MASTER

Vision-Based Reinforcement Learning Controller Of Robotic Arm Manipulator Buzz Wire Demonstrator

Sriram, Tarun Bhargav

Award date:
2021

Vision-Based Reinforcement Learning Controller Of Robotic Arm Manipulator

Buzz Wire Demonstrator

Graduation Project Report

Tarun Bhargav Sriram
Student Number: 1444646

Master of Science in Systems and Control
Department of Electrical Engineering
Control Systems Group

This report was made in accordance with the TU/e Code of Scientific Conduct for the Master thesis

Supervisors	Lennart van Bremen (DEMCON)
	Stefan Heijmans (DEMCON)
	Bas Vet (DEMCON)
	Maarten Schoukens (TU/e)
	Roland Toth (TU/e)

Eindhoven
August 30, 2021

Abstract

This report presents the implementation of a vision-based control system for the Buzz-Wire Demonstrator Project, carried out at Demcon Advanced Mechatronics, in collaboration with the Eindhoven University of Technology. The demonstrator setup is the buzz-wire game where a metal loop is moved along a curved metallic wire while avoiding contact between the two. A vision-based reinforcement learning controller with feature extraction from image observations is presented. The design and construction of the deep-learning-based feature extraction system and the Deep Q-Learning-based reinforcement learning agent in the vision-based control pipeline are discussed. The control policies obtained are evaluated and compared for their performance in completing the 2D buzz-wire game. An end-to-end vision-based reinforcement learning control model is also investigated. This model makes action decisions based on raw pixel inputs. The results for its performance in the 2D buzz-wire task are outlined, along with the challenges presented in obtaining such a model.

Contents

Chapter 1

Introduction

Robotics and automation have become ubiquitous in manufacturing processes due to the advantages of speed in production while maintaining accuracy over repetitive tasks. These advantages are also being explored in other domains like personal assistance, surveillance, and in agriculture, where the need for increased productivity, precision, performance, and overall competitiveness is highly sought after.

Ensuring suitable behaviour and performance of robotic systems has been the subject of intense study in domains like automatic control and allied fields. Classical notions of control and system modelling heavily rely on first-principles methods based on the known physics of the system, or, in case of frequency response methods, require linear system behaviour. This can become cumbersome and prone to modelling errors based on approximations [1]. These limitations have been addressed by modern and advanced control techniques that take into account the various changes the system experiences during its operation and provide suitably optimized solutions online. The use of artificial intelligence (AI) as a platform for developing systems that make complex decisions automatically, has been researched in various domains for increasing throughput. The use of data-driven techniques for both system identification and control have seen increased popularity over recent years [2]. In particular, machine learning methods have gained attention due to their application in perception (computer vision [3]) and control of robotic sytems [4, 5].

At DEMCON Advanced Mechatronics, one such application being explored is that of a fruit-picking robot (Figure 1.1). This robot is an arm manipulator that reaches for a fruit hanging from a plant, plucks it, and places it in a container. The robot has to first sense the object (fruit) in its right location in the environment, and with a camera sensor that provides RGB images, this is done using object detection algorithms. The arm is then moved to that location and the fruit is picked by means of actuating servomotors present in the arm's joints. Furthermore, the robot must navigate while avoiding obstacles like twigs or branches and other foliage around the fruit, which do not have a fixed location. Additionally, external forces like wind or other disturbances might move these obstacles for a given localised fruit. Such a high-level task within an uncertain environment requires the development of robust and generalized solutions, which can be constructed using machine learning algorithms.

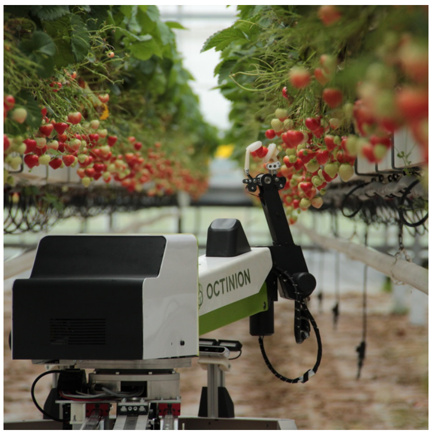


Figure 1.1: Fruit picking robot [6].

Machine Learning (ML) is concerned with developing algorithms and techniques that allow computers to create structure from available data. A statistical model is estimated to describe a mapping between two variables, and this mapping can be used to extrapolate to new and unseen data points, with reasonable accuracy. The input-output data of the system in question can be used to make a machine learning model with the inherent behaviour of the system (system identification), which can then substitute the real system in simulations for designing a controller. This ML model can also encapsulate system behaviour which may not have been modelled using first-principles, due to lack of system insight or high system complexity. ML models also provide the advantage of learning the control behaviour that is input to the system based on a reference trajectory, without requiring the construction of complex control structures by hand [7].

A particularly interesting branch of machine learning is the paradigm of Reinforcement Learning, where an *agent* learns the decision-making mechanism based on interactions with the surrounding world. The agent interacts with the *environment* by taking *actions* (Figure 1.2). An action can be defined as a high level reference trajectory that the agent has to follow (e.g., joint angle positions in an arm manipulator, position or velocity of the wheels of a mobile robot, valve positions in a pneumatic system, etc.). The agent then observes information about the state of the environment, as well as a *reward* that tells the agent about the quality of the action taken. Based on this information a subsequent action is chosen that, according to the agent, is likely to provide a higher reward signal. The series of actions and perceived information form the agent's experience and the decision-making process is solely based on this experience of former interactions and their resulting rewards. The objective of reinforcement learning is for the agent to devise a strategy (*policy*) to maximize its reward and achieve necessary performance [8].

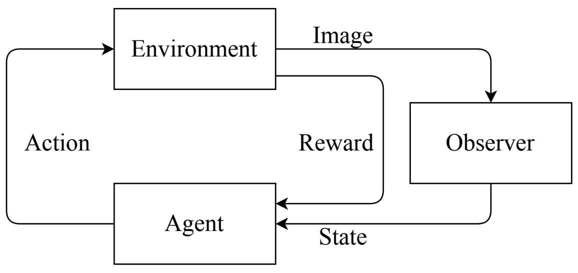


Figure 1.2: Reinforcement Learning Framework.

Reinforcement learning provides a general and intuitive robotic control framework for autonomous learning and sequential decision making under uncertainty and is scalable to many different operating conditions as well as applications [9]. The need for sophisticated prior knowledge is overcome, and all required information to achieve a particular goal is obtained through multiple trials.

While reinforcement learning provides the advantage of high-level decision-making, it also comes with challenges in development, especially in deep RL techniques. Most RL algorithms require a large dataset as well as considerable memory and computational resources. Experience data can be challenging to obtain in case of physical systems like robot learning due to time taken for the experiment. In case of transfer learning from a simulation to a real world setup, obtaining a policy from simulations that can easily generalize to a physical experiment can be difficult [10]. Formulation of a suitable reward function is crucial in RL training since the reward is the only signal with which the agent learns quality of its actions, and thus the overall policy [9]. These, along with sensitivity of RL algorithms to hyperparameter choices, are some challenges seen in RL, described in [10] and [11]. However, the generalized decision-making that results from learning trial-and-error experience is a powerful advantage to consider [12].

Reinforcement Learning techniques in control applications are attracting increased interest due to the similarities seen in the domain of optimal control [13], where these methods can be applied to systems with nonlinearities and uncertain or unknown dynamics. A tabular method of storing trial-and-error experience and assessing the quality of decisions made using this experience is the Q-learning algorithm [8]. Deep Reinforcement Learning replaces tabular methods with a neural network that approximates quality of actions taken. Deep Q-Learning [14] is one of these algorithms that naturally extends from Q-Learning, where a neural network approximator (Deep Q-Network) is used. This reduces memory complexity for high-dimensional input information and for continuous control actions [15]. An appropriate example for such a continuous control/large state and action dimension task is that of controlling an arm manipulator, which is a component of the fruit-picking robot application. Different deep RL methods are explored in common arm manipulation tasks of *pick-and-place* and *reach* in [16].

Apart from the controller design, an in-depth analysis and a suitable implementation of the observer need to be made. The sensors that gather information from the environment do not necessarily present information that is directly useful to the agent, either qualitatively (type of data) or quantitatively (need for scaling, frequency of sampling). Hence an intermediate step of translating the sensor data to information that is useful in the decision-making process is required. Vision systems are commonly employed in robotics due to the ease of installation and the observation being information-rich. Relevant data can be extracted from an image using prevalent techniques in image processing and computer vision. Such an implementation is called Vision-In-The-Loop (VITL), where the image sensing and conversion to relevant features are performed in sequence with the robot action. The camera sensors can output data in formats like colour images, depth information, etc. The type of sensor (or the type of sensor data) used depends on the specific use case and the cost of purchase.

Early methods of feature extraction from images used standard image processing techniques to extract information from the image based on detailed knowledge of the environment. However, these methods do not scale well to factors like changes in the environment or for re-usability in other applications. Robot vision applications tend to operate in environments with many features in the surroundings (eg. agriculture [17]) and it is crucial to extract the right information. Computer Vision (CV) based on machine learning provides this flexibility due to the data-driven, general-purpose learning procedures [18]. Convolutional Neural Networks (CNNs) have been widely used in object detection and classification tasks. They consist of successive filters which learn the mapping between image data to specific features. CNNs have shown to perform better than hand-made feature detectors in object detection and segmentation [19]. They are also very versatile and can support transfer learning, where the convolutional layers trained for a specific task can successfully be used to solve other problems [20] [21].

