardware. Simulations are particularly advantageous in robotic learning research as the huge data requirement for deep RL approaches means it can be more effective to take advantage of increasingly powerful and ubiquitous computational resources to cheaply and quickly generate synthetic data to accelerate the learning as opposed to the high costs intrinsic to real-world data collection experiments. Simulation also carries numerous advantages [23], including alleviating risk of damage to real-world hardware, faster than real-time operation and parallelisation, instant access to robots without purchase, easy reconfigurability and limited requirements for human intervention.

Although simulations cannot capture all dynamics of a real-world environment, they can provide a basis for evaluating the feasibility of control algorithms for specific tasks. Furthermore, control systems trained in simulation can often be fine-tuned to operate in the physical world with a greatly reduced training requirement than training in the physical environment from scratch [24]. Simulations of control tasks therefore offer a valuable basis by which to develop complex robotic control systems.

1.8 Research Aims and Scope

The overarching aim and motivation of this work is that we wish to generate a solution to automating accurate fixtureless manufacturing tasks in dynamic, stochastic, non-deterministic environments. Specifically, we are concerned with addressing challenges in the automation of highly accurate aerospace manufacturing tasks such as drilling, as encapsulated by the Airbus Shopfloor Challenge [13]. Whilst solving addressing these challenges in its entirety is far beyond the scope of this research work, in this work we would like to be able to examine the potential of particular control algorithms to this task environment, as a first-step to developing automation systems in this domain. Assessments of the efficacy of different robotic control approaches can take place utilising simulations of the task environment, with the view that any control policy successfully derived via simulation can potentially be applied to a physical robotic control system with some further tuning.

We hypothesise that a control policy for a dextrous and collaborative robotic manipulator to act in such an environment can be achieved using a dual-camera visual servoing-based control methodology and derived through a deep RL algorithm and using an ANN to model the control policy. Simulations are commonly employed in RL research for testing the feasibility of and optimising RL learning algorithms to particular tasks, but at present, no such simulations exist for multi-camera visual servoing for accurate dextrous robotic manipulation in fixtureless manufacturing tasks. In this work, we seek to develop a simulation that accurately reflects the challenges of an aerospace fixtureless manufacturing environment, and evaluate whether this complex control task can be solved using a deep RL learning approach applied to a dextrous collaborative robot employing hybrid visual servoing.

1.9 Chapter Summary

In this chapter,we have introduced the background to this research, touching on Industry 4.0, collaborative robots, fixtureless manufacturing environments such as aerospace manufacturing and the challenges they pose to automation, visual servoing for robotic control and the application of ML to this domain, as well as the value of simulations for deriving ML-based control policies. We then identified robotic control limitations in fixtureless manufacturing environments, specifically in aerospace manufacturing tasks such as drilling, exemplified by the Airbus Shopfloor Challenge. We have hypothesised a potential solution to this robotic manufacturing challenge in the form of a hybrid visual servoing control policy, derived via deep RL and set out our research aims. In the following section, we perform a literature review of the state-of-the-art as it relates to our research aims.

Chapter 2

Literature Review

In this chapter, we will conduct a literature review of the state-of-the-art as it relates to industrial robotic control. The review is structured as follows: first we review the state of contemporary robotic manipulator technology including operation and models; next we review the state-of-the-art in computer vision and visual servoing methods; after this, we review the theory and state-of-the-art of ML and ANNs, with a particular focus on RL; we then dive into the state-of-the-art research in deep RL for robotics including algorithms, simulations, and simulation-to-reality transfer methods before reviewing practical considerations in deep RL and robotics research including as experimental best practice and software implementation libraries; finally, we discuss gaps in existing research and propose a novel solution to the challenges of automation in aerospace manufacturing tasks.

2.1 Industrial Robotic Manipulators

Robot manipulators are widely used in manufacturing [3] due to their simplicity, flexibility, versatility, general applicability and reconfigurability. Robotic manipulators consist of a series of rigid links connected by motor-actuated joints that extend from a fixed base to an end-effector. Most manipulators built for general application have six degrees of freedom (DoF) which allow them to position their end-effector at any point in 3D space within its reach. Each degree of freedom corresponds to translational movement in the X, Y, Z axes as well as rotational movement about each of these axes.

There are a growing number of Human-Robot Collaboration (HRC) compatible, collaborative robots, or 'cobots', that have been developed for industrial applications. Collaborative robots is a relatively recent and active field of research with the main aims of researchers and manufacturers being to improve the aspect of human safety during human-robot interaction while increasing payload capacity and maintaining and enhancing the mobility and flexibility of these robots [25], with a close objective of achieving systems that can recognise, work with, and adapt to human or other robot behaviours in unstructured environments [26]. Since 2008, this technology started to emerge in industry with early cobot examples such as Universal Robots UR5

[27], Rethink Robotics Baxter [28], and KUKA LBR iiwa [29]. Many manufacturers have now released their own cobots [30], with some of the major contributors and manufactures in the field being Universal Robotics, Rethink Robotics, ABB, KUKA, and Omron [25]. Hentout et al. surveyed a range of industrial collaborative robots, along with their specification and applications. We summarise this survey, with the addition of the KUKA KMR iiwa [31] and Fetch Mobile Manipulator [141], in Table 2.1. A detailed overview of the typical architectures and features of collaborative robots is also given in [5].

Several of the aforementioned robots, such as the LBR iiwa, Baxter, and Yumi IRB robots are classified as kinematically redundant manipulators as they possess seven DoF, compared to traditional manipulators which have six DoF respectively. A kinematically redundant manipulator may be defined as a robot manipulator that possesses more joints than those required to execute its task such that the same task at the end-effector level can be executed in several different ways at the joint level. As 6 DoF is required for a manipulator to reach any point in a 3D workspace, a seventh degree of freedom provides redundancy that allows the manipulator to reach any point in its workspace in an infinite number of ways. This provides the robot with an increased level of dexterity that may be used to avoid singularities, specific joint limits, and workspace obstacles, as well as minimise joint torque and energy usage or in general optimise appropriate performance indexes [20], but also introduces additional complexity into the inverse kinematics calculations required for motion planning.

While traditional industrial robots have required extensive safety infrastructure to avoid human injury due to their high speeds and must operate in more open and predictable workspaces due to their more limited six DoF and sensing capabilities, HRC-compatible robot manipulators can operate in more inhibitive spaces with minimal safety infrastructure and in the presence of humans due to their kinematic redundancy, joint-torque sensors that can detect collisions with sensitivity, and lighter weight [29]. This allows them to be deployed with more flexibility and thus faster changeover times, and even opens them up to applications in mobile robotics as demonstrated with the KUKA KMR iiwa [31].

Finally, a general advantage of newer robots and typically associated with HRC-compatible robots is that they are programmable in more sophisticated and modern programming languages than many many traditional manipulators. For example, software for the KUKA LBR iiwa is written in the Java programming language and simple programming and configuration is made possible in the Sunrise Workbench IDE (Integrated Development Environment). Java is an extremely popular object-orientated language with a thriving ecosystem of documentation and support. Its application to an industrial robot means that realising complex algorithms and interfacing with a range of external devices suddenly became much easier compared to traditional industrial robotic programming languages such as KUKA Robot Language (KRL). Furthermore, other industrial cobots such as the Universal Robotics UR series may be programmed in the Python scripting programming language which is considered to be intuitive, especially for those new to programming.

Robot	Company	Specifications	Sensors
Yumi IRB 14000	ABB	Dual-arm body (7-DoF each); Action resumption only by human remote control; Collision-free path;	Camera-based object tracking; Collision detection through force sensor;
LBR iiwa	KUKA	Contact detection; Velocity and force reduction on collision; Single arm with 7-DoF;	Torque sensors; Force sensors;
KMR iiwa	KUKA	LBR iiwa with omnidirectional mobile base;	Laser sensors
Baxter	Rethink Robotics	Dual-arm (7 + 7-dof);	Embedded torque sensors; Integrated camera per arm; Vision-guided movement and object detection; 360 degrees sonar; Front camera;
UR3 510	Universal Robots	6-DoF in single arm; Collision detection; Robot stops upon collision; Speed reduction to 20%;	Force sensors; Speed reduction in directly programming;
Robonaut	NASA	Dual arms with complete hands and fingers; Each arm has 7-DoF; Each finger has 3-DoF; Elastic joints;	Stereo-vision camera; Infrared camera; High-resolution auxiliary cameras; Miniaturised 6-axis load cells; Force sensing in joints;
APAS	Bosch	6-DoF;	Touchless triggering sensor; 3D stereo camera; 2D monochrome camera;
COMAU dual-arm robot	COMAU	Dual anthropomorphous arms with 6-DoF;	Proximity and tactile sensors; Vision system; Force/torque sensor;
Rob@Work 3	Fraunhofer IPA	6-DoF arm; Omnidirectional mobile base;	3D camera; Stereo-camera systems; Laser scanners;
DLR-LWR III	Institute of Robotics and Mechatronics	7-DoF;	Joint torque sensors; Redundant position sensing; Wrist force-torque sensor;
DLR LWR 4/5	KUKA-DLR	7-DoF;	Torque sensors;
Mobile Manipulator	Fetch Robotics	7-Dof arm with gripper and omnidirectional mobile base	Laser sensors; Stereoscopic camera;

Table 2.1: Industrial collaborative robot manipulators and their features. Summarised from [30].

2.2 Computer Vision and Visual Servoing

Computer Vision includes methods for processing and understanding images, to produce numeric or symbolic information about an environment [32]. Computer vision has gained popularity in robotics applications due to the emergence of low-cost 2D and 3D cameras, and breakthroughs in computer vision image processing tasks including image classification, object detection, image segmentation, path planning, trajectory generation, environment mapping, as well as visual servoing [33]–[35]. According to Cherubini, It is one of the most common sensor-based control methodologies for collaborative robot applications, along with touch-based, audio-based, and distance-based control [32].

Since 2012, and the breakthrough by Krizhevsky et al. in using artificial neural networks for image classification [19], as well as the increasing availability of greater computer processing power it has been observed that research into techniques employing machine learning and ANNs in the visual servoing domain has increased rapidly. According to [33], 98% of research in visual servoing in the year 2021 employed some kind of AI-based technique. Singh states that recently, machine learning methods combined with optimisation algorithms like model predictive control, genetic algorithms, or Kalman filter techniques are the current state-of-the-art for predicting optimal and efficient robot trajectories in vision-guided robotics systems but comments on the extent of research happening in the field of deep reinforcement learning in this research context, viewing it as a future-oriented technology where robots learn tasks with minimal human intervention [33].

Further detail and applications relating to sensor-based robotic control, including vision-based control can be found in [32] and [33]. We also direct the reader to the works of Feng et al. for an overview of state-of-the-art computer vision algorithms for different image processing tasks [34] and Zhou et al. for an overview of computer vision applications in manufacturing[35].

2.2.1 Visual Servoing Methods

In terms of camera positioning, there are two possible visual servoing configurations for robot manipulators [16], [32], [36], [37]:

- **Eye-to-Hand** An external vision system observes the robot in its environment and provides perception of both the robot and its wider environment. As these approaches are necessarily situated at a distance from the robot, their greater awareness comes at a cost of diminished accuracy and potential view obstruction by the robot itself.

- **Eye-in-Hand** A local vision system is attached to the robotic end-effector or "hand". This results in a dynamically framed visual input that varies according to the robot's specific movements. These methods provide a local perception concerned with the positioning of the robotic end-effector which limits environmental awareness, but yields detailed feedback and thus greater end-effector accuracy.

These methods can be combined to great effect. In the work of [16], a two camera approach is used for visual guidance and feedback in a manipulator-based pick and place task. The robot, its workspace, and the target are in the field of view of a fixed camera providing visual guidance for the control system, meanwhile another camera attached to the end effector offers feedback. This set up provides the advantage that even when the end-effector camera cannot see the target due to its specific configuration, the target remains within the field-of-view of the fixed camera. Therefore, the robot is directed towards the goal with the eye-to-hand camera, and then switches to visual feedback control when the object is fully visible for the eye-in-hand camera.

Furthermore, the addition of robotic joint position inputs, combined with visual inputs is established. Termed '2.5D Visual Servoing', the technique has been shown to improve the stability, convergence and overall robustness of visual servoing systems [38]. Other decompositions of visual servoing are given in [38].

2.3 Artificial Neural Networks & Deep Learning

An **Artificial Neural Network** (ANN) is a computational model inspired by how human and animal brains process information and learn from experience. They consist of connected layers of simple computational units known as neurons that process weighted input values and generate an output vector based on an internal activation function.

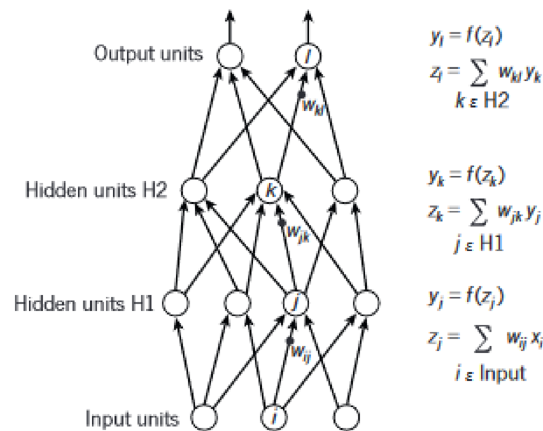

Figure 2.1: Architecture of a Feedforward Neural Network with three input units, two output units, and two hidden layers [39]

Neural networks are composed of nodes or units connected by directed links. A link from unit i to unit j serves to propagate the activation a_i from i to j. Each link has a numeric weight $w_{i,j}$ associated with it, which determines the strength and sign of the connection. Each unit j computes a weighted sum of its inputs:

$$in_j = \sum_{i=0}^{n} w_{i,j} a_i \tag{2.1}$$

Then applies an activation function g to this sum to derive the output [40, p. 728]:

$$a_j = g(in_j) = g(\sum_{i=0}^{n} w_{i,j} a_i) \tag{2.2}$$

Several common activation functions include sigmoid, tanh, ReLU (Rectified Linear Unit), ELU (Exponential Linear Unit) and Radial Basis Function [41].

Neural networks are usually arranged in layers, such that each unit receives input only from units in the immediately preceding layer. There are three types of layer in a neural network: the input layer, output layer, and hidden layer. The input layer is the means of providing an input vector to the network for it to process. The output layer outputs the result of the network's computation. Hidden layers are additional computational units that are not directly accessible from outside of the network [40, p. 729].

There are many different types of neural network architecture [42] with endless variation due to the current popularity of the field. We briefly describe two of the most common architectures: Feedforward networks and Convolutional Neural Networks:

- **Feedforward Networks** Also called **Multi-Layer Perceptrons**, feedforward networks have connections in one direction only such that they are a directed acyclic graph. Every node receives input from "upstream" nodes and delivers output to "downstream" nodes. There are no loops. Feedforward networks are usually fully-connected such that every node in a preceding layer acts as input to every node in the successive layer. A feed-forward network represents a function of its current input and thus it has no internal state other than the values of the weights themselves [40, p. 729].

- **Convolutional Neural Networks (CNN)** Convolutional Neural Networks [43] are a special type of Feedforward Network that are designed to process data stored in array format. Many data modalities can be expressed in arrays: signals [44] and sequences are expressed one-dimensionally, images [19] or audio spectrograms are expressed bi-dimensionally, and video [45] and volumetric images can be expressed tri-dimensionally.

  Units in a convolutional layer are organised in feature maps, within which each unit is connected to local patches in the feature maps of the previous layer through a set of weights. The result of this local weighted sum is then passed through an activation function (usually ReLU). All units in a feature map share the same set of weights. Different feature maps in a layer use different weights. This architecture design is useful for two reasons. First of all in data such as images, local groups of values are often highly correlated such as being temporally or spatially related, forming distinctive local features. Second, the local statistics of images and other signals are invariant to location, such that a feature could appear several times

in an input vector, hence the idea of units at different locations sharing the same weights and detecting the same pattern in different parts of the input vector.

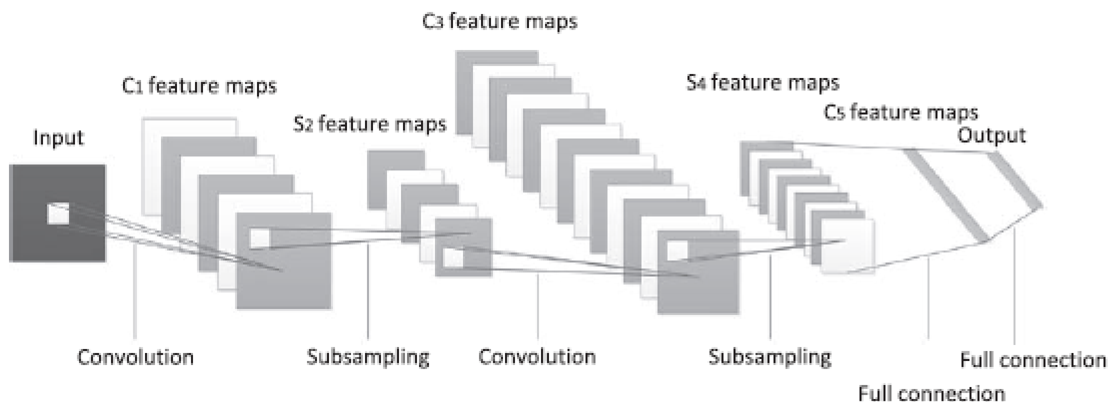

Figure 2.2: Structure of a Convolutional Neural Network with 4 hidden layers [41]

The hidden layers of CNNs may either be convolutional layers or pooling layers. Convolutional layers perform the mathematical convolution function and are used to detect local conjunctions of features from the previous layer. In contrast, pooling layers, merge semantically similar features together by computing the maximum of a local patch of units in one feature map (or in a few feature maps), thereby reducing the dimension of the representation and creating an invariance to small shifts and distortions.

Recent CNN architectures have used as many as 10 to 20 layers of ReLUs, hundreds of millions of weights, and billions of connections between units. Usually, there are two or three stages of stacked convolution, non-linearity and pooling layers, followed by more convolutional and fully-connected layers [39].

The behaviour of a model is defined by training the network using a machine learning algorithm. A **Machine Learning** (ML) algorithm is is able to learn from the data presented to it [46, p. 99]. As defined by Mitchell, "[a] computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E" [47, p. 2].

Artificial Neural Networks are one set of methods in the wider field of ML algorithms belonging to **Artificial Intelligence** (AI).

An ANN with many layers is known as a deep neural network, and training these kinds of networks has been called **Deep Learning**.

Deep learning and ML in general have been receiving much attention from the scientific community, industry, and the wider public in recent years due to many breakthrough successes in a range of different problems. For instance, deep learning

has been making strides in many problems that were previously intractable including pattern recognition [19], natural language processing [48], speech recognition [49], object detection [50], games playing [51], [52], and synthesis [53] to name a few, and it has been found to be generally useful for solving many cross-domain optimisation, prediction, and control problems as well as for automating a range of tasks.

The success of artificial neural networks in so many wide-ranging domains is due to their Universal Approximation property which states that a feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions arbitrarily well [54]. Or that width-bounded networks with an increasing depth can approximate any continuous convex function arbitrarily well [55], [56]. While a feedforward network with a single hidden layer is sufficient to represent any function, the layer may be infeasibly large and may fail to learn and generalise correctly. Therefore, in many circumstances, using deeper models reduces the number of units required to represent a desired function and reduces the amount of generalisation error [46, p. 199].

Artificial Neural Networks have existed for decades [57], but it is only recently that they have become much more widely used due to the increase in computing power informally observed by Moore's Law, and the advent of computational parallelisation of ANNs on GPUs (Graphics Processing Units) due to their structure and repetitive computation lending themselves to be processed more efficiently on a GPU with time-savings orders of magnitude greater than when compared to equivalent processing on a CPU (Central Processing Unit) [58] making them feasible for greater use.

Though highly successful at a wide range of tasks, the main caveats of deep learning models remain that they are relatively difficult to engineer at present with their theoretical properties relatively unexplored leading to their design often being more art than science, and the immense amount of data and computational resources that are often needed to train them.

However, due to the vast quantity of data that is being generated in the digital age and the seemingly rapid advancement arising from the currently high level of interest in the field has meant that ML and deep learning are becoming increasingly viable for solving a wide-range of industrial problems.

2.4 Learning Algorithms

AI Learning algorithms can be classified into three general types depending on the feedback that is available for the system to learn from [40], however it is also common to use a combination of these techniques:

2.4.1 Supervised Learning

In **Supervised Learning** an agent observes a series of example input–output pairs and learns a function that maps from the input to the output [40, p. 695].

Supervised learning algorithms are trained on a dataset of features where each example is associated with a label or target. Roughly speaking, supervised learning

involves observing several examples of a random vector x and an associated value or vector y, and learning to predict y from x, usually by estimating $P(y|x)$.

Supervised learning is the most common form of ML used today [39] and is most commonly used for regression and classification tasks such as object recognition, transcription, linguistic translation, synthesis, and denoising [46, p. 103-105].

An important challenge in Supervised Learning is that of *generalisation*, that the model not only classifies the input data correctly but also classifies previously unseen inputs correctly. These performance metrics are measured by the training error and the test error respectively, which a supervised learning algorithm seeks to minimise. The performance of a supervised learning model is assessed by having two distinct subsets of inputs: the training set, and the test set. We compute the error measure on the training set, and we reduce the training error. Following this, we test the model on unseen data and compute the test error. For a model to be performing well, it must be able to *generalise* its function to unseen data and it is therefore important that the model does not underfit or overfit. *Underfitting* occurs when the model is not able to obtain a sufficiently low error value on the training set and therefore does not approximate the target function well enough, whereas *overfitting* occurs when the gap between the training error and test error is too large and as such its learned function is too specific to the training data and not generalisable.

In practice, supervised learning models are assessed using methods such as k-fold cross validation, where the dataset is partitioned into k non-overlapping subsets and the test error is estimated by taking the average test error across k trials where on trial i, the i-th subset of the data is used as the test set and the rest of the data is used as the training set [46, p. 110-122].

To ensure robust behaviour towards every input faced by a supervised learning model, you would have to ensure that it had been trained on that specific feature previously. However, in many real-world scenarios or in situations with complex input vectors such as vision this is an impossible feat. This is why the concept of generalisability is so important.

We can consider artificially augmenting the dataset with additional data, as the best way to make ML models generalise is to train them on more data. This can be reasonably straightforward in some tasks. For example, in some image-based tasks there is often a need to be invariant to a wide variety of image transformations such as lighting, noise, position, or orientation. As these transformations can be easily simulated, we can generate new (x, y) pairs easily just by transforming the x inputs in our training set. Operations like translating training images a few pixels in each direction can often greatly improve generalisation, even if the model has already been designed to be partially translation invariant by using CNN-based convolution and pooling techniques. Other operations such as rotating or scaling the image have also proven quite effective [46, p. 240-241].

Neural networks are typically not very robust to noise. However, injecting noise into the neural network input has been shown to improve robustness [59], as has injecting noise within the hidden units themselves given the noise is carefully tuned [60].

Though demonstrably effective, the main issues associated with supervised learn-

ing are the requirement for vast labelled datasets which are difficult to obtain, time-consuming to produce, (possibly) intrinsically difficult to label correctly and succinctly, as well as the high computational cost to achieve generalisation that generally requires specialist hardware [39].

2.4.2 Unsupervised Learning

In **Unsupervised Learning** an agent learns patterns in its input without being given any explicit feedback [40, p. 694].

Unsupervised learning algorithms experience a dataset containing many features, then learn useful properties of the structure of this dataset. In the context of deep learning, we usually want to learn the entire probability distribution that generated a dataset, whether explicitly as in density estimation or implicitly for tasks like synthesis or denoising. Some other unsupervised learning algorithms perform other roles, like clustering, which consists of dividing the dataset into clusters of similar examples. Roughly speaking, unsupervised learning involves observing several examples of a random vector x, and attempting to implicitly or explicitly learn the probability distribution $p(x)$ or some interesting properties of that distribution.

Despite the popularity of supervised learning today, it is expected that unsupervised learning is likely to play a larger role in the long term as this is more akin to how humans learn who can generally learn to understand the world unsupervised [39]. Currently, it is most commonly used for density estimation in support of other tasks such as clustering, anomaly detection, synthesis and denoising. [46, p. 105].

Unsupervised learning can be used to supplement supervised learning models in an approach known as **Semi-supervised Learning** where some examples include a supervision target but others do not. In semi-supervised learning, both unlabelled examples from $P(x)$ and labelled examples from $P(x, y)$ are used to estimate $P(y|x)$ or predict y from x. Unsupervised learning can provide useful cues for how to group examples in the representation space as examples that cluster tightly in the input space should be mapped to similar representations [46, p. 243-244].

Examples of unsupervised learning include Self-organising maps [61].

2.4.3 Reinforcement Learning

Reinforcement Learning is about learning from interaction how to behave in order to achieve a goal. [21, p. 55].

The learner and decision maker is an agent who continually interacts with an external environment. The agent selects actions and the environment responds to these actions by presenting new states to the agent [1] and returning a scalar numerical reward signal representing how desirable that state is to the agent which the agent seeks to maximise over time through its choice of actions.

[1] The terms agent, environment, and action are equivalent to a controller, controlled system (or plant), and control signal in engineering terms [21, p. 37].

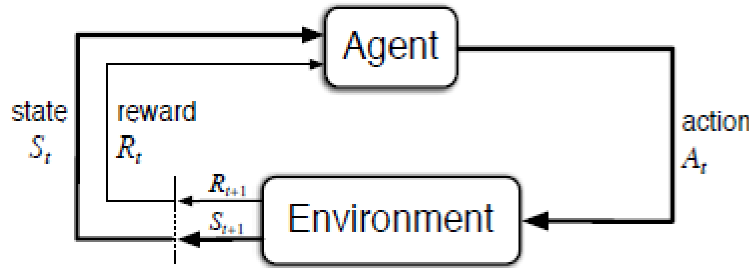

Figure 2.3: Agent-environment interaction in a Markov Decision Process [21, p. 38]

The agent and environment interact in a sequence of discrete time steps, $t = 0, 1, 2, 3, ...$ At each time step t, the agent receives some representation of the environment's state, $S_t \in S$, and on that basis selects an action, $A_t \in A(s)$. One time step later, partly as a consequence of its action, the agent receives a numerical reward, $(r_{t+1} \in R \subset \mathbb{R}$, and finds itself in a new state, S_{t+1}.

The boundary between agent and environment is typically not the same as the physical boundary of a robot and the world around it. For example, the motors and mechanical linkages of a robot and its sensing hardware are usually considered parts of the environment rather than the agent. Rewards are computed by the learning system, but are considered external to the agent. In general, anything that cannot be changed arbitrarily by the agent is considered to be outside of it and thus part of its environment.

Definition of Terms

Actions are any decisions that can be made by the agent that affect its state that it wants to learn how to make. These can range from low-level controls such as the voltages applied to the motors of a robotic arm, or high-level decisions such as a change of the agent's intentions or computation.

States determine all information available to the agent that are relevant for solving its current task and as such their representation is determined by the task designer: for example they could be low-level sensations such as direct sensor readings, or they could be more high-level and abstract such as symbolic descriptions. States can be immediate or based on memory of past sensations.

In general, actions can be any decisions we want to learn how to make, and the states can be anything we can know that might be useful in making them.

Rewards are a simple scalar number $(r_t \in \mathbb{R}$ computed by the learning system at each time step. The agent's goal is to maximise the total amount of reward it receives, which means not maximising immediate reward but rather cumulative reward.

It is therefore useful to introduce a *value function* v as a secondary predictor of expected rewards which determines the value of a given state as the total amount of reward an agent can expect to accumulate in the future from that state onwards. Rewards determine the immediate, intrinsic desirability of environmental states, whereas

values indicate the long-term desirability of states after taking into account the states that are likely to follow, and the rewards available in those states. For example, a state might always yield a low immediate reward but still have a high value because it is regularly followed by other states that yield high rewards, and vice versa. Values are more useful with regards to making and evaluating decisions, however, they are much harder to determine as they must be estimated and re-estimated from sequences of observations over an agent's entire lifetime, whereas rewards are given directly and immediately by the environment [21, p. 37-55].

An agent's behaviour is determined by its *policy*, π_t. A policy is a mapping from observed environmental states to probabilities of each action that can be taken by the agent in those states. Given π_t, $\pi_t(a|s)$ is the probability that $A_t = a$ if $S_t = s$. Reinforcement learning methods specify how the agent changes its policy as a result of its experience with the goal being to maximise the total amount of reward it receives over the long run. The reward signal is the primary basis for altering the policy; if an action selected by the policy is followed by low reward, then the policy may be changed to select an alternative action in that situation in the future [21, p. 37-55]. Policies can either be deterministic or stochastic in nature, and their implementation may take several forms, from simple functions and lookup tables [62] to more advanced approaches such as search processes and neural networks [51].

Some reinforcement learning systems also make use of an environmental *model* that allows an agent to make inferences about how an environment behaves over time and to enable planning of sequenced actions to bring about certain desirable states. Methods for solving reinforcement learning problems that use models and planning are model-based methods. Both model-based [44], [63]–[65] and model-free, reactive methods [51], [66]–[68] are highly successful.

Example

A remarkable variety of problems in robotics may be naturally phrased as problems of reinforcement learning. Reinforcement learning enables a robot to autonomously discover optimal behaviour through trial-and-error interactions with its environment based on an objective function designed by the designer of the control task to measure the one-step performance of the robot. Thus, reinforcement learning offers a framework and set of tools to robotics for the design of sophisticated and hard-to-engineer behaviours [69].

An intuitive example of where reinforcement learning may be applied in a robotic manipulator-based pick-and-place task is described by Barto and Sutton [21, p. 40]:

"Consider using reinforcement learning to control the motion of a robot arm in a repetitive pick-and-place task. If we want to learn movements that are fast and smooth, the learning agent will have to control the motors directly and have low-latency information about the current positions and velocities of the mechanical linkages. The actions in this case might be the voltages applied to each motor at each joint, and the states might be the latest readings of joint angles and velocities. The reward might be +1 for each object successfully picked up and placed. To encourage smooth movements,

on each time step a small, negative reward can be given as a function of the moment-to-moment "jerkiness" of the motion."

Applied examples similar to the above include the work of [66], [70], [71].

Markov Decision Processes

The paradigm of RL deals with learning in sequential decision making problems in which there is limited feedback. Wiering et al. identify three classes of algorithms that deal with the problem of sequential decision making including programming solutions, search and planning, and learning. They emphasised learning as the most advantageous technique for uncertain environments as it encapsulates advantages including simpler system design, coping with uncertainty and changing situations, and solving problems generally for every state rather than specific planning from one state to another. In fact, RL is associated with the most difficult sequential decision making problems where limited or no prior knowledge about the environment is known. Although a model of the environment can be used or learned, it is not necessary in order to compute optimal policies as everything can be learned from interaction with the environment in RL and the task of the algorithm is to interact with the environment in order to gain knowledge about how to optimise its behaviour while being guided by evaluative feedback in the form of rewards.

Markov Decision Processes (MDP) are an intuitive and fundamental formalism for decision-theoretic planning, reinforcement learning, and other learning problems in stochastic domains. They can be seen as stochastic extensions of finite automata and as Markov processes augmented with actions. MDPs have proven a versatile abstract model for describing reinforcement learning environments and a majority of the literature focuses on solving reinforcement learning problems in the context of a MDP [72].

The elements of the RL problem as described in section 2.4.3 can be formalised using the MDP framework. A MDP is a 5-tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ with the following elements:

- $\mathcal{S}$ set of all states.

- $\mathcal{A}$ set of all actions.

- $\mathcal{P}$ the transition dynamics, where $P(s'|s,a)$ defines the distribution of the next state s' by taking action a in state s, where $s, s' \in S, a \in A$. We also denote the initial state distribution $P(s_0)$ as p_0.

- $\mathcal{R}$ set of all possible rewards. In the following, we denote the instantaneous scalar reward received by the agent by taking action a_t from state s_t as $r_{t+1}(s_t, a_t)$, and use r_{t+1} as short for $r_{t+1}(s_t, a_t)$. Other definitions of the reward function exist that depend only on the state itself, in which $r(s)$ refers to the reward signal that the agent receives by arriving at state s.

- γ a discount factor in the interval $[0, 1]$ determines the present value of future rewards such that a reward received k time steps in the future is worth only γ^{k-1} times what it would be worth if it were received immediately If $\gamma = 0$, the agent is myopic in the sense it is only concerned with maximising immediate rewards. As γ approaches 1, future rewards are taken into account more strongly and the agent becomes more far-sighted.

In an MDP, the agent takes an action a_t in state s_t, receives a reward r_{t+1}, and transitions to the next state s_{t+1} following the transition dynamics $P(s_{t+1}|s_t, a_t)$.

MDPs may be episodic or continuing. In episodic MDPs there exists a terminal state (such as when achieving a goal) that terminates the current episode once reached, whereas continuing MDPs do not break down naturally into distinct episodes with a terminal state, such as process control tasks. For an episodic MDP with a time horizon T, an episode will still be terminated after a maximum of T time steps, even if a terminal state was not reached.