The use of vision in robotic arm manipulation has been the subject of study in the domain of visual-servoing. There are two main methods prevalant in visual servoing (VS) - Position-Based (PBVS) and Image-Based (IBVS) [22]. PBVS utilizes known geometric models of the target to determine its pose (position and orientation) with respect to the camera from the observations. The robot is then moved to that location through the joint controller. IBVS directly uses image features to control the motion of the robot, without first estimating the pose. This method is advantageous since it does not depend on the accuracy of camera calibration and the pose is implicitly encoded with respect to the target in the observations. This can be used when the robot task is more generalised and targets are not fixed, which is inherent in fruit picking tasks.

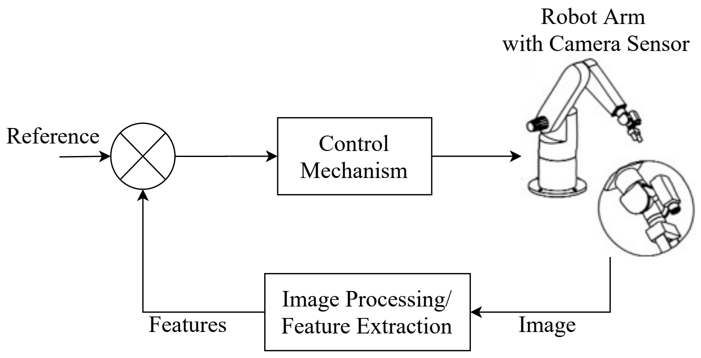


Figure 1.3: Vision-in-the-loop system for controlling a robotic arm

Combining the flexible feature extraction capability of computer vision and generalized action selection (control) of reinforcement learning for a arm manipulator task leads to vision-based reinforcement learning control (or vision-in-the-loop RL control, Figure 1.3). This system shown in the figure consists of a camera mounted on the robot arm's wrist. The camera observes a local region of interest, from which relevant features are extracted. These features are fed to the controller which decides on what action the arm has to take based on this information.

Simple tasks like reaching, where the controller provides path planning for the arm to reach a goal position in space has been implemented with the help of Deep-Q Networks where images are directly fed as state inputs to the network [23].

The design of visual perception components using computer vision techniques in robotic control are often considered as a separate task from the controller design, where feature engineering is a crucial step. However, the use of reinforcement learning has shown that the perception component can be integrated in the action decision pipeline to give a single visual control block. These algorithms provide "end-to-end" policies, where control actions are output directly from the raw input pixel data. Such policies showed better performance compared to splitting the vision and path-plannning tasks for select examples as seen in [24]. They also show the ability to eliminate the need for feature engineering. However, these techniques so far are application specific without much evidence of generalizability to other tasks.

Interesting results on vision-based control using RL are presented in [25] which shows the complex behavioural skill of door opening, and in [26] where CNN based deep RL is used to achieve the task of grasping different objects based only on image observation. These results show how the end-to-end nature of RL methods for combining the perception and control in a single unit can be used in complex tasks.

The fruit-picking application presents a similar challenging task as outlined previously. However, in order to gain competence in developing a complex application like fruit picking, a simplified problem is first considered. This includes the challenges presented by fruit picking in terms of path planning and perception, while having reduced complexity in terms of creating a visual observation of the environment and an experimental setup that can be easily assembled to demonstrate the technology used. Such an experimental setup is the Buzz-Wire game which can emulate the challenges of perception and generalized path planning as in the fruit picking application, while having a simple construction.

1.1 The Buzz Wire Experiment

The Automated Buzz Wire Experiment serves as a simple, but challenging test-bed for dynamic path planning, Computer Vision, and Machine Learning Control. In this experiment, the goal is to move a circular loop from one end of the wire to the other, without touching the wire. The loop and wire are generally made of metal, such that when they come in contact a circuit is completed, and an indication of the contact is made either by a buzzing sound or by a light source. To make the problem challenging, the loop progresses over an arbitrarily bent wire. This task is automated using a robot arm that holds the loop at its end-effector and has to sense the wire using a camera mounted on its penultimate link.

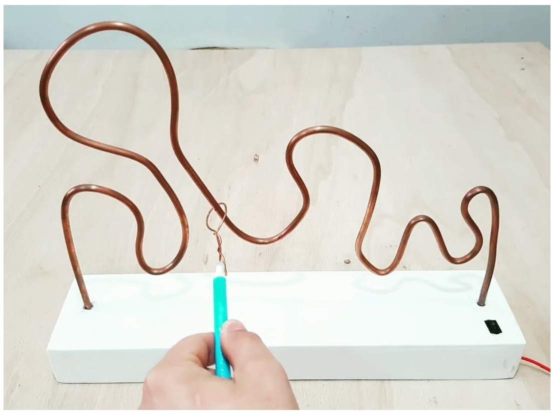


Figure 1.4: Buzz Wire Setup [27].

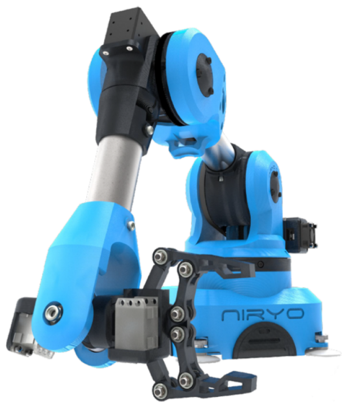


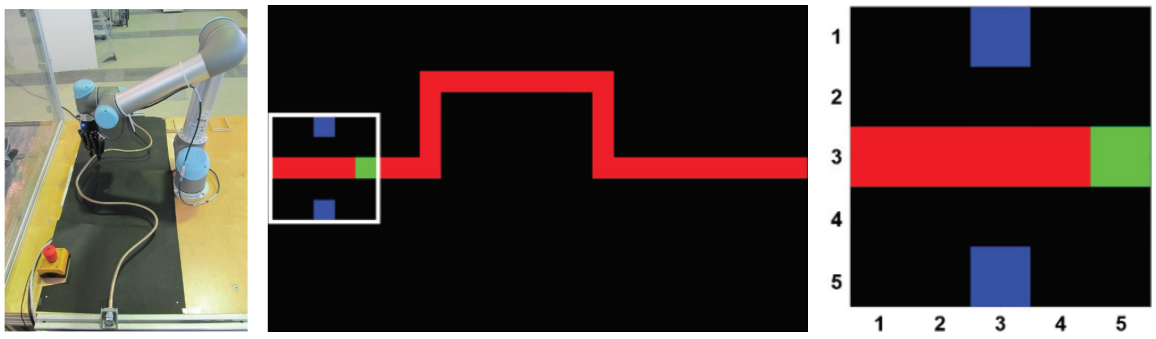
Figure 1.5: Niryo One Arm.

The metallic loop is attached to the end effector of a robot arm-manipulator that has sufficient range to traverse the wire trajectory. The robot arm chosen in this project is the Niryo One (Figure 1.1), which is a six-axis arm manipulator [28]. The arm is controlled using a reinforcement learning agent, which maps state observations (position and orientation of the loop, future goal point) and rewards, to actions that the robot needs to take to achieve the next state. These actions are defined in terms of the position that the end-effector of the arm needs to take, and underlying commands for the joint movements are calculated in the development suite provided by Niryo. Appropriate commands for joint motor forces are sent to the arm, which then physically moves along the wire. The agent is trained on different wire shapes and the resulting control policy is then evaluated on an arbitrarily bent wire, the shape of which is unknown to the controller.

Training an RL algorithm requires experience data based on trial-and-error with the environment. While a real-world system provides the most accurate representation of the experience, obtaining this data can be challenging and time consuming. Constant human supervision is necessary to ensure proper operation of the components in the setup, and this may become cumbersome. Hence, simulation environments are created, where the training data obtained is faster. However, care must be taken to include as much information that occurs in the real setup as possible to reduce the sim-to-real gap explained previously. Constructing a 3D simulation environment is still quite complex where creating large number of three-dimensional objects as the different wire confgurations is difficult, and ensuring proper behaviour of the various objects based on physics requires considerable effort. Therefore, the buzz wire experiment is further simplified to a 2D simulation environment, which is the main focus of this project, and in which the algorithms for solving the buzz-wire problem are developed.

Vision-based reinforcement learning control for the buzz-wire experiment is studied in [29] and [30]. The difference noted between these two studies is the positioning of the camera sensor. In [29], the camera is placed on the robot arm, which observes the agent locally. [30] uses a global frame of reference, from which a local image is extracted. The output at the end of the vision stage is the same as shown in Figure 1.6(c).

The features used in both articles are shown in Figure 1.6(b), where the image of the local observation of the wire and the loop is downsampled to a 5×5 matrix (local observation in Figure 1.6(c) used in [29]). The image is then one-hot encoded to four objects - the wire, the loop, background, or future goal point. This results in a $5 \times 5 \times 4$ feature matrix that is fed to an agent trained with Q-learning [8] or a modified Q-learning algorithm [30].



(a) Experimental setup. (b) Global feature space. (c) Local observation.

Figure 1.6: The Buzz-Wire experiment as described in [30].

Both articles provide the same feature extraction methods using standard image processing techniques (greyscale conversion, morphology - dilation and erosion, encoding).

While the current project deals with similar aspects of vision-based RL as explained in the previous articles, the main differences are two-fold. First, unlike the previous cases, the current project deals with dynamic trajectory tracking without complete prior knowledge of the trajectory to be followed. In this case, the vision sensor provides only local observation around a small region of the end-effector of the arm. Second, while previous articles have dealt with estimating actions directly from highly downsampled images through a policy network, in this project the network is designed to extract relevant position and orientation features from the images that describe the agent's location and future goal. These features are fed to the subsequent RL agent, thus resulting in a more generalized vision-in-the-loop control policy for the buzz-wire task, without losing information by downsampling the observation. Different training cases of the RL agent, where observations are either extracted features or direct pixel inputs, are investigated and the performance of these methods are compared.

These tasks will help in gaining insight into the objective of designing a controller for the more complex fruit-picking problem, where subtasks like object detection, localisation and locomotion are employed in the control pipeline. The method under investigation presents a unified approach to solve these tasks in a simple manner.

The positioning of the problem under consideration with respect to existing literature has been explained thus far. A formal definition of the problem and the main deliverables of the project are discussed in the following section.

1.2 Problem Statement

The goal of this project is to design a vision based reinforcement learning controller that enables the robot arm to play the buzz-wire game successfully for arbitrary wire trajectories. The 2D Simulation is of primary interest in this project to understand how to construct the vision and control algorithms and gain insight into how they may be adapted to more complex simulation environments. The main research question can be formulated as:

> *How to design and develop a vision-based reinforcement learning controller for a 6-axis robot arm, that decides on actions from observed images, to follow the trajectory of a buzz-wire?*

This overall problem can be divided into two sub-tasks: Feature Extraction and Decision Making, which fits into the robot control framework as shown in Figure 1.3, where the control mechanism represents what actions are to be taken at a given time step. The following sub-problems enumerate the intermediate steps to be taken to achieve the goal of constructing the vision-based controller for the 2D buzz wire game.

1. *How to construct machine learning models for extracting features from images (feature extraction) and the RL controller (decision making) subtasks?*

2. *Can the separate models be merged to provide a single pipeline for obtaining action decisions from images and perform vision-based control task suitably well? Is there any modification or further training necessary for obtaining the required performance?*

3. *How to quantify and measure performance of the vision-based controller?*

4. *How to construct a reinforcement learning controller that takes actions based directly on image information, without the need for an intermediate feature extraction step? What challenges occur and what alterations need to be made to the algorithm?*

1.3 Organization of The Report

Chapter 2 provides background into the concepts of reinforcement learning and computer vision, which are used in the experiments. Chapter 3 explains the 2D Simulation environment and its different components that were built for this project and the experimental framework for the vision-based reinforcement learning control as the separate problems of observation (computer vision) and control (reinforcement learning). Chapter 4 lists the experimental results and the behaviour of the RL policies are analysed in detail. Chapter 5 gives concluding arguments and a summary of the primary objectives of the project. A brief recommendation for future work to be carried out is also given. Appendices are provided and referenced as necessary throughout the report.

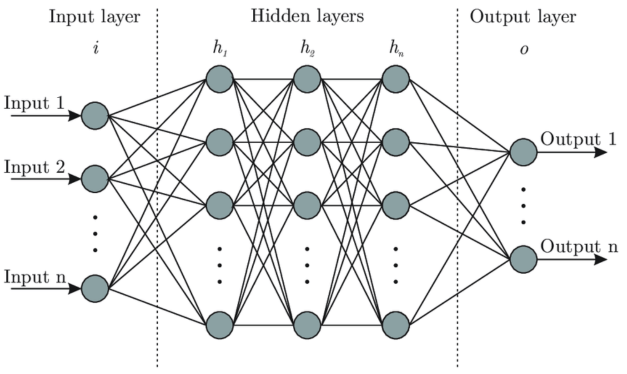
Chapter 2

Reinforcement Learning and Computer Vision

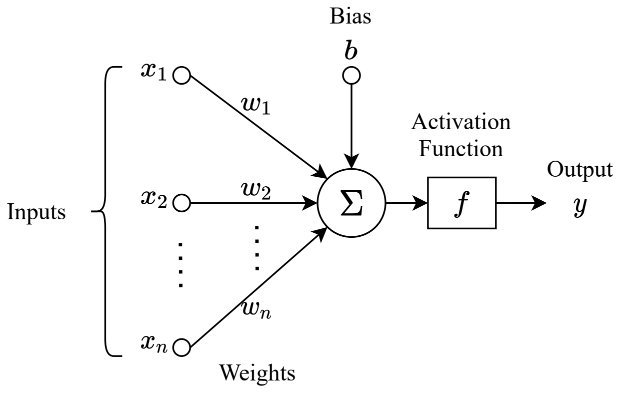
The major themes in this project are the use of machine learning methods in control and vision, especially of reinforcement learning for controlling a robot arm and the use of neural networks in both control and vision. The following sections present a brief introduction to these concepts and discuss some of the literature that are relevant to this project.

2.1 Neural Networks

A neural network is a collection of interconnected neurons in multiple layers, that provides a mapping between input and output variables. It consists of an input layer that collects the input variables, hidden layers that consist of neurons, and an output layer that provides the network output based on internal neuron operations. The most basic unit of a neural network is the neuron shown in Figure 2.1(b). The strength of the connection between two neurons is referred to as a weight. In the neuron, a weighted sum of it's inputs is calculated and passed through an *activation* function, which is usually a nonlinear function (sigmoid, tanh, ReLU, etc.).



(a) Neural Network.



(b) Operations of a neuron.

Figure 2.1: Neural Network and Neuron Operations.

A neural network is referred to as a 'deep' network when it has many hidden layers in it's architecture. The neural network learns the input-output mapping by altering it's weights over multiple training steps. A reference signal at the output of the network gives an error signal, and the learning algorithm seeks to minimize this error for a given input by changing these weights through the backpropagation algorithm [31].

Neural networks have been used in function approximation [32] as well as in image classification tasks [3]. Thus, neural networks form the basis for deep reinforcement learning, where a value function is approximated, and computer vision, which utilizes convolutional networks to extract features from an input image.

2.2 Reinforcement Learning

As mentioned in Chapter 1, Reinforcement Learning (RL) is concerned with making decisions or actions that maximises a reward signal. This decision making ability is learnt through direct interaction between an agent and its environment over several experiments, and the experience that is gained from these experiments is used to choose optimal actions that obtain maximum rewards for the given task, and eventually reaching the goal [8].

The Deep Q-learning algorithm is the RL algorithm of choice for this project since it builds on the results of the previous literature with Q-learning, and is naturally applied to a more complex setting with larger images and features as well as a larger action space. Deep Q-Learning is also among the simpler deep RL techniques.

This also allows the coupling of the convolutional neural network based feature extraction mechanism of the observer with the agent network that decides the actions, to form a single network that maps images observed from the camera to actions that the robot has to take. To understand the working of the Deep Q-Learning algorithm, the following terms are first introduced which deal with setting up the reinforcement learning problem in general.

Reinforcement Learning experiments are modelled as Markov Decision Processes (MDP, Figure 2.2). At time $t \in \mathbb{N}$ the set of natural numbers, the agent receives information about the environments state $s_t \in \mathcal{S}$, where $\mathcal{S}$ is the space of all possible states of the agent in the environment, and the quality of the state as reward $r_t \in \mathbb{R}$ (real number space). The agent then performs an action $a_t \in \mathcal{A}$, selected from the space of all possible agent actions $\mathcal{A}$ and the state of the environment changes to $s_{t+1} \in \mathcal{S}$ in the next sampled time step $t+1$, and consequently a reward r_{t+1} is obtained for this new state. The probability that the environment moves to state s_{t+1} given the action a_t from the current state s_t is denoted as $p(s_{t+1}|s_t, a_t)$. This probability defines the dynamics of the MDP. A sequence of events in the MDP gives rise to a *trajectory* defined as $(s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, \dots)$.

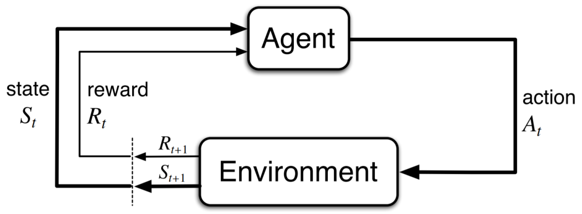


Figure 2.2: Markov Decision Process schematic [8].

The goal of the agent is to obtain a *policy* (π) that maps states to actions, such that the cumulative reward signal is maximised over a given time horizon in the experiment. In formal terms, the problem statement translates to finding the optimal policy $\pi_\star : \mathcal{S} \rightarrow \mathcal{A}$ that is given by

$$\pi_\star = \arg\max_{\pi} \sum_{t=0}^{N} r_t\left(s_t, \pi\left(s_t\right)\right), \tag{2.1}$$

with N the maximum number of steps taken in the environment and where r_t, and s_t denote the reward, and state at time $t \in \mathbb{N}$, respectively. $\pi(s_t)$ represents the action taken under policy π given the agent is in state s_t.

In case of a partially observable environment, where the problem becomes a Partially Observable MDP (POMDP), the policy is a function of the observation $o_t \in \mathcal{O}$ (observation space), which is a subset of the state space $\mathcal{S}$. For simplicity, we define the following terms considering fully observable state.

Value Functions

The reward signal is a good metric for determining the quality of the state in the immediate time step. However, since we wish to maximise the reward over multiple time steps and because this information is not easily available, an estimate of the cumulative reward that can be obtained by the agent must be calculated, from the given state. This leads to the *value function* V defined for a state s in the state-space under a given action-selection policy π [8] as

$$V_\pi(s) = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right] \text{ for all } s \in \mathcal{S}, \tag{2.2}$$

where $\gamma \in [0,1]$ is a reward discount factor that determines the importance of future rewards on the quality of the current state. This value function represents how good a state (s) is for the agent to be in, and is the expected total reward for an agent starting from the current state s.