The objective of reinforcement learning is to maximise the cumulative future reward or expected return. The discounted return is defined as follows [21, p. 37-55] [73]:

$$G_t = r_{t+1}, \gamma r_{t+2}, \gamma^2 r_{t+3} + ... + \gamma^{T-t-1} r_T \tag{2.3}$$

$$G_t = \sum_{k=t}^{T} \gamma^{k-t} r_{k+1} \tag{2.4}$$

There also exist various extensions of MDPs for differing environments. For example, *Partially Observable Markov Decision Problems* (POMDP) are employed in environments where the state dynamics of the agent are governed by a Markov process, but the state is not fully observable and is instead inferred from observations probabilistically related to the state. In these cases it is possible that the observation history τ_t may be required to describe the current state s_t [74].

Algorithms

Generally, there are two kinds of reinforcement learning algorithms: *value-based policy-based* methods [73]:

- **Value-based Methods** Value-based approaches estimate the values of being in a given state (known as Q values), and then extract control policies from the estimated values to produce an optimal action in each state based on the state(s) that action will probabilistically lead to [73] [40, p. 652].

  One of the best-known value-based algorithms is Watkins' Q-Learning [62]. State-of-the-art value iteration methods include DQN [51] and its various extensions [75], and QT-Opt [71].

- **Policy-based Methods** Policy-based approaches operate on parametrised policies, and search for a set of parameters that maximise the policy objective function. Unlike value-based methods, policy-based methods do not maintain value

estimations, but work directly on the policies themselves. Policy-based methods generally give much more effective solutions than value-based approaches in high-dimensional and continuous action spaces as they can learn stochastic policies rather than just deterministic policies, and have better convergence properties [73].

Both algorithmic types have been implemented for both traditional lookup tables [62] and neural network parametrisations [51]. Recent successes of Deep Reinforcement Learning have extended the aforementioned value-based, policy-based and actor-critic algorithms to high-dimensional domains, by deploying deep neural networks as powerful non-linear function approximators for the optimal value functions $V^*(s), Q^*(s,a), A^*(s,a)$, and the optimal policies $\pi^*(a|s), \mu^*(s)$. They usually take the observations as input (e.g, raw pixel images from Atari emulators [51], joint angles of robot arms [76], or a combination [44]), and output either the Q-values, from which greedy actions are selected, or policies that can be directly used to execute agents [73]. Convolutional neural networks can be used to directly parametrise a reinforcement learning policy and have shown to be effective for vision-based control tasks [77].

Actor-Critic Methods Actor-critic methods [78] are specific types of policy-based methods combine elements from both value and policy-based methods. They extend the traditional reinforcement learning paradigm by decomposing the agent into two separate functions: the actor which is a policy-based function that selects actions; and the critic which is a value-based function that estimates the value of a given state the critic must learn about and critique whatever policy is currently being followed by the actor. This scalar signal is the sole output of the critic and drives all learning in both actor and critic.

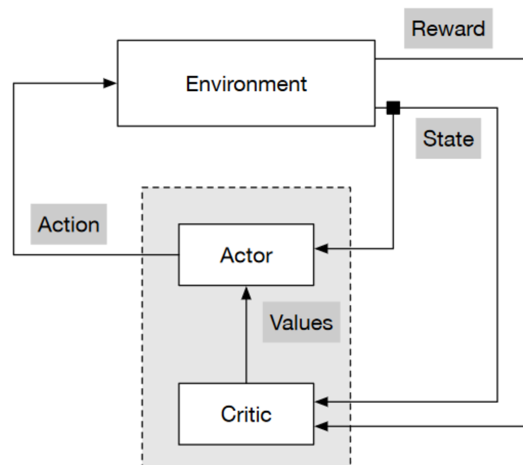

Figure 2.4: Actor Critic Algorithmic Structure [79]

The critic learns an advantage function $A(s, a)$ to compute the advantage of taking an action in a given state compared to remaining in the current state as a means to reduce variance. The advantage function $A(s, a)$ is defined as:

$$A(s, a) = Q(s, a) - V(s) \tag{2.5}$$

where for N states until the terminal state, $Q(s, a)$ is the Q-value function which estimates the overall expected reward assuming the agent is in state s and performs action a, and then continues playing until the end of the episode following some policy π. Defined as:

$$Q(s, a) = \sum_{n=0}^{N} \gamma^n r_n \tag{2.6}$$

and $V(s)$ is the value function which estimates the overall expected reward assuming the agent is in state s and then continues playing until the end of the episode following some policy π. Defined as:

$$V(s) = \sum_{n=0}^{N} \gamma^n r_n \tag{2.7}$$

While the definitions of $Q(s, a)$ and $V(s)$ are very similar, the reward (r of $V(s)$ is the expected reward from just being in state s *before* any action is taken, whereas r in $Q(s, a)$ is the expected reward *after* a certain action is taken.

State of the art deep actor-critic methods include the A3C algorithm [80] (and superseding variant A2C [81]), ACKTR [81], DDPG [67], [70], ACER [82], TD3 [83], SAC [84], [85], TRPO [86], PPO [87] and SAC [84].

Deep actor-critic architectures may either be separate if the actor and critic are parametrised as entirely separate networks, or combined if some of their structure overlaps [67], [81], [88].

While separate architectures are said to be simpler to implement and more stable, shared architectures are advantageous for their ability to share lower level feature processing which is particularly useful for large state representations that require a lot of non-linear interpretation to retrieve meaningful features such as images, audio or video. Shared architectures threfore encourage better generic learning of the low-level features and faster processing per network update [88]–[90].

This approach is somewhat reflected in the literature by works including Schulman et al. and Wu et al. who employ a shared architecture for Atari (pixel-based) tasks with discrete actions and separate architectures for continuous control tasks with low-dimensional state-spaces [81], [87], [91]. But contrastingly, Wu et al. also reported better empirical performance by separated architectures in pixel-based continuous control tasks [81]. It is unclear why this is the case. It is also unclear what the most effective architecture for state-spaces comprising both images and low-dimensional numerical inputs.

Challenges

There are several problems that make reinforcement learning challenging:

Exploration-Exploitation Trade-Off To obtain a high cumulative reward, an agent should repeatedly choose previously tried actions that were found to be successful. However, discovering these actions means it has to try actions not selected before. The agent has to exploit what it has already experienced in order to obtain reward, but it also has to explore a variety of actions in order to make better action selections in the future. Therefore, the agent must develop a strategy to try a variety of actions and progressively favour those that appear to be best [21, p. 2].

Exploration strategies are important for allowing the agent to discover different states outside of what it has seen before. Different strategies have been proposed for exploration, however some of the most common approaches are randomly generating actions, or adding noise to the policy-chosen action [91], [92]. For instance, Andrychrowicz proposed sampling a random valid action with probability 20% or otherwise taking the output of the policy network and adding independent noise to every coordinate with standard deviation equal to 5% of the total range of allowed values [91].

Ibarz noted that adding uncorrelated random noise into the action space for exploration can cause jerky motions which are unsafe to execute on physical robots due to risk of damaging the gearbox and actuators. Jerky actions can be overcome fairly effectively using a number of approaches including reward shaping, mimicking smooth reference trajectories, learning additive feedback together with a trajectory generator, sampling temporal coherent noise, or smoothing the action sequence with low-pass filters[92].

Credit Assignment Problem Feedback for a desirable or undesirable action in sequential control problems is often delayed and not instantaneous, so how do you determine which decision(s) resulted in the final outcome and how do you assign credit for all actions along a trajectory sequence [21, p. 301][73]?

Sparse Reward Problem The success of a reinforcement learning application often strongly depends on how well the reward signal frames the problem and how well it assesses progress in solving it.

The usual way to use reinforcement learning to solve a problem is to reward the agent according to its success in solving the problem. Whilst this can be easy for many problems, some problems have goals that are difficult to translate into reward signals, especially for skilful performance in a complex task.

Even when there is a simple goal that is easy to identify, delivering non-zero rewards frequently enough to allow the agent to achieve the goal once, let alone achieving it efficiently in stochastic environments is challenging. Further, reinforcement learning agents can discover unexpected ways to make their environments deliver reward, some of which might be undesirable, or even dangerous [21, p.386]. This can be seen in the work of Popov et al. where a poorly defined reward function for a

manipulator-based block-stacking task led to the algorithm finding workarounds for optimising rewards that were inconsistent with the desired goal [66].

Intuitively, the Sparse Reward Problem could be addressed by rewarding the agent for achieving subgoals that the designer thinks are important for achieving the overall goal, but augmenting the reward signal with supplemental rewards may lead the agent to behave very differently from what was intended and result in not achieving the overall goal at all. A better method of providing guidance is to augment the value-function approximation with an initial guess of what it should ultimately be rather than laboriously shaping the reward signal [93].

Though they can be difficult to learn from, sparse reward signals can be advantageous in terms of observed performance as the learner can discover novel and potentially preferable solutions to a task. Conversely, well-shaped rewards inject bias into learned control policy which can potentially push the policy in suboptimal directions [73].

Reward signal design is often left to an informal trial-and-error process in practice. If the agent fails to learn, learns too slowly, or learns incorrect behaviour, the reward signal is tweaked by the designer and tries again until the agent produces acceptable results. The designer is therefore judging the agent's performance by criteria which they are attempting to translate into an appropriate reward signal. It is clear that the hand-coded nature of many reward signals is a hindrance to the scalability of models and cross-task applicability, yet unfortunately the theory of designing reward functions is relatively unexplored [21, p.386].

Several supplementary methods have been proposed in the literature to work more effectively with binary and sparse reward signals. *Curriculum learning* where a learning system learns by being exposed to progressively more difficult problems [94] has been applied to reinforcement learning contexts by Florensa et al. in the form of *Reverse Curriculum Learning* [95] and Generative Adversarial Network (GAN)-based *Automatic Goal Generation* [96]. In a similar way, Tavakoli et al. proposed a *Prioritised Restart Distribution* method where episodic initial states are more carefully chosen to be related to trajectories leading to 'significant' past experiences whilst maintaining exploration (e.g., nearby goals) [97]. Andrychowicz et al. adapt the experience replay method first proposed with DQN [51] by reassessing previous action trajectories with a subset of different goal states in a method they call *Hindsight Experience Replay* with the underlying idea that just because a trajectory was not successful for achieving a particular goal, it may still provide useful experience for achieving an alternative goal [91]. Riedmiller et al. proposed, alongside the policy learning for the main task, learning policies for a set of auxiliary tasks whose supervision signals can be easily obtained by the activation of certain sensors while concurrently learning a scheduling policy to sequence these policies in an approach they called *Scheduled Auxiliary Control* [98]. Pathak et al. also proposed *Curiosity Driven Learning* where extrinsic environmental rewards are supplemented with intrinsic agent-generated rewards for environment exploration [99], [100]. Finally, there are *Teach by Demonstration* methods such as *Imitation Learning* where the system generalises to new scenarios from expert demonstrations in a similar manner to classification, and *Inverse Reinforcement Learning* whereby a reward signal is extracted

from observation and the agent's behaviour is optimised according to that reward signal [101].

2.5 Deep Reinforcement Learning in Robotics

Deep learning and AI algorithms have been applied successfully in robotic control tasks, and remarkable progress has been made in multi-sensory and visual control-based tasks [8], [14], [16], [71], [77] particularly considering the sensory complexity of vision sensors.

Though by no means exhaustive, there are several interesting lines of research that are being used to improve various aspects of deep RL in robotics.

2.5.1 Parallelised Algorithms

As we have previously mentioned, one of the main caveats of deep learning is the long time needed to train models due to the vast quantities of data needed. One method for reducing this training time is using algorithms that take advantage of parallelisation by learning across multiple agents simultaneously to train a single model such as A3C / A2C [80], ACKTR [81] or QT-Opt [71].

For instance, Kalashnikov et al. developed QT-Opt which was a distributed deep RL algorithm that combined a deep Q-Learning algorithm with large-scale distributed optimisation so that multiple robotic agents can contribute to learning simultaneously, thus improving the scalability of training. In the work accompanying the algorithm, Kalashnikov et al. utilised seven physical LBR iiwa which they trained for 8000 hours over a period of four months to derive a control system for a vision-based pick-and-place task that was able to grasp a diverse range of objects with a high success rate [71].

Their approach demonstrated merit in terms of scalability, general applicability, self-supervision and demonstrable real-world capability, but the approach generally suffered in terms of the poor sample efficiency of the Q-Learning based method leading to long training times, and the work relied on immense computational and robotic resources. These are significant factors when you consider the specificity of the task and the fact that you may have to repeat the procedure and derive a new model for a new task environment.

This demonstrates both the benefits of parallelisation whilst simultaneously highlighting the need for sample efficient [2] algorithms [73], [81].

Whilst parallelism as a means of sharing experience is useful, it is limited by the number of physical agents that are available, and by temporal constraints on data gathering. Therefore it is useful to consider the use of simulated agents for training.

[2] The Sample Efficiency of a learning algorithm is defined as the speed of convergence per number of timesteps [81]

2.5.2 Simulated Environments

ML algorithms may be trained either off-line or on-line. Within robotics, off-line training can be completed by either training with a batch of predefined historically gathered data or by generating simulation-based experience before task execution, whereas online training is done by learning in real-time in a physical environment with unseen temporally-constrained real-world data [24].

Simulators are widely used in the robotics community as they allow for real world systems to be quickly and cheaply prototyped without the need for physical access to hardware. Simulations are particularly advantageous in robotic learning research as the huge data requirement for deep RL approaches means it can be more effective to take advantage of increasingly powerful and ubiquitous computational resources to cheaply and quickly generate synthetic data to accelerate the learning as opposed to the high costs intrinsic to real-world data collection experiments. The use of simulation over reality carries numerous advantages, including:

- No risk of wear or damage to real-world hardware;
- Many instantiations of a simulation can run in parallel, potentially speeding up training time through parallelisation;
- Faster than real-time operation (depending on computational resources);
- Instant access to robots without having to purchase;
- Easy reconfigurability;
- Human intervention is not required [23];

Generally, a new simulation needs to be created for each specific task environment to train robots for new tasks. However, for more generic tasks there exists an increasing number of publicly available RL environments such as the OpenAI Gym [102], and numerous others [103]–[105] with the purpose of providing a benchmark for measuring the effectiveness of various RL algorithms in solving certain types of environments, hence accelerating research in RL and related areas by standardising problems and making comparison of results and algorithmic evaluation easier. Their intent is comparable to the use of standard datasets for benchmarking in supervised learning problems [19].

Despite the many advantages of simulators in robotic learning, real-world experience will always be more useful for training as simulations may not capture all the dynamics of the real-world. In fact, ML algorithms trained exclusively on simulation may not always translate well to real-world environments [24] due to the *Simulation-Reality Gap* where discrepancies between real sensory readings and the modalities of synthetic renderings brought about by the necessity to abstract, approximate, or remove certain physical phenomena can impose major challenges and prevent control systems created in simulation from performing to the same standard in reality due to underspecification [8], [23], [73], [106].

In robotics contexts, these 'gaps' usually relate to actuators (i.e. torque characteristics, gear backlash,...), sensors (i.e. sensor noise, latencies, and faults), temporal dynamics, and the physics that govern interactions between robots and objects in their environment [23].

However, there is research into several methods for overcoming the simulation-reality gap. Tai et al. gave a thorough overview of the various methods for vision-based deep RL in robotics including domain adaptation methods utilising generative adversarial networks [53], [107], noise augmentation to simulate real-world error [8], [71], [108], and domain randomisation to improve invariance to certain environmental parameters [109]. For example, Kalashnikov et al. used some of these techniques during their simulated training by cropping the camera image with a random anchor to support viewpoint invariance and adjusting the brightness, contrast, and saturation of training images by randomly sampling within bounded limits to simulate generic visual noise such as lighting fluctuations [71], while Tobin et al. used domain randomisation during simulation of a pose estimation task to adapt to a physical environment with no prior training by manipulating simulation settings at random such as position and texture of objects; position, orientation and field of view of camera; and the type and amount of random noise added to images [109].

However, perhaps the best method for overcoming the simulation-to-reality gap is through simulation-based *Transfer Learning*.

Transfer Learning can be an efficient trade-off of simulated and real-world training where a control system is initially trained off-line in simulation, before successive fine-tuning on-line in the physical domain [8], [22], [24]. This approach has the benefits of being able to generate a wide range of experience while also decreasing the time to learn the general task structure by taking advantage of powerful computational resources unconstrained by the physical realm and efficient parallel algorithms to train multiple simulated agents simultaneously, and providing opportunity to learn previously unmodelled environmental dynamics and achieve robust real-world behaviour.

2.5.3 Transfer Learning and Associated Methods

Transfer learning does not only apply to robots trained in simulation and then fine-tuned in the real world. The general concept of pre-training in a differing environment before successive fine-tuning in the application environment can be seen in several competing research areas based on the premise that robots will only become ubiquitously useful when they require just a few attempts to teach themselves to perform different tasks, even with complex bodies and in dynamic environments [22]. These research areas include *Learning by Demonstration*, and *Few-Shot Learning* methods based on leveraging prior unsupervised / semi-supervised training.

Learning by Demonstration

Learning by Demonstration is a method of learning tasks where an agent is taught by another observable agent (possibly human) who is already an expert at the task with the aim of benefiting from the expert whilst leaving open the possibility of performing

even better [21, p.386]. Observation can come in many forms; from sensor-based teleoperation [110], to visual detection [77], to physical guiding [111], [112].

Learning by Demonstration usually achieved by one of two methods: *Imitation Learning*, a supervised learning-based method similar to classification that learns an expert's behaviour directly; or *Inverse Reinforcement Learning*, where a reward signal is extracted from a demonstration and an RL algorithm is used with that reward function to learn an optimal policy. Whilst Inverse Reinforcement Learning cannot extract a reward signal exactly, it is possible to find a plausible set of reward-signal candidates, with some strong assumptions about the environment required [101].

These methods have been successfully applied to robotic manipulators including the KUKA LBR iiwa for the learning, optimisation, and generalisation of task trajectories [111], [112].

An overview of methods for robotic learning by demonstration is presented in [113].

Few-shot Learning

Learning by Demonstration is also thematically similar to another line of research known as *Few-Shot Learning* where agents learn to perform generalised tasks based on very few prior demonstrations of that task.

State-of-the-art few-shot learning methods tend to rely on the previously mentioned idea of transfer learning where prior experience is utilised to generalise quickly to new tasks. For instance, both [77] and [22] leverage their few-shot learning with prior sessions of unsupervised / semi-supervised learning.

In the work of Marjaninejad et al. they devised a new two-phase RL-based algorithm that they call G2P, or *General-to-Particular* where the initial phase consists of generating a random unsupervised control sequence of actions, referred to as 'motor babbling' to train a multi-layer perceptron representing an inverse map between system inputs (motor activation levels) and desired outputs (system kinematics: joint angles, angular velocities and angular accelerations). This allows the system to gain a sparse but diverse familiarity with the state-action space without exploring all possible control sequences before the second phase of the algorithm which is a period of simple task-specific RL to learn high level control. This technique allowed a 2 DoF tendon-driven manipulator to achieve rapid proficiency in cyclical control behaviours such as running with good resilience to perturbations [22].

Finn et al. on the other hand utilise a *Meta-Learning* approach whereby agents adapt to new tasks based on pre-trained experience in loosely-related tasks [114]–[116]. In their work, they pre-train a model for general invariant vision-based manipulation tasks in dynamic environments with a series of prior demonstrations of a range of tasks through both robotic teleoperation and observation of the human performing a task without the robot at all in a range of environments, with a range of objects. Following this, the system was able to adapt to new tasks after a single demonstration even if it has not completed that specific task before in the specific environment configuration and even when presented with different camera angles [77].

This work along with others [64] demonstrates the usefulness that unsupervised

and semi-supervised methods have in pre-training an RL system, particularly in tasks with visual input [64], [77].

2.6 Practical Considerations in Deep RL and Robotics

ML and RL research have unique challenges due to the many stochastic aspects of their application and implementation and the often immense computational resources required to obtain results. Certain procedures should be considered to not only produce working models, but also follow best practice and report experimental research in such a way that is beneficial to the deep RL research community.

2.6.1 Performance Metrics

Determining your goals in terms of which error metric to use and the level of performance required is said to be the necessary first step in developing a machine learning application as it is these metrics that will guide all future actions[46, p. 410-412]. Some common performance metrics in the RL literature include average return and episode length among others [117].

2.6.2 Reproducibility

Research progress in RL has been impeded by reproducibility problems across the literature that have produced challenges with verifying the efficacy of methods and thus make it more difficult to contrasting works. To overcome this, it is recommended that when reporting RL methodology that all hyperparameters, implementation details, experimental setup, and evaluation methods for both baseline comparison methods and novel work are reported and when reporting on the performance data of an algorithm it is helpful to run many trials across different random network seeds and report performance averages and ranges due to the high variance across trials and random seeds in RL [118].

2.6.3 Ablation Studies

Recent concerns of reproducibility in deep RL have also led to the increased use of ablation studies [71]. Ablation studies describe a procedure where certain parts of a system are removed or altered in order to gain a better understanding of that system's behaviour.

There is no one-size-fits-all approach to performing ablation studies in deep learning as it is dependent on the system, application, and desired performance characteristics. However, some examples in the literature include:

- Kalashnikov et al. compared the performance sensitivity of their deep learning-based robotic grasping system by performing several studies including: altering the state representation; altering the discount factors and time-based reward

penalty; altering the episodic termination condition; altering the specific learning algorithm; altering the size of the training set; performing tasks with differing characteristics [71].

- Tobin et al. compared the performance sensitivity of their simulation-to-reality-trained pose estimation system by performing studies such as: altering the number of training images; altering the number of unique textures seen in training; altering use of random noise in pre-processing; randomisation of camera position in training; and use of pre-trained weights; and found that their system was sensitive to all of the above except for use of random noise [109].

- Girshick et al. tested the performance of their CNN-based object detection algorithm by performing studies where different parts of the network were removed and found that removal of significant portions of certain parts of the network resulted in surprisingly little performance loss [119].

- Fujimoto et al. proposed improvements to deep actor-critic algorithms including Clipped Double Q-Learning, delayed policy updates, and target policy smoothing under an umbrella algorithm called TD3. Their algorithm's effectiveness was empirically proved by comparing the performance of the vanilla network architecture with the addition of one of their proposed improvements at a time, and comparing the full TD3 architecture with a single proposed improvement removed at a time [83].

2.6.4 Hyperparameter Optimisation

Deep learning algorithms are governed by a set of hyperparameters that affect an algorithm's behaviour across various performance measures such as runtime, memory cost, model quality and generalisation capabilities. Hyperparameters can have significantly different effects across algorithms and environments [118] and as such finding an optimal set of hyperparameters is extremely difficult as good hyperparameter values are application dependent. While hyperparameter values can be set manually based on intuition, this requires expert knowledge of the domain and of how such parameters can affect various learning algorithms, (although sometimes referencing values from similar work in the field may suffice [120]). To compensate for this, there exist automatic hyperparameter tuning methods, however these vary in computational efficiency, search effectiveness, and implementation complexity that are more suitable for when suitable values are not known.

The two most common and simple methods to implement are **grid search** and **random search**. Grid searches are generally employed where there are a very small number of hyperparameters to optimise. Usually, a finite set of values is chosen for each hyperparameter and a model is trained for every combination of hyperparameter values with the values yielding the best performance chosen for use. Grid searches are often performed repeatedly with hyperparameter values iteratively refined to get closer to an optimal value.

However, the main problem with grid searches is that their computational cost grows exponentially with the number of hyperparameters. While there are some ways of mitigating this through parallelisation, the fixed hyperparameter values may not cover the search space effectively in a computationally efficient way. Furthermore, in practice not all hyperparameters are equally important to tune as some have greater influence on model sensitivity than others and so it can be redundant to run repeated tests on nearly identical hyperparameter subsets with no material difference in model performance and thus no real insight into useful parameter values .

Random searches have been found to converge much faster to good hyperparameters than grid searches where there are several hyperparameters that do not strongly affect model performance. In a random search, you would define a marginal distribution over each discrete hyperparameter or a uniform distribution e.g. on a logarithmic scale for continuous valued hyperparameters and then run a search of randomly sampled hyperparameter values for as many trials as required. Similarly to grid search, multiple iterations of random search may be carried out with refined parameter values but the mechanism of random search means sensitive parameters can be identified more quickly and as a result may be chosen for more thorough exploration in equivalent computational time. Random searches also offer a number of other number of other benefits for general experimental practice including early termination, asynchronicity, being able to add additional experiments or restart failed experiments without disruption, and compatibility with additional controlled experiments [46, p. 415-423][121].

Grid and random searches are the most commonly used methods of hyperparameter optimisation due to their implementation simplicity. More complex hyperparameter optimisation methods include population (genetic algorithm)-based methods [122], RL-based methods[123], on the fly methods [124], and Bayesian optimisation [125], as well as further research in the closely related field of Neural Architecture Search (NAS) [126], [127].

2.6.5 Software Implementation

There are a multitude of open-source software libraries for developing ML algorithms and robotic simulations that can vary in terms of their intended application, features, performance, base language and other facets.

Robotic Simulators

There are many different robotic simulators that can be used, depending on the application [128]. Collins et al. [23] compared the effectiveness of different simulation software for simulating robotic manipulation environments in terms of their real-world reproduction accuracy against a 'ground-truth' physical robot. In their study, they created a downselected list comprising of *"mature, well maintained simulators with active communities and good documentation practices* [which also had] *a common programming language* [and provided] *access to the Robot Operating System (ROS)"* to identify the simulators best suited to facilitate robotics research which left them

with the MuJoCo [129], PyBullet [130], and V-Rep [131] simulators, the latter of which also had three different physics simulators: ODE, Vortex, and Newton.

The results showed that no one simulator was best at reproducing all of the three tasks that the authors experimented with and that PyBullet was the most accurate in two out of the three tasks whilst V-Rep running the Newton physics engine performed better in the other. However, there was very little overall difference in the total error across all three tasks for Pybullet, V-Rep (Newton) and V-Rep (Vortex). MuJoCo on the other hand had a much larger cumulative error due to its poor performance in one particular task. The authors conclude that the simulation of the kinematic model and control of manipulators is largely solved when compared with the real world, but that considerable developments are necessary for interactions between simulated objects as the physics behind contacts remains a complex problem that is difficult to replicate in simulated environments [23].

Deep Learning Libraries

The development of open-source, industry-backed software libraries for building ML models written in high-level languages such as Python have contributed to the widespread research and application of deep learning methods seen today. Shi [132] and Bahrampour [133] et al. identified and compared several state-of-the-art and contemporary deep learning libraries including Tensorflow [134], Torch [135] / PyTorch [136], Deep Learning4j [137], Caffe [138], and Theano [139] amongst others.

2.7 Research Gap Discussion

Fixtureless assembly environments pose many challenges to automation due to their non-determinism but are sometimes necessary due to economic factors and / or spatial and geometric constraints brought about by work pieces and the plants surrounding them. The use of collaborative and redundant robot manipulators equipped with visual servoing, and in particular, a hybrid of eye-in-hand and eye-to-hand visual servoing, could be effective for robotic manipulation tasks in fixtureless environments that require a high degree of accuracy and where the precise end-effector location is not known relative to the robot's coordinate system but instead determined from specific visual features. Deep reinforcement learning methods have shown promise in vision-based robotic manipulation tasks but are uncommon in industrial settings due to the highly task specific setup required and immense computational resources required for training reinforcement learning models. Progress in the field is being sped-up by the introduction of better experimental reporting practices for new algorithms and the use of standardised benchmark tasks in the form of simulations to assess and compare the effectiveness of reinforcement learning algorithms to specific problems.

We have identified two publicly available, open-source simulated RL environments for robotic manipulation that encapsulate some elements of the accurate fixtureless manufacturing problem, but with different goals and different problem framings.

One of these environments is the FetchReach environment in the OpenAI Gym [140], [91]. FetchReach incorporates the Fetch Robotics Mobile Manipulator robot (described in Table 2.1), a collaborative mobile robot with a 7-Dof manipulator and gripper end-effector and stereoscopic vision camera. The goal in fetch reach is to move the robot's gripper to a target position. The goal is 3-dimensional and describes the desired position of the end-effector for reaching. Rewards may be either: sparse and binary, where the agent obtains a reward of 0 if the object is at the target location (within a tolerance of 5 cm) and -1 otherwise; or dense, where the linear distance from the goal is returned as a direct reward. Actions are 4-dimensional: 3 dimensions specify the desired gripper movement in Cartesian coordinates and the last dimension controls opening and closing of the gripper. Observations include the Cartesian position of the gripper, its linear velocity and the position and linear velocity of the robot's gripper.

There are several limitations to this environment which would make it non-analogous to the challenges of accurate fixtureless manufacturing in aerospace manufacturing environments. Firstly, the choice of robot is not suitable for heavy industrial manufacturing tasks such as those aerospace manufacturing. The Fetch Mobile Manipulator is designed for warehouse logistics tasks as indicated by its pre-attached gripper end-effector and low payload capacity (6 kg). Secondly, the dexterity of the robot is limited by the MDP formulation, which only specifies actions in terms of a Cartesian coordinate for the end-effector position rather than the joint-level control which is required to operate in dynamic, changing environments. Thirdly, the accuracy requirements of the task is extremely generous, within 5 cm of the desired position. This is nowhere near accurate enough for aerospace manufacturing applications which require sub mm accuracy [9]. Finally, environment information to direct the agent's learning in terms of the state observation and reward signal in the MDP are too accessible. Both of FetchReach's reward functions assume knowledge of the precise distance of the end-effector from the goal to compute the reward, and the observation vector also explicitly includes the exact goal to inform the agent's decisions, thus the state is fully observable. But, it is common in fixtureless manufacturing tasks that the exact position of a goal may be difficult to communicate as the environment is non-deterministic and lack of precise knowledge about a desired end-effector position introduces considerable difficulty to a control task. The authors of FetchReach themselves concede their task as "very easy to learn".

A practical way to indicate a goal in fixtureless environments if the desired position is not explicitly known by the control system can be via a visual sensory cue. This kind of scenario is addressed by the other relevant RL environment we have identified, KUKACamBulletEnv in the PyBullet RL Gym Environments [130] based on the work of Kalashnikov et al. [71] (see: Section 2.5) Here the goal of the RL agent is to learn to perform a pick-and-place task with the KUKA LBR iiwa robot using observations consisting of a single high resolution eye-to-hand camera image (472x472 pixel) and internal joint position information,and a binary reward based on the success of grasping an object.

This environment also possesses limitations to learning robotic control polices for accurate manipulation in fixtureless environment. Firstly, the environment is over-

specified to a pick-and-place task rather than a generic positional alignment task and lacks emphasis on specific alignments or accuracy so long as the goal (grasping an object) is achieved. Secondly, a high-resolution sensory image is included in the observation of the MDP specification. High resolution image require large neural network architectures and lots of training, as demonstrated by the training period of 8000 hours over a period of four months, even after training in simulation [71]. Finally, only a single eye-to-hand camera image is utilised in the visual servoing configuration which will place limitations on the accuracy than can be learned by the control policy.

Based on the limitations present in the existing publicly available RL environments for emulating accurate robotic manipulation in dynamic fixtureless manufacturing environments and exploring hybrid visual servoing approaches to such problems, we feel compelled to design and implement a RL environment that more accurately reflects the challenges of accurate robotic manipulation in fixtureless manufacturing and incorporates hybrid visual servoing as a means to overcome these challenges.

2.8 Novelty

Creation of a new benchmark task in the form of a simulated RL gym environment emulating a generic accurate manipulation task within the aforementioned fixtureless manufacturing environment can only speed up progress towards solutions in this area, while assessment of current algorithmic techniques for solving this task to a suitable standard and within reasonable computational time and resources will provide clues to the most effective techniques to use within this particular problem domain. Therefore, the basis of this work and its novelty is to define and implement a task environment that uniquely differs in both objective and observation constraints to the aforementioned RL environments of FetchReach and KUKACamBulletEnv so as to more accurately reflect the challenges of accurate fixtureless manufacturing, and to conduct experiments assessing whether deep reinforcement learning is a feasible methodology for solving this manufacturing challenge. The control policy required to solve such a task environment encapsulates an end-to-end accurate and efficient object detection, pose estimation, and robotic trajectory computation algorithm.

Based on limitations of the prior works, our novel RL environment will need to:

- Utilise a robotic manipulator that is suited to heavy industrial manufacturing tasks;
- Present an environment state that is only partially observable through visual cues and incorporate a hybrid visual servoing observation;
- Provide a high level of robotic control dexterity to handle a dynamically changing environment;
- Assess that control policies possess a high degree of accuracy;
- Encapsulate the stochastic nature of the task;

- Be generic enough such that a learned control policy can be translated to many accurate manufacturing tasks that involve robotic manipulation and uncertain / dynamic task environments;
- Limit the amount of computation and training that may be required to derive a feasible control policy via RL;

To the author's knowledge, such an environment does not yet exist as an openly available benchmark for RL as the application of deep reinforcement learning for hybrid VS tasks is relatively unexplored. It is hoped that an implementation of the described environment will speed up research towards solving tasks where such a setup may be necessary. This environment may also be used as a test-bed for the performative and computational effectiveness of low-resolution hybrid VS and robotic joint sensor-based observations for positional alignment tasks requiring sub mm accuracy and precision. To the author's knowledge, the combination of all of these research elements is a relatively unexplored area of the field.

2.9 Chapter Summary

In this chapter we have performed a literature review examining the state-of-the-art as it relates to visual servoing and ML methods for robotic control in fixtureless manufacturing tasks. We have identified gaps in the research and are proposing a novel solution to the challenges of developing robotic control policies for fixtureless manufacturing applications such as aerospace manufacturing. In the next chapter, we will set out the methodology, informed by our literature review, by which we hope to achieve our research aims and test our hypotheses.