The *action-value function* Q, or simply, the Q-function [8], gives information about the quality of the action that an agent can take, given a policy π and the current state s. This is defined as

$$Q_\pi(s,a) = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right] \text{ for all } s \in \mathcal{S} \text{ and for all } a \in \mathcal{A}. \tag{2.3}$$

The objective in RL is to find an optimal policy π_* that maximises these expected returns $V_\pi(s)$ and $Q_\pi(s,a)$. There may be more than one optimal policy π_*, but they have the same optimum returns $V_*(s)$ and $Q_*(s,a)$ given by

$$V_*(s) = \max_\pi V_\pi(s) \text{ for all } s \in \mathcal{S} \text{ and } Q_*(s,a) = \max_\pi Q_\pi(s,a) \text{ for all } s \in \mathcal{S} \text{ and for all } a \in \mathcal{A}. \tag{2.4}$$

where $V_\pi(s)$ and $Q_\pi(s,a)$ are defined in (2.2) and (2.3) respectively. The value functions can be written as a relationship between the value of a state and the value of successor states, looking ahead from the current state. The Bellman optimality equation given by

$$V_*(s) = \max_a \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \gamma V_*\left(s'\right)\right] \tag{2.5}$$

tells us that the value of a state under an optimal policy π_* must equal expected return for the best action from that state. Similarly, the Bellman optimality equation for the action-value function is

$$Q_*(s,a) = \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \gamma \max_{a'} Q_*\left(s', a'\right)\right]. \tag{2.6}$$

This method of estimating the value functions from the Bellman equations is difficult to perform when there is no knowledge of the system dynamics. An alternate method for the same is to incrementally update the value function for every time step, rather than at the end of a sequence of events (episode). The update rule for the incremental update is given generally as

$$\text{New Estimate} \leftarrow \text{Old Estimate} + \text{Step Size} \times [\text{Target} - \text{Old Estimate}]. \tag{2.7}$$

The update equation for the state value function is given accordingly as

$$V_\pi\left(s_t\right) = V_\pi\left(s_t\right) + \alpha\left[r_{t+1} + \gamma V_\pi\left(s_{t+1}\right) - V_\pi\left(s_t\right)\right] \tag{2.8}$$

where α is the step size parameter ($\alpha \in \mathbb{R}$ set of real numbers).

Action Selection

The quality of the action that an agent takes can be evaluated based on the action-value function and the given policy. It is desired that the action taken results in the highest reward at the end of the experiment. The agent can choose a random action from the action-space and accummulate information regarding the quality of various actions (exploration). The agent can also take the action that it knows from experience to result in the highest reward at that instant (exploitation). However, if the agent only exploits, it may ignore actions that it has not yet known to provide a higher reward, and if the agent only explores, then it may not maximise its return at the end of the experiment. Hence the agent must create a balance between exploration and exploitation. One method that incorporates this alternation between exploration and exploitation is the $\epsilon-$greedy method, where the agent takes a random action with a probability $\epsilon \in [0,1]$ and otherwise takes the *greedy* action, i.e., the action that results in highest reward at that instant. This $\epsilon-$greedy method is depicted as

$$a_t = \begin{cases} \text{argmax}_a\, Q_\pi(a) & \text{with probability } 1-\epsilon \\ \text{a random action} & \text{with probability } \epsilon \end{cases} \tag{2.9}$$

2.2.1 Deep Q-Networks

Instead of evaluating the quality of the state reached in successive steps, the quality of the actions taken from the current state can be used to obtain the optimal policy π_*. In this case, the action-value function, or Q-function can be used and updated incrementally as

$$Q_\pi\left(s_t, a_t\right) = Q_\pi\left(s_t, a_t\right) + \alpha\left[r_{t+1} + \gamma\ \text{max}_{a_{t+1}} Q_\pi\left(s_{t+1}, a_{t+1}\right) - Q_\pi\left(s_t, a_t\right)\right]. \tag{2.10}$$

The objective of Q-learning is to learn the Q value of all state-action pairs in the environment. For finite and discrete state and action spaces, this information can be easily stored in *Q-tables* and as the agent obtains new information about the expected reward for an action given a state, this table is updated to reflect the experience from the action. The agent explores several actions over multiple experiments until an optimum is reached for each state-action pair. This information is then used to define an optimal policy that maximises the cumulative rewards in the experiment. Q-learning is an iterative off-policy method, where the Q-table is updated regardless of the type of action selection policy used.

In case there are a large number of states and actions in the system, or even a continuous state and action space, forming a Q table becomes infeasible. A neural network function approximator can be used to estimate the Q value for a given state-action pair. With many hidden layers, the network becomes 'deep', which leads to Deep Q-Learning [14] that will be a subject of study in this project (Figure 2.3).

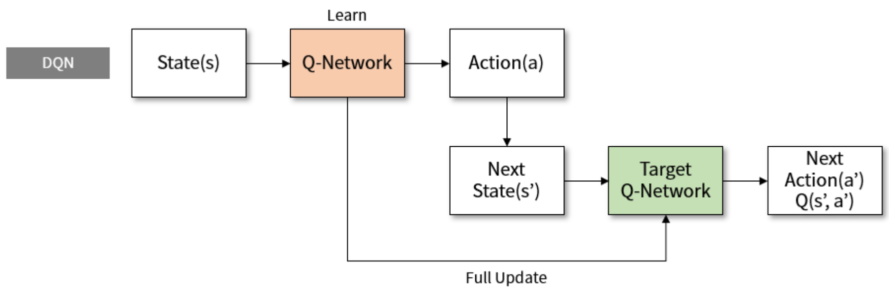


Figure 2.3: Deep-Q Learning schematic with Target Network [33].

The Deep Q-Network (DQN) approximates the Q value for each action in the action space given a state input, as the output neurons of the network, and the action with the highest Q value is chosen by the agent in a greedy policy scheme. The parameters of the network are updated through backpropagation based on the Temporal Difference (TD) Loss given by

$$L_t\left(\theta_t\right) = \mathbb{E}_{s_t, a_t, r_t, s_{t+1} \sim \mathcal{D}}\left[\left(y_t - Q\left(s_t, a_t; \theta_t\right)\right)^2\right], \tag{2.11}$$

where $y_t = r + \gamma\ \text{max}_{a_{t+1}} Q\left(s_{t+1}, a_{t+1}; \theta_{t-1}\right)$ is the TD target return and $y_t - Q$ is the TD error calculated at the output of the Q-network. The parameters of the Q-network at a given time step is θ. However, using this loss to update the Q-network implies the target is continuously changing as the parameters of the Q network change, which makes the target unstable and training difficult. To avoid this, the use of a target network was proposed in [34], which has the same architecture as the main Q-network, but the parameters of the target network are updated with a given frequency, usually more than every iteration of the Q-network update. The Q-network is updated every step of the simulation and these parameters are copied to the target network at a given number of steps. This leads to the TD target $y_t = r + \gamma \max_{a_{t+1}} Q\left(s_{t+1}, a_{t+1}; \theta_t^{target}\right)$ where θ^{target} are the parameters of the target network, which makes the target return more stable leading to better training of the agent.

Experience Replay is also introduced in the training process [35][36], where the state transitions, rewards and actions for a number of episodes are stored and the agent is trained on mini-batches of this data. This avoids overfitting the agent to only recent experiences and reuses past transitions to avoid catastrophic forgetting. Table 2.1 summarizes the deep-Q learning algorithm.

```
Deep Q-Learning with Experience Replay and Target Network
Initialize primary network $Q_\theta$, target network $Q_{\theta^{target}}$, replay buffer $\mathcal{D}$
for each iteration (epoch):
  Data Collection -
  for each environment step:
    Observe state $s_t$ and select action $a_t$ based on policy $\pi(a_t, s_t)$
    Execute $a_t$ and observe next state $s_{t+1}$ and reward $r_t$
    Store $(s_t, a_t, r_t, s_{t+1})$ in replay buffer $\mathcal{D}$
  Network Training -
  for each update step:
    sample experience $e_t = (s_t, a_t, r_t, s_{t+1})$ $\mathcal{D}$
    Compute target value:
      $y_t = r + \gamma \max_{a_{t+1}} Q\left(s_{t+1}, a_{t+1}; \theta_t^{target}\right)$
    Perform gradient step with loss $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
    Update target network weights:  $\theta^{target} \leftarrow \theta$
```

Table 2.1: Deep Q-Learning algorithm summarized.

2.3 Computer Vision

The input to the RL agent is the feature vector that is extracted from the image observation. This is achieved through computer vision, where a convolutional neural network is trained on various example images and corresponding feature vector references. An example of such a convolutional neural network is shown in Figure 2.4, where an object recognition task is portrayed. To understand how a convolutional neural network is constructed, the following concepts are introduced.

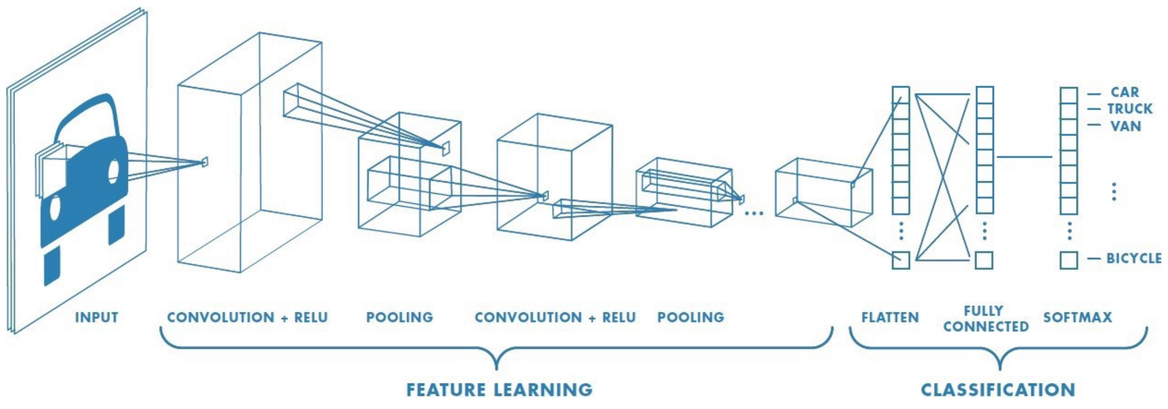


Figure 2.4: Convolutional Neural Network (CNN) architecture used for image classification [37].

A camera is attached to the penultimate link of the robot arm and acts as the main sensor to observe the system state, which is the agent (loop) located at a given pose with respect to the wire. The camera provides RGB images, from which specific information must be extracted and fed to the RL agent. This feature information can be the end-effector position and orientation of the arm, and a local goal that the arm should achieve (new position), etc. A natural first step would be to explore standard image processing techniques where transformations (‘smoothing’, ‘sharpening’, etc.) are applied to the image to result in an output from which the ‘features’ are easily extracted. The features can be defined based on the specific problem as particular regions in the image or values calculated from image characteristics. However, defining these transformations are quite specific to the use case, and with a change in environments, new transformations have to be calculated. Hence, these techniques are not scalable to feature extraction tasks in more complex images, and not flexible to changes in environments. This problem is addressed in the use of machine learning-based computer vision techniques, which provide a common framework for different feature extraction problems, thereby providing both scalability and flexibility for use cases.

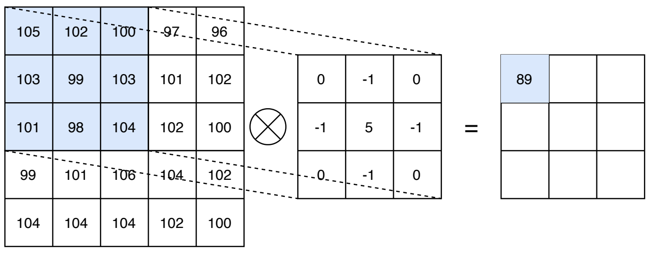
Neural Networks have been extensively used for image classification and feature extraction tasks. The most common example is the classification of handwritten digits [38] which acts as a benchmark for different machine learning algorithms. Convolutional Neural Networks (CNNs) are widely used for feature extraction and classification in cases like document recognition [3], image classification [39] and in more advanced visual cognition tasks.

2.3.1 Convolutional Neural Networks

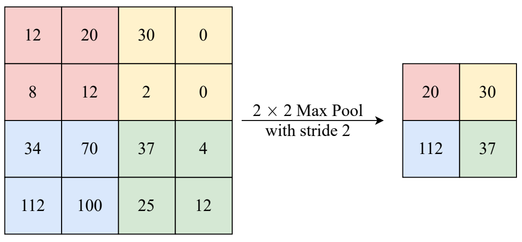
A convolutional neural network (CNN) consists of three main types of layers: convolutional layers, pooling layers, and densely-connected layers.

A convolutional layer consists of multiple filters (kernels) whose values are trainable. These filters are applied to the input matrix (image), through the convolution operation, and the output is the filtered image in the different filter channels. These trained filters provide information about the local pixel relationships, like edges and contours, and more complex relations in deeper layers.

The pooling layers are usually alternated with convolutional layers. These layers down-sample the input matrix (image), thus reducing its dimensionality. Pooling is done in various ways, like max, min, or average pooling, where the maximum, minimum or, the average value of the selected region is chosen as output and the rest discarded.



(a) Convolution.



(b) Pooling.

Figure 2.5: Convolution and Pooling Operations.

Dense layers (fully-connected) can be applied at the end of convolutional layers and are used to project the features encoded in the output of convolutional layers to higher dimensions, which is then used to estimate the output label.

The advantages that CNN’s bring compared to regular densely connected neural networks is the reduced number of trainable parameters, due to shared parametrization in the filters. This reduces training time and can avoid over-fitting. The filters learned and applied on the image produce spatial feature maps, that provide information about the relationship between nearby pixels in an image. These can provide insight into the complex intermediate transformations occurring in the feature encoding, that may not have been arrived at using standard image processing methods.

Chapter 3

The 2D Simulation Environment

Reinforcement Learning techniques require a lot of data to train the agent for obtaining an optimal policy, due to the trial-and-error nature of learning. This may not be feasible when training a real-world agent since number of experimentation cycles may be limited, executing a sequence of actions may require relatively higher experimentation time, and damage occurring in the hardware to wear or some unforeseen actions. It is therefore crucial to begin development of a reinforcement learning solution through simulation, which can provide faster experimentation time while being able to run a large number of experimental episodes, and also provide a platform to understand and eliminate analytical errors that may have been ignored in designing the problem.

A trade-off exists between how well the simulation mimics the real-world behaviour of all components of the experiment and how simplistic the simulation can be to test the algorithms for control and observation. Understanding this *sim-2-real* gap is necessary in designing the workflow of the development and defining checkpoints from where knowledge from a simpler experiment can be used in a more complex one. For the buzz wire experiment in particular, a two dimensional simulation of the environment is chosen as the starting point. This provides enough freedom to depict complexity in terms of sufficiently identifying the different objects in consideration, like the metallic loop and the wire, and in terms of applying the reinforcement learning and machine vision techniques. It was shown that the RL policy generated from the 2D Simulation explained below is able to be successfully applied on a hardware setup, with certain assumptions [40].

In this section, the 2D Simulation setting is explained and the different components of this simulation are shown in detail. Experimental results for generating the RL policies as well as their evaluation in the 2D Buzz Wire Problem are shown in the following chapter.

3.1 Experimental Setup

The 2D Simulation experiment for the buzz wire experiment should be framed in terms of the Markov Decision Process setting as in Figure 2.2. The different components of this setting is explained in the following sections. The environment was constructed using the Open AI Gym toolkit [41], which can be used to develop and compare reinforcement learnig algorithms by abstracting the experimental setting through APIs. The agent and the training process were created using the TF-Agents toolkit [42], which provides APIs for implementing, training and deploying RL agents.

3.1.1 Environment and Observation

The global view of the simulation environment is shown in Figure 3.1. This consists of the wire between two walls, coloured in red, that depicts the region where the agent must not hit. The agent or the loop is coloured in blue (top half) and green (bottom half) so that conflicts in detecting angle of the agent that are 180° apart are resolved. This enables the inclusion of wire shapes that turn backward in the global view, and the differential colouring lets the agent know which direction is forward along the wire. The wires can be arbitrarily shaped in the environment as well as randomly rotated. The given example in Figure 3.1 shows a ramped sinusoidal wave.

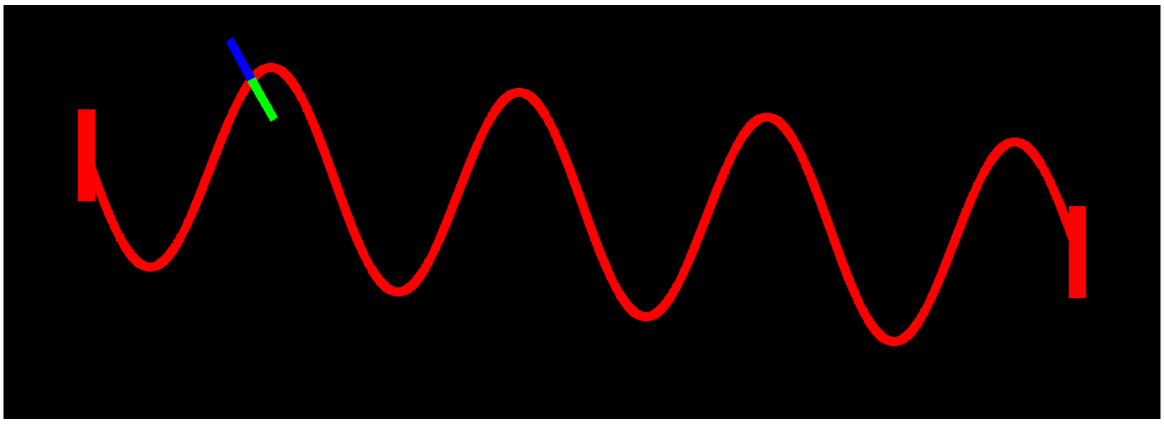


Figure 3.1: Global view of 2D Buzz Wire Simulation environment.

In the given environment, the eye-in-hand configuration of the camera is simulated, where the camera is situated on the wrist-link of the robot arm. The robot then observes only a local region around the agent. To simulate this in the 2D environment, the following example of a local observation is seen in Figure 3.1.1. The image size is 256×256×3 for the three colour channels.