Chapter 3

Methodology

In this section, we outline the methodology we have followed in the pursuit of the research aims stated in Section 1.8. We begin by giving an overview of our methodology in terms of the steps required to achieve our research aims in Section 3.1. In the remaining sections, we will provide the justification for our methodology including: literature review strategy; the choice of robotic manipulator in 3.3; the reasoning for a deep reinforcement learning approach in 3.4; the motivation for producing a simulated environment in 3.5; the choice of software for each aspect of our technical work in 3.6; and our experimental approach in 3.7.

Please note, that in this chapter we only justify the methodology that has been followed to enable us to test our hypothesis. Justification and discussion of specific design choices, including the task description, MDP specification, and algorithmic choices takes place in the following chapter: System Design.

3.1 Methodology Overview

In order to establish whether hybrid visual servoing and deep reinforcement learning can be applied successfully to the problem of end-effector positioning in robotic fixtureless manufacturing environments such as aerospace manufacturing applications, experiments need to be conducted demonstrating a control policy trained via a reinforcement learning algorithm successfully aligning with relevant points in space, ascertained from visual cues, where it could perform a fixtureless manufacturing task. In order to conduct these experiments, a series of steps will need to be followed to produce the required resources and select appropriate infrastructure for testing our hypothesis.

Firstly, we describe a task encapsulating a fixtureless manufacturing problem present at facilities such as Airbus, and include in this description our proposed solution to the problem in terms of physical hardware for which we have selected the KUKA LBR iiwa robotic manipulator. This will serve as a high-level description of the research challenge at hand and will reflect the robotic control challenge appropriately. Then, we will formally model the task as a MDP so that a reinforcement learning algorithm can interact with the task environment and be used to train a robotic con-

trol policy. This modelling will consist of definitions for all key MDP components, including the observation / state, actions, reward function / goals, and initial and final state distributions. Following this, we will select an appropriate reinforcement learning algorithm for solving the task environment given the specified characteristics of the MDP. Next, we will design a neural network architecture appropriate to the specification, characteristics and perceived difficulty of the MDP that will serve as the basis of a robotic control policy, trained via reinforcement learning. The choice of reinforcement learning algorithm will influence some aspects of the specific neural network model.

Following this system design, software implementation will take place of the three aforementioned elements: the MDP as a reinforcement learning environment, the reinforcement learning algorithm, and a neural network model. The MDP will serve as the primary basis for the software implementation of the RL environment in terms of its function, but the environment itself will adhere to the conventions of the `PyEnvironment` class in the TF Agents reinforcement learning library to ensure it can be used with TF Agents' built-in reinforcement learning algorithms. The underlying physical modelling of the reinforcement learning environment will be a simulation in the PyBullet software library. A pre-made 3D model of the KUKA LBR iiwa will be imported from the PyBullet library to be utilised in the simulation, and additional assets will be created in the 3D modelling software, Blender. A training script will then be written in Python and consisting of code for the generation of a neural network using the Tensorflow library, initiation of the reinforcement learning algorithm and training loop using elements from the TF Agents library, and additional code for recording experimental data using the Comet.ml library.

Finally, a series of experiments will be conducted to test our hypothesis and determine the efficacy of deep reinforcement learning and hybrid visual servoing for robotic manipulation in fixtureless environments. Optimal, or even functional, hyperparameter settings for the reinforcement learning algorithm and specific neural network architecture will not be known in advance, thus the goal of these experiments will be to discover, and then optimise, a set of working hyperparameters that allow the RL agent to derive a control policy, encoded as a deep neural network, to solve the simulated task environment after a period of training using a deep reinforcement learning algorithm. Hyperparameter optimisation will be conducted using a random search method across a selection of key algorithm parameters including number of training episodes, learning rates, random seed, and replay buffer and exploration settings. Experiments will be executed on the Google Colab cloud computing service. Evaluation of agent performance will take place through visual observation of the agent acting in the simulation environment at select intervals, the recording of the average return over the number of episodes - a value that should increase if the agent is solving the task successfully with increasing frequency, the average episode length - which will decrease if the agent is solving the environment more efficiently, and the loss - which dictates convergence of the model during training. Analysis of the collected data will allow us to evaluate whether deep reinforcement learning and hybrid visual servoing can be usefully applied to fixtureless manufacturing environments.

Having set out our methodology, in the following sections, we will describe the

justification for the methodology followed.

3.2 Literature Review

An idea of the research aims and general hypothesis was already known based on prior knowledge in related fields and an understanding about the research problem domain from the advertised Airbus Shopfloor Challenge. Based on this, key words were identified such as "deep reinforcement learning", "collaborative robotics", "aerospace manufacturing", "computer vision", "Industry 4.0" and used to guide the initial literature search. Many of the initial works considered would have included survey and literature review papers to give a broader view of the various research areas. From this, more precise terms could be derived to further guide reading including "fixtureless manufacturing", "visual servoing", "redundant manipulators", and "simulation-to-reality gap", "transfer learning" and so on until many specific and related works were identified, and a research gap could be found. From the large number of papers reviewed, specific research questions had been derived and a general methodology to approach the problem with had been identified. At this point, very specific works and resources were identified to address my specific research challenges including looking at overviews of software methods, defining MDPs, experimental approaches in deep RL etc. Following on from this, the specific methodology was derived and followed.

3.3 Robot Specification

Figure 3.1: KUKA LBR iiwa R800 [29]

The KUKA LBR iiwa has been selected as an appropriate platform for this research. While several different robotic manipulators could perhaps be used to similar intentions, the iiwa has been selected for the following suitability reasons.

- **Kinematic Redundancy** As a redundant manipulator the iiwa is generally applicable to a wide array of industrial tasks (see: Section 1.2).

Joint	Range	Velocity
A1	±170°	±98°s
A2	±120°	±98°s
A3	±170°	±100°s
A4	±120°	±130°s
A5	±170°	±140°s
A6	±120°	±180°s
A7	±175°	±180°s

Table 3.1: Maximum range and velocities of motion for KUKA LBR iiwa R800 joints [142].

- **Accuracy** The iiwa has an end-effector accuracy of ± 0.1 mm [142] which makes it suitable for tasks that require a high level of accuracy such as drilling and other tasks in the aerospace manufacturing domain (see Section 1.4) which require accuracies of ± .25 mm [9]. In fact, the winners of the previously mentioned Shopfloor Challenge utilised a KUKA LBR iiwa robot in their approach, demonstrating the robot's promise to the task domain [143].

- **Sensors and Hardware** The iiwa is equipped with joint torque and position sensors and its movement can be determined with forces, velocities, angular rotation, or vector coordinates which gives access to several control paradigms and offers flexibility in formulating the MDP in Section 4.2.

- **Industrial Popularity** KUKA is one of 'the big four' globally leading robotics companies (along with ABB, Fanuc, and Yasukawa). In particular, the KUKA LBR iiwa has proven very popular due to its innovative technology and has subsequently been used in industry [144] and the subject of a swathe of research activity [70], [71], [107], [110]–[112], [145]–[152]. Adding to the strong base of research on an already popular and current commercial robot increases the likelihood of future application and further research. Furthermore, the popularity of the iiwa has meant that models of the robot already exist in each of the previously specified simulation software.

- **Ease of Programming and Software Integration** Though the iiwa is not natively as intuitive to program as some other platforms such as Universal Robotics UR series (which emphasise ease of use), applications for the LBR iiwa are written in the Java programming language with the tailored Sunrise Workbench IDE and Java is an extremely popular object-orientated language with a thriving ecosystem of documentation and support [147]. Not only this, the LBR iiwa can also be integrated with ROS [152] which not only decreases the technical gap between this work (written in the Python programming language with various open-source packages) and physical implementation, but better equips it for the challenges of Industry 4.0 that ROS seeks to bridge [153]. ROS compatibility was highlighted as a key requirement for selecting appropriate robotic simulation software by Collins et al. [23] (see: Section 2.6.5).

- **Safety** The LBR iiwa has numerous, highly configurable safety features from both a hard-and-software point of view that enable it to comply with modern safety standards including workspace monitoring, velocity monitoring, and collision detection. While these features do not make an application automatically safe, for instance when sharp tools are involved, many applications will not require any further safety equipment [147], thus increasing the likelihood of physical deployment in industry.

- **Connectivity** The iiwa has been designed to be Internet of Things compatible with its external network and hardware connectivity. Not only is this consistent with vision of Industry 4.0, thus increasing the potential lifetime of the system, this connectivity is essential for communicating with other systems running computationally demanding software such as neural networks, and for easily integrating with sensory hardware. This connectivity will also allow it to communicate with ROS applications [152].

- **Extensibility** Some of the fixtureless manufacturing problems we are seeking to address through this work may require the use of a mobile manipulator in their solution. The existence of the KMR iiwa, a combination of an LBR iiwa with a mobile platform [31] would appear to be the most straightforward way of extending the results of this work to mobile domains due to the tight integration of manipulation and mobility robotics. While other redundant mobile manipulators, such as the Fetch Reach [141] exist, it is designed for warehouse logistics tasks rather than heavy industrial applications such as drilling in aerospace manufacturing and comes pre-attached with a gripper as its end-effector while also possessing a lower payload capacity (6 kg) than the KUKA LBR iiwa (7 or 14 kg).

3.4 Deep Reinforcement Learning

The complexity of fixtureless manufacturing environments such as aerospace manufacturing require the use of artificial neural networks and deep reinforcement learning to control a robot in which the robot's working environment is dynamically and stochastically reconfigured and its precise sensory inputs may be difficult to predict. Discussion and justification of the specific algorithm and neural network architectures employed can be found in Sections 4.3 and 4.4.

3.5 Simulation

A simulated reinforcement learning environment will be implemented to provide a basis for training a neural network-based control policy to solve our fixtureless manufacturing task. The use of simulated environments is also well established in reinforcement learning research for benchmarking the performance of RL algorithms and he role of simulated environments in developing robotic control policies, and their advantages, was laid out in Section 2.5.2 of the literature review. Although simulations

are limited in the sense they lack deployment in the real-world and there are issues with the simulation-to-reality gap, they are a useful approach for prototyping control polices that can be used or fine-tuned in the real-world. Their advantages include faster training time due to the rapid parallelised computation of control scenarios without real-world encumberment, limiting the risk of damage to robotic hardware, and rapid reconfigurability [23]. Faster computation simply means that more experiments, and a greater hyperparameter search, can be conducted in a shorter amount of time. Furthermore, as we described in Section 2.5.3, models trained via simulation can be fine-tuned in the physical world using methods such as transfer learning, with less training required in the physical world and fewer of the inconveniences it poses as a result. In this sense, the simulation presented here is a stepping-stone towards physical deployment.

The simulation environment will be designed so as to reasonably reflect the real-world physical challenge. The model of the LBR iiwa present in the PyBullet library has been donated by KUKA themselves, and so will closely reflect the dynamics of the physical robot. However, no simulation could fully replicate all real-world dynamics and stochasticity so the likelishood is that any control policy trained exclusively in simulation should also be fine-tuned via transfer learning in a physical environment.

We will use Blender [154] for the production of simple assets for use in the PyBullet environment.

3.6 Software Libraries

The software implementation of this work will leverage prior-written software libraries. For simulation of our environment we will be using PyBullet[130], for the network model and deep learning algorithm we will be using Tensorflow[134] and Tensorflow Agents[155], and for experiment management we will be using Comet.ml [156], thus the software will be programmed in Python[157]. Tensorflow, PyBullet, and Python are all widely used within their respective application with good levels of online support to ease development and increase the likelihood of code reuse. At the time of writing, the latest versions of these packages were Tensorflow 2.6, Tensorflow Agents 0.9, PyBullet 3.2.0, and Comet.ml 3.15 and these were the versions used for implementation. Below, we state further justification for the adoption of these software technologies.

3.6.1 Machine Learning Library

Tensorflow is both the most widely used and fastest growing machine learning framework across a number of measures including wide research and industry usage [158]. In terms of its functionality, features, and ease of use it is fairly similar to other major industrially backed libraries such as PyTorch and the choice of deep learning library was fairly arbitrary here. However, it does have a dedicated library for reinforcement learning, Tensorflow Agents, which is endorsed for use with PyBullet[159] thus making it attractive for this work. Though there are several different reinforce-

ment learning libraries available, TF Agents is thought to be the best equipped for the scope of this work due to its ease of integration with other system components, relative ease of plugging in custom environments, range of high quality contemporary algorithmic implementations, and its tight integration with Tensorflow that allows for greater experimentation with network architectures including multimodal inputs, which was not offered in otherwise similar frameworks such as Stable Baselines at the time of implementation decision, although has been added since [160], [161].

3.6.2 Simulation Library

PyBullet is an easy to use Python module for physics simulation for robotics and machine learning with a focus on simulation-to-real transfer. PyBullet can load articulated bodies from various file formats and provides forward dynamics simulation, inverse dynamics computation, forward and inverse kinematics, collision detection, and supports virtual cameras [130], thus supporting all of the requirements for our simulated task environment. As discussed in Section 2.6.5, all of the main robotic simulation software have similar functionality and performance with regards to simulating robotic manipulation environments but PyBullet had a slight edge in terms of accuracy for most tasks[23] which is of particular importance to this work.

Furthermore, a related piece of work in the literature by Kalashnikov et al. from Google[71] (which our work builds upon in some regards), employed PyBullet for the simulation aspects of their work as a precursor to physical implementation. Their simulation environment, along with other variations using an LBR iiwa model, are available in the PyBullet libraries by default which allows us to quickly develop our simulation by using related examples for guidance as well as circumventing the need for building our own robotic model by using the included PyBullet KUKA LBR iiwa URDF model. PyBullet also encourages integration with TF-Agents for reinforcement learning purposes [159].

3.6.3 Experiment Management Library

Comet.ml is one of the most popular tools used by machine learning practitioners. It is a cloud-based meta-machine learning platform allowing data scientists to track, compare, explain, and optimise experiments and models. Comet.ml offers a Python library for data scientists to integrate their code with Comet and track work in the application with features including:

- Enabling the sharing of results with guest users
- Integrating well with ML libraries including Tensorflow
- Ability to log code, hyperparameters, metrics, dependencies, system metrics, and more
- Accessibility and simple experiment comparison
- Visualisation for images, videos, text, and tabular data

- Automated approaches for hyperparameter optimisation, including grid, random and bayesian methods.

While other experiment management libraries such as Neptune and Tensorboard do exist [162], the choice is relatively unimportant and Comet.ml fulfils the requirements for this work in terms of logging experimental data and assets, generating graphs for comparison and analysis, integrating with the rest of the deployed software libraries, and providing an accessible method for the sharing of data for reproducibility purposes [156]. As we stated in the literature review, reproducibility has been a serious issue in reinforcement learning research to date [118] so this was considered an important feature.

3.6.4 Programming Language

Python is generally accepted as the go-to choice for contemporary ML development as it is free, open-source, offers extensive libraries for scientific computing such as NumPy [163], and it is regarded as simple to setup, integrate, program, and read which is useful for quick implementation and prototyping. Its widespread use means that many community resources are available to ease development [164].

Tensorflow, PyBullet and Comet.ml are all supported in Python which allows for tight integration between the simulation, learning algorithm, and experimental management.

3.7 Experimental Approach

Following the algorithmic approach selection, the discovery and optimisation of a set of hyperparameters that allow an agent to learn to solve a task (in a reasonable amount of time) is the basis of all current applied reinforcement learning research. Hyperparameters can have significantly different effects across algorithms and environments [118] and finding an optimal set of hyperparameters can be extremely difficult.

In Section 2.6.4 of the literature review, we identified several strategies for hyperparameter optimisation including simpler grid and random searches, and more complex optimisation methods such as Bayesian optimisation, and Neural Architecture Search (NAS). Whilst ideally, we would look to incorporate these promising and more complex methods, their additional implementation complexity put them outside of the scope of this work. We instead employ a random search method, with the potential for later grid searches should sensitive hyperparameters be identified, due to their general effectiveness and ease of implementation.

Grid and random searches are amongst the most common methods employed in hyperparameter optimisation. Random searches have been found to generally converge to good hyperparameters faster than grid searches and the mechanism of random search means sensitive parameters can be identified more quickly than a grid search, and the resulting parameters can be selected for more thorough exploration than using a grid search method. Grid searches are generally used where there are

a very small number of hyperparameters to optimise, which is not the case with the number of overall parameters the deep reinforcement learning algorithms employed in this research are comprised of, but could be the case should sensitive hyperparameters be identified. Random searches also offer a number of other benefits for general experimental practice including early termination, asynchronicity, the ability to add additional experiments or restart failed experiments without disruption, and compatibility within additional controlled experiments [46, p. 415-423][121].

Furthermore, in the hope of reducing the number of trials required to determine a feasible set of hyperparameters, where possible we will set hyperparameter default values according to existing work in the literature, a strategy recommended by Andrychowicz [120]).

3.8 Chapter Summary

In this chapter, we have stated the methodology by which we will test the efficacy of deep reinforcement learning and hybrid visual servoing for robotic manipulation in fixtureless environments, and stated the reasoning and justifications by which we have pursued this methodology. In the following chapter, we will describe, justify and discuss our specific design choices in the implementation of a specific fixtureless control task, its MDP specification, and algorithmic choices relating to the reinforcement learning algorithm and neural network encoding.

Chapter 4

System Design

In this chapter we define and describe the objective of this research in terms of introducing a specific task environment representative of the aforementioned research area and its translation into a reinforcement learning problem before proposing a neural network-based control policy trained with deep reinforcement learning techniques within the described task environment to achieve the desired behaviour.

The chapter is organised as follows: In Section 4.1, we formally describe a robotic manipulation task representative certain problems in fixtureless manufacturing environments and explain the motivation of this work; in 4.2 we formulate our task environment as a Markov Decision Process; in 4.3 we describe our algorithm for solving the MDP with deep reinforcement learning techniques; and in 4.4 we propose a general neural network architecture for parametrising the agent's policy.

4.1 Task Description

The task environment consists of a planar work surface and a redundant manipulator, the KUKA LBR iiwa R800, with its base fixed to the surface. The surface is arbitrarily sized but the edges are such that they exist outside of the reach of the manipulator wherever it is attached, and is therefore assumed to at least be in the order of several metres across.

There are two RGB cameras present in the environment representative of a hybrid visual servoing (VS) setup: the first, known as the eye-in-hand (EIH) camera, is attached to the robotic end effector and moves with it; the second, known as the eye-to-hand (ETH) camera,is in a fixed location elevated above the work surface and placed behind and to the side of the robot. The ETH camera should be directed such that it captures the surface area within the working envelope of the LBR iiwa completely at a sensible position and angle that maximises visual coverage of the working area, whilst also being close enough so as to capture as much detail as possible within a given camera resolution, minimise potential for occlusion from the robotic manipulator, and ensure a good overhead view of the robotic end-effector as it approaches the work surface. An example of such positioning is given in Figure 4.1 with the ETH camera directed at the workspace from above facing diagonally at

the work surface, capturing both the work surface and the manipulator and its end effector. Depending on the distance from the work surface, the angle of inclination could be placed between 30° and 60° as an example.

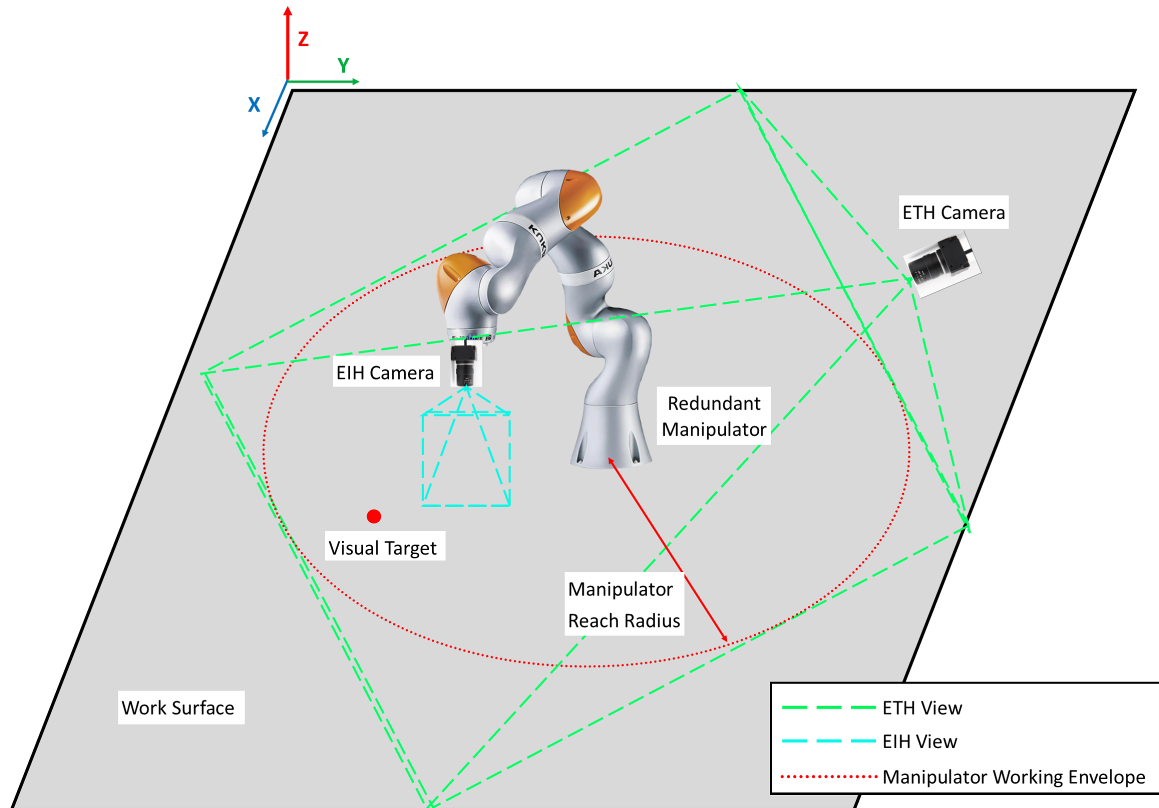

Figure 4.1: Task setup.

A small visual feature will be present on the plane of the work surface in a random location with reach of the robot manipulator in the form of a simple geometric shape with a colour / texture that is easily distinguished from the surface itself. The target feature will remain in its defined position for a set amount of time, after which it will visually disappear and a new feature will appear on the surface plane in a different, random location position within the robot's work envelope. Only one target will be visible on the work surface at any given time. The system is closed-loop with no human intervention and targets are automatically generated.

The task within this environment is for the robot to align its end effector with the focal point of the feature within some bound of accuracy a such that the flat surface of the end-effector is both facing and parallel with the work plane of the work surface and a fixed displacement, h above it in the order of several cm for the purpose of providing space for the EIH camera and some arbitrary end-effector tool. The robot must not collide with the work surface at any time.

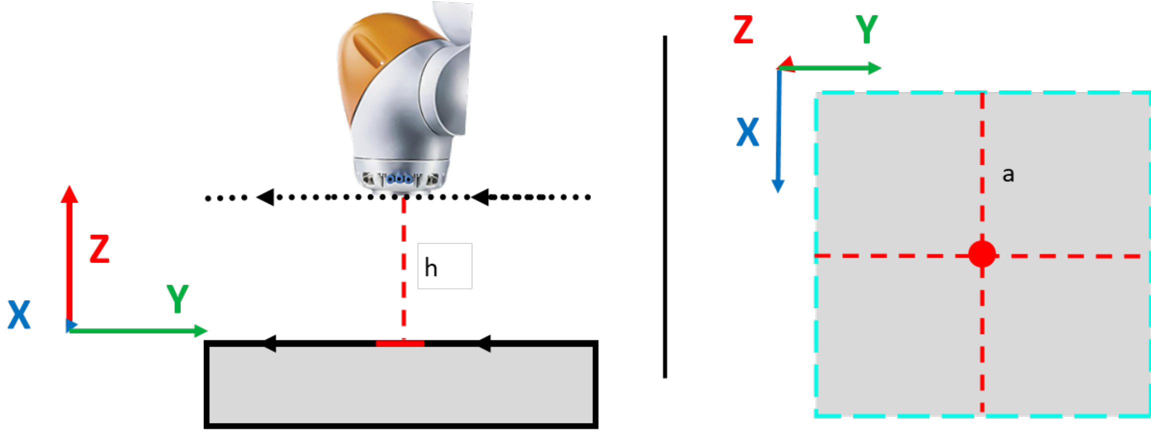

Figure 4.2: Goal attainment. **Left**: profile view; **Right**: end-effector view

This should be achieved by means of an agent acting within the environment with access to sensory data including live pixel data from the EIH and ETH cameras and numerical data from the robot's internal joint position and joint torque sensors, that is able to transform the relative orientation and translation of the robotic end-effector in 3D Space by affecting the positions of the seven actuators / degrees of freedom of the redundant manipulator. An agent can be said to have completed the task successfully if the robotic end-effector is aligned with the point in space above the visual cue at the correct orientation to within a fixed sub mm margin of accuracy and achieved this within the given time. As only one target is visible at any given time, tasks can be divided into 'episodes' based on the position of current visual cue.

4.2 Environment Modelling - Markov Decision Process

We model the described task environment E as an agent interacting with a variable-time episodic Markov Decision Process (MDP) with a state space $\mathcal{S}$, action space $\mathcal{A} = \mathbb{R}^N$, transition dynamics $\mathcal{P}$, reward function $\mathcal{R}$, and discount factor γ. Additionally we define E to have a set of goals $\mathcal{G}$, and an initial state-goal distribution $p(s_0, g)$.

Each episode is defined by selecting an initial state and a goal from the distribution $p(s_0, g)$. At each timestep t in the episode the agent receives a representation of the environment's state s in the form of an observation o_t and updates its internal state $s_t \in \mathcal{S}$. The agent then executes a an action $a_t \in \mathcal{A}$ based on s_t according to its parametrised policy $\pi_\theta(a|s_t)$. The environment in turn produces a scalar reward $r(s_t, a_t)$ and transitions to the next state s_{t+1} according to the transition probability $P(s_{t+1}|s_t, a_t)$.

The process continues until the agent reaches a state in a set of terminal states $s_t \in \mathcal{T}, \mathcal{T} \subset \mathcal{S}$. The set of terminal states $\mathcal{T}$ in any given episode is defined by the goal g during that episode. Goals describe the desired outcome of a task and dictate

the sequence of actions by the agent's policy to bring about a goal state such that $\pi : \mathcal{S} \times \mathcal{G} \rightarrow \mathcal{A}$. Every goal $g \in \mathcal{G}$ corresponds to some reward function $r_g : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$. Goals are extracted from the observation o.

In the section below, we define specific elements of the MDP mathematically. We give details on their technical implementation in Section 5.2.

4.2.1 States

The state of the system is represented in the physics engine and consists of angles and velocities of all robot joints as well as positions and rotations of all objects including the robot base, work surface, cameras, and goal.

4.2.2 Goals

Goals represent a desired end-effector position that is a fixed distance above the work surface within some fixed measure of accuracy ϵ. Since they can only appear on the surface at constant Z, goals can be described in two dimensions X and Y corresponding to their 2D position on the work surface in relation to the origin of the LBR iiwa's base, the origin point of the environment.

However, the purpose of a goal is that the LBR iiwa should position its end effector above it. Thus goals must also be generated to be physically attainable with regards to the LBR iiwa working envelope shown in Figures 4.3 and 4.4 and the constraints of the agent's task. Given a minimum and maximum working envelope radius d_{min} and d_{max} and a goal with X and Y coordinates g_x and g_y, a goal is valid if it satisfies the following Equations 4.1, 4.2, and 4.3.

$$-d_{max} \leq g_x \leq -d_{min} \vee d_{min} \leq g_x \leq d_{max} \tag{4.1}$$

$$-d_{max} \leq g_y \leq -d_{min} \vee d_{min} \leq g_y \leq d_{max} \tag{4.2}$$

$$g = (g_x, g_y), \quad g \in \mathcal{G}, \quad \mathcal{G} = \mathbb{R}^2 \tag{4.3}$$

Each goal corresponds to a particular reward signal $r_g \in \mathcal{R}$ and a set of terminal states $\mathcal{T} \subset \mathcal{S}$. At least one component of the reward signal r_g will rely on a function f_g to measure whether the goal has been achieved to compute a reward signal and terminate the episode if the agent successfully reaches a terminal goal state.

f_g is a binary reward signal based on whether the goal has been achieved or not within the desired accuracy ϵ. Given s_g and s_e are the respective positions of the goal and the end-effector in the state:

$$f_g(s) = [|s_g - s_e| \leq \epsilon] \tag{4.4}$$

4.2.3 Observations

Observations provide information about the state to the agent to inform its decision making. In our environment, observations are taken using various vision sensors and

the robot's own internal sensors. The observation space is multimodal and at any given timestep, the agent's observation o is:

$$o = (I_{EIH}, I_{ETH}, J_p, J_v) \tag{4.5}$$

where I_{EIH} and I_{ETH} are the images from the monocular EIH and ETH cameras and J_p and J_v are the robot joint positions and velocities respectively at time t.

I_{EIH} and I_{ETH} are encoded as $i_h \times i_w \times i_d$ matrices where i_h, i_w, i_d are the image's height, width, and depth respectively. Note that the dimensions of I_{EIH} and I_{ETH} are not necessarily identical. Meanwhile J_p and J_v are seven dimensional real-valued vectors of the robotic joint angles and velocities where each component has a value in the relevant ranges of Table 3.1.

The results of [71] and other work has shown that richer state representation results in faster convergence and better final performance than purely image-based states, thus we supplement the images with robotic joint position and velocity information, similar to approaches such as 2.5D visual servoing [38] Indeed, different sensors provide complementary information about an environment and to effectively perform a task, the robot should measure multiple feedback modalities and integrate multiple sensors into its control system [32].

Remembering that the environment is goal-based, and that at every timestep the agent should observe as input not only the current state but also the current goal to determine how it should act. Here, the goal is not given explicitly like some other work in the literature (e.g [91], [140]), rather it is a feature to be extracted from the observed images I_{EIH}, I_{ETH} and as such the accuracy of the definition may change based on the observations.

For this reason, the environment may in general be partially observed due to factors such as occlusion, sensor positioning and sensor resolution. However here, like some other work in the field ([67], [71]), we assume that the current observation provides all necessary state information due to the ETH camera configuration and treat the environment as fully observable such that $s_t = o_t$[1]. While a POMDP (see: Section 2.4.3) formulation would be most general, it has been found in practice that resulting control policies have still exhibited moderate robustness to occlusions [71] and extension to a POMDP is straightforward.

Whilst the observation does not provide all information about the state that could be provided (see: Section 4.2.1), it provides sufficient extractable information about variable parameters such as the robot join positions, goal position etc. and omits fixed parameters such as the robot's base position, work surface position etc. that are not required for a representation of the environment to be built. No other prior knowledge is given about objects, physics, or motion planning is provided to the model aside from the knowledge can be extracted autonomously from the data.

[1]Therefore, the terms observation and state are occasionally used interchangeably.

4.2.4 Actions

The agent's actions, $a \in \mathcal{A}$ consist of a seven dimensional vector where each component has a real value in the range $[-1, 1]$ such that $\mathcal{A} \in \mathbb{R}^7$.

$$a = (a_1, \ldots, a_7), \quad -1 \leq a_i \leq 1, \quad a_i \in \mathbb{R} \tag{4.6}$$

Each component in the vector represents an action based on positional control in the joint space rather than other forms of control for the LBR iiwa such as velocity or force control, Policy outputs are in the normalised interval $[-1, 1]$ so that they may then be scaled to the robot's actual joint value by multiplying the policy output value by the range of that joint as detailed in Table 3.1. Normalisation is often necessary for neural network input / output computation and also increases possibility for model reuse. If we rewrite Table 3.1 as a vector j:

$$j = (170, 120, 170, 120, 170, 120, 175) \tag{4.7}$$

Then the vector p of actual robotic joint positions relative to the zero angle is the Hadamard product of a and j such that

$$p = a \odot j \tag{4.8}$$

It is worth noting that positional actions are not necessarily achieved within a single timestep as this is dependent on the maximum velocity of the robot and the length of real time between each algorithmic timestep. The agent will merely act towards a positional value rather than necessarily achieving it. The velocity input will indicate the amount of movement occurring at any given timestep.

4.2.5 Initial State-Goal Distribution

Every episode starts by randomly sampling an initial state s_0 and goal g pair from some distribution $p(s_0, g)$. Once sampled, the goal remains fixed for that episode.

Studies exploring different strategies for generating the initial state-goal, are also proposed in Appendix D.

4.2.6 Rewards

The reward function r provides feedback that directs the agent towards achieving its goals and thus determine how the agent will act. In this environment, the reward function should encourage behaviour that not only achieves the goal set out in Goals, but also achieve these goals in a timely and efficient way. Efficiency could be defined according to several metrics, especially on redundant manipulators as demonstrated by the LBR iiwa motion planning algorithm which prioritises different metrics depending on whether moving in a linear or point-to-point manner, but here we will prioritise the time taken to reach a pose, relative to the linear distance from the initial robot state.

It was made clear in Section 2.4.3 that better performance is generally achieved using sparser reward signals as demonstrated in related work including [71], [91], but with consequences that the learning is more challenging and may require additional training time or the use of supplemental algorithmic techniques to make learning tractable. Here we define several reward subfunctions that reward different kinds of behaviour with different levels of sparsity that can be combined to shape the reward function as necessary.

We propose five subfunctions that may be incorporated into an experimental reward function r: r_g which provides essential goal-related feedback; r_c which provides feedback relating to collisions; r_e which encourages the agent to perform efficiently; and r_o and r_p which provide rotational and positional feedback to direct the agent towards achieving its goal along a gradient.

$$r(s) = r_g(s) + r_c(s) + r_e(s) + r_o(s) + r_p(s) \tag{4.9}$$

Goal Attainment, r_g

The sparsest possible reward function should simply assess whether a goal has been achieved by implementing the inequality in Equation 4.4. Some implementation of r_g, of knowing that the goal has been achieved, must be included in any variation of the reward function. It is important that the implementation is binary as described and not amended to provide continuous feedback as this would yield a gradient that would make the problem substantially easier to solve. While this information is easily determined in a parametrised simulation, it is substantially more difficult to determine in a real world physical environment and the MDP should reflect this.

Collision Evasion, r_c

Collisions are highly undesirable as they risk damaging the robot in physical environments. The agent should not collide with an object under any circumstances and a collision will be treated as a terminal state, ending the episode and incurring a large penalty.

$$r_c(s_t) = \left\{ \begin{array}{ll} -r^*, & \text{if collision detected} \\ 0, & \text{otherwise} \end{array} \right\} \tag{4.10}$$

Efficiency, r_e

It is well documented in the literature that adding a small reward penalty across time steps does well to encourage performance from a control system where speed is desirable, and is more effective for increasing long-term performance than decreasing the discount factor γ [40], [71].

A small penalty can be applied at every timestep to encourage the agent to achieve its goal in as few timesteps as possible. Our implementation applies a small negative reward at every timestep prior to termination where the goal state is not achieved as shown in Equation 4.11.

$$r_e(s_t) = k, k \leq 0 \tag{4.11}$$

Rotational Alignment, r_o

As a way of providing guidance in the search space and compensating for the other sparse reward signal r_g, reward shaping could be provided via a reward gradient that encourages movement towards the goal position.

There are two gradients that can be provided. The first is to guide towards the correct end effector orientation r_o and the second is to guide towards the correct end-effector position

The most simple and intuitive heuristics that could be used for guidance could be based on Euclidean distances, d in both the orientation and position spaces between the goal orientation and position and the end-effector orientation and position by computing the forward kinematics on data attained from the robot's positional sensors where the reward inversely changes with regards to the distance magnitude.

The Euclidean distance d between two vectors e, g is described as:

$$d_n(e, g) = \sqrt{(e_1 - g_1)^2 + \cdots + (e_n - g_n)^2} \tag{4.12}$$

where n is the number of arguments required to describe the vector.

To apply this heuristic to orientation, the agent must orientate its end-effector in the way seen in Figure 4.1 If the orientation of the goal in relation to the robot's base and end-effector is known, as it is in this particular task, then feedback can be given to the agent to encourage alignment with this orientation.

Given this, we can calculate the absolute maximum rotational distance that the end-effector may be at any time from the goal orientation and we can reward the agent proportionally to how close its actual orientation is to the goal value based on the maximum possible rotational displacement from the goal, $\theta_{max} = 180 - \epsilon_o$.

Given we calculate the Euclidean distance between the end-effector orientation e_o and goal orientation g_o expressed as vectors.

$$\delta_p = d(e_\theta - g_\theta) \tag{4.13}$$

The rotational reward r_o can be calculated:

$$r_o(s_t) = \left\{ \begin{array}{ll} k \times r^*, & \text{if } \delta_o \leq \epsilon_o \\ k \times r^* \times \frac{(\theta_{max} - \delta_o)}{\theta_{max}}, & \text{if } \delta_o > \epsilon_o \end{array} \right\} \tag{4.14}$$

where k is some constant.

Positional Alignment, r_p

We employ a similar method to r_p as we do r_o. However, instead of taking θ_{max} as the maximum possible distance, we make an approximation based on the robot's range as determined by its working envelope in Figures 4.3 and 4.4.

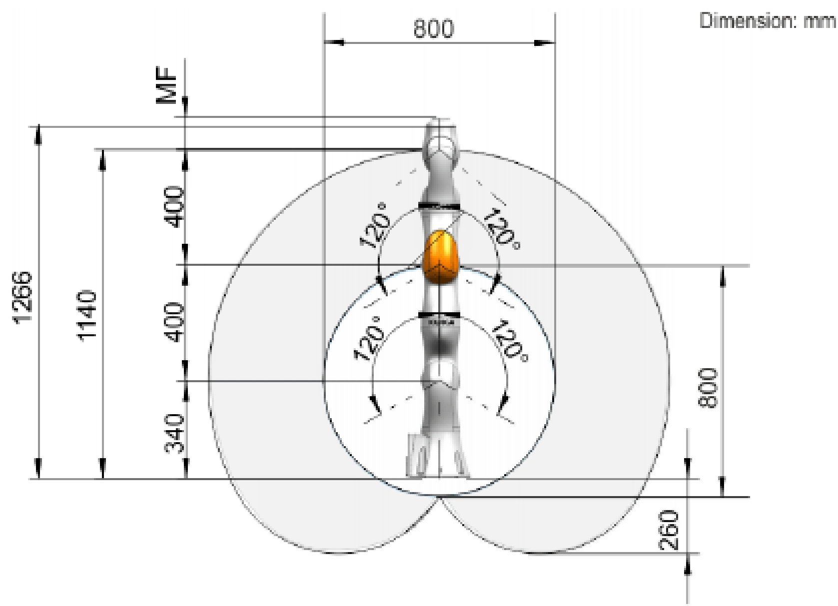

Figure 4.3: KUKA LBR iiwa working envelope, side view [142]

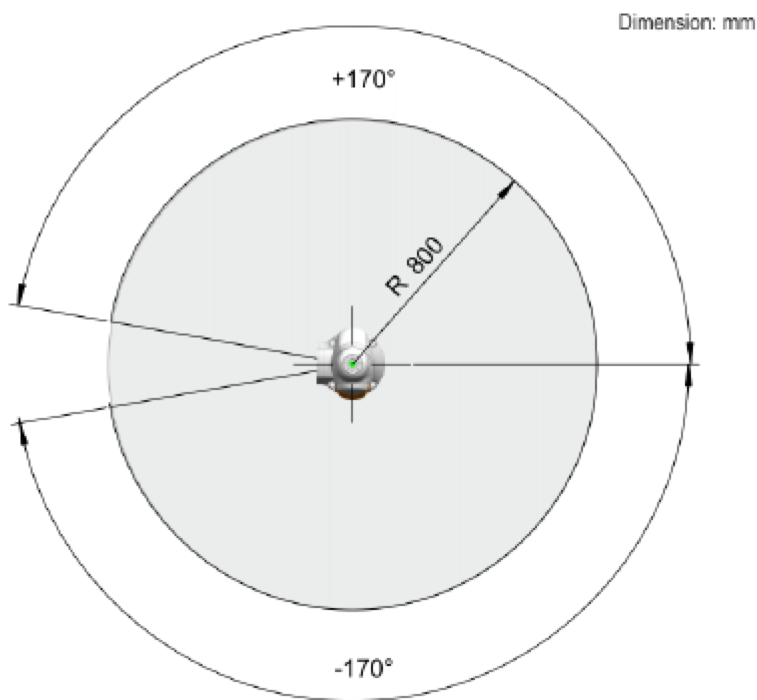

Figure 4.4: KUKA LBR iiwa working envelope, top view [142]

We can therefore make the rough assumption that the maximum possible distance that the robot may be from the goal position is $l_{max} = (2 \times l) - \epsilon_p$ where l is the reach of the robot manipulator.

Given we calculate the Euclidean distance between the end-effector position e_p and goal position g_p expressed as coordinate vectors.

$$\delta_p = d(e_p - g_p) \tag{4.15}$$

The positional reward r_p can be calculated:

$$r_p(s_t) = \left\{ \begin{array}{ll} k \times r^*, & \text{if } \delta_p \leq \epsilon_p \\ k \times r^* \times \frac{(l_{max} - \delta_p)}{l_{max}}, & \text{if } \delta_p > \epsilon_p \end{array} \right\} \tag{4.16}$$

where k is some constant.

Although, r_g, r_c and r_e are sparse enough.that they are unlikely to lead to the development of suboptimal policies, it is possible that the more heavily shaping functions r_o, r_d could lead to the agent learning sub-optimal behaviour and performing sub-optimal trajectories. For example, while the implementations described here generally reward alignment with a specific final orientation or axis position at any point in time regardless of whether other measurements are correct, it is possible that more efficient trajectories exist where divergence from the final orientation, or travelling non-linearly during the trajectory would be a more efficient trajectory overall due to the LBR iiwa's physical constraints. Therefore, the functions r_o and r_p should be carefully studied with regards to their shaping potential and may require tuning of the the discount factor γ to generate more efficient trajectories and movement behaviour.

4.3 Reinforcement Learning Algorithm

4.3.1 Learning Algorithm

There are a number of deep RL algorithms with different efficacies and applicability to different problem domains [73]. The algorithm we employ needs to be sample efficient and capable of learning in an environment with high dimensional observation spaces and continuous action spaces. We believe that the SAC algorithm [165] is best suited for learning on our particular environment for the following reasons.

Firstly, policy-based RL algorithms are generally much more effective in high-dimensional and continuous action spaces as there is more generalisation of the environment interaction (and thus fewer parameters to learn) and they can learn stochastic policies which are better for interacting with dynamic environments than deterministic policies, and have better convergence properties [73]. Secondly, if using a policy-based learning algorithm, it is also preferable to use an actor-critic method as these are known to reduce variance in the reward signal and these methods have been also been shown to be effective in robotics domains [81], [165]. Thirdly, neural networks are necessary for scaling up reinforcement learning methods to high dimensional state-spaces including computer vision applications, therefore the algorithm employed must be applicable to training neural network policy parametrisations. Fourthly, the algorithm should be off-policy as these are known to be generally an order of magnitude more sample efficient for training as samples can be reused without overfitting and sample efficiency is key for robotics applications [92].

However, based on these criteria and the current state-of-the-art alone, several other algorithms could be deemed suitable for this task including ACKTR [81], PPO [87], or SAC [165] as they have all yielded promising results on robotic control problems. But the performance of state-of-the-art RL methods depends on careful setting of the hyperparameters, and even then performance can vary substantially between runs with off-policy algorithms tending to suffer even more from these issues than on-policy policy gradient methods [92], [118]. RL algorithms should be robust to hyperparameter settings through their design, but this is an extremely challenging area of research. One approach towards this is developing algorithms that automatically tune their own hyperparameters. This is exemplified most recently by SAC which contains automated temperature tuning and has been demonstrated to greatly reduce the need for hyperparameter tuning across domains [92]. With SAC requiring the fewest hyperparameter settings of the three aforementioned algorithms, as well as being the most recently developed and currently accepted as the state-of-the-art for general robotics RL applications, this outlines it as the most favourable algorithm for application.

For reference, the SAC algorithm is given in Appendix A. We will be using the implementation featured in the TF Agents Library [155].

4.3.2 Training Procedure

The training procedure used follows the same steps as the SAC Minitaur example in the TF Agents Docs [166]. That is, an initial period of random data collection to seed the replay buffer followed by training and periodic evaluation.

4.3.3 Exploration Policy

The exploration policy is relatively simple and follows the example used in [166]. We seed the replay buffer by collecting experience using a random policy for 500 episodes, or approximately 10,000 steps. The agent will then begin collecting experience using the neural network-based policy initialised with random (normalised) weights inducing further random exploration.

4.4 Neural Network Architecture

Designing a neural network to parametrise an RL agent can be non-trivial as there are a multitude of design parameters that affect a network's performance including: number of layers, number of nodes in a layer, activation functions, initialisation, and even its code-base implementation. Choosing good parameters is made more difficult by inconsistent or incomplete reporting of parameters in the literature making comparison of various approaches difficult.[118]. Furthermore, the success of a particular architecture is somewhat application-dependent, though certain techniques are known to work well across a range of applications. Although the neural network architecture

is not always the most significant contributor to function modelling, the size / depth of the network must reflect the complexity of the function to approximate. [54]–[56].

In 4.4 we describe the general network architecture that we believe would be capable and effective in learning to solve our environment and illustrate the general network architectures for the actor and critic networks in Figures 4.4.1 and 4.4.1 respectively, then in 4.4.1 we discuss some of the reasoning behind some of our chosen network properties based on other works in the literature, referencing the works [71], [81], [91], [167] in particular.

We have limited the definition of some network parameters to the experiments in Section 6. This is to acknowledge that a good network structure with specified parameters will need to be determined through optimisation and would be the subject of Future Work.

4.4.1 Network Description

We employ an actor-critic approach with an architecture that contains several sub-networks and a mixture of convolutional and fully connected layers. The actor and critic networks are separate but identical with the exception of their inputs and outputs.

The networks may take as input the pixel values of two RGB images, which may differ in size, and two arrays of numeric inputs representing the robotic joint position and velocity values respectively, which in the case of the KUKA LBR iiwa is fixed at seven joints. All input and output values are continuous and scaled to the range $[-1, 1]$. All four inputs are modular and initially processed separately with dedicated and customised pre-processing layers. The image inputs should be pre-processed with a convolutional neural network (CNN), while the vector inputs should be processed by a fully connected, feed forward, multi layer perceptron (MLP).

Following pre-processing, the inputs are flattened (if necessary) and concatenated into one input array which is processed by another fully-connected, feed forward MLP before generating the respective network output. The actor network outputs a normalised action a in the form of a seven-valued array representing a joint position for each of the robot's joints as defined in Equation 4.6. The critic network takes an additional input which is the output of the actor network which can also have its own dedicated pre-processing prior to concatenation. The critic network outputs a single value calculating the advantage of the current state and action combination.

All activation functions within the network are ReLUs with the exception of the output layer of the actor network which uses the tanh activation function.

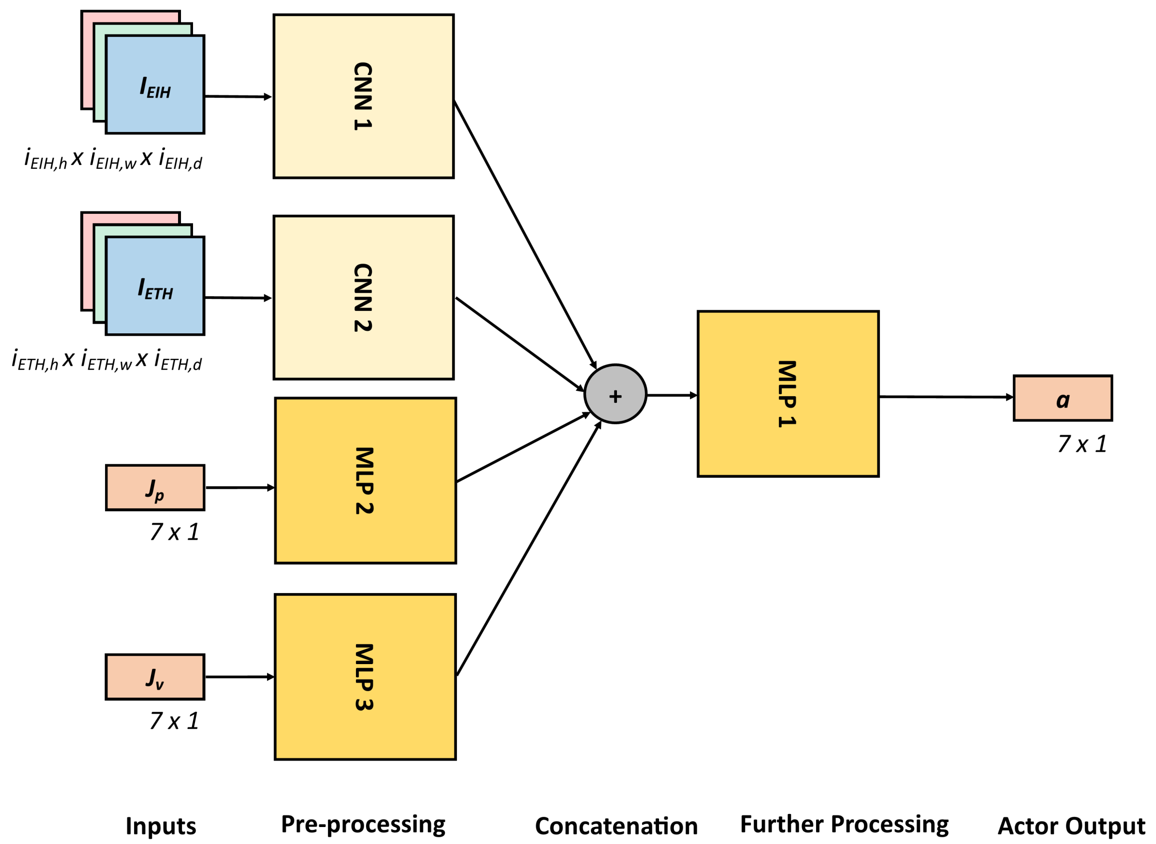

Figure 4.5: General actor network architecture.

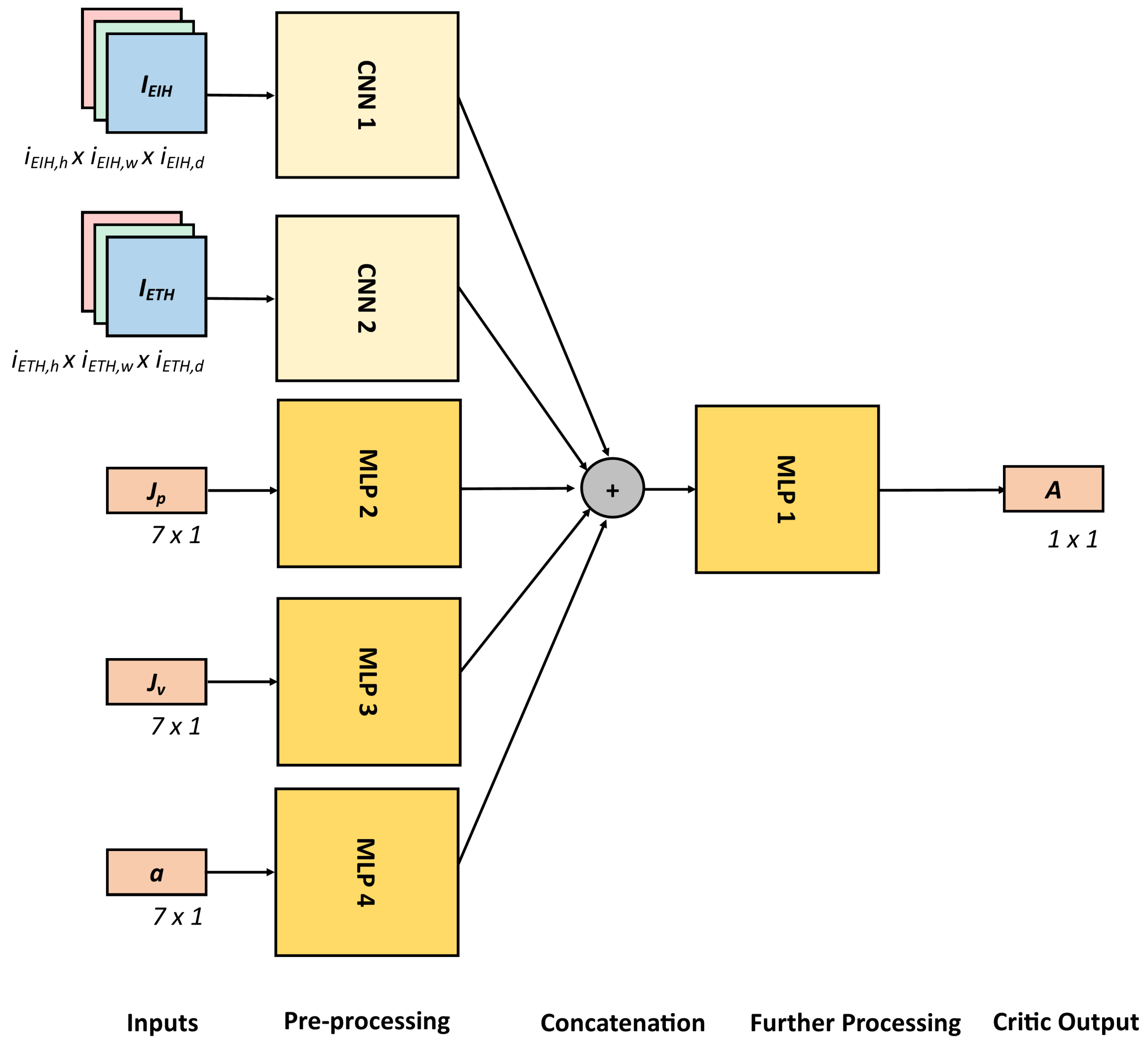

Figure 4.6: General critic network architecture.

4.4.2 Network Discussion

Structure

The state-action function we wish the agent to learn is complex and not linearly separable and necessitates deep neural networks with hidden layers. A majority of contemporary work in ML and RL utilise deep neural networks as we previously discussed that adding more layers can reduce the overall size of the networks thus making them more computationally efficient. Recognising the many successes CNNs have enjoyed in computer vision, our networks will also contain convolutional layers as we have a partially image-based input.

The actor-critic approach was chosen due to the success of such approaches to robotic control in work such as [67], [80], [81]. Actor-critic architectures can either be separate networks or contain shared parameters [67], [81], [88]. There is no consensus in the literature as to whether combined or separate networks are more effective. However, the main perceived advantage of a combined network is that it is more computationally efficient to train and store as you only need to update one network in each training step, whereas the perceived advantage of separate networks is that they can learn their own dedicated function and thus may be easier to train. Since we are making use of cloud computing resources and the general size of our networks is likely to be fairly small, we have opted for separate architectures for simplicity and the perceived training advantage.

Inputs

Each network can take as input up to two images of size $i_h \times iw \times i_d$ pixels, as well as two arrays of seven mono-dimensional numeric inputs representing the robotic joint position and velocity values (see: Section 4.2.3 . All input values are continuous and normalised to the range $[-1, 1]$ as neural networks have difficulty dealing with inputs of different magnitudes and normalisation is recommended by Andrychrowicz [120] and used across work in the literature.. The modularity of the networks prior to concatenation mean that inputs can be added or removed as necessary. Hence the structure of the actor and critic networks are very similar.

Hidden Layers and Outputs

It is currently established in the literature that CNNs are the standard for processing image-based data in neural networks. However, we believe that our network does not require extensive convolution because: (i) we are mostly interested in the location of distinctly coloured features in the input images and thus the features are relatively simple; (ii) we are not interested in extracting complex features e.g. the robot's pose from images as this is encoded by the joint position data and the rest environment is otherwise static; and (iii) the resolution of our images is be reasonably low for efficiency reasons which limits the detail of features anyway.

For instance, [81] make use of 42×42 pixel images for continuous control tasks and their entire architecture consisted of two convolutional layers with 32 filters of size 3×3 with stride 2 followed by a fully connected layer of 256 hidden units. Whilst this can't necessarily be translated to our domain due to the added complexity, it gives an indication of the scale that might be needed.

Furthermore, our networks do not need to be homogeneously convolutional as we are dealing with low-dimensional numerical inputs which are not spatially related for the robotic joint positions, velocities, and actions for which simpler processing with an MLP is more appropriate.

This is reflected in [81] where convolutional layers are used for pixel-based continuous control tasks but not continuous control tasks with a low-dimensional input. The structure and scale of our hidden layers is also adapted from [81] where we have

used 64 hidden units per layer in a two-layer fully-connected network for the joint sensor input and two convolutional layers containing 32 filters of size 3×3 and stride 2. It is also common practice to use dense MLP layers to simplify the output of any neural network as shown frequently in the literature [71], [81], [91], [134].

Concatenation

Our network accepts multiple input sources which requires a concatenation of input values. Due to their differing formats, it is sensible to group inputs into images and robotic joint values and process them in similar ways based on their type before concatenating them later. However, since one set of inputs should be processed by a CNN, and the other should be processed by a MLP, concatenation is not straightforward and there needs to be conversion of one type of format into the other. Though their network is substantially different, we take a similar approach to Kalashnikov et al. who concatenate image and low-dimensional sensory data after some initial processing on both inputs separately before further processing together [71]. Although since our network is smaller and also favours modularity, we instead flatten the output from the convolutional layers into a single fully-connected layer to be instantly processed by the output MLP, rather than convert the sensor data into an additional convolutional channel and processing with further convolutional layers.

Activation Functions

ReLUs are the most commonly employed activation function for hidden units in CNNs and we employ them here for every layer except the output layers which use the tanh activation function and bound output values to the normalised range [-1,1], which is suitable for scaling. The use of tanh and bounding for the output layers is recommended by Andrychowicz [91], [120].

4.5 Chapter Summary

In this chapter, we have presented our specific design choices in the implementation of a specific fixtureless control task, the MDP specification of this task, and algorithmic choices relating to the reinforcement learning algorithm and neural network encoding, as well as justifying and discussing the design of each component. In the following chapter, we describe key features of environment, training algorithm, and 3D models in terms of their specific software implementation.

Chapter 5

Software Implementation

In this chapter, we describe key features of the software implementation for our environment, training algorithm, and 3D models.

The most relevant resources used for reference during the writing of this software included the Tensorflow and TF-Agents documentation and examples, in particular the SAC Minitaur [166], custom network [168], and custom environment [169] examples, the PyBullet documentation and examples [159] in particular the KukaCam Environment [170], the Comet.ml docs and examples [156], the Blender documentation [154], and numerous other online resources for general programming queries such as the Python documentation and Stack Overflow.

The chapter is organised as follows: In Section 3.6, we describe and explain the software libraries used for our software implementation; then in 5.1 we describe the 3D models used for modelling and rendering our simulation environment; and in 5.2 we describe our PyBullet environment implementation including key functions, algorithms and variables.

5.1 3D Modelling

The visual render of the environment requires three 3D models for the KUKA LBR iiwa, work surface, and target. The LBR iiwa is a .urdf model imported from the PyBullet library [130] whereas the work surface and target are simple .obj files created in Blender [154]. The work surface object is a grey square plain of dimensions 2m x 2m. If the robot is placed in the centre of the object, this means that the outside edge of the work surface is just out of the robot's reach in any direction. The target object is a red circular plain with 1cm diameter which will be placed just above the work surface plain and is clearly visible against the work surface. The simplicity of the objects means they can scaled easily which may be particularly helpful for different target sizes and striking the right balance between image input resolution and target size.

The floor and target .obj files can be found at the code repository in Appendix B.

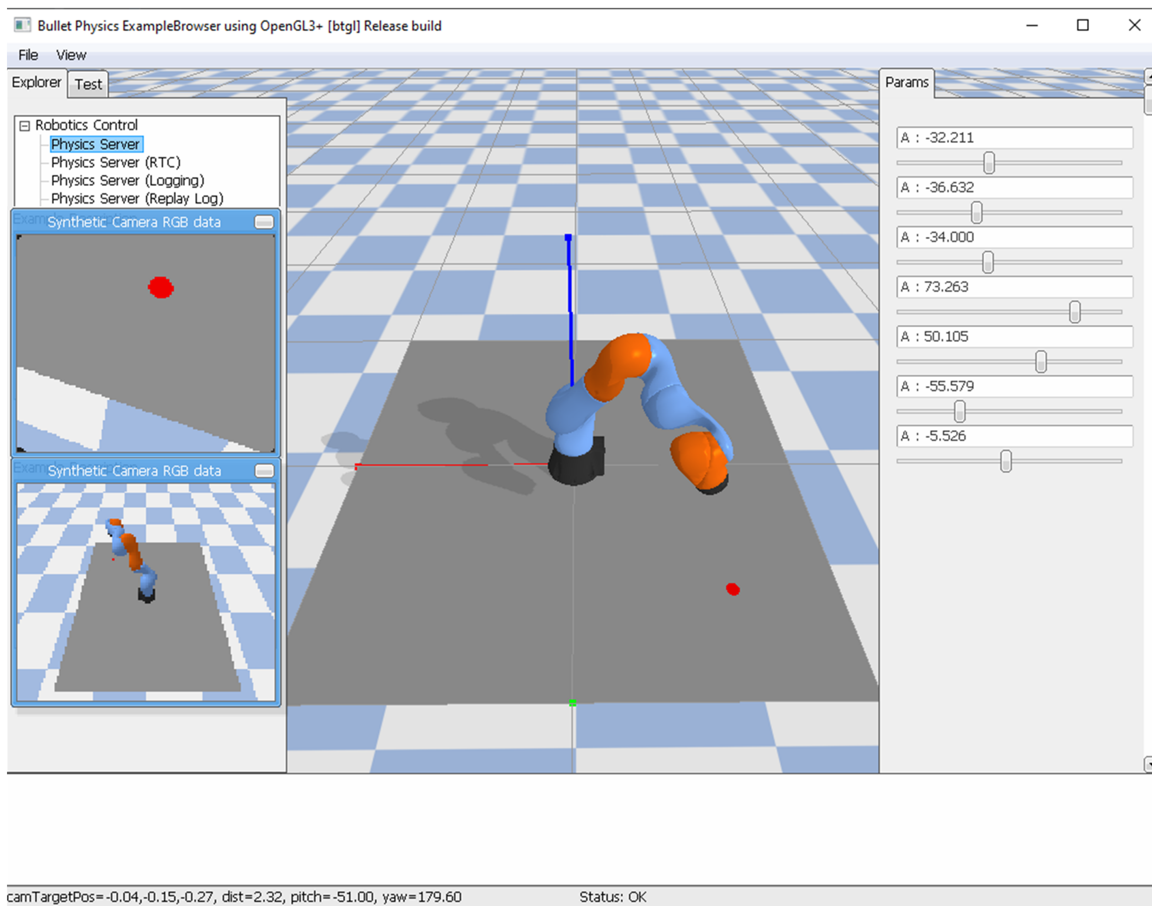

Figure 5.1: Render of simulated environment in PyBullet GUI showing a typical placement of the floor, target and robot, with windowed eye-in-hand and eye-to-hand camera views in the left panel and joint angles A1:7 in degrees in the right panel.

5.2 PyBullet Environment

Our RL environment has been implemented as a TF-Agents `PyEnvironment` class so that it can interface with the TF Agents RL algorithm implementations. A PyEnvironment interface was chosen as it is native method for implementing custom environments in our chosen framework TF Agents, however it would be trivial to convert the environment to other standards such as an OpenAI Gym which have nearly identical environment formatting and indeed the TF Agents library also accepts OpenAI Gym Environments as input. Each custom environment that implements the PyEnvironment interface must contain (as a minimum) custom implementations of the following methods with inputs and outputs in a defined format[169].

- `action_spec()` Describes the TensorSpecs of the action expected by `step(action)`
- `observation_spec()` Defines the TensorSpec of observations provided by the environment.

- `reset()` Returns the current TimeStep after resetting the Environment.
- `step(action)` Applies the action and returns the new TimeStep

As well as PyEnvironment specific functions, our environment implementation also contains a number of other functions that are called by the above functions to assist with modelling the environment.

- `getJointStates()` Returns the robot's current position and velocity of all joints.
- `getKukaJointLimits()` Returns the robot's upper and lower joint limits as described in Table 3.1.
- `getKukaVelocityLimits()` Returns the robot's velocity limits.
- `setKukaJointAngles(jointPositions)` Sets the desired positions for the robot's joints which the robot will move towards during the following simulation steps until instructed otherwise at the next algorithmic time step. Will not move beyond joint physical joint limits in Table 3.1.
- `_getEyeInHandCamera()` Returns an RGB image from the eye in hand camera with the dimensions set in the environment constructor (see: Section 5.2.1).
- `_getEyeToHandCamera()` Returns an RGB image from the eye in hand camera with the dimensions and at the location and orientation set in the environment constructor (see: Section 5.2.1).
- `_pixelArray2RGBArray(rgba_seq, img_dep, img_res)` Converts RGBA data from `getEyeToHandCamera` / `getEyeInHandCamera` into 3 separate arrays of RGB data.
- `getObservation()` Collects observations from all enabled sources and concatenates them into one observation so as to be in the correct format to be returned by the RL algorithm.
- `normalisedAction2JointAngles(action)` Converts normalised joint action values [0..1] into equivalent radian values for each robotic joint range.
- `normaliseJointAngles(jointPositions)` Converts a set of robotic joint positions in radians to their normalised value, relevant to the respective joint range i.e. 0 is the minimum joint angle for that particular joint and 1 is the maximum, regardless of if the joint range differs between joints.
- `_termination()` Checks if the conditions for termination have been met. I.e. if the step counter has exceeded the maximum number of steps in an episode, a collision has occurred, or the goal has been achieved.

• `_reward(reward_goal, reward_collision, reward_time, reward_rotation, reward_posit`
Calculates the reward for a given timestep (see: 5.2.8).

• `generateGoal(x, y, size)` Generates a new goal and renders it in the environment. Goal postion can be set manually with inputs `x, y` or will otherwise be randomly generated within the robot's working envelope as defined by *KUKA_MAX_RANGE* and *KUKA_MIN_RANGE* (see: Table 5.2). The size of the goal can also be set with the size parameter but is set to 1cm diameter by default.

• `generateFloor(size)` Renders floor in environment. Size of floor can be set with the size parameter but is set to $1m^2$ by default.

• `generateRobot(jointPositions)` Sets robot's initial position and renders robot in environment. Initial position is set 0 for all joint angles i.e. each joint is at the midpoint of its range and the robot is vertically straight.