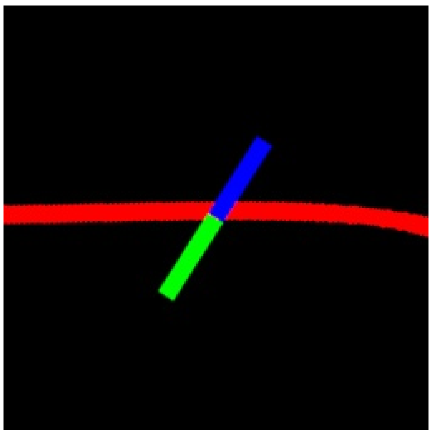
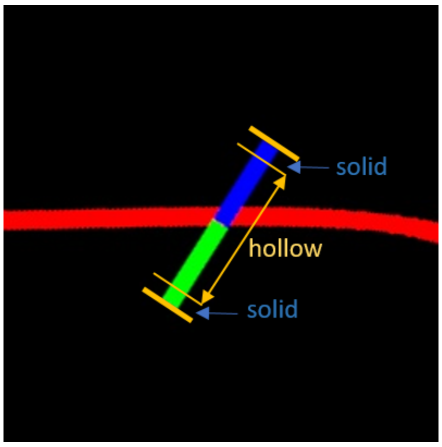


Figure 3.2: Local observation with agent and a portion of the surrounding wire.

3.1.2 Agent

The agent itself is constructed to include thickness due to the material of the loop and a hollow part as shown in Figure 3.3. The two ends of the agent are solid to simulate material thickness, which is considered equal to the width of the agent. Figure 3.4 shows the loop in a 3-Dimensional setting where the wire (in grey) passes through the hollow part of the loop (in red).



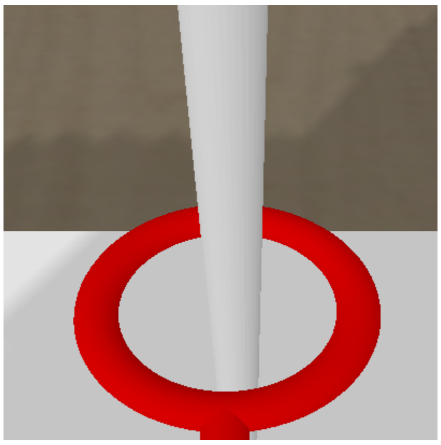
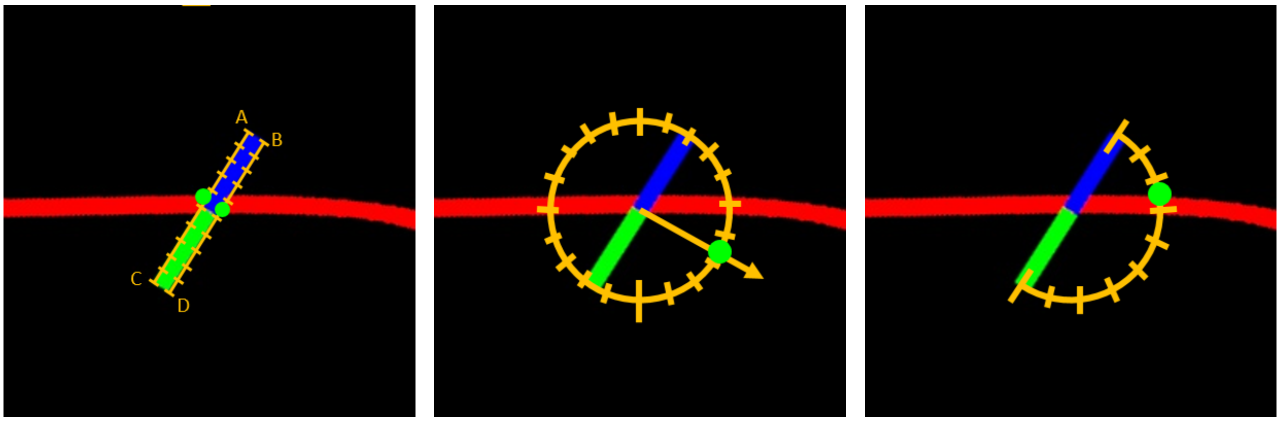


Figure 3.3: Agent construction (solid and hollow parts). Figure 3.4: 3D View of hollow part of agent.

In the local observation, the agent is allowed to move in the rotational degree of freedom (θ), which simulates the placement of the camera on the wrist link of a 6-axis arm manipulator.

3.1.3 States

The underlying state that is extracted from the observation model is explained as follows. Specific information like the agent's location, orientation and a local goal to be reached is extracted from the local observation as seen in Figure 3.1.1. The choice of these features was the subject of study in [43]. These features are encoded in a one-hot encoding (OHE) scheme, so that the feature extraction stage is a classification problem, which is a common task in machine learning-based vision. In the OHE scheme, the labels are vectors of zeros with a single one placed in the relevant position of the class occurrence. This is reformulated into predicting the agent pose, orientation and future orientation.



(a) Agent Location. (b) Agent Orientation. (c) Future-wire Orientation.

Figure 3.5: Encoded Features.

Feature	Hyperparameter	One-Hot-Encoded vector
1 - agent location front edge (BD)	# sections = 7	$[0,0,0,1,0,0,0,0]^T$
2 - agent location back edge (AC)	# sections = 7	$[0,0,0,1,0,0,0,0]^T$
3 - agent orientation	# sections = 21	$[0,0,0,0,0,0,1,0,0,\cdots,0,0,0]^T$
4 - future wire orientation	radius = 5 # sections = 9	$[0,0,0,0,0,1,0,0,0]^T$

Table 3.1: One-Hot-Encoded features for example in Figure 3.5[43].

This feature formulation results in four multi-class classification problems, where one class (encoding) is to be predicted in each of the four features.

These OHE vectors are chosen as the state of the system as they provide low dimensional feature input to the reinforcement learning agent that gives the advantage of reduced computation time as well as greater speed of learning. Concise agent properties like distance of loop corners from the wire, angle of the agent (in radians), and future orientation was also considered in the feature extraction problem. However, since these quantities are real valued and obtained from SVD operations, they are not accurately estimated by neural networks, and the classification problem is a much easier task with CNNs. Hence, the OHE features are chosen as the input to the reinforcement learning model.

In a single step, the agent can take a discretized action in +1, 0, or -1 times the step-size of motion. This combination of three possible movements for the three degrees of freedom gives 27 possible actions in a given step. These actions are indexed as 0 to 26 as shown in Appendix A.

3.1.4 Action

The agent in the 2D Simulation has three degrees of freedom: translation along longitudinal axis of the loop (u), translation along lateral axis of loop (w), and rotation around loop centre (θ). Figure 3.6 depicts the directions in which the agents can move.

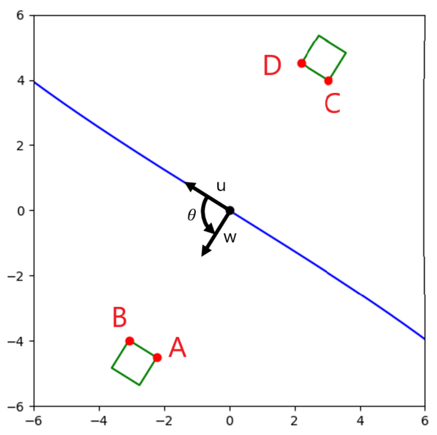


Figure 3.6: Action directions [40].

3.1.5 Reward and Episode Termination

The reward scheme used in this project is derived from [40], where the policies obtained with a dense reward scheme based on euclidean distance and a sparse reward scheme were tested. It was found that the policy obtained from the sparse reward experiment performed better generalization compared to the dense reward. This is likely due to the fact that the euclidean distance based reward does not encode information regarding the path taken along the wire. Instead the absolute distance between agent location and reference point, which may vary for wires of different amplitudes, frequency and shape. In this project, the sparse reward is chosen due to it's proven performance. The reward function at a given time step k is given by

$$r_k = \begin{cases} r_{\text{hit}} & \text{if agent hits wire,} \\ r_{\text{finish}} & \text{if agent reaches finish point,} \\ 0, & \text{else.} \end{cases} \tag{3.1}$$

In the following experiments $r_{hit} = -100$ and $r_{finish} = 2000$. The reward for hitting the wire is accumulated from current time step at which agent hits the wire up to the 400th step in that episode. Episode termination occurs either when the agent reaches the goal or at 400 steps in the episode.

This cumulative negative reward gives an idea of how much distance is left to travel along the wire before the goal is reached. However, in case of no action taken, where the reward accumulated is zero, there is no progress along the wire. A negative reward of higher magnitude implies the agent is closer to the finish.

3.2 Observation and Control

The previous sections provide a brief explanation of the experimental setting of the 2D buzz wire simulation. These components are constructed in the Open AI Gym toolkit and the algorithms in the following sections are developed for obtaining a vision-based reinforcement learning control solution.

The vision-based control task is divided into two components as shown in Figure 1.3, where the vision system is a convolutional neural network that extracts relevant features from the image observation and the controller consists of a path planning module using reinforcement learning for the actions mentioned previously.

3.2.1 Feature Extractor Network

The construction of the feature extractor network was the subject of study in [43]. The relevant features extracted by this network and the choice of the network architecture are discussed and the results are briefly mentioned. The Feature Extractor Network (FENet) has the architecture shown in Figure 3.7.