We describe in greater detail below the key functions of the implementation, ignoring those which are simple and fully explained in functional terms by their short description above.

5.2.1 Constructor

When creating a new instance of an environment, there are multiple options to tweak its behaviour which can be set via the constructor method. These variables and their behaviour are described in Table 5.1 and their effect on algorithmic performance is described in 6.

Variable	Data Type	Default Value	Function
seed	*int*	*None*	Randomisation seed
timesteps	*int*	*20*	The maximum number of steps that can be taken by the agent per episode. Also written as T
discount	*float*	1.0	RL Discount Factor. Also written as γ
eih_input	*bool*	*True*	If *True*, then returns EIH camera input at each timestep.
eth_input	*bool*	*True*	If *True*, then returns ETH camera input at each timestep.
position_input	*bool*	*True*	If *True*, then returns robotic joint positions at each timestep.
velocity_input	*bool*	*True*	If *True*, then returns robotic joint velocities at each timestep.
eih_camera_ resolution	*int*	112	Sets the resolution of the EIH camera input (equal length and width).
eth_camera_resolution	*int*	112	Sets the resolution of the ETH camera input (equal length and width).
eih_channels	*int*	3	Sets the depth of the EIH camera input. This value should either be 1 or 3 for monochromatic or RGB images.
eth_channels	*int*	3	Sets the depth of the ETH camera input. This value should be 1 for monochromatic or 3 for RGB images.
target_position	*[float, float]*	*None*	If set, target is placed in a repeatable position on the worksurface for each episode.
reward_goal	*bool*	*True*	If *True*, include whether goal has been achieved in reward signal.
reward_collision	*bool*	*True*	If *True*, include collision detection in reward signal.
reward_time	*bool*	*True*	If *True*, include time penalties in reward signal.
reward_position	*bool*	*True*	If *True*, include rewards for lateral positioning towards target position in reward signal.
reward_rotation	*bool*	*True*	If *True*, include penalties for deviation from target orientation in reward signal.
normalise_observation	*bool*	*True*	If *True*, observations are returned normalised in the range [0..1] for images and [-1..1] for joint data.

Table 5.1: Environment customisation options available in class constructor.

5.2.2 Constants

As well as all of the accessible variables in Table 5.1 there are also defined several immutable constants which are described in 5.2.

Constant	Data Type	Value	Function
KUKA_MIN_RANGE	*float*	0.40	Specifies inner range (M) of LBR iiwa's working envelope where a target can be placed and the robot can align
KUKA_MAX_RANGE	*float*	0.78	Specifies outer range (M) of LBR iiwa's working envelope where a target can be placed and the robot can align
KUKA_VELOCITY_LIMIT	*float*	1.0	Specifies a limit on the maximum velocity. Required to prevent simulation exceeding joint limits during a timestep but not linked to the LBR iiwa's real velocity limits
SIMULATION_STEP_DELTA	*float*	$\frac{1}{240}$	Real time between each simulation step. Also written as $\delta t'$.
SIMULATION_STEPS_PER_TIMESTEP	*int*	24	Number of simulation steps in one algorithmic timestep. Also written as t'.
POSITIONAL_TOLERANCE	*float*	0.001	Accuracy requirement for end-effector position to goal in m. Also written as ϵ_p.
ROTATIONAL_TOLERANCE	*float*	1.0	Accuracy requirement for end-effector orientation to goal in deg. Also written as ϵ_o.
VERTICAL_DISTANCE	*float*	0.05	Height above work surface to align with in m. Also written as h.
MAX_REWARD	*float*	1.0	The maximum reward that the environment can return. May be positive or negative. Also written as r^*.
TIME_PENALTY	*float*	0.05	Penalty applied per timestep if *reward_time* is set.
POSITION_REWARD	*float*	0.4	Maximum reward that can be awarded for positional alignment if *reward_position* is set.
ROTATION_REWARD	*float*	0.4	Maximum reward that can be awarded for rotational alignment if *reward_rotation* is set.

Table 5.2: Environment class constants

5.2.3 Observation Spec

The agent's observations are highly customisable and dependent on the relevant input variables passed to the environment's constructor including: `eih_input`; `eth_input`; `eih_camera_resolution`; `eth_camera_resolution`; `eih_image_depth`; `eth_image_depth`; `position_input`; `velocity_input`;

Depending on the variables flagged, the function `observation_spec()` will generate an observation specification in the format of a list of `tf_agents.specs.ArraySpec` as required by implementing the PyEnvironment interface. In our particular instance, each component of the observation is represented as a `tf_agents.specs.BoundedArraySpec` as shown in table 5.3.

Component	**Matrix Shape**	**Data Type**	**Minimum**	**Maximum**
I_{EIH}	$i_h^{EIH} \times i_w^{EIH} \times i_d^{EIH}$	*float*	0	255
I_{ETH}	$i_h^{ETH} \times i_w^{ETH} \times i_d^{ETH}$	*float*	0	255
J_p	7×1	*float*	$-j_i$	$+j_i$
J_v	7×1	*float*	None	None

Table 5.3: Observation Spec Components.

There are two aspects of the observation spec that are worth remarking upon. Firstly, that there are no bounds for joint_velocities due to current limitations of PyBullet not factoring velocity limits into motor control commands. [159, p. 23]. Secondly, that the the RGB data is represented by a matrix of floating point values even though they could naturially be represented as integers in the range [0..255] this is for the sake of compatibility and having uniform data types across all possible observation specifications. Pre-and-post-processing will round floats to their nearest integer value.

The simple function used to systematically include elements in the observation specification with their specified customisation is shown in 1.

Algorithm 1 Observation Spec Function

1: *observation_spec* $\leftarrow$ []
2: **if** *eih_input* **then**
3: Append $I_{EIH}(i_{EIH,h} \times i_{EIH,w} \times i_{EIH,d})$ to *observation_spec*
4: **end if**
5: **if** *eth_input* **then**
6: Append $I_{ETH}(i_{ETH,h} \times i_{ETH,w} \times i_{ETH,d})$ to *observation_spec*
7: **end if**
8: **if** *position_input* **then**
9: Append J_p to *observation_spec*
10: **end if**
11: **if** *velocity_input* **then**
12: Append J_v to *observation_spec*
13: **end ifreturn** *observation_spec*

5.2.4 Action Spec

The action spec is immutable in our environment and not subject to any customisation options. The `action_spec()` methods is required to return an action spec in the format of a `tf_agents.specs.ArraySpec`. In our case we have represented it as a `tf_agents.specs.BoundedArraySpec` with normalised bounds as shown in table 5.4.

Matrix Shape	Data Type	Minimum	Maximum
7×1	*float*	-1.0	1.0

Table 5.4: Action Spec

5.2.5 Observation Function

The logic in this function is very similar to that of the observation spec function in that only the specified observations are collected and included in the function output. Each individual observation is retrieved via a dedicated function that extracts the observation directly from the PyBullet simulation which are *getEyeInHandCamera()*, *getEyeToHandCamera()*, and *getJointStates()* repsectively.

Algorithm 2 Observation Function

1: $o_t \leftarrow []$
2: **if** *eih_input* **then**
3: $\quad o_{eih} \leftarrow getEyeInHandCamera()$
4: $\quad$ Append o_{eih} to o_t
5: **end if**
6: **if** *eth_input* **then**
7: $\quad o_{eth} \leftarrow getEyeToHandCamera()$
8: $\quad$ Append o_{eth} to o_t
9: **end if**
10: **if** $position_input \lor velocity_input$ **then**
11: $\quad o_p, o_v \leftarrow getJointStates())$
12: $\quad$ **if** *position_input* **then**
13: $\quad\quad$ Append o_p to o_t
14: $\quad$ **end if**
15: $\quad$ **if** *velocity_input* **then**
16: $\quad\quad$ Append o_v to o_t
17: $\quad$ **end if**
18: **end if**
$\quad$ **return** o_t

5.2.6 Reset Function

The reset function initialises the PyBullet environment for a new episode by resetting the timestep, goal achievement, collision detection, and termination variables,

t, *goal_achieved*, *collision*,and *terminated* respectively and setting the initial state-goal by initialising the robot to its initial position and generating a new goal. An observation o_0 of the initial state s_0 at initial timestep t_0 is then returned.

Algorithm 3 Environment Reset Function

```
1: terminate ← False                                  ▷ Reset episode variables
2: goal_achieved ← False
3: collision ← False
4: t ← 0
5: generateRobot()                                    ▷ Initialise 3D Models
6: generateFloor()
7: generateGoal()
8: o_0 ← getObservation()                             ▷ Get initial observation
       return o_0
```

5.2.7 Step Function

The step function is responsible for advancing the environment by one algorithmic time step. It takes as input the action that the agent should perform in the next step, performs that action and updates the simulation, checks whether the episode should be terminated, and then returns an observation of the environment and a reward after completing that action.

Algorithm 4 Step Function

```
1: setKukaJointAngles(a_t)                    ▷ Set desired action position
2: for t' = 0 to T' do        ▷ Advance simulation in real-time until next timestep
3:     E.stepSimulation()
4: end for
5: t = t + 1
6: terminate ← termination()           ▷ Check if episode should be terminated
7: o_{t+1} ← getObservation()                ▷ Get observation at new timestep
8: r_{t+1} ← reward()                        ▷ Calculate Reward in current state
       return o_{t+1}, r_{t+1}, terminate
```

Every episode consists of up to 50 algorithmic timesteps, t. Each algorithmic timestep t is comprised of 24 PyBullet steps t' of length $\delta t' 1/240$ seconds. Therefore, each episode may have a maximum length of 10 seconds in real-time execution (not processing time). These values were set based on examples in the literature and PyBullet examples [71], [91], [130] with the total admissible episode length being adapted generously to allow the LBR iiwa enough time to satisfactorily position itself within the simulated velocity settings.

5.2.8 Reward Function

The reward function includes the five shaping options described in Section 4.2.6. Each shaping option can be activated using the environment constructor with the variables *reward_goal, reward_collision, reward_time, reward_rotation, reward_position* where they are later passed as arguments to the reward function during the step function (see: Algorithm 4). Assuming each shaping option of the reward function is activated, the reward function is sequenced such that the collision detection reward takes precedence and discards any other reward calculations if a collision is detected. The time penalty also adjusts any other reward calculated before it. The goal reward and the position / rotation rewards are mutually exclusive as the position / rotation reward shapings are only calculated if the goal has not been achieved, which is not the case if the conditions for the goal reward have been met and at this point the variable *goal_achieved* is *True*.

Algorithm 5 Reward Function

1: $r_t \leftarrow 0$
2: $s_e \leftarrow$ computeForwardKinematics() ▷ Determine end-effector state.
3: $distance \leftarrow d_(e_p, g_p)$ ▷ Calculate translational distance from goal.
4: $rotation \leftarrow d_(e_\theta, g_\theta)$ ▷ Calculate rotational distance from goal.
5: **if** *reward_goal* **then** ▷ Calculate r_g
6:     **if** $|position| < \epsilon_p \wedge |rotation| < \epsilon_o$ **then**
7:         $r_t \leftarrow MAX_REWARD$
8:         $goal_achieved \leftarrow True$
9:     **end if**
10: **end if**
11: **if** *reward_position* $\wedge \neg$*goal_achieved* **then** ▷ Calculate r_p
12:     $p_{max} \leftarrow (2 \times l) - \epsilon_p$
13:     $r_p \leftarrow POSITION_REWARD \times MAX_REWARD$
14:     **if** $|distance| \leq POSITIONAL_TOLERANCE$ **then**
15:         $r_t \leftarrow r_t + r_p$
16:     **else**
17:         $r_t \leftarrow r_t + r_p \times ((p_{max} - |distance|) \div p_{max})$
18:     **end if**
19: **end if**
20: **if** *reward_rotation* $\wedge \neg$*goal_achieved* **then** ▷ Calculate r_o
21:     $o_{max} \leftarrow (2 \times l) - \epsilon_p$
22:     $r_o \leftarrow ROTATION_REWARD \times MAX_REWARD$
23:     **if** $|rotation| \leq ROTATIONAL_TOLERANCE$ **then**
24:         $r_t \leftarrow r_t + r_o$
25:     **else**
26:         $r_t \leftarrow r_t + r_o \times ((o_{max} - |rotation|) \div o_{max})$
27:     **end if**
28: **end if**
29: **if** *reward_time* $\wedge \neg$*goal_achieved* **then** ▷ Calculate r_e
30:     $r_t \leftarrow r_t - TIME_PENALTY)$
31: **end if**
32: **if** *reward_collision* **then** ▷ Calculate r_c
33:     **if** getContactPoints() ≥ 0 **then**
34:         $r_t \leftarrow -MAX_REWARD$
35:         $collision \leftarrow True$
36:     **end if**
37: **end if**
        return r_t

5.2.9 Goal Generation

For training, goals are randomly generated (with respect to the seed) within the inner and outer bounds of the robot's working envelope which are defined as *KUKA_MIN_RANGE*

and *KUKA_MAX_RANGE* within the code and set to 0.4 m and 0.78 m respectively as shown in Table 5.2. These bounds conservatively estimate the robot's working envelope while also satisfying the task constraints. However,slight adjustment may also be possible to reflect the robot's true reach with the ability to satisfy the goal constraints and edge cases where the robot is at the outer limit of its joint rotation

Unless explicit values of x and y are specified, generation of the goal is carried out in *generateGoal()* with the algorithm 6.

Algorithm 6 Random Goal Generation in generateGoal()

```
1:  if x ∧ y then                                   ▷ Manually assign g_x and g_y if set
2:      g_x, g_y ← x, y
3:      Assign goal g(g_x, g_y)
4:  else
5:      for i=0; i ≤ 4; i++ do              ▷ Generate four random coordinates in range
6:          coordinate[i]←random_float(KUKA_MIN_RANGE,KUKA_MAX_RANGE)
7:      end for
8:      x_coords ← [coordinate[0], -coordinate[1]]
9:      y_coords ← [coordinate[2], -coordinate[3]]
10:     g_x ← pick_random(x_coords)  ▷ Pick -tive or +tive coordinate value for g_x, g_y
11:     g_y ← pick_random(y_coords)
12:     Assign goal g(g_x, g_y)
13: end if
14: ...
```

5.2.10 Termination Function

There are three conditions that trigger termination of an episode: achieving a goal; encountering a collision; or exceeding the maximum imposed number of timesteps T.

The termination function only explicitly checks one of these conditions which is that the number of timesteps has been exceeded, and checks the if the flags *collision* or *goal_achieved* have been set when they are checked elsewhere in the reward function in order to calculate a reward.

Algorithm 7 Termination Function

```
1: if collision ∨ goal_achieved ∨ t > T then
2:     terminated ← True
3: end if
4: return terminated
```

5.3 Chapter Summary

In this chapter, we described key features of the software implementation for our environment, training algorithm, and 3D models. In the following chapter, we detail

the experimental work undertaken to test the hypothesis with the goal of achieving our research aims.

Chapter 6

Experiments and Results

In this chapter, we describe the experimental work undertaken with the goals of (1) determining if the task environment can be solved using a deep RL approach, and (2) locating an optimal set of hyperparameters for training a RL agent to solve the task environment.
The chapter is organised as follows: First, we give an overview of the experimental work carried out in Section 6.1; then we describe the data we have collected for each experiment and the manner in which it was collected in 6.2; then, we describe the parametrisation of the experiments carried out in 6.3, before presenting and discussing our results in 6.4 and 6.5 respectively.

6.1 Experimental Work

A series of 14 experiments were conducted in line with a random search-based hyperparameter optimisation process using a simple random search method. The purpose of these experiments was to first identify and then further optimise a feasible combination of hyperparameters that would allow the RL agent to learn a policy for solving the RL task environment, while also testing parameters across several different random seeds.

We present the results of the 14 experiments that were run in the context of optimisation. (Many more were run and are accessible in Comet.ml but were part of testing or debugging and are not necessarily useful for comparison). Experiments were run using the Google Colab cloud computing service and carried out in series as only one instance is allowed to run at a time. The hardware used by Google Colab is not consistent so training time is not a useful metric in this instance. The total amount of computation time for training across all experiments included in our analysis is approximately 150 hours. The length of experiments varied from 3,000 episodes (~60,000 steps) to 25,000 episodes (~400,000 steps). Some experiments were intentionally of differing length to trial different training lengths, however some terminated early for no apparent reason. Several experiments are run for 25,000 episodes as this equates to just under 24 hours training time which is the maximum length of time you can run a single experiment on Google Colab before termination.

The hyperparameters adjusted include random seed, number of episodes, initial collection episodes, replay buffer size, and actor / critic / alpha learning rates.

6.2 Data Collection and Evaluation Metrics

We have used the Comet.ml platform [156] for the recording and management of all experimental data. All collected experimental data is available online in an interactive format in Appendix C. For each experiment, we store the following information for reproducibility purposes:

- **Hyperparameters** All hyper parameters will be recorded for each experiment. This will include the training, RL, and environment hyperparameters outlined in Table 6.2 as well as a description of the model used.

- **Code** Both the training script, and the environment code will be recorded in case of future changes.

- **Output Log** The output text of each experiment will be recorded. This will include training updates and any errors or warning encountered.

- **Training Time** Although training time is purely hardware dependent, anecdotally it can be used for general comparison.

- **System information** Characteristics of the computational hardware will be recorded.

We will also capture the following data as metrics at fixed intervals (default=500 episodes) to evaluate how well the agent is performing in the environment as it learns.

- **Average Reward** The average reward the agent earns in an episode calculated as the sum of rewards in an episode.

- **Loss** Reward prediction error.

- **Average Episode Length** Gives an indication of early termination frequency either due to collisions or achieving the goal if it is less than the maximum number of timesteps.

- **Video** Recordings of the EIH and ETH input across five episodes will be made at each evaluation interval as well as at the beginning and end of training.

6.3 List of Experiments

The hyperparameters adjusted include random seed, number of episodes, initial collection episodes, replay buffer size, and actor / critic / alpha learning rates. Table 6.1 lists the settings used for each experiment across these parameters, along with

the key name and colour for that experiment. Unless otherwise stated, the default values in Table 6.2, and Figures 6.3 and 6.3 are the hyperparameter values and neural network configurations used during each experiment. As previously stated, the full list of hyperparameter settings and results can be found in Appendix C.

Key	Seed	Episodes	Init. Coll. Eps.	Repl. Buff. Cap.	Critic L.R.	Actor L.R.	Alpha L.R.
A	1234567	10000	1000	10000	0.00003	0.00003	0.00003
B	1234567	25000	5000	10000	0.0003	0.0003	0.0003
C	1234567	20000	10000	50000	0.0003	0.0003	0.0003
D	1234567	3500	1000	50000	0.0003	0.0003	0.0003
E	1234567	13000	500	10000	0.0003	0.0003	0.0003
F	123456	3500	500	10000	0.0003	0.0003	0.0003
G	123456	25000	500	10000	0.0003	0.0003	0.0003
H	12345	6000	500	10000	0.0003	0.0003	0.0003
I	1234	25000	500	10000	0.0003	0.0003	0.0003
J	123	4500	500	10000	0.0003	0.0003	0.0003
K	123	25000	500	10000	0.0003	0.0003	0.0003
L	12	6000	500	10000	0.0003	0.0003	0.0003
M	12	25000	500	10000	0.0003	0.0003	0.0003
N	1234567	25000	500	10000	0.0003	0.0003	0.0003

Table 6.1: Non-default hyperparameter settings for each experiment.

Hyperparameter	Function	Default
Training Hyperparameters		
Number of Episodes	Number of episodes to train for	25000
Initial Collection Episodes	Episodes to seed replay buffer with before training	500
Collect Steps Per Iteration	Steps in each episode before termination	20
Number of Evaluation Episodes	Episodes over which to evaluate agent performance	20
Log Interval	Frequency of agent evaluation	500
Replay Buffer Capacity	Amount of steps to store in replay buffer	10000
Batch Size	Experience to use for training at each update	256
Seed	Randomisation seed for reproducibility	1234567
Reinforcement Learning Hyperparameters		
Critic Learning Rate	Critic network learning rate	0.0003
Actor Learning Rate	Actor network learning rate	0.0003
Alpha Learning Rate	Entropy regularization coefficient for SAC	0.0003
Critic Optimiser	Critic Optimisation Function	ADAM
Actor Optimiser	Actor Optimisation Function	ADAM
Alpha Optimiser	Alpha Optimisation Function	ADAM
Target Update Tau	Factor for soft update of target networks for SAC	0.005
Target Update Period	Period for soft update of target networks for SAC	1
Discount Rate / Gamma	Value of future rewards	0.99
Environment Hyperparameters		
EIH Input	Include EIH input in observation	True
ETH Input	Include ETH input in observation	True
Position Input	Include joint positions input in observation	True
Velocity Input	Include joint velocities in observation	True
EIH Resolution	Resolution of EIH input	112x112
ETH Resolution	Resolution of ETH input	112x112
EIH Channels	Colour channels in EIH input	3
ETH Channels	Colour channels in ETH input	3
Reward Goal	Include goal attainment in reward signal	True
Reward Collision	Include collision penalty in reward signal	True
Reward Time	Include time penalty in reward signal	True
Reward Position	Include positional reward in reward signal	True
Reward Rotation	include rotational reward in reward signal	True

Table 6.2: (Non-exhaustive) list of hyperparameters including training, reinforcement learning, and environment hyperparameters with their function and the default value used for experiments.

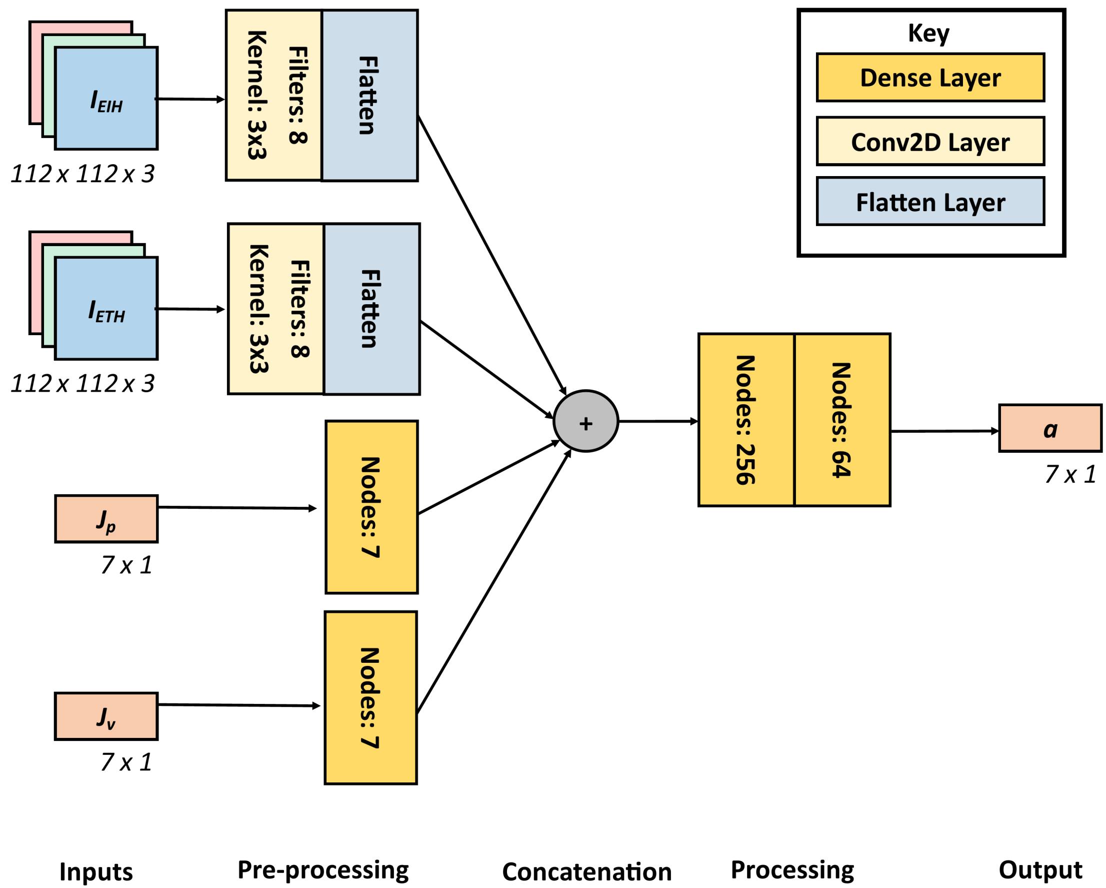

Figure 6.1: Specific actor network used for experiments. Key for layer types is shown. Unless stated, the default parameters are used for each layer type as they appear in [171] and [172] respectively.

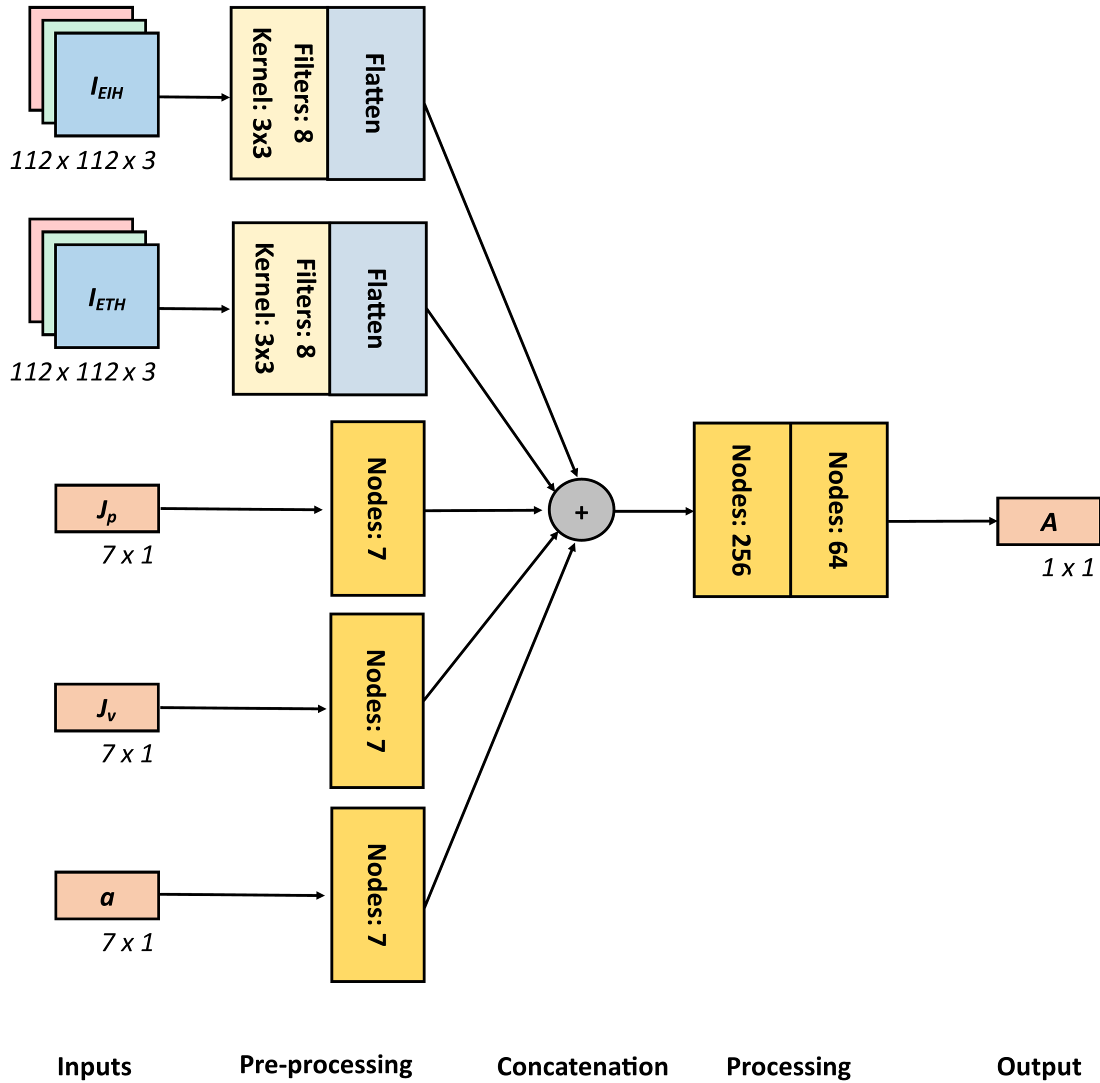

Figure 6.2: Specific critic network used for experiments. Key and other parameters as defined in Figure 6.3.

6.4 Results

We present the results of our experiments below. In 6.4.1 we present four graphs illustrating the metrics of all experiments undertaken A-N. Figure 6.4.1 shows the average return, Figure 6.4.1 shows the average episode length, and Figures 6.4.1 and 6.4.1 show the loss with and without experiment A, which had different learning rate settings to all other experiments.

In 6.4.2 we present three graphs illustrating the same metrics in the same way, but we only display experiments G, H, I, K, M and N. All six of these experiments

all possessed identical hyperparameters except for their seed and all ran for 25,000 episodes, except for experiment H which ran for 6,000 episodes, so they are readily comparable.

6.4.1 Summary of all experiments

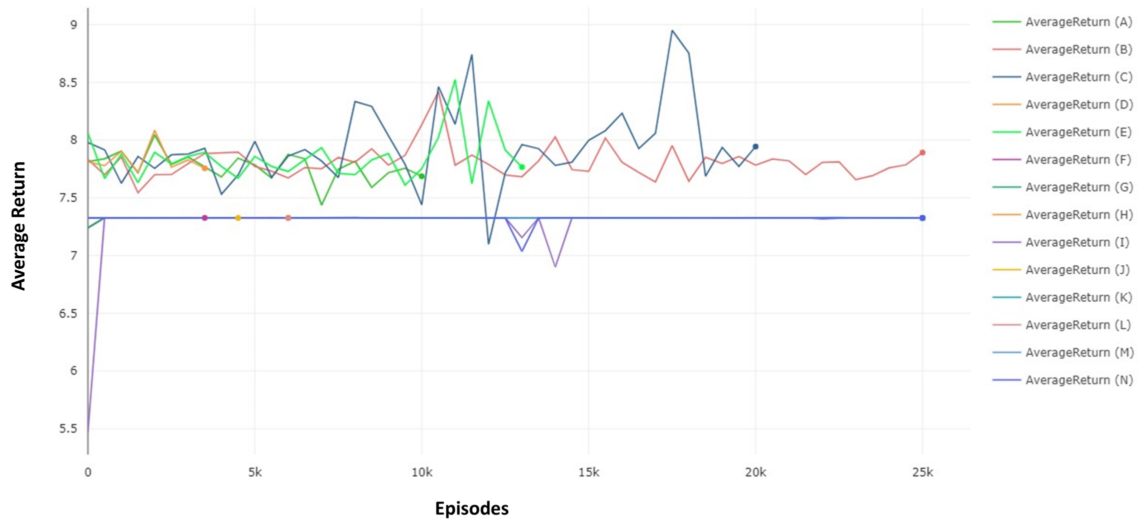

Figure 6.3: Average return across episodes for all experiments

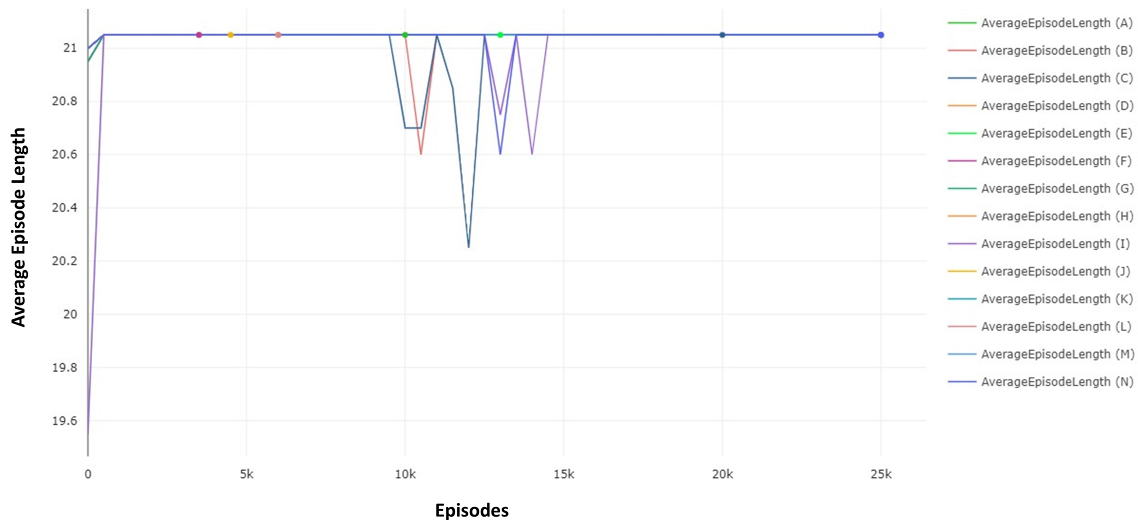

Figure 6.4: Average episode length across episodes for all experiments

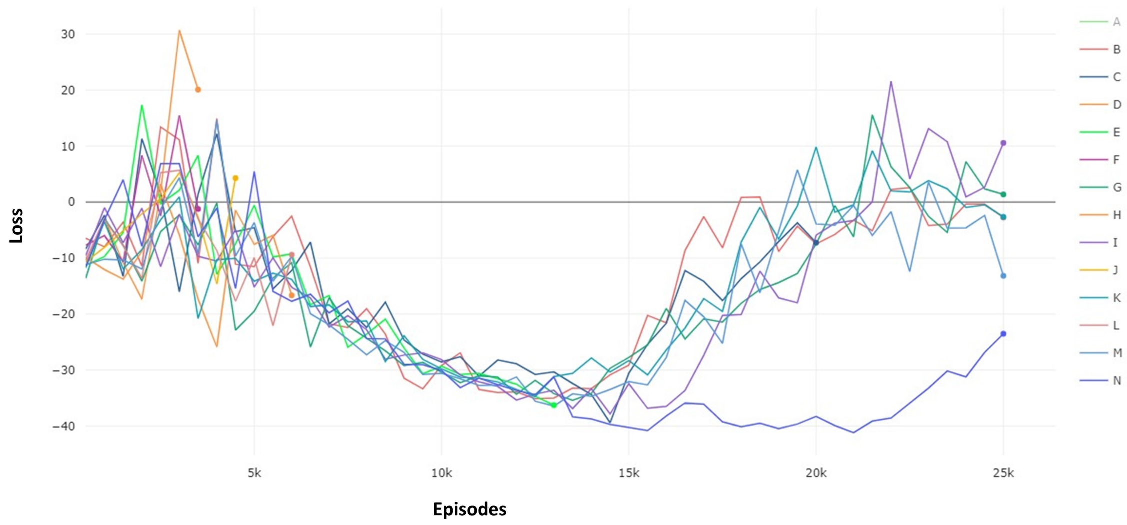

Figure 6.5: Loss across episodes for experiments where actor / critic / alpha learning rate = 0.0003

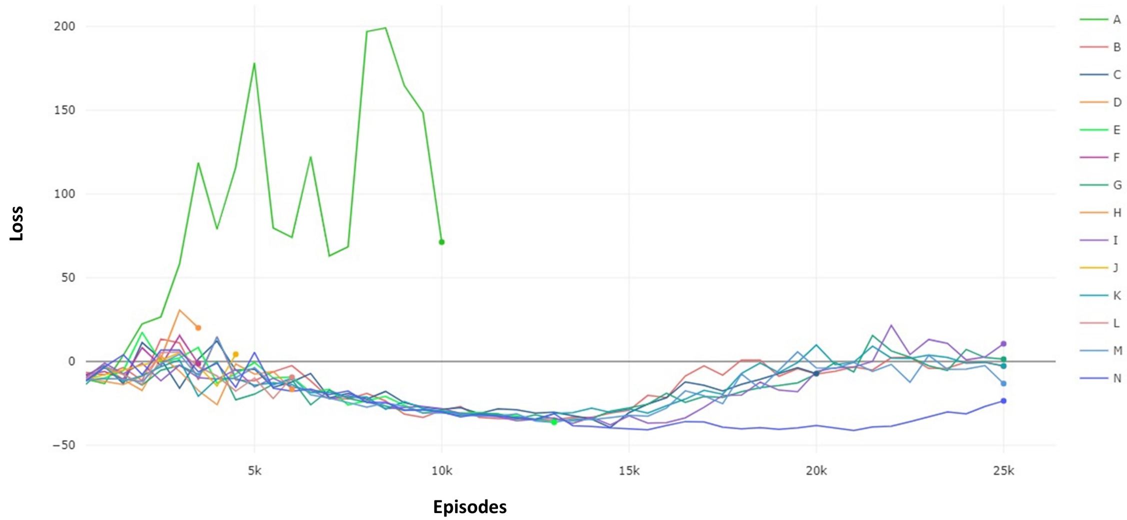

Figure 6.6: Loss across episodes for all experiments

6.4.2 Seeding

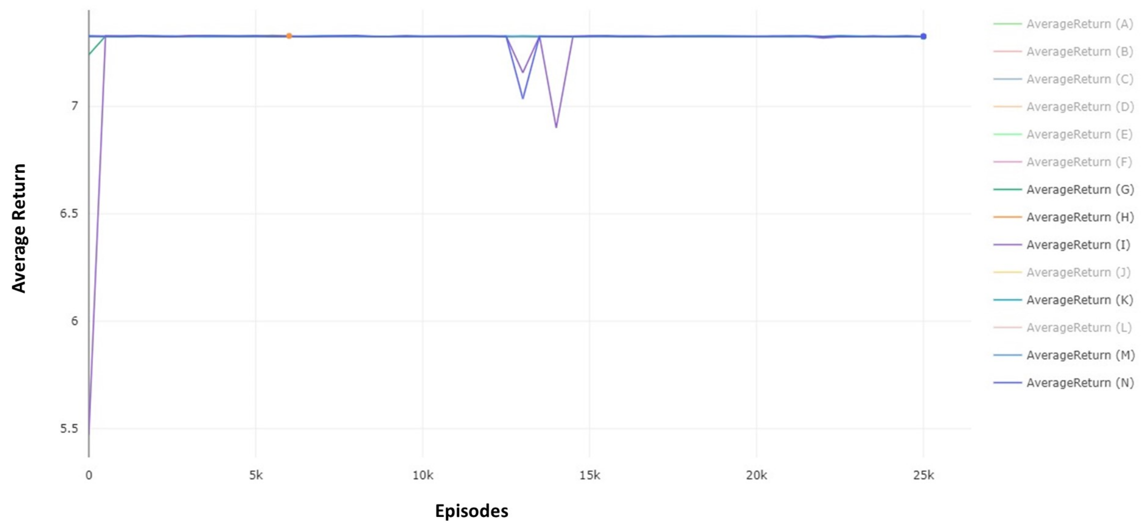

Figure 6.7: Average return for experiments where all parameters fixed except for seed

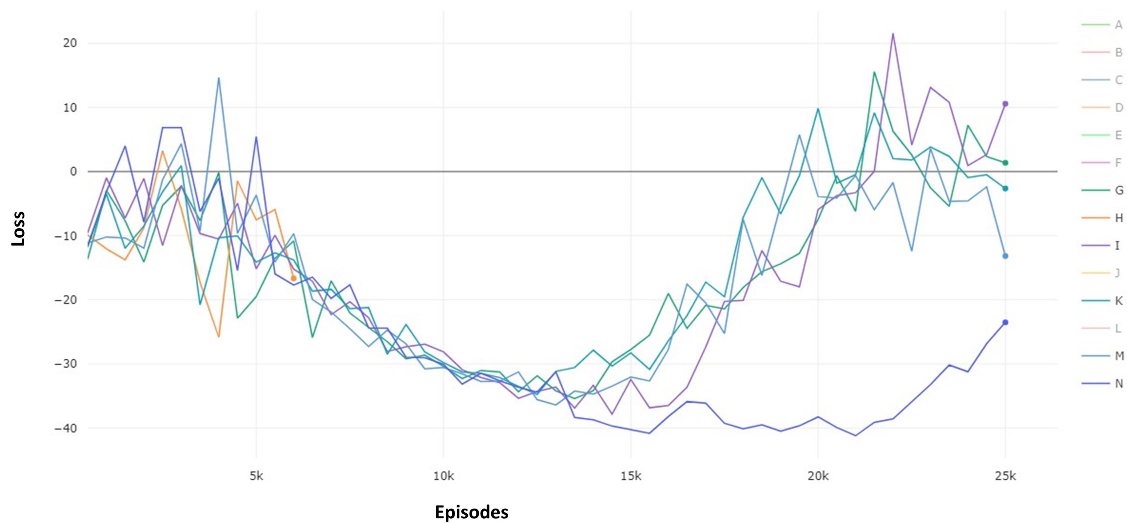

Figure 6.8: Loss for experiments where all parameters fixed except for seed

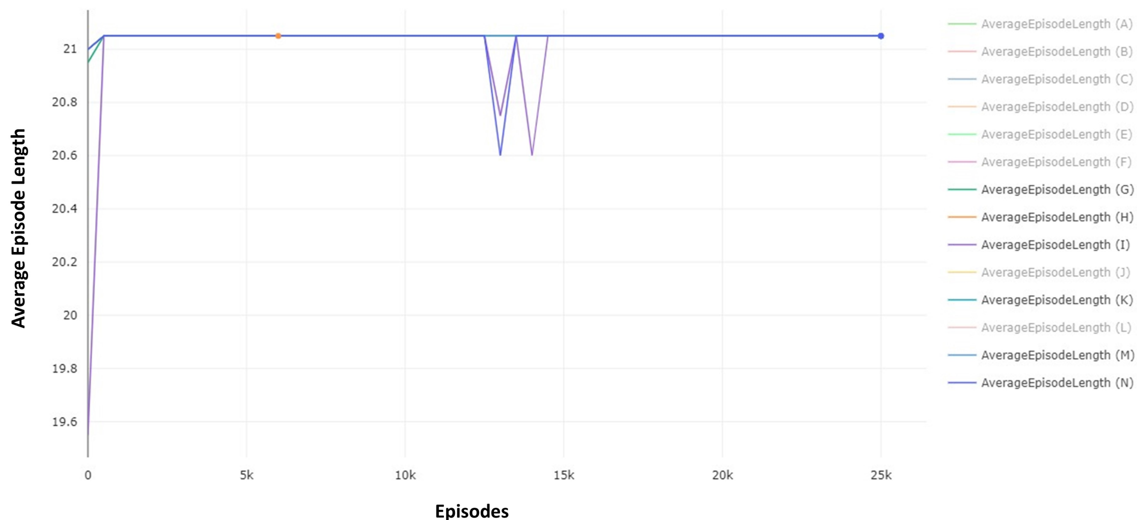

Figure 6.9: Average episode length for experiments where all parameters fixed except for seed

6.5 Discussion

The results collected were generally quite unremarkable and offer little observable sign of improvement, let alone of learning a policy capable of attaining the goal. Only a slim number of experiments were conducted and hyperparameter settings were not explored in enough depth. As such it is difficult to gain much insight into the efficacy of our deep reinforcement learning approach to solving this control task as it would be difficult to argue that the observed behaviour after training is any more advantageous that the initial random policy in terms of average return and goal attainment as across all trials of varying training length and parametrisation the goal was not achieved a single time.

The longest trials were over 25,000 episodes or approximately 500,000 steps with an initial 500 episodes or 10,000 steps of random exploration. As no observable improvement is seen in each agent's average return even with reward shaping, this suggests that the agent has not trained long enough to learn the mapping between the observation space and its actions.

The agent's poor behaviour in this environment, and the training time needed to solve this could also be influenced by other factors. We list several possibilities (i) the environment is too challenging, as the randomly positioned goal means the agent cannot find a correlation between inputs and reward unless difficult-to-process image inputs are included; (ii) efficient algorithm and hyperparameter settings have not been used; (iii) the exploration strategy is ineffective or inefficient; (iv) the structure of the neural network is not sufficient to learn the necessary complexity of the control function. It is clear that simply more experiments are needed to determine the correct set of hyperparameters and algorithmic features that are needed to solve the

control problem as we know that, in principle, a function for such behaviour should be learnable as demonstrated by other visual servoing works in the literature [71].

However, there was some interesting behaviour to be observed across this set of experiments. We can see in Figure 6.4.1 the stark differences in the loss between Experiment A, which had reduced learning rates, and all other experiments, as well as minor differences in loss. We can also see how all experiments with the same hyperparameter settings generally follow a similar loss trajectory across different seeds as can be observed in Figure 6.4.2.

The fairly static average return and episode lengths in Figures 6.4.1 and 6.4.1 also suggest that the agent is not exploring the environment well or that perhaps the reward function is not well defined. These are concerning results considering the reward function is heavily shaped for this set of experiments.

The most interesting and promising behaviour is that there does appear to be some rudimentary training going on to improve the agent's behaviour in relation to avoiding collisions. In most experiments, the agent would eventually discover collisions if training for long enough, usually between 10,000 and 20,000 episodes, and then go through a period of relatively frequent collisions before generally avoiding collisions again until training is terminated.[1] This suggests that some coarse training is occurring and the agent is learning to avoid collisions as these have an impact on average return as observed by the concurrent sharp dips in average return and average episode lengths in Figure and that since the agent does not discover collisions often until quite far in the training, that perhaps it is trying to follow some kind of broad reward gradient, in that moving towards the floor from its fixed initial position is likely to increase the reward, even if features for the image inputs are not yet learned. In fact, we can see in all experiments of length 25,000 the loss is being minimised and therefore that training is taking place. But because it never finds the goal and the optimal reward, and rarely collides or receives a reward outside of 0.3 to 0.5, the agent does not have a good idea of its potential return and this training is not particularly useful.

These observations and their conclusions should be taken with a pinch of salt and further experiments are needed across a wider range of hyperparameter settings and training procedures.

6.6 Further Experiments and Ablation Studies

Further experiments were proposed to study agent learning and performance characteristics within differing environmental configurations and parametrisations. Outlines of these experiments can be found in Appendix D and will be a topic of future work.

[1]Collisions were reported whenever they occurred during training, but as they did not always occur during the evaluation period they are not necessarily reflected in the figures above. A collision is observed in the evaluation period where there is a concurrent dip in both average return and average episode length, for instance in experiments I and N at Episode 13,000 in Figures 6.4.2 and 6.4.2 and confirmed by video recordings in Appendix C.

6.7 Chapter Summary

In this chapter, we have described the experimental work undertaken to test our hypothesis and discussed what our results mean in the context of our original research aims. In the following chapter, we critically evaluate and discuss the research work undertaken.

Chapter 7

Project Challenges and Discussion

This research project took place part-time over a period of three years. Over that length of time, the direction of research and features of the final implementation were subject to many revisions, refinement and development as more research was carried out, challenges or limitations became apparent, or technical issues were encountered. In this section, we describe some of the challenges encountered over the course of the project, the actions taken to overcome these and different approaches taken, and we summarise how elements of the system progressed.

7.1 Environment

The definition and scope of the Markov Decision Process changed several times throughout the project as it became clear what features would be needed to accurately reflect the problem environment and its difficulty, while ensuring that features were rich enough to possess all information that the environment may be solvable via reinforcement learning.

While designing the environment, it was clear that there were many ways in which the task could be customised. Indeed, while the simulated task environment is intended to contain similarities to specific industrial automation challenges and to be used as a basis for trialling control algorithms to solve those real-world problems, it also does not replicate any specific industrial task as we are only concerned with positioning. With this in mind, rather than make prohibitive design decisions that would limit the environment in one way or another, it made sense to provide many customisation options so that the environment is truly extensible and can be tailored to specific task requirements with default settings in place. This is most apparent in the observation customisation options where each input in Equation 4.5 can be toggled to be included in the observation or not, but there are other options too including image resolution, goal size and requirements, a randomised or fixed initial state (robot and goal positioning), reward function sparsity and more possible within the scope of Future Work.

However, even within the included features and scope of customisation, the implementation of many components was subject to much revision over the course of

the project. For instance, the reward function went through many iterations. While it was always essential that the reward function should reward the goal and punish collisions with these actions being fairly binary and easy to compute, the other sub-functions were optional and only implemented as means of assisting the agent achieve the desired behaviour While the time penalty function was based on works such as Kalashnikov et al. [71], ambiguity in the text left open the interpretation one of several ways: (i) apply penalty at final timestep only; (ii) apply penalty per timestep and increase until final step; or the chosen implementation (iii) apply penalty per timestep. Similarly, while the orientation and positioning reward functions could have been implemented several ways. While initial ideas pondered the use of a more complex positioning reward function that rewarded positioning along the X-Y and Z axes separately to reflect the nature of the goal and the 2D work surface or even punishing rather than rewarding distance from the goal, eventually a heuristic based on rewarding linear distance from the end effector in-terms of position and orientation was derived. However, it is questionable whether this was the best approach based on some of the preliminary results we have obtained and it is possible that this heuristic could shape the agent to travel along sub-optimal trajectories. This is also reflected in the API of the LBR iiwa allowing you to specify whether the robot should travel linearly in a point-to-point fashion or via a more efficient route [142]. A better solution could be a heuristic based on inverse-kinematics and the shortest distance in the joint space to achieve the goal position and orientation, or having a reward that changes non-linearly with distance to the goal (e.g. exponential) so that there is a steeper gradient to follow.

It is also possible that the best action parametrisation was not to use the robot's joint space. While this gives you direct control of each joint for fine motor control, it may have been more efficient to encode the action space via 3D coordinates that describe where the robot's end-effector should be to achieve the goal and to rely on a point-to-point inverse kinematics approach to perform an action as used in the proprietary KUKA API for the LBR iiwa. This would significantly reduce the action space, at the cost of diminished control and it is not clear which approach would be more worthwhile.

There were also decisions made on whether to use hard or soft termination. Hard termination would have penalised the agent once the maximum number of timesteps had elapsed in an episode, whereas soft termination simply resets the environment when reaching the final timestep without any special reward behaviour primarily to gather experience more efficiently. It was decided that soft-termination would be a more appropriate option as there is no inherent time-limit to the task described and efficiency is more desirable than necessary and efficiency should be handled by the time subfunction, r_t regardless, however it is possible that hard termination may produce more efficient learning.

Of course, some of these design decisions may be trivial and may only subtly affect the overall behaviour of the system and difficulty of the environment but nonetheless the options are worth mentioning and possibly exploring in future work, along with exploration of reward function sparsity.

7.2 Network

The general design of the network was adjusted several times. Originally the design envisaged stacking the two image matrices together and passing between 2 and 6 image layers through a single CNN for preprocessing depending on the respective image depths, and joining the position and velocity inputs together into a single vector of size 1x14 to be processed by a single FCN before the image pre-preprocessing CNN and the vector processing FCN were input to another FCN. However this was later replaced in favour of separate pre-processing networks for each input before connection to a single FCN. This was employed for two reasons. Firstly it allowed us to leverage pre-existing network classes in the TF-Agents Library, namely the ActorDistributionNetwork, and ValueNetwork classes, thus speeding up development time and improving the likelihood of code stability. Secondly, it allows us to change the input combination more readily without having to make significant changes to unrelated parts of the network. For example, using different image depths and resolutions in each camera, or none, some, or all of the possible input combinations. This design is likely to be computationally and behaviourally similar to the original design while allowing freedom for greater input combination. However, it does limit the customisation of the network class components.

All experiments also made use of separate actor and critic networks, however, it would also be possible to implement a custom shared architecture and this may be favourable in some cases where computational power is limited.

Although an original self-contained network was utilised for this work, it is possible that it would be more effective to instead train small transformer networks for pre-and-post processing of a pre-trained image classification network or to modify such a network with transfer learning to speed up the learning of complex image features.

7.3 Algorithmic Choices

ACKTR [81] was originally proposed as the algorithm most likely to be able to solve the environment as it had shown that it could be applied to simpler visual servoing problems [173] and is generally seen as one of the state-of-the-art reinforcement learning algorithms. However, with ACKTR not being supported in TF Agents at the time of writing and it being out of scope for this project to implement the algorithm from scratch, this meant it was necessary to adopt a different algorithm. Two algorithms that presented themselves that are state-of-the-art, meet the criteria for application described in Section 4.3 and also happened to be available in TF Agents were PPO [87] and SAC [84]. While either of these algorithms could have been used, SAC was chosen because it had fewer hyperparameters which should have made optimisation simpler, and there was tacit implication in various online resources such as [166] that SAC was well suited for robotics applications. Although, as no conclusive results were drawn about its efficacy it is impossible to say whether this algorithm is indeed the most suitable for solving this environment.

Aside from the main learning algorithm, auxiliary parts of the algorithm such as

the exploration strategy could also be improved. The current strategy replicates that used in the TF Agents SAC Minitaur example where the replay buffer is seeded with experience generated via a random policy for a certain number of timesteps before tweaking the randomly generated policy, similar to examples such as [22]. Whilst this approach is probably sufficient for an observation space consisting of joint angles only and with a fairly dense natural reward signal, it is probably too simple for image-based observations and more comprehensive exploration, and perhaps initial guiding should be used. For example, guiding the robot through a reasonable trajectory with a fixed probability until performance is at a sufficient level [71], selecting a random action with a fixed probability, and adding randomly generated noise to actions [91], [92]. Even adjusting the step length so that more frames are elapsed in the simulation between timesteps could be a way of providing more exploration and reducing jerky motion.

7.4 Software Implementation

The software implementation took place over the course of about a year with decisions about suitable software packages taking place in the months prior to that during the design phase.

Software resources for reinforcement learning are far less developed than for other machine learning methods such as supervised learning. Many general purpose machine learning resources are usually tailored towards supervised learning due to its wider applicability. As such, there were few relevant examples of the necessary software packages that could be followed either directly from the documentation, or other 3rd party resources, and there were no comparable end-to-end examples that included all of the custom features needed for our environment including a custom environment with multi-modal input and corresponding network configuration, an actor-critic algorithmic implementation, and specific robot required by this project, experiment logging, and other required features.

The relevant parts of existing examples needed to be pieced together to achieve the desired result. The process of achieving this was difficult and messy with a steep learning curve and development was laden with many unintuitive software errors to debug along the way. As someone new to these libraries and to practical deep reinforcement learning more generally, TF Agents seemed to require a lot of boiler plate code and setting of parameters that were not immediately obvious. As a result there was persistent questioning over whether the right approach and best practice was being followed and subsequent follow-up research during implementation. In fact, the setup for TF Agents appears to be more complex and customisable than other packages such as Stable Baselines. Yet, Stable Baselines did not possess the functionality required for all required features, notably multimodal input, until they were included in an update to the library late in the implementation stage, at which point switching to a new framework would have been unconducive to the project's success.

Another hurdle encountered was the platform dependency of some essential pack-

ages. The examples studied made use of the Reverb package [174] for the replay buffer required by the SAC algorithm. However, at the time of writing the package only works on Linux-based operating systems. A decision was made that rather than pursue the implementation of an alternative replay buffer system in the limited time available, that experiments would be exclusively run on the Linux-based Google Colab cloud computing platform [175] instead of the originally planned hybrid approach using both cloud computing resources and a local GPU-equipped, Windows-based machine. Although, it was later discovered that a hybrid process could have been set up through the Colab interface using the local runtime option, but this functionality was not known about until nearly the end of the project.

7.5 3D Render

The 3D environment created and the objects within it were fairly simple. No doubt, this environment could be altered to be made more realistic, challenging, and thus transferable to real-world application such as by adding realistic textures, fleshing out the 3D objects and background environment, and adding sensor noise to robotic joints and images, and there were ideas to refine the environment further if a successful algorithmic approach was derived. Some of these ideas are described in following chapter in Future Work. Even so, the intrinsic control problem without all these additional challenges is evidently still quite difficult to solve with a reinforcement learning approach.

7.6 Experiments

Due to the challenging implementation period, only limited time was left for gathering data in experiments and unfortunately far fewer experiments were run than desired or was necessary. Training in reinforcement learning is always challenging for any non-trivial problem and environments with large observation and action spaces such as this will always require a lot of training and thus computation time to learn an effective policy.

Alongside this, the delicacy of hyperparameter settings needing iterative tuning and the stochastic nature of reinforcement learning requiring running experiments with the same hyperparameters across multiple seeds [118] means that many long running experiments were needed to even scratch the surface of finding good hyperparameters. For instance, Kalashnikov et al. whose approach had similar aims to ours stated needing experience in the orders of millions of timesteps, even with a heavily guided exploration policy [71]. With the time and computational resources available, it was simply not possible to run the number of experiments needed to explore the hyperparameter space thoroughly for the length needed.

However, there were also physical limitations that hindered the collection of data and the running of experiments in the choice of appropriate hardware that was available. Although the Google Colab cloud computation system did have obvious advantages in terms of computational power over the local hardware available, it also

had limitations. Colab has a maximum allowed time of twenty-four hours for any single experiment before being terminated, and it also has a tendency to terminate processes prematurely in line with its scheduling policy. Fortunately, it is not necessary for reinforcement learning experiments to be the same length to be able to compare them, but it was incredibly frustrating when an experiment would left to run with the expectation of running for nearly twenty-four hours, and then finding it had terminated after as little as four which led to the repetition of some experiments.

These limitations could be overcome in some fairly obvious ways if more implementation time was available. Firstly, implementing a policy saving and recall function [176] so that we could train a policy for a number of steps, and then recall it and continue training in a separate instance in the case of premature termination. Secondly, running on a dedicated local machine would allow unrestricted training time, but potentially on inferior hardware. Nevertheless, exploring the use of a local runtime via the Colab interface (thus avoiding package compatibility errors) would be worthwhile.

If a solution for uninterrupted training was achieved through one of the above methods, then it would also have been preferable to implement a better strategy for hyperparameter optimisation. The random search produced very scattergun results in the limited number of runs. We had attempted to implement a Bayesian optimisation strategy using the Comet.ml Optimizer function [156] but unfortunately unresolved errors and the limited time remaining forced us to abandon this approach in favour of the random search and a series of single experiments.

The metrics used to gain insight into agent behaviour may also be insufficient in that the average return as it is calculated by default sums rewards across all timesteps in an episode. For instance, a typical average return for an episode as it is currently measures is around 8.0 as shown in Section 6.4 which is equivalent to a reward of 0.4 for each timestep for an episode length of 20 timesteps. However, if the agent was to achieve the goal after say 5 timesteps, its average return may be less than if it had not achieved the goal but completed all 20 timesteps. Similarly, the agent could have a collision on the final timestep and still have a higher average return than an episode in which the goal is achieved. This is clearly unsuitable and should be revised to better assess goal-based behaviour and while also taking into account shaped and non-terminal rewards.

It was also intended that additional evaluation metrics would have been collected to gain a finer understanding of an agents behaviour. However, these metrics are only useful for comparing agents capable of achieving the goal or performing interesting behaviour and unfortunately we did not achieve this kind of behaviour. Intended metrics included measuring the success rate of the agent in terms of the frequency of achieving its goal, the collision frequency, the accuracy and precision of the agent in achieving the goal, and the agent's sample efficiency e.g. how many episodes it takes to achieve $x\%$ success rate.

7.7 Physical Deployment

During the original scoping of this research, it was originally envisioned that control policies learned via simulation would be deployed on a physical KUKA LBR iiwa robot in a real-world replication of the simulated task environment. This deployment would have included steps such as setting up relevant hardware such as a laser projector and vision sensors with the LBR iiwa to replicate the task environment, integrating the physical hardware with the software produced for simulation, steps to overcome the simulation-to-reality gap and transfer learning of control policies to work in a physical environment with the minimal amount of training. This scoping was too ambitious given the time available and the challenges encountered during project development so the project scope was revised so as to only cover the implementation of a simulated environment and RL-based control policy to determine the efficacy of RL for solving the simulated task environment and optimise algorithmic hyperparameters according to this, with the view that this work could be used as a stepping stone to physical deployment in the future. Although, we were not able to take this research as far as physical deployment, we did review techniques and research relating to sim-2-real transfer and transfer learning in the literature review of this work in Sections 2.5.2 and 2.5.3 so that it is clear to researchers who may seek to continue this work or conduct similar work what the next steps are, and how simulated RL environments fit into the broader picture and necessary end-goal.

7.8 Chapter Summary

In this chapter, we have discussed the challenges experienced while undertaking this research. In the final chapter, we state our conclusions from this research and outline avenues for future work, as well as plans for dissemination.

Chapter 8

Conclusions and Future Work

In this chapter, we conclude the thesis by summarising what has been achieved in this research, and its next steps. First, we summarise the contributions and outcomes of this work and the main conclusions that can be drawn from this research in Section 8.1; then we outline several areas of future work that could be conducted based on the limitations of this work in 8.2; before stating plans for the dissemination of various aspects of this work in 8.3.

8.1 Conclusion

In this project we have proposed and implemented a novel reinforcement learning environment in simulation with a hybrid visual servoing and robotic joint sensor observation-based approach to a generic yet extensible and customisable fixtureless robotic positioning task involving a modern collaborative industrial robot, the KUKA LBR iiwa. This work was motivated by real-world industrial manufacturing challenges that have not been sufficiently addressed by the current state-of-the-art, with the aim of progressing research in this area towards developing control systems that can be deployed on real-world manufacturing problems. To this end, we also trialled the performance of state-of-the art reinforcement learning methods in solving this complex task environment and testing the efficacy of hybrid visual servoing methods, to little success.

The main contribution of this project is the development of an open-source novel reinforcement learning environment developed to a common reinforcement learning environment software format, along with an example training script demonstrating the application of contemporary deep reinforcement learning techniques to a complex environment. This contribution will surely aid research progress in both the specific kinds of manufacturing problems encapsulated by the environment, as well as reinforcement learning more generally. By making the code open-source, we have contributed well documented resources to a relatively slim ecosystem of support for the development of custom reinforcement learning algorithms and environments in the contemporary frameworks of TF Agents and PyBullet respectively, which will help to mitigate implementation challenges faced by other researchers and practitioners in

carrying out similar work.

While the initial experiments carried out at the time of writing are somewhat disappointing in that we were not able to obtain a set of hyperparameters to learn an effective control function in the given time to solve the task environment and we were not able to explore the effect of different environmental characteristics on performance, we remain hopeful that such hyperparameters can be found or that an alternative algorithmic approach may yield better results or that some elements of the environment can be fine-tuned to aid learning. We have outlined potential avenues of exploration in the Future Work section below.

Nonetheless, there are several lessons to take forward from this project which should also motivate future research. It is clear that the formulation and implementation of control problems into reinforcement learning environments and subsequent training of agents remains fairly challenging with a high level of specialist knowledge required to setup such algorithms, large amounts of computation needed for training, and limited guarantee of return as opposed to some of the more well resourced and proven machine learning methods such as supervised learning which has found common usage in industrial applications [39]. This will remain a barrier to wider adoption of reinforcement learning methods to industry until methods are more accessible, well resourced, and better understood. However, the potential application areas of reinforcement learning are huge and its capability to solve incredibly difficult problems [52], [71] highlight why it remains such an active area of research.

8.2 Future Work

Below, we outline potential avenues of future work based on the outcomes of this project.

Implementing a Capable Agent Although it was not possible to learn an effective control policy in any of the experiments run under this project in the time available, it is certainly possible that such a policy could be learned if similar work in the field is anything to go by. Therefore, the obvious direction of future work is in actually implementing a reinforcement learning agent capable of solving the task environment with a learned control policy.

There are multiple ways this could be achieved. For instance, simply increasing the amount of training time with the existing approach could yield results and it is possible that the agent simply has not gathered enough experience in the experiments run so far. Alternatively, the existing hyperparameters may just need to be tweaked to learn more efficiently and perhaps even other algorithmic hyperparameters need to be explored in such as the behaviour of the experience replay or neural network components which have not been explored within the scope of this project. We would also look at implementing a more efficient hyperparameter search such as Bayesian Optimisation.

Alternative algorithms and supplementary algorithmic techniques could also be applied to the problem domain. While vanilla SAC was used as the learning algorithm

for this work, it is possible that other deep RL algorithms such as ACKTR [81] or PPO [87] or the addition of supplementary techniques such as Hindsight Experience Replay [91] or curriculum learning [95] will have effective application to this particular problem class.

Studying Environment Behaviour The environment code was written with flexibility in mind to enable different specifications of the MDP including subsections of the full observation space, alternative initial state distributions, and a reward function with multiple components that can be enabled or not. It would be interesting to study the characteristics of different environmental specifications and their impact on agent learning and performance. Some proposed experiments for studying the behaviour of different environmental parameters and their effect on agent learning are described in Appendix D.

Environment Refinement and Extension As we highlighted in 7, it is possible that elements of the heavily shaped reward function could be improved with an alternative measure to the linear distance between the end-effector and goal such as an inverse kinematics-based distance across the joint space or having a reward that increases non-linearly (e.g. exponentially) with linear distance to the goal.

The environment was also intended to be customisable and extensible and there are many ways in which the generic task environment could be extended to better capture some real-world challenges. For instance, the environment could encapsulate all of the stages of a real-world manufacturing task by including a specific end-effector attachment and learning to perform a specific task such as e.g. pick-and-place or drilling. This would also include more objects within the task environment and would require the agent to learn more comprehensive object detection and spatial reasoning. You could even make the existing task more challenging, and realistic, by introducing non-planar surfaces to the working environment, stochastic or restricted workspaces, multiple simultaneous goals, non-static goals or even a non-static manipulator reflecting the form and function of the KMR iiwa for instance. Some of these extensions could require additional sensors and actuators to be included in the observation and action space such as end-effectors, a mobile platform, or 3D vision and might require learning more challenging behaviour such as complicated sequencing and positioning or alternative approaches such as force-torque control.

It should also be noted, that generally there are few options for customisation when importing environments from libraries such as PyBullet or Gym because it is important for benchmarking that an environments behaviour is repeatable and having customisation options may obfuscate this. The usual best practice is to have named variations of the same environment with customised parameters and alternative behaviour 'baked in'. It would be useful to come up with a set of named variations with diverse parametrisations that can be chosen by the user and be used for benchmarking. The environmental behavioural studies mentioned above and described in Appendix D could assist in narrowing down potential candidates with marked behavioural differences.

Software Compatibility Once it was decided that the TF Agents package would be used for the algorithmic side of the work, the environment was written following the TF Agents PyEnvironment format [177]. However, it would be useful to add compatibility for the environment in the OpenAI Gym format to enable it to be used with alternative RL packages such as Stable Baselines [178]. The difference between the two formats is trivial but adding compatibility with both would increase ease of access for practitioners.

Physical Validation The motivation of this work was in providing a solution towards known manufacturing system engineering challenges. Assuming that the task environment can be solved by a reinforcement learning approach, the logical endpoint is to transfer a learned control policy to a real-world robot and manufacturing environment. As discussed in Section 2.5.2, sim-to-real transfer presents its own challenges and these would need to be explored with regards to applying the learned control policy to the real world in an efficient way such as transfer learning.