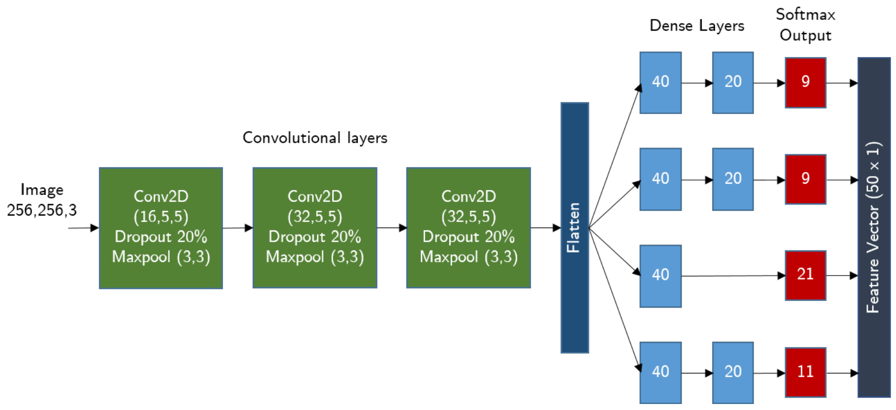


Figure 3.7: Feature Extractor Network (FENet) Architecture.

The branching network gives rise to four separate classification problems that are suitable for simultaneously producing one-hot encoded feature estimates that are required for the observations as seen in Figure 3.5. All layers use the ReLU activation function, except the final output layer which uses the softmax function suitable for classification. This network was trained on a dataset of image examples and corresponding feature outputs as explained in [43], and shows a reasonably good feature extraction performance as shown in Table 3.2.

Feature	# of classes	Accuracy %
1 - Forward Edge	9	80
2 - Back Edge	9	80
3 - Orientation	21	98
4 - Future Point Orientation	11	70

Table 3.2: Feature Extraction Accuracy of FENet.

This forms the vision system that provides relevant information to the following stage in the control pipeline, the path planning and control module that is characterized as a deep-Q network.

3.2.2 Deep-Q Network

As explained in Section 2.2.1, the RL algorithm of choice is the Deep-Q Learning algorithm, in which the Deep-Q Network (DQN) is an integral part. This DQN is the agent that performs the path planning and control in the 2D Buzz-Wire simulation control pipeline. The architecture of the DQN trained from ground truth features was investigated in [40] and the same architecture will be used to evaluate the control performance in the vision-based control system where the features are now estimated by the FENet. Figure 3.8 shows the DQN architecture used in the 2D simulation experiments.

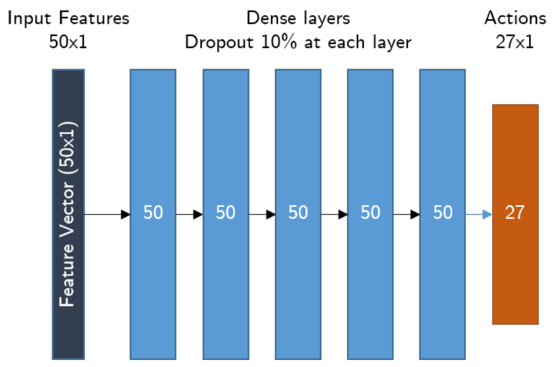


Figure 3.8: Deep-Q Network Architecture.

The DQN agent is trained on various sinusoidal wire confgurations with randomized amplitude, frequency, phase and global orientation, within reasonable bounds of the agent (loop) being able to traverse along the wire. The input to the DQN agent is the ground truth features calculated by hand in the simulation environment. These ground truth features provide the most accurate representation of the required features. The result of the training process is a *policy*(control mechanism) with which the agent chooses actions given a particular observed state.

3.2.3 Measuring Performance

To understand how well a policy generated from training performs on the buzz wire task, it must be evaluated on example wires to see whetehr the agent is able to travel along the wire and reach the goal, without hitting the wire. For this a set of 50 fixed sinusoidal wire confgurations are sampled from the same distribution of sine waves with randomized amplitude, frequency, phase and global orientation are chosen. The performance of a policy is then measured in the number of evaluation wires it is able to complete. If the policy is able to complete more number of wires in this set, the policy is able to generalize well to the sinusoidal buzz wire cases. If it is able to complete only a few number, such a policy is deemed poor.

A policy must accommodate for imperfect observations by the vision system (FENet) in order to navigate the wire. Additionally, noise can be added externally to the observations. The robustness of a policy is also tested by inducing these noisy conditions. If a policy is able to complete many wires under this artificial noise, it is considered robust. Further, to measure a policy's generalizability beyond the training set, it is evaluated on 50 multisinusoidal wires with 10 sine components. These wire configurations are not part of the training set, and a policy is considered generalizable to the 2D buzz wire task if it completes many multisinusoidal wires.

With this method of performance measurement, absolute performance of a policy cannot be estimated. It can only be compared with the performance of another policy. This is because the number of wire configurations that can be sampled from the sinusoidal class of waves with randomised parameters mentioned earlier are infinite. It is impossible to truly measure absolute performance over the infinite number of configurations. Hence, a set of 50 wires are chosen that represent various levels of difficulty and the policies are compared over this set.

Chapter 4

Experimental Results

In this chapter, the different training phases are explained, where vision-based RL models with feature engineering and without are generated. The policies obtained from these training phases are evaluated and their behaviour is analysed. Policy π_{GT} is generated by training the RL agent in an environment where the observations are calculated from the ground truth location and orientation of the loop. Policy π_{EF} is generated by training the agent in an environment with observations extracted from local images of the environment, as explained in Chapter 3. The end-to-end policy is obtained by training a DQN model with convolutional layers on image observation data. These policies are then compared for their performance.

4.1 Training Phases

The two networks explained in the previous chapter (Figures 3.7 and 3.8) were trained separately on their respective datasets and their performance for given sub-tasks has been recorded in [40], where the policy π_{GT} was obtained, and [43], where the feature extraction mechanism was constructed. The combination of the two to provide a vision-based controller is studied in this project. Figure 4.1 shows the resulting vision-based controller with the FENet and DQN as part of the control pipeline.

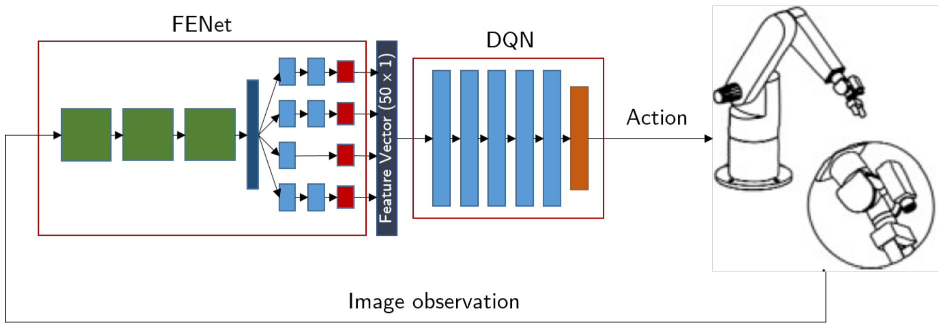


Figure 4.1: Combined vision-based reinforcement learning controller.

The two components trained separately must first be evaluated for their performance when they are directly coupled with each other. This presents the first training phase, where the feature observations in the environment are the output of the FENet and the DQN trained on ground truth features estimates the action that has to be taken for navigating the 2D buzz wire. The control policy of the DQN in this case is denoted as π_{GT}. This policy is expected to perform poorly since the output of the FENet is not completely accurate and estimation noise creeps in with some erroneous features being used. The robustness of the control policy can also be tested in this case.

The next training phase is where the DQN is trained on the noisy observations of the environment that are output by the FENet and not the ground truth, for the same training dataset of randomised sinusoidal wires. This control policy is denoted as π_{EF}, and its performance with estimation noise and artificially added noise (to test robustness) is compared with that of policy π_{GT}.

Both policies are then evaluated on wires made of multisines, which were not part of the original training dataset, to test how well they can generalize over unseen scenarios.

The final training phase investigated in this project is that of an end-to-end architecture, where the tasks of feature extraction and action selection are performed by a single model. This eliminates the feature engineering step and allows the agent to decide what features are useful in completing the goal. Figure 4.2 shows the architecture of the end-to-end model.

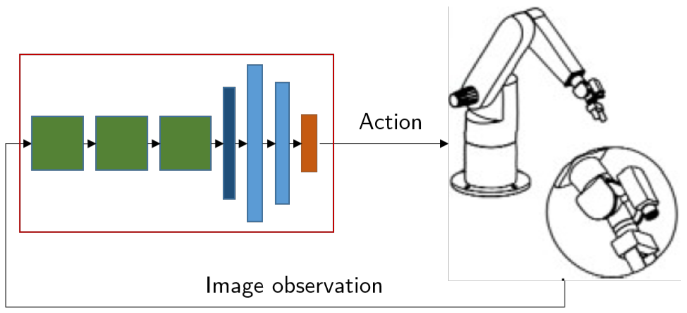


Figure 4.2: End-to-end visual reinforcement learning controller.

Obtaining π_{GT} was the subject of study in [40]. The results from this study are carried forward in this project. The following section shows the training result for obtaining π_{EF}.

Table 4.1 summarizes the different training phases considered in the 2D buzz-wire experiment.

Training Phase	Description
Direct Coupling	Policy π_{GT} evaluated on agent in environment with FENet output observation
DQN trained on estimated features	Policy π_{EF} evaluated on agent in environment with FENet output observation
End-to-end training	Agent trained on environment with image observation

Table 4.1: Training Phases.