8.3 Dissemination

An academic paper summarising key contributions from this work is intended to be published in due course to bring attention to the software developed and the various research challenges surrounding its development with the aim of assisting research progress in this area. Target publications include journals such as IEEE Transactions on Industrial Informatics and publication is expected by the end of 2022.

In addition to this, I will be producing a technical guide illustrating the process of implementing a reinforcement learning environment and applying relevant algorithms to it in the context of a TF Agents / PyBullet framework in order to improve the resources available to reinforcement learning practitioners. This will be published in a less formal, but highly visible forums such as Towards Data Science or Medium - websites commonly used by practitioners for practical tutorials.

The developed software has already been made open-source and publicly available (see Apprendix B). However, this work may also be of interest to the developers of libraries such as PyBullet to include as one of their included reinforcement learning libraries and offer an additional environment for their users to test. This would increase the visibility of the software and assist in progressing research in this area, developing the skills of practitioners, and adding to the code-base of an established and widely-used project. The developers of this library will be approached following publication.

8.4 Chapter Summary

In this chapter, we have presented the main contributions and conclusions from this work, as well as identifying limitations and proposing the next steps for developing this

work further towards achieving its aims of solving automation challenges in accurate fixtureless manufacturing environments.

Appendices

Appendix A

SAC Algorithm

Algorithm 1 Soft Actor-Critic

1: Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
2: Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
3: **repeat**
4: Observe state s and select action $a \sim \pi_\theta(\cdot|s)$
5: Execute a in the environment
6: Observe next state s', reward r, and done signal d to indicate whether s' is terminal
7: Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
8: If s' is terminal, reset environment state.
9: **if** it's time to update **then**
10: **for** j in range(however many updates) **do**
11: Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
12: Compute targets for the Q functions:

$$y(r, s', d) = r + \gamma(1 - d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi_\theta(\cdot|s')$$

13: Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d)\in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$

14: Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s\in B} \left(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta\left(\tilde{a}_\theta(s)|\, s\right)\right),$$

where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt θ via the reparametrization trick.
15: Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho\phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2$$

16: **end for**
17: **end if**
18: **until** convergence

111

Figure A.1: SAC algorithm pseudocode as it appears in [179]

Appendix B

Code Repository

The software relevant to this project including the *KukaHybridVisualServoingEnv.py* environment class, *RunTraining.py* training script, *EnvDebugger.py* simple debugging script, *floor.obj* and *target.obj* 3D object files, and *requirements.txt* listing the required software packages can be found at the following publicly available repository:
https://github.com/alexjameswilliams/KukaHybridVisualServoing

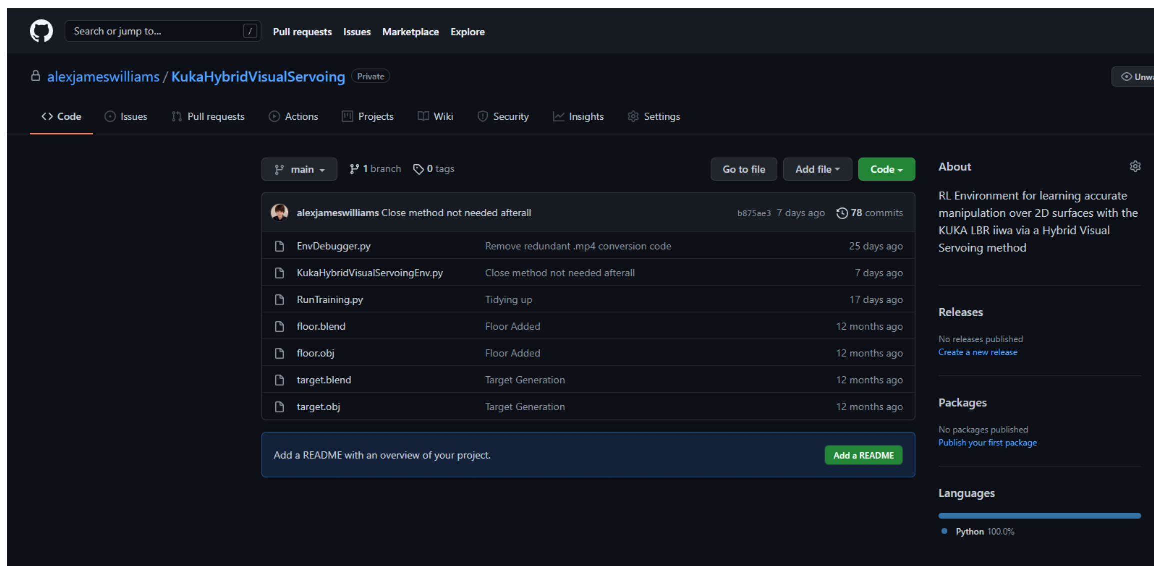

Figure B.1: Github repository.

Appendix C

Experiment Data

All experimental data, including hyperparameter settings, results, and videos can be found at the publicly available Comet.ml repository:
https://www.comet.ml/alexjameswilliams/kukahybridvs/

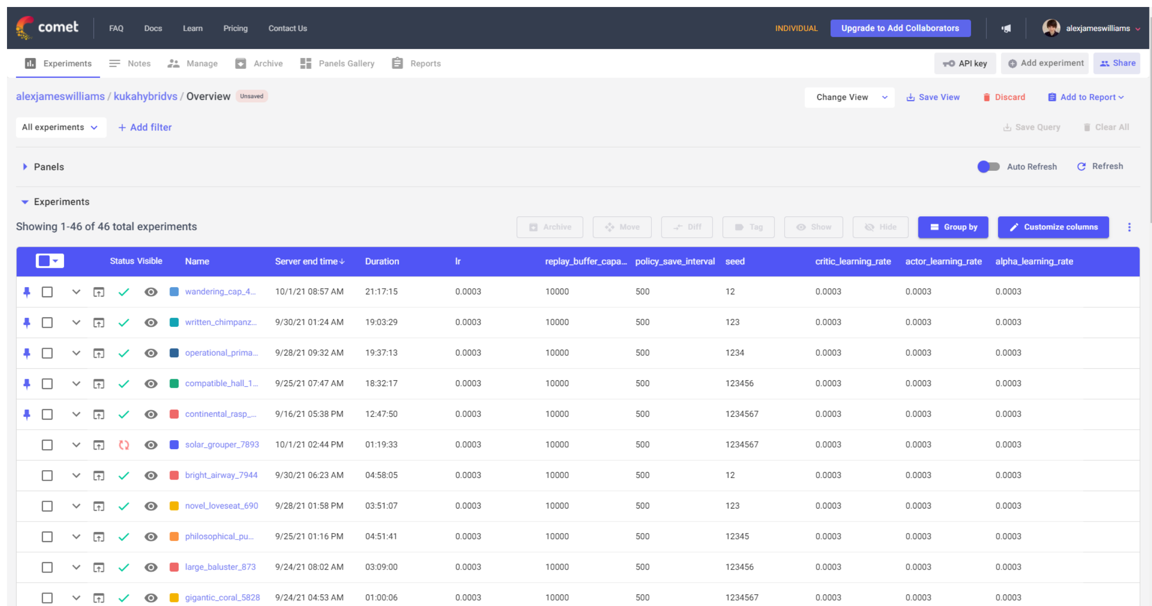

Figure C.1: Comet.ML repository with composite graphs and hyperlink access to each experiment.

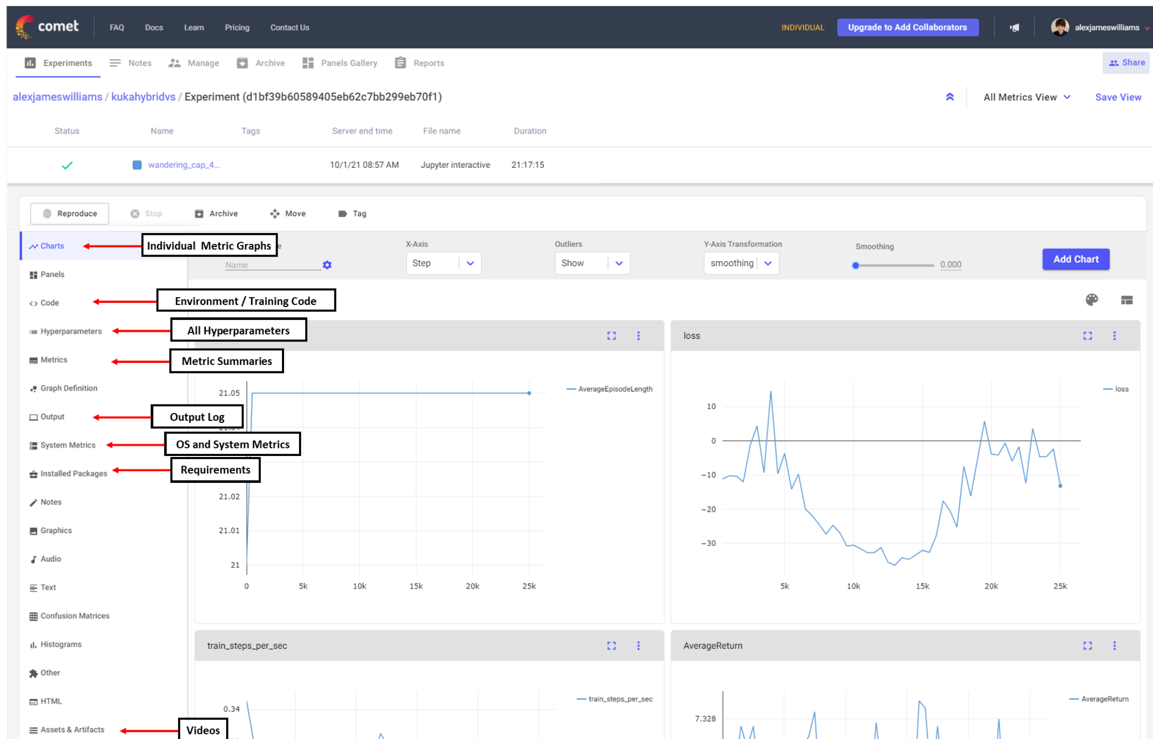

Figure C.2: Location of recorded data in comet interface for an individual experiment

Appendix D

Proposed Experiments and Ablation Studies for Future Work

Over the course of this project, several experiments were devised to study how the behaviour, performance, and ability to learn of the agent could be affected by different parametrisations of the environment. These experiments were out of scope for this thesis but we briefly describe their motivation and characteristics here as potential avenues of future work.

D.1 Observation Modularity

Some works have demonstrated that better performance was achievable by adding domain-specific state features such as robotic sensor information to image observations rather than simply images by themselves [71]. This experiment seeks to look at the differences in performance between different observation compositions for our specific environment by varying the number and combination of the four input sources described in 4.2.3: EIH and ETH cameras, and joint positions and velocities.

A table of possible input combinations is given in Table D.1. Note, at least one image input should be present in all permutations, otherwise it is impossible for the agent to locate the goal.

Observation, o
(I_{EIH})
(I_{ETH})
(I_{EIH}, J_v)
(I_{ETH}, J_v)
(I_{EIH}, J_p)
(I_{ETH}, J_p)
(I_{EIH}, J_p, J_v)
(I_{ETH}, J_p, J_v)
(I_{EIH}, I_{ETH})
(I_{EIH}, I_{ETH}, J_p)
(I_{EIH}, I_{ETH}, J_v)
$(I_{EIH}, I_{ETH}, J_p, J_v)$

Table D.1: Possible valid observation permutations.

D.2 Camera Resolution

Image processing via convolutional layers is one of the largest contributors to computation time because of the number of pixels that need to be processed (far greater than the number of variables in joint sensor data). However, at least one image is required as input for an agent to detect the goal, placing a bottleneck on computation. Therefore, we will explore how different image resolutions, image depth / colouring options, and image input combinations affect agent performance and training time.

D.3 Sim-2-Real

The simulated environment will not accurately represent the complexity of a real-world physical environment. Prior to running any experiments in the real world, experiments could be run exploring the effect of some well-used techniques for simulating real-world behaviour on agent performance. Such environment augmentations could include domain randomisation and the addition of noise to actions and observations.

D.4 Reward Sparsity

The reward function is perhaps the most significant affector of agent behaviour. Functions may either be sparse or shaped with denser and more detailed rewards with benefits and drawbacks to both approaches as we described in Section 2.4.3. In this study, we would train an agent using different levels of sparsity and quantitatively and qualitatively assess how the agent's performance, behaviour, and training efficiency is affected.

We can affect the sparsity of the overall reward function by toggling the various sub-functions that comprise it. Five sub-functions were proposed in 4.2.6 to form the reward function r: (r_g, r_c, r_e, r_o, and r_t). The feedback from each subfunction is different, but they can generally be split into three groups relating to their intended objective. These groupings could provide the basis for forming experimental subsets rather than exhausting every possible implementation of r. r_g and r_c are the most minimal requirements that need to be met in order to achieve a *goal* safely and must be included in any reward function. r_e relates to achieving the goal in the most *efficient* way. r_o and r_t provide *gradients* to direct the agent towards the goal. Based on these groupings, the following experiments could be carried out:

Goal, r_g, r_c	**Efficiency,** r_e	**Gradient,** r_o, r_t	**Reward Function,** r
Y	N	N	(r_g, r_c)
Y	N	*	(r_g, r_c, r_o)
Y	N	*	(r_g, r_c, r_t)
Y	N	Y	(r_g, r_c, r_o, r_t)
Y	Y	N	(r_g, r_c, r_e)
Y	Y	Y	$(r_g, r_c, r_e, r_o, r_t)$

Table D.2: Reward Function Combinations.

D.5 Initial State-Goal Distribution

The initial state-goal distribution, $p(s_0, g)$ could have a significant effect on the level of performance demonstrated by the agent and the amount of state-space exploration required to achieve a goal by applying limits on the number of reward-inducing state-action trajectories, which would reduce training time. Intuitively, fixing the initial state and goal in each episode so that the robot begins in the same configuration each time would allow the agent to learn a goal-efficient policy quicker. But, more useful behaviour would be the ability of the agent to achieve any new goal from any given start configuration as this removes the need for intermediary reconfiguration, which would be inefficient (and perhaps impossible) in real-world environments.

For each episode, we can set independently whether $p(s_1)$ and $p(g)$ are fixed or random. If they are fixed then an arbitrary state or goal is generated and used for all episodes thereafter until the simulation terminates. If they are random then an arbitrary state or goal is generated and used for that episode only. Permutations for initial state-goal distribution variations which could be explored are shown in Table D.3.

Initial State, s_0	**Goal, g**
Fixed	Fixed
Fixed	Random
Random	Fixed
Random	Random

Table D.3: Initial State-Goal Distribution Function Combinations.

D.6 Curriculum Learning

The focus of this work is achieving a high level of accuracy adoptable in specifically highly accurate manufacturing processes such as those within the aerospace industry which may have an accuracy around $\pm$ 0.25 mm [9]. The KUKA LBR iiwa also has a similar accuracy of $\pm$ 0.1 mm so ideally goals should be achieved with accuracies as close to these values as possible, but achieving this level of accuracy, particularly with sparse reward signals, is challenging. Curriculum learning is a supplemental technique to RL that assists the agent in learning a complex task by forcing it to learn tasks that become progressively more difficult [94]–[96].

We propose applying a version of curriculum learning to augment the reward signal in our environment whereby the relative difficulty of the goal increases relative to the success rate of the agent over a previous number of episodes reaches a minimum threshold. 'Difficulty' could arise in several ways, however the most straightforward way of increasing the difficulty of the task may be to reduce the accuracy tolerance ϵ, or increase the initial distance from the goal in the initial state-goal configuration.

Bibliography

[1] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld, and M. Hoffmann, "Industry 4.0," *Business & Information Systems Engineering*, vol. 6, no. 4, pp. 239–242, Aug. 2014, ISSN: 1867-0202. DOI: 10.1007/s12599-014-0334-4. [Online]. Available: https://doi.org/10.1007/s12599-014-0334-4.

[2] M. R. Pedersen, L. Nalpantidis, R. S. Andersen, *et al.*, "Robot skills for manufacturing: From concept to industrial deployment," *Robotics and Computer-Integrated Manufacturing*, vol. 37, pp. 282–291, Feb. 2016, ISSN: 0736-5845. DOI: 10.1016/j.rcim.2015.04.002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0736584515000575.

[3] P. J. Alhama Blanco, F. J. Abu-Dakka, and M. Abderrahim, "Practical Use of Robot Manipulators as Intelligent Manufacturing Systems," *Sensors (Basel, Switzerland)*, vol. 18, no. 9, p. 2877, Aug. 2018, ISSN: 1424-8220. DOI: 10.3390/s18092877. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6164753/ (visited on 09/20/2022).

[4] International Organization for Standardization, "ISO 10218-1:2011," Standard, 2011. [Online]. Available: http://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/05/13/51330.html.

[5] F. Vicentini, "Collaborative Robotics: A Survey," *Journal of Mechanical Design*, vol. 143, no. 4, Oct. 2020, ISSN: 1050-0472. DOI: 10.1115/1.4046238. [Online]. Available: https://doi.org/10.1115/1.4046238 (visited on 09/30/2022).

[6] International Organization for Standardization, "ISO/TS 15066:2016," Standard, 2016. [Online]. Available: http://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/06/29/62996.html.

[7] P. Corke, *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*, 1st. Springer Publishing Company, Incorporated, 2013, ISBN: 3-642-20143-1 978-3-642-20143-1.

[8] Y. Litvak, A. Biess, and A. Bar-Hillel, "Learning Pose Estimation for High-Precision Robotic Assembly Using Simulated Depth Images," *arXiv:1809.10699 [cs]*, p. 8, Sep. 2018, arXiv: 1809.10699. [Online]. Available: http://arxiv.org/abs/1809.10699 (visited on 07/10/2019).

[9] R. Devlieg, "Expanding the Use of Robotics in Airframe Assembly Via Accurate Robot Technology," *SAE Int. J. Aerosp.*, vol. 3, pp. 198–203, 2010. DOI: 10.4271/2010-01-1846. [Online]. Available: https://doi.org/10.4271/2010-01-1846.

[10] T. M. Anandan, *Aerospace Manufacturing on Board with Robots*, en, Feb. 2016. [Online]. Available: https://www.robotics.org/content-detail.cfm/Industrial-Robotics-Industry-Insights/Aerospace-Manufacturing-on-Board-with-Robots/content_id/5960 (visited on 08/21/2018).

[11] B. Brumson, *Robots in Aerospace Applications*, Jul. 2012. [Online]. Available: https://www.robotics.org/content-detail.cfm/Industrial-Robotics-Industry-Insights/Robots-in-Aerospace-Applications/content_id/3591 (visited on 08/21/2018).

[12] Airbus Group, *Registration now open for #ICRA16's Airbus Shopfloor Challenge — Robohub*, Oct. 2015. [Online]. Available: https://robohub.org/registration-now-open-for-icra16s-airbus-shopfloor-challenge/ (visited on 08/21/2018).

[13] Airbus Group, *The Airbus Shopfloor Challenge*, en, 2016. [Online]. Available: http://company.airbus.com/careers/Working-for-Airbus/Airbus-Shopfloor-Challenge-2016.html (visited on 08/21/2018).

[14] O. Zettinig, B. Fuerst, R. Kojcev, *et al.*, "Toward real-time 3D ultrasound registration-based visual servoing for interventional navigation," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, Stockholm: IEEE, May 2016, pp. 945–950, ISBN: 978-1-4673-8026-3. DOI: 10.1109/ICRA.2016.7487226. [Online]. Available: https://ieeexplore.ieee.org/document/7487226.

[15] F. Janabi-Sharifi, "Visual Servoing: Theory and Applications," in *Opto-Mechatronic Systems Handbook*, H. Cho, Ed., vol. 20025549, CRC Press, Sep. 2002, ISBN: 978-0-8493-1162-8 978-1-4200-4069-2. DOI: 10.1201/9781420040692.ch15. [Online]. Available: http://www.crcnetbase.com/doi/abs/10.1201/9781420040692.ch15.

[16] A. T. Vijayan, A. Sankar, and S. A. P., "A comparative study on the performance of neural networks in visual guidance and feedback applications," *Automatika*, vol. 58, no. 3, pp. 336–346, Jul. 2017, ISSN: 0005-1144, 1848-3380. DOI: 10.1080/00051144.2018.1437683. [Online]. Available: https://www.tandfonline.com/doi/full/10.1080/00051144.2018.1437683.

[17] J. Canny, "A Computational Approach to Edge Detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679–698, Nov. 1986, ISSN: 0162-8828. DOI: 10.1109/TPAMI.1986.4767851.

[18] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," in *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds., Curran Associates, Inc., 2015, pp. 91–99. [Online]. Available: http://papers.nips.cc/paper/5638-faster-r-cnn-towards-real-time-object-detection-with-region-proposal-networks.pdf.

[19] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2012, ISSN: 00010782. DOI: 10.1145/3065386. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3098997.3065386.

[20] S. Chiaverini, G. Oriolo, and I. D. Walker, "Kinematically Redundant Manipulators," in *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 245–268, ISBN: 978-3-540-30301-5. DOI: 10.1007/978-3-540-30301-5_12. [Online]. Available: https://doi.org/10.1007/978-3-540-30301-5_12.

[21] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. MIT Press, Nov. 2018, ISBN: 978-0-262-03924-6. [Online]. Available: https://mitpress.mit.edu/books/reinforcement-learning-second-edition.

[22] A. Marjaninejad, D. Urbina-Meléndez, B. A. Cohn, and F. J. Valero-Cuevas, "Autonomous functional movements in a tendon-driven limb via limited experience," *Nature Machine Intelligence*, vol. 1, no. 3, p. 144, Mar. 2019, ISSN: 2522-5839. DOI: 10.1038/s42256-019-0029-0. [Online]. Available: https://www.nature.com/articles/s42256-019-0029-0.

[23] J. Collins, D. Howard, and J. Leitner, "Quantifying the Reality Gap in Robotic Manipulation Tasks," en, *arXiv:1811.01484 [cs]*, Nov. 2018, arXiv: 1811.01484. [Online]. Available: http://arxiv.org/abs/1811.01484 (visited on 02/24/2020).

[24] L. Jin, S. Li, J. Yu, and J. He, "Robot manipulator control using neural networks: A survey," *Neurocomputing*, vol. 285, pp. 23–34, Apr. 2018, ISSN: 09252312. DOI: 10.1016/j.neucom.2018.01.002. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0925231218300158.

[25] F. Sherwani, M. M. Asad, and B. Ibrahim, "Collaborative Robots and Industrial Revolution 4.0 (IR 4.0)," in *2020 International Conference on Emerging Trends in Smart Technologies (ICETST)*, Mar. 2020, pp. 1–5. DOI: 10.1109/ICETST49965.2020.9080724.

[26] J. Guiochet, M. Machin, and H. Waeselynck, "Safety-critical advanced robots: A survey," en, *Robotics and Autonomous Systems*, vol. 94, pp. 43–52, Aug. 2017, ISSN: 0921-8890. DOI: 10.1016/j.robot.2017.04.004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0921889016300768 (visited on 09/30/2022).

[27] Universal Robotics, *Universal Robotics*, Company Website. [Online]. Available: https://www.universal-robots.com/ (visited on 04/25/2019).

[28] Rethink Robotics, *Rethink Robotics*, Company Website. [Online]. Available: https://www.rethinkrobotics.com/ (visited on 04/25/2019).

[29] KUKA Roboter GmbH, *KUKA LBR iiwa Brochure*, English, 2017. [Online]. Available: https://www.kuka.com/en-gb/products/robotics-systems/industrial-robots/lbr-iiwa.

[30] A. Hentout, M. Aouache, A. Maoudj, and I. Akli, "Human–robot interaction in industrial collaborative robotics: A literature review of the decade 2008–2017," *Advanced Robotics*, vol. 33, no. 15-16, pp. 764–799, Aug. 2019, Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/01691864.2019.1636714, ISSN: 0169-1864. DOI: 10.1080/01691864.2019.1636714. [Online]. Available: https://doi.org/10.1080/01691864.2019.1636714 (visited on 09/30/2022).

[31] KUKA, *KMR iiwa*, en-GB, 2018. [Online]. Available: https://www.kuka.com/en-gb/products/mobility/mobile-robots/kmr-iiwa (visited on 08/09/2018).

[32] A. Cherubini and D. Navarro-Alarcon, "Sensor-Based Control for Collaborative Robots: Fundamentals, Challenges, and Opportunities," *Frontiers in Neurorobotics*, vol. 14, 2021, ISSN: 1662-5218. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fnbot.2020.576846 (visited on 09/30/2022).

[33] A. Singh, V. Kalaichelvi, and R. Karthikeyan, "A survey on vision guided robotic systems with intelligent control strategies for autonomous tasks," *Cogent Engineering*, vol. 9, no. 1, A. H. Darwish, Ed., p. 2 050 020, Dec. 2022, Publisher: Cogent OA _eprint: https://doi.org/10.1080/23311916.2022.2050020, ISSN: null. DOI: 10.1080/23311916.2022.2050020. [Online]. Available: https://doi.org/10.1080/23311916.2022.2050020 (visited on 10/03/2022).

[34] X. Feng, Y. Jiang, X. Yang, M. Du, and X. Li, "Computer vision algorithms and hardware implementations: A survey," en, *Integration*, vol. 69, pp. 309–320, Nov. 2019, ISSN: 0167-9260. DOI: 10.1016/j.vlsi.2019.07.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167926019301762 (visited on 10/03/2022).

[35] L. Zhou, L. Zhang, and N. Konz, "Computer Vision Techniques in Manufacturing," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, pp. 1–13, 2022, Conference Name: IEEE Transactions on Systems, Man, and Cybernetics: Systems, ISSN: 2168-2232. DOI: 10.1109/TSMC.2022.3166397.

[36] F. Chaumette and S. Hutchinson, "Visual servo control. I. Basic approaches," *IEEE Robotics Automation Magazine*, vol. 13, no. 4, pp. 82–90, Dec. 2006, ISSN: 1070-9932. DOI: 10.1109/MRA.2006.250573.

[37] F. Chaumette and S. Hutchinson, "Visual servo control. II. Advanced approaches [Tutorial]," *IEEE Robotics Automation Magazine*, vol. 14, no. 1, pp. 109–118, Mar. 2007, ISSN: 1070-9932. DOI: 10.1109/MRA.2007.339609.

[38] X. Sun, X. Zhu, P. Wang, and H. Chen, "A Review of Robot Control with Visual Servoing," in *2018 IEEE 8th Annual International Conference on CYBER Technology in Automation, Control, and Intelligent Systems (CYBER)*, ISSN: 2379-7711, Jul. 2018, pp. 116–121. DOI: 10.1109/CYBER.2018.8688060.

[39] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015, ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature14539. [Online]. Available: http://www.nature.com/articles/nature14539.

[40] P. Norvig and S. Russell, *Artificial Intelligence: A Modern Approach*, 3rd. Prentice Hall, 2009, ISBN: 978-0-13-207148-2.

[41] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi, “A survey of deep neural network architectures and their applications,” *Neurocomputing*, vol. 234, pp. 11–26, Apr. 2017, ISSN: 0925-2312. DOI: 10.1016/j.neucom.2016.12.038. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925231216315533.

[42] F. v. Veen, *The Neural Network Zoo*, Sep. 2016. [Online]. Available: https://www.asimovinstitute.org/neural-network-zoo/ (visited on 07/03/2019).

[43] Y. Lecun, “Generalization and network design strategies,” English (US), *Connectionism in perspective*, 1989. [Online]. Available: https://nyuscholars.nyu.edu/en/publications/generalization-and-network-design-strategies.

[44] S. Levine, C. Finn, T. Darrell, and P. Abbeel, “End-to-end Training of Deep Visuomotor Policies,” *J. Mach. Learn. Res.*, vol. 17, no. 1, pp. 1334–1373, Jan. 2016, ISSN: 1532-4435. [Online]. Available: http://dl.acm.org/citation.cfm?id=2946645.2946684.

[45] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, “Large-Scale Video Classification with Convolutional Neural Networks,” in *2014 IEEE Conference on Computer Vision and Pattern Recognition*, Jun. 2014, pp. 1725–1732. DOI: 10.1109/CVPR.2014.223.

[46] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[47] T. M. Mitchell, *Machine Learning* (McGraw-Hill series in computer science). New York: McGraw-Hill, 1997, ISBN: 978-0-07-042807-2.

[48] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language Models are Unsupervised Multitask Learners,” p. 24, 2019.

[49] G. Hinton, L. Deng, D. Yu, *et al.*, “Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups,” *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, Nov. 2012, ISSN: 1053-5888. DOI: 10.1109/MSP.2012.2205597.

[50] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You Only Look Once: Unified, Real-Time Object Detection,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, USA: IEEE, Jun. 2016, pp. 779–788, ISBN: 978-1-4673-8851-1. DOI: 10.1109/CVPR.2016.91. [Online]. Available: http://ieeexplore.ieee.org/document/7780460/.

[51] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015, ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature14236. [Online]. Available: http://www.nature.com/articles/nature14236.

[52] D. Silver, J. Schrittwieser, K. Simonyan, *et al.*, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, p. 354, Oct. 2017, ISSN: 1476-4687. DOI: 10.1038/nature24270. [Online]. Available: https://www.nature.com/articles/nature24270.

[53] I. Goodfellow, J. Pouget-Abadie, M. Mirza, *et al.*, "Generative Adversarial Nets," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2014, pp. 2672–2680. [Online]. Available: http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf.

[54] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural Networks*, vol. 4, no. 2, pp. 251–257, Jan. 1991, ISSN: 0893-6080. DOI: 10.1016/0893-6080(91)90009-T. [Online]. Available: http://www.sciencedirect.com/science/article/pii/089360809190009T.

[55] Z. Lu, H. Pu, F. Wang, Z. Hu, and L. Wang, "The Expressive Power of Neural Networks: A View from the Width," in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, *et al.*, Eds., Curran Associates, Inc., 2017, pp. 6231–6239. [Online]. Available: http://papers.nips.cc/paper/7203-the-expressive-power-of-neural-networks-a-view-from-the-width.pdf.

[56] B. Hanin, "Universal Function Approximation by Deep Neural Nets with Bounded Width and ReLU Activations," *arXiv:1708.02691 [cs, math, stat]*, Aug. 2017, arXiv: 1708.02691. [Online]. Available: http://arxiv.org/abs/1708.02691 (visited on 06/27/2019).

[57] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain.," *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958, ISSN: 1939-1471, 0033-295X. DOI: 10.1037/h0042519. [Online]. Available: http://doi.apa.org/getdoi.cfm?doi=10.1037/h0042519.

[58] K.-S. Oh and K. Jung, "GPU implementation of neural networks," *Pattern Recognition*, vol. 37, no. 6, pp. 1311–1314, Jun. 2004, ISSN: 0031-3203. DOI: 10.1016/j.patcog.2004.01.013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0031320304000524.

[59] J. Sietsma and R. J. Dow, "Creating artificial neural networks that generalize," *Neural Networks*, vol. 4, no. 1, pp. 67–79, 1991, ISSN: 0893-6080(Print). DOI: 10.1016/0893-6080(91)90033-2.

[60] B. Poole, J. Sohl-Dickstein, and S. Ganguli, "Analyzing noise in autoencoders and deep networks," *arXiv:1406.1831 [cs]*, Jun. 2014, arXiv: 1406.1831. [Online]. Available: http://arxiv.org/abs/1406.1831 (visited on 08/09/2019).

[61] T. Kohonen, "Self-organized formation of topologically correct feature maps," *Biological Cybernetics*, vol. 43, no. 1, pp. 59–69, Jan. 1982, ISSN: 1432-0770. DOI: 10.1007/BF00337288. [Online]. Available: https://doi.org/10.1007/BF00337288.

[62] C. J. C. H. Watkins, "Learning from delayed rewards," PhD Thesis, King's College, Cambridge, 1989.

[63] C. Finn, Xin Yu Tan, Yan Duan, T. Darrell, S. Levine, and P. Abbeel, "Deep spatial autoencoders for visuomotor learning," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, Stockholm, Sweden: IEEE, May 2016, pp. 512–519, ISBN: 978-1-4673-8026-3. DOI: 10.1109/ICRA.2016.7487173. [Online]. Available: http://ieeexplore.ieee.org/document/7487173/.

[64] F. Ebert, C. Finn, S. Dasari, A. Xie, A. Lee, and S. Levine, "Visual Foresight: Model-Based Deep Reinforcement Learning for Vision-Based Robotic Control," *arXiv:1812.00568 [cs]*, Dec. 2018, arXiv: 1812.00568. [Online]. Available: http://arxiv.org/abs/1812.00568 (visited on 06/26/2019).

[65] D. Hafner, T. Lillicrap, I. Fischer, *et al.*, "Learning Latent Dynamics for Planning from Pixels," en, in *International Conference on Machine Learning*, May 2019, pp. 2555–2565. [Online]. Available: http://proceedings.mlr.press/v97/hafner19a.html (visited on 10/01/2019).

[66] I. Popov, N. Heess, T. Lillicrap, *et al.*, "Data-efficient Deep Reinforcement Learning for Dexterous Manipulation," *arXiv:1704.03073 [cs]*, Apr. 2017, arXiv: 1704.03073. [Online]. Available: http://arxiv.org/abs/1704.03073 (visited on 06/30/2019).