4.2 Training results

The most important hyperparameters used in the training experiment to generate the policy π_{EF} are shown in Table 4.2. The exploration rate ϵ induces randomness in selecting an action from the action space in an ϵ-greedy algorithm. A high rate of exploration was chosen as the starting value so that the agent can get fairly good estimates on how good taking all possible actions are. This probability is decayed linearly over the experiment and the policy becomes more greedy towards the final epochs, choosing only the best actions and reinforcing their action value estimate. The rewards chosen for this experiment were

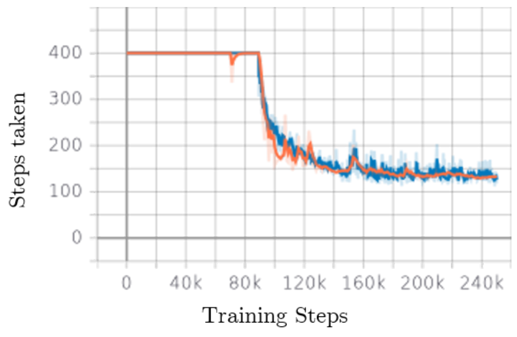
Hyperparameter	Value
ϵ - Probability of random actions	$1.0 \rightarrow 0.0001$ (linear decay)
α - learning rate	$10^{-2} \rightarrow 10^{-4}$ (linear decay)
γ - Discount factor	0.4
Replay buffer capacity	10^7
Batch Size	256
Target network update frequency	100
Experience collection steps per epoch	2000
Training steps per epoch	100
Number of training epochs	2500
Evaluation examples	50
Number of training epochs after which evaluation occurs	10

Table 4.2: Hyperparameters chosen for the training experiment in generating π_{EF}.

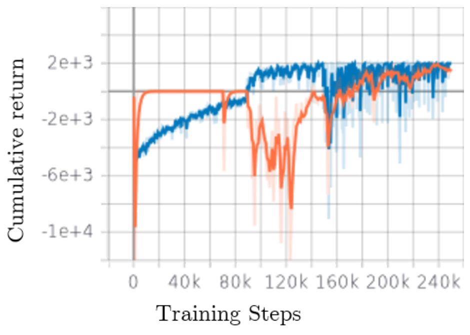
A small reward discount factor (γ) means the agent estimates rewards more into the immediate future steps and is beneficial in case of POMDPs where the end goal is not visible at the outset. A large γ weights the reward estimates well into the future and is suitable for finite horizon episode lengths. A central value of $\gamma = 0.4$ was chosen by considering the trade-off provided by both the extreme conditions. A large replay buffer capacity is beneficial in Q-learning since more historical data can be stored, and the agent can be trained on experiences that are well into the past. Before training the network, some considerable amount of trial-and-error experience needs to be accumulated, where the agent can test its ability to make the correct action decisions over multiple steps in an episode. For this a value of 2000 collection steps was a reasonable choice. After the data collection is done, the agent is trained over the experience data in the replay buffer (Table 2.1). This is done for 100 training steps. With a batch size of 256, the DQN is trained over 25600 examples during the training phase in epoch of the Deep Q-Learning algorithm.

If the target network is updated very frequently, it presents a moving target problem where the main Q-network tries to adapt itself to its past value, causing unstable learning. A slow target network update frequency, however, causes slow learning and convergence of the policy. Based on this trade-off the target update frequency chosen was 100 steps.

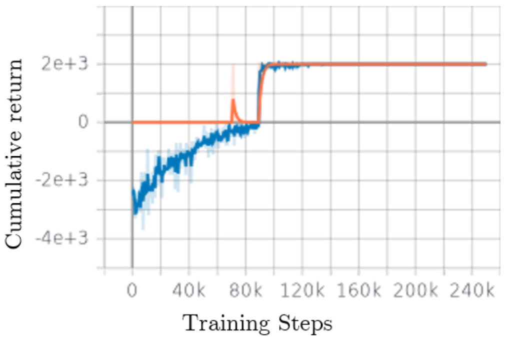
The training progression for generating π_{EF} is shown in Figure 4.3. Since one training epoch consists of 100 training steps, the x-axis of the graphs show a total of 250000 data points for 2500 epochs. The number of environment steps taken in an episode is shown in Figure 4.3(a). The finish point for the sinusoidal wires used in the training dataset is around 140 steps (can vary due to different amplitude wires). A stable episode length around this value shows that the agent's policy is able to complete wire trajectories. The Figures 4.3(b), 4.3(d) and 4.3(c) show average, minimum and maximum cumulative returns in each set of episodes used in training. The training data consists of multiple episodes where the agent performs actions in the environment, and among the maximum and minimum return over all these episodes are recorded. The average return is the numerical average of the returns obtained in all the episodes recorded between training points. The objective is to stabilize the number of steps taken in an episode to complete the wire and to maximize the returns, which results in the agent completing the buzz wire.



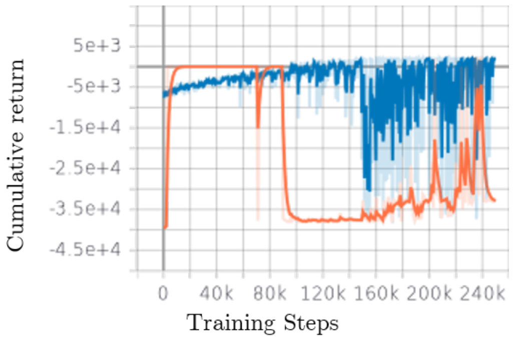
(a) Episode Length.



(b) Average Returns.



(c) Maximum Returns.



(d) Minimum Returns.

Figure 4.3: Training progression for Policy π_{EF}. Blue shows the training curve and orange shows the evaluation curve (plots from TensorBoard).

Initially, the agent has a high probability of exploration ($\epsilon \sim 1.0$). This makes the agent choose actions which do not result in completing the wire. Hence a maximum of 400 episodic steps are taken in the environment (Figure 4.3(a)). These episode steps pertain to the data collection phase in the Deep Q-Learning algorithm (Table 2.1). Correspondingly in the returns graphs, the training returns is negative, which implies the agent hits the wire. At around 90k training steps, the agent learns to find positive rewards, i.e., the agent completes wires. This is seen in the maximum returns graph where the training returns is at the maximum of 2000. As the training progresses, we see the average returns becoming increasingly positive, which signifies multiple configurations of wires being completed and the agent achieves a degree of generalization. The policy state with maximum average return is saved for the same reason and used in further evaluation experiments.

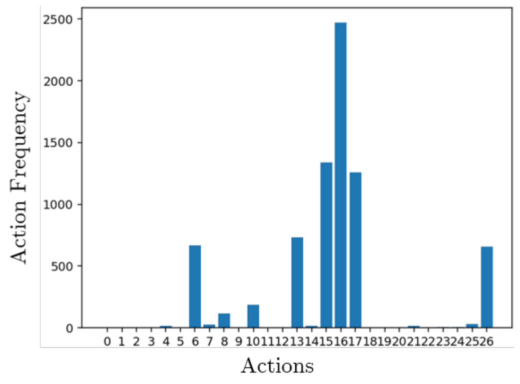
4.3 Policy Evaluation

The policies obtained in the two experiments are evaluated for a set of 50 evaluation wires with randomised frequency, amplitude, phase and rotation. The evaluation environment used to test these policies consists of the observations that are output from the feature extractor network. The performance of these policies can be seen for an example wire in the evaluation set in the following two videos, where the policies are evaluated on the same wire.

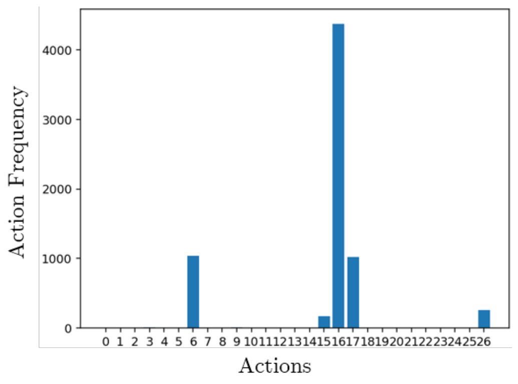
π_{GT} evaluation on example wire (click to view)

π_{EF} evaluation on example wire (click to view)

The two policies show distinct behavioural difference in the way they navigate the wire. π_{GT} shows jittery actions, but continuous forward motion along the wire. In contrast, π_{EF} shows smoother motion along the wire without the jitter. This evidence is also seen in the frequency of actions taken as shown in Figure 4.4.



(a) Frequency of actions with π_{GT}.



(b) Frequency of actions with π_{EF}.

Figure 4.4: Frequency of actions with different policies over 50 evaluation wires.

These plots show the cumulative number of times a particular action is taken as listed in Table A.1 when the policies are evaluated in the 50 test wires. Comparing Figures 4.4(a) for π_{GT} and 4.4(b) for π_{EF}, we see that actions 15 and 17 (from Table A.1) are more frequent in the former case than the latter. This explains the consecutive forward and backward rotations of the agent when moving forward in π_{GT}, and the smoother motion in π_{EF} is shown by the lower freuency in these actions and a more dominant frequency of action 16 which is forward motion without rotating the agent.

Policy π_{GT} also makes the agent position itself more towards its edges, with respect to the wire, and is hence at risk of hitting the wire more frequently. Contrarily, policy π_{EF} causes the agent position with respect to the wire to be more at the agent centre, and hence takes more cautious actions than policy π_{GT}.

Over the 50 evaluation wires, π_{EF} performs better in terms of number of wires completed as compared to π_{GT}. Table 4.3 shows the number of wires completed in the evaluation by each of these policies. π_{GT} performs considerably poor with only 7 wires completed among the 50 evaluation wires. In fact, evaluating π_{GT} on this environment with estimated feature observations is the direct coupling training phase. The poor performance shows the need for further training, with the DQN agent trained on the noisy observations due to the FENet