[67] T. P. Lillicrap, J. J. Hunt, A. Pritzel, *et al.*, "Continuous control with deep reinforcement learning," *arXiv:1509.02971 [cs, stat]*, Sep. 2015, arXiv: 1509.02971. [Online]. Available: http://arxiv.org/abs/1509.02971 (visited on 04/25/2019).

[68] I. Siradjuddin, L. Behera, T. M. McGinnity, and S. Coleman, "Image-Based Visual Servoing of a 7-DOF Robot Manipulator Using an Adaptive Distributed Fuzzy PD Controller," *IEEE/ASME Transactions on Mechatronics*, vol. 19, no. 2, pp. 512–523, Apr. 2014, ISSN: 1083-4435. DOI: 10.1109/TMECH.2013.2245337.

[69] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, Sep. 2013, ISSN: 0278-3649, 1741-3176. DOI: 10.1177/0278364913495721. [Online]. Available: http://journals.sagepub.com/doi/10.1177/0278364913495721.

[70] A. Wilson, *Learning Inverse Kinematics of Kuka iiwa using Deep Reinforcement Learning using DDPG, in pyBullet*, Nov. 2017. [Online]. Available: https://www.youtube.com/watch?v=httoZpXy4nw.

[71] D. Kalashnikov, A. Irpan, P. Pastor, *et al.*, "Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation," in *Conference on Robot Learning*, Oct. 2018, pp. 651–673. [Online]. Available: http://proceedings.mlr.press/v87/kalashnikov18a.html (visited on 08/29/2019).

[72] M. van Otterlo and M. Wiering, "Reinforcement Learning and Markov Decision Processes," en, in *Reinforcement Learning*, M. Wiering and M. van Otterlo, Eds., vol. 12, Series Title: Adaptation, Learning, and Optimization, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 3–42, ISBN: 978-3-642-27644-6 978-3-642-27645-3. DOI: 10.1007/978-3-642-27645-3_1. [Online]. Available: http://link.springer.com/10.1007/978-3-642-27645-3_1 (visited on 09/25/2022).

[73] L. Tai, J. Zhang, M. Liu, J. Boedecker, and W. Burgard, "A Survey of Deep Network Solutions for Learning Control in Robotics : From Reinforcement to Imitation," in *arXiv:1612.07139 [cs]*, arXiv: 1612.07139, Dec. 2016. [Online]. Available: http://arxiv.org/abs/1612.07139 (visited on 03/03/2019).

[74] H. Li, X. Liao, and L. Carin, "Multi-task Reinforcement Learning in Partially Observable Stochastic Environments," *The Journal of Machine Learning Research*, vol. 10, pp. 1131–1186, Jun. 2009, ISSN: 1532-4435.

[75] M. Hessel, J. Modayil, H. v. Hasselt, *et al.*, "Rainbow: Combining Improvements in Deep Reinforcement Learning," in *Thirty-Second AAAI Conference on Artificial Intelligence*, Apr. 2018. [Online]. Available: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17204 (visited on 08/29/2019).

[76] S. Gu, E. Holly, T. Lillicrap, and S. Levine, "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, May 2017, pp. 3389–3396. DOI: 10.1109/ICRA.2017.7989385.

[77] C. Finn, T. Yu, T. Zhang, P. Abbeel, and S. Levine, "One-Shot Visual Imitation Learning via Meta-Learning," in *Conference on Robot Learning*, Oct. 2017, pp. 357–368. [Online]. Available: http://proceedings.mlr.press/v78/finn17a.html.

[78] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-13, no. 5, pp. 834–846, Sep. 1983, ISSN: 0018-9472. DOI: 10.1109/TSMC.1983.6313077.

[79] E. Diederichs, "Reinforcement Learning - A Technical Introduction," en, *Journal of Autonomous Intelligence*, vol. 2, no. 2, p. 25, Jul. 2019, ISSN: 2630-5046. DOI: 10.32629/jai.v2i2.45. [Online]. Available: http://jai.front-sci.com/index.php/jai/article/view/45 (visited on 09/26/2022).

[80] V. Mnih, A. P. Badia, M. Mirza, *et al.*, "Asynchronous Methods for Deep Reinforcement Learning," in *Proceedings of The 33rd International Conference on Machine Learning*, M. F. Balcan and K. Q. Weinberger, Eds., ser. Proceedings of Machine Learning Research, vol. 48, New York, New York, USA: PMLR, Jun. 2016, pp. 1928–1937. [Online]. Available: http://proceedings.mlr.press/v48/mniha16.html.

[81] Y. Wu, E. Mansimov, R. B. Grosse, S. Liao, and J. Ba, "Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation," in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, *et al.*, Eds., Curran Associates, Inc., 2017, pp. 5279–5288. [Online]. Available: http://papers.nips.cc/paper/7112-scalable-trust-region-method-for-deep-reinforcement-learning-using-kronecker-factored-approximation.pdf.

[82] Z. Wang, V. Bapst, N. Heess, *et al.*, "Sample Efficient Actor-Critic with Experience Replay," *arXiv:1611.01224 [cs]*, Nov. 2016, arXiv: 1611.01224. [Online]. Available: http://arxiv.org/abs/1611.01224 (visited on 03/12/2019).

[83] S. Fujimoto, H. Hoof, and D. Meger, "Addressing Function Approximation Error in Actor-Critic Methods," en, in *International Conference on Machine Learning*, Jul. 2018, pp. 1587–1596. [Online]. Available: http://proceedings.mlr.press/v80/fujimoto18a.html (visited on 01/15/2020).

[84] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor," *arXiv:1801.01290 [cs, stat]*, Jan. 2018, arXiv: 1801.01290. [Online]. Available: http://arxiv.org/abs/1801.01290 (visited on 03/12/2019).

[85] T. Haarnoja, A. Zhou, K. Hartikainen, *et al.*, "Soft Actor-Critic Algorithms and Applications," *arXiv:1812.05905 [cs, stat]*, Jan. 2019, arXiv: 1812.05905. [Online]. Available: http://arxiv.org/abs/1812.05905 (visited on 11/01/2020).

[86] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust Region Policy Optimization," in *Proceedings of the 32nd International Conference on Machine Learning*, F. Bach and D. Blei, Eds., ser. Proceedings of Machine Learning Research, vol. 37, Lille, France: PMLR, Jul. 2015, pp. 1889–1897. [Online]. Available: http://proceedings.mlr.press/v37/schulman15.html.

[87] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," *arXiv:1707.06347 [cs]*, Jul. 2017, arXiv: 1707.06347. [Online]. Available: http://arxiv.org/abs/1707.06347 (visited on 06/29/2019).

[88] S. Levine, *CS 294-112: Actor-Critic Algorithms*, Lecture Slides, University of California, Berkeley, Sep. 2017. [Online]. Available: http://rail.eecs.berkeley.edu/deeprlcourse-fa17/f17docs/lecture_5_actor_critic_pdf (visited on 05/31/2020).

[89] *Reinforcement learning - How do shared parameters in actor-critic models work?* Library Catalog: stackoverflow.com. [Online]. Available: https://stackoverflow.com/questions/56312962/how-do-shared-parameters-in-actor-critic-models-work (visited on 05/31/2020).

[90] *Tensorflow - Confusion about neural network architecture for the actor critic reinforcement learning algorithm*, Library Catalog: datascience.stackexchange.com. [Online]. Available: https://datascience.stackexchange.com/questions/35814/confusion-about-neural-network-architecture-for-the-actor-critic-reinforcement-l (visited on 05/29/2020).

[91] M. Andrychowicz, F. Wolski, A. Ray, *et al.*, “Hindsight Experience Replay,” en, *arXiv:1707.01495 [cs]*, Feb. 2018, arXiv: 1707.01495. [Online]. Available: http://arxiv.org/abs/1707.01495 (visited on 01/28/2020).

[92] J. Ibarz, J. Tan, C. Finn, M. Kalakrishnan, P. Pastor, and S. Levine, “How to train your robot with deep reinforcement learning: Lessons we have learned,” en, *The International Journal of Robotics Research*, vol. 40, no. 4-5, pp. 698–721, Apr. 2021, Publisher: SAGE Publications Ltd STM, ISSN: 0278-3649. DOI: 10.1177/0278364920987859. [Online]. Available: https://doi.org/10.1177/0278364920987859 (visited on 06/07/2021).

[93] E. Wiewiora, “Potential-Based Shaping and Q-Value Initialization are Equivalent,” *Journal of Artificial Intelligence Research*, vol. 19, pp. 205–208, Sep. 2003, ISSN: 1076-9757. DOI: 10.1613/jair.1190. [Online]. Available: https://jair.org/index.php/jair/article/view/10338.

[94] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, “Curriculum learning,” in *Proceedings of the 26th Annual International Conference on Machine Learning - ICML ’09*, Montreal, Quebec, Canada: ACM Press, 2009, pp. 1–8, ISBN: 978-1-60558-516-1. DOI: 10.1145/1553374.1553380. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1553374.1553380.

[95] C. Florensa, D. Held, M. Wulfmeier, M. Zhang, and P. Abbeel, “Reverse Curriculum Generation for Reinforcement Learning,” *arXiv:1707.05300 [cs]*, Jul. 2017, arXiv: 1707.05300. [Online]. Available: http://arxiv.org/abs/1707.05300 (visited on 07/16/2019).

[96] C. Florensa, D. Held, X. Geng, and P. Abbeel, “Automatic Goal Generation for Reinforcement Learning Agents,” en, *arXiv:1705.06366 [cs]*, Jul. 2018, arXiv: 1705.06366. [Online]. Available: http://arxiv.org/abs/1705.06366 (visited on 01/28/2020).

[97] A. Tavakoli, V. Levdik, R. Islam, and P. Kormushev, “Prioritizing Starting States for Reinforcement Learning,” en, *arXiv:1811.11298 [cs, stat]*, Jan. 2019, arXiv: 1811.11298. [Online]. Available: http://arxiv.org/abs/1811.11298 (visited on 03/10/2020).

[98] M. Riedmiller, R. Hafner, T. Lampe, *et al.*, “Learning by Playing Solving Sparse Reward Tasks from Scratch,” in *International Conference on Machine Learning*, Jul. 2018, pp. 4344–4353. [Online]. Available: http://proceedings.mlr.press/v80/riedmiller18a.html.

[99] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, “Curiosity-driven Exploration by Self-supervised Prediction,” in *International Conference on Machine Learning*, Jul. 2017, pp. 2778–2787. [Online]. Available: http://proceedings.mlr.press/v70/pathak17a.html (visited on 08/29/2019).

[100] Y. Burda, H. Edwards, D. Pathak, A. Storkey, T. Darrell, and A. A. Efros, “Large-Scale Study of Curiosity-Driven Learning,” *arXiv:1808.04355 [cs, stat]*, Aug. 2018, arXiv: 1808.04355. [Online]. Available: http://arxiv.org/abs/1808.04355 (visited on 03/11/2019).

[101] B. Piot, M. Geist, and O. Pietquin, "Bridging the Gap Between Imitation Learning and Inverse Reinforcement Learning," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 8, pp. 1814–1826, Aug. 2017, ISSN: 2162-237X, 2162-2388. DOI: 10.1109/TNNLS.2016.2543000. [Online]. Available: http://ieeexplore.ieee.org/document/7464854/.

[102] G. Brockman, V. Cheung, L. Pettersson, *et al.*, "OpenAI Gym," *arXiv:1606.01540 [cs]*, Jun. 2016, arXiv: 1606.01540. [Online]. Available: http://arxiv.org/abs/1606.01540 (visited on 07/15/2019).

[103] *Bullet Real-Time Physics Simulation — Home of Bullet and PyBullet: Physics simulation for games, visual effects, robotics and reinforcement learning.* [Online]. Available: https://pybullet.org/wordpress/ (visited on 07/15/2019).

[104] X. Gao, R. Gong, T. Shu, X. Xie, S. Wang, and S.-C. Zhu, "VRKitchen: An Interactive 3D Virtual Environment for Task-oriented Learning," *arXiv:1903.05757 [cs]*, Mar. 2019, arXiv: 1903.05757. [Online]. Available: http://arxiv.org/abs/1903.05757 (visited on 10/07/2020).

[105] J. K. Terry, B. Black, M. Jayakumar, *et al.*, "PettingZoo: Gym for Multi-Agent Reinforcement Learning," *arXiv preprint arXiv:2009.14471*, 2020.

[106] A. D'Amour, K. Heller, D. Moldovan, *et al.*, "Underspecification Presents Challenges for Credibility in Modern Machine Learning," *arXiv:2011.03395 [cs, stat]*, Nov. 2020, arXiv: 2011.03395. [Online]. Available: http://arxiv.org/abs/2011.03395 (visited on 11/20/2020).

[107] S. James, P. Wohlhart, M. Kalakrishnan, *et al.*, "Sim-To-Real via Sim-To-Sim: Data-Efficient Robotic Grasping via Randomized-To-Canonical Adaptation Networks," en, in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Long Beach, CA, USA: IEEE, Jun. 2019, pp. 12 619–12 629, ISBN: 978-1-72813-293-8. DOI: 10.1109/CVPR.2019.01291. [Online]. Available: https://ieeexplore.ieee.org/document/8954361/ (visited on 01/14/2020).

[108] Q. Bateux, E. Marchand, J. Leitner, F. Chaumette, and P. Corke, "Training Deep Neural Networks for Visual Servoing," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, May 2018, pp. 1–8. DOI: 10.1109/ICRA.2018.8461068.

[109] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, ISSN: 2153-0866, Sep. 2017, pp. 23–30. DOI: 10.1109/IROS.2017.8202133.

[110] J. Grasshoff, L. Hansen, I. Kuhlemann, and K. Ehlers, "7DoF Hand and Arm Tracking for Teleoperation of Anthropomorphic Robots," in *Proceedings of ISR 2016: 47st International Symposium on Robotics*, Jun. 2016, pp. 1–8.

[111] C. Li, C. Yang, Z. Ju, and A. S. K. Annamalai, "An enhanced teaching interface for a robot using DMP and GMR," *International Journal of Intelligent Robotics and Applications*, vol. 2, no. 1, pp. 110–121, Mar. 2018, ISSN: 2366-598X. DOI: 10.1007/s41315-018-0046-x. [Online]. Available: https://doi.org/10.1007/s41315-018-0046-x.

[112] C. Li, C. Yang, and C. Giannetti, "Segmentation and generalisation for writing skills transfer from humans to robots," *Cognitive Computation and Systems*, vol. 1, no. 1, pp. 20–25, 2019, ISSN: 2517-7567. DOI: 10.1049/ccs.2018.0005.

[113] B. D. Argall, S. Chernova, M. Veloso, and B. Browning, "A survey of robot learning from demonstration," *Robotics and Autonomous Systems*, vol. 57, no. 5, pp. 469–483, May 2009, ISSN: 09218890. DOI: 10.1016/j.robot.2008.10.024. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0921889008001772.

[114] J. X. Wang, Z. Kurth-Nelson, D. Tirumala, *et al.*, "Learning to reinforcement learn," *arXiv:1611.05763 [cs, stat]*, Nov. 2016, arXiv: 1611.05763. [Online]. Available: http://arxiv.org/abs/1611.05763 (visited on 02/21/2019).

[115] L. Metz, N. Maheswaranathan, B. Cheung, and J. Sohl-Dickstein, "Meta-Learning Update Rules for Unsupervised Representation Learning," *arXiv:1804.00222 [cs, stat]*, Mar. 2018, arXiv: 1804.00222. [Online]. Available: http://arxiv.org/abs/1804.00222 (visited on 02/27/2019).

[116] C. Finn, P. Abbeel, and S. Levine, "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks," in *International Conference on Machine Learning*, Jul. 2017, pp. 1126–1135. [Online]. Available: http://proceedings.mlr.press/v70/finn17a.html.

[117] S. M. Jordan, Y. Chandak, D. Cohen, M. Zhang, and P. S. Thomas, "Evaluating the Performance of Reinforcement Learning Algorithms," *arXiv:2006.16958 [cs, stat]*, Aug. 2020, arXiv: 2006.16958. [Online]. Available: http://arxiv.org/abs/2006.16958 (visited on 11/23/2020).

[118] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep Reinforcement Learning that Matters," en, *arXiv:1709.06560 [cs, stat]*, Jan. 2019, arXiv: 1709.06560. [Online]. Available: http://arxiv.org/abs/1709.06560 (visited on 01/06/2020).

[119] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation," en, in *2014 IEEE Conference on Computer Vision and Pattern Recognition*, Columbus, OH, USA: IEEE, Jun. 2014, pp. 580–587, ISBN: 978-1-4799-5118-5. DOI: 10.1109/CVPR.2014.81. [Online]. Available: http://ieeexplore.ieee.org/document/6909475/ (visited on 01/15/2020).

[120] M. Andrychowicz, A. Raichuk, P. Stańczyk, *et al.*, "What Matters In On-Policy Reinforcement Learning? A Large-Scale Empirical Study," *arXiv:2006.05990 [cs, stat]*, Jun. 2020, arXiv: 2006.05990. [Online]. Available: http://arxiv.org/abs/2006.05990 (visited on 02/07/2021).

[121] J. Bergstra and Y. Bengio, "Random Search for Hyper-Parameter Optimization," *Journal of Machine Learning Research*, vol. 13, no. 10, pp. 281–305, 2012, ISSN: 1533-7928. [Online]. Available: http://jmlr.org/papers/v13/bergstra12a.html (visited on 10/31/2020).

[122] M. Jaderberg, V. Dalibard, S. Osindero, *et al.*, "Population Based Training of Neural Networks," *arXiv:1711.09846 [cs]*, Nov. 2017, arXiv: 1711.09846. [Online]. Available: http://arxiv.org/abs/1711.09846 (visited on 11/08/2020).

[123] H. S. Jomaa, J. Grabocka, and L. Schmidt-Thieme, "Hyp-RL : Hyperparameter Optimization by Reinforcement Learning," *arXiv:1906.11527 [cs, stat]*, Jun. 2019, arXiv: 1906.11527. [Online]. Available: http://arxiv.org/abs/1906.11527 (visited on 11/08/2020).

[124] S. Paul, V. Kurin, and S. Whiteson, "Fast Efficient Hyperparameter Tuning for Policy Gradients," *arXiv:1902.06583 [cs, stat]*, Sep. 2019, arXiv: 1902.06583. [Online]. Available: http://arxiv.org/abs/1902.06583 (visited on 11/08/2020).

[125] I. Dewancker, M. McCourt, and S. Clark, "Bayesian Optimization for Machine Learning : A Practical Guidebook," *arXiv:1612.04858 [cs]*, Dec. 2016, arXiv: 1612.04858. [Online]. Available: http://arxiv.org/abs/1612.04858 (visited on 11/08/2020).

[126] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning Transferable Architectures for Scalable Image Recognition," en, *arXiv:1707.07012 [cs, stat]*, Jul. 2017, arXiv: 1707.07012. [Online]. Available: http://arxiv.org/abs/1707.07012 (visited on 03/27/2019).

[127] H. Cai, L. Zhu, and S. Han, "ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware," *arXiv:1812.00332 [cs, stat]*, Dec. 2018, arXiv: 1812.00332. [Online]. Available: http://arxiv.org/abs/1812.00332 (visited on 03/27/2019).

[128] J. Collins, S. Chand, A. Vanderkop, and D. Howard, "A Review of Physics Simulators for Robotic Applications," *IEEE Access*, vol. 9, pp. 51 416–51 431, 2021, Conference Name: IEEE Access, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3068769.

[129] E. Todorov, T. Erez, and Y. Tassa, "MuJoCo: A physics engine for model-based control," en, in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Vilamoura-Algarve, Portugal: IEEE, Oct. 2012, pp. 5026–5033, ISBN: 978-1-4673-1736-8 978-1-4673-1737-5 978-1-4673-1735-1. DOI: 10.1109/IROS.2012.6386109. [Online]. Available: http://ieeexplore.ieee.org/document/6386109/ (visited on 02/24/2020).

[130] E. Coumans and Y. Bai, *PyBullet, a Python module for physics simulation for games, robotics and machine learning*. 2016. [Online]. Available: http://pybullet.org.

[131] E. Rohmer, S. P. N. Singh, and M. Freese, "V-REP: A versatile and scalable robot simulation framework," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Tokyo: IEEE, Nov. 2013, pp. 1321–1326, ISBN: 978-1-4673-6358-7 978-1-4673-6357-0. DOI: 10.1109/IROS.2013.6696520. [Online]. Available: http://ieeexplore.ieee.org/document/6696520/.

[132] S. Shi, Q. Wang, P. Xu, and X. Chu, "Benchmarking State-of-the-Art Deep Learning Software Tools," in *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, Nov. 2016, pp. 99–104. DOI: 10.1109/CCBD.2016.029.

[133] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah, "Comparative Study of Deep Learning Software Frameworks," *arXiv:1511.06435 [cs]*, Nov. 2015, arXiv: 1511.06435. [Online]. Available: http://arxiv.org/abs/1511.06435 (visited on 07/02/2019).

[134] M. Abadi, A. Agarwal, P. Barham, *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," *arXiv:1603.04467 [cs]*, Mar. 2016, arXiv: 1603.04467. [Online]. Available: http://arxiv.org/abs/1603.04467 (visited on 07/02/2019).

[135] R. Collobert, S. Bengio, and J. Marithoz, *Torch: A Modular Machine Learning Software Library.* 2002.

[136] A. Paszke, S. Gross, S. Chintala, *et al.*, "Automatic differentiation in PyTorch," 2017.

[137] Eclipse Deeplearning4j Development Team, *Deeplearning4j: Open-source distributed deep learning for the JVM.* [Online]. Available: http://deeplearning4j.org.

[138] Y. Jia, E. Shelhamer, J. Donahue, *et al.*, "Caffe: Convolutional Architecture for Fast Feature Embedding," in *Proceedings of the ACM International Conference on Multimedia - MM '14*, Orlando, Florida, USA: ACM Press, 2014, pp. 675–678, ISBN: 978-1-4503-3063-3. DOI: 10.1145/2647868.2654889. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2647868.2654889.

[139] T. T. D. Team, R. Al-Rfou, G. Alain, *et al.*, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv:1605.02688 [cs]*, May 2016, arXiv: 1605.02688. [Online]. Available: http://arxiv.org/abs/1605.02688 (visited on 07/10/2019).

[140] M. Plappert, M. Andrychowicz, A. Ray, *et al.*, "Multi-Goal Reinforcement Learning: Challenging Robotics Environments and Request for Research," en, *arXiv:1802.09464 [cs]*, Mar. 2018, arXiv: 1802.09464. [Online]. Available: http://arxiv.org/abs/1802.09464 (visited on 02/24/2020).

[141] M. Wise, M. Ferguson, D. King, E. Diehr, and D. Dymesich, "Fetch & Freight: Standard Platforms for Service Robot Applications," en, p. 6, 2016. [Online]. Available: http://docs.fetchrobotics.com/FetchAndFreight2016.pdf.

[142] KUKA, "LBR iiwa 7 R800, LBR iiwa 14 R820 Specification," Tech. Rep., May 2016.

[143] F. v. Drigalski, *Meet the winners from the 2016 Airbus Shopfloor Challenge*, 2016. [Online]. Available: http://company.airbus.com/dam/assets/airbusgroup/int/en/jobs-and-careers/working-at-airbus-group/Airbus-Shopfloor-challenge/Meet-the-winners-from-the-2016-Airbus-Shopfloor-Challenge-_-Robohub/Meet%20the%20winners%20from%20the%202016%20Airbus%20Shopfloor%20Challenge%20_%20Robohub.pdf.

[144] KUKA Roboter GmbH, *HRC system at BMW's production plant*, en-GB, Jun. 2017. [Online]. Available: https://www.kuka.com/en-gb/industries/solutions-database/2017/06/solution-systems-bmw-dingolfing (visited on 01/03/2020).

[145] V. Chawda and G. Niemeyer, "Toward torque control of a KUKA LBR IIWA for physical human-robot interaction," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Vancouver, BC: IEEE, Sep. 2017, pp. 6387–6392, ISBN: 978-1-5386-2682-5. DOI: 10.1109/IROS.2017.8206543. [Online]. Available: http://ieeexplore.ieee.org/document/8206543/.

[146] V. Chawda and G. Niemeyer, "Toward controlling a KUKA LBR IIWA for interactive tracking," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, Singapore, Singapore: IEEE, May 2017, pp. 1808–1814, ISBN: 978-1-5090-4633-1. DOI: 10.1109/ICRA.2017.7989213. [Online]. Available: http://ieeexplore.ieee.org/document/7989213/.

[147] J. Braumann, S. Stumm, and S. Brell-Cokcan, "Towards New Robotic Design Tools: Using Collaborative Robots within the Creative Industry," in *ACADIA//2016: Posthuman Frontiers: Data, Designers, and Cognitive Machines, Proceedings of the 36th Annual Conference of the Association for Computer Aided Design in Architecture*, 2016, pp. 164–173.

[148] J. Hallersbro and N. Josefsson, "Vision supported collaborative robot in pick-and-place operations: A coexistent material handling station between humans and a collaborative robot supported by vision systems," English, M.S. thesis, Chalmers University of Technology, Gothenburg, Sweden, 2018. [Online]. Available: http://publications.lib.chalmers.se/records/fulltext/255167/255167.pdf (visited on 01/31/2019).

[149] J. Guérin, O. Gibaru, E. Nyiri, and S. Thiery, "Learning local trajectories for high precision robotic tasks: Application to KUKA LBR iiwa Cartesian positioning," in *IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society*, Oct. 2016, pp. 5316–5321. DOI: 10.1109/IECON.2016.7793388.

[150] W. K. H. Ko, Y. Wu, K. P. Tee, and J. Buchli, "Towards Industrial Robot Learning from Demonstration," in *Proceedings of the 3rd International Conference on Human-Agent Interaction*, ser. HAI '15, New York, NY, USA: ACM, 2015, pp. 235–238, ISBN: 978-1-4503-3527-0. DOI: 10.1145/2814940.2814984. [Online]. Available: http://doi.acm.org/10.1145/2814940.2814984.

[151] R. Madhavan, "The 2016 Airbus Shopfloor Challenge [Competitions]," *IEEE Robotics Automation Magazine*, vol. 23, no. 3, pp. 21–22, Sep. 2016, ISSN: 1558-223X. DOI: 10.1109/MRA.2016.2587919.

[152] S. Mokaram, J. M. Aitken, U. Martinez-Hernandez, *et al.*, "A ROS-integrated API for the KUKA LBR iiwa collaborative robot**The authors acknowledge support from the EPSRC Centre for Innovative Manufacturing in Intelligent Automation, in undertaking this research work under grant reference number EP/I033467/1, and the University of Sheffield Impact, Innovation and Knowledge Exchange grant "Human Robot Interaction Development". Equipment has been provided under the EPSRC Great Technologies Capital Call: Robotics and Autonomous Systems.," en, *IFAC-PapersOnLine*, 20th IFAC World Congress, vol. 50, no. 1, pp. 15 859–15 864, Jul. 2017, ISSN: 2405-8963. DOI: 10.1016/j.ifacol.2017.08.2331. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2405896317331464 (visited on 02/26/2020).

[153] Stanford Artificial Intelligence Laboratory et al., *Robotic Operating System*, May 2018. [Online]. Available: https://www.ros.org.

[154] B. O. Community, *Blender - A 3D modelling and rendering package*. Stichting Blender Foundation, Amsterdam: Blender Foundation, 2018. [Online]. Available: http://www.blender.org.

[155] Sergio Guadarrama, Anoop Korattikara, Oscar Ramirez, *et al.*, *TF-Agents: A library for Reinforcement Learning in TensorFlow*, 2018. [Online]. Available: https://github.com/tensorflow/agents.

[156] Comet.ML, *Comet.ML home page*, 2021. [Online]. Available: https://www.comet.ml/.

[157] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009, ISBN: 1-4414-1269-7.

[158] J. Hale, *Which Deep Learning Framework is Growing Fastest?* en-US, May 2019. [Online]. Available: https://www.kdnuggets.com/which-deep-learning-framework-is-growing-fastest.html/ (visited on 11/08/2020).

[159] E. Coumans and Y. Bai, *PyBullet Quickstart Guide*, en-GB, 2016. [Online]. Available: https://docs.google.com/document/d/10sXEhzFRSnvFcl3XxNGhnD4N2SedqwdA (visited on 01/29/2020).

[160] T. Simonini, *Choosing a Deep Reinforcement Learning Library*, en, Jun. 2019. [Online]. Available: https://medium.com/data-from-the-trenches/choosing-a-deep-reinforcement-learning-library-890fb0307092 (visited on 11/19/2020).

[161] P. Winder, *A Comparison of Reinforcement Learning Frameworks: Dopamine, RLLib, Keras-RL, Coach, TRFL, Tensorforce, Coach and more*, en-gb, Jul. 2019. [Online]. Available: https://WinderResearch.com/a-comparison-of-reinforcement-learning-frameworks-dopamine-rllib-keras-rl-coach-trfl-tensorforce-coach-and-more/ (visited on 11/19/2020).

[162] P. Jenkner, *The Best Comet.ml Alternatives*, en-US, May 2020. [Online]. Available: https://neptune.ai/blog/the-best-comet-ml-alternatives (visited on 10/04/2022).

[163] C. R. Harris, K. J. Millman, S. J. van der Walt, *et al.*, "Array programming with NumPy," en, *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020, Number: 7825 Publisher: Nature Publishing Group, ISSN: 1476-4687. DOI: 10.1038/s41586-020-2649-2. [Online]. Available: https://www.nature.com/articles/s41586-020-2649-2 (visited on 11/08/2020).

[164] C. D. Costa, *Best Python Libraries for Machine Learning and Deep Learning*, en, Mar. 2020. [Online]. Available: https://towardsdatascience.com/best-python-libraries-for-machine-learning-and-deep-learning-b0bd40c7e8c (visited on 11/08/2020).

[165] T. Haarnoja, S. Ha, A. Zhou, J. Tan, G. Tucker, and S. Levine, "Learning to Walk Via Deep Reinforcement Learning," en, in *Robotics: Science and Systems XV*, Robotics: Science and Systems Foundation, Jun. 2019, ISBN: 978-0-9923747-5-4. DOI: 10.15607/RSS.2019.XV.011. [Online]. Available: http://www.roboticsproceedings.org/rss15/p11.pdf (visited on 06/07/2021).

[166] Tensorflow, *SAC minitaur with the Actor-Learner API — TensorFlow Agents*, Aug. 2021. [Online]. Available: https://www.tensorflow.org/agents/tutorials/7_SAC_minitaur_tutorial (visited on 09/13/2021).

[167] A. Kappeler, S. Yoo, Q. Dai, and A. K. Katsaggelos, "Video Super-Resolution With Convolutional Neural Networks," *IEEE Transactions on Computational Imaging*, vol. 2, no. 2, pp. 109–122, Jun. 2016, Conference Name: IEEE Transactions on Computational Imaging, ISSN: 2333-9403. DOI: 10.1109/TCI.2016.2532323.

[168] Tensorflow, *Networks — TensorFlow Agents*, May 2021. [Online]. Available: https://www.tensorflow.org/agents/tutorials/8_networks_tutorial (visited on 09/13/2021).

[169] Tensorflow, *Environments — TensorFlow Agents*, May 2021. [Online]. Available: https://www.tensorflow.org/agents/tutorials/2_environments_tutorial (visited on 09/13/2021).

[170] Bullet Physics SDK, *kukaCamGymEnv - Bullet Physics SDK*, original-date: 2011-04-12T18:45:08Z, Sep. 2021. [Online]. Available: https://github.com/bulletphysics/bullet3/blob/b638300fe41f0a44d3edb766772044c7b3056d7d/examples/pybullet/gym/pybullet_envs/bullet/kukaCamGymEnv.py (visited on 09/13/2021).

[171] Tensorflow, *Tf.keras.layers.Dense — TensorFlow Core v2.6.0*, Sep. 2021. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense (visited on 10/28/2021).

[172] Tensorflow, *Tf.keras.layers.Conv2D — TensorFlow Core v2.6.0*, Oct. 2021. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D (visited on 10/28/2021).

[173] Y. Wu, E. Mansimov, S. Liao, A. Radford, and J. Schulman, *OpenAI Baselines: ACKTR & A2C*, Aug. 2017. [Online]. Available: https://blog.openai.com/baselines-acktr-a2c/ (visited on 02/20/2019).

[174] *Reverb*, original-date: 2020-05-01T09:17:02Z, Sep. 2021. [Online]. Available: https://github.com/deepmind/reverb (visited on 09/15/2021).

[175] *Google Colaboratory*, en. [Online]. Available: https://colab.research.google.com/ (visited on 05/16/2021).

[176] Tensorflow, *Checkpointer and PolicySaver — TensorFlow Agents*, Sep. 2021. [Online]. Available: https://www.tensorflow.org/agents/tutorials/10_checkpointer_policysaver_tutorial (visited on 09/30/2021).

[177] Tensorflow, *Tf_agents.environments.PyEnvironment — TensorFlow Agents*, Aug. 2021. [Online]. Available: https://www.tensorflow.org/agents/api_docs/python/tf_agents/environments/PyEnvironment (visited on 09/21/2021).

[178] A. Hill, A. Raffin, M. Ernestus, *et al.*, *Stable Baselines*, Publication Title: GitHub repository, 2018. [Online]. Available: https://github.com/hill-a/stable-baselines.

[179] *Soft Actor-Critic — Spinning Up documentation*. [Online]. Available: https://spinningup.openai.com/en/latest/algorithms/sac.html (visited on 09/25/2021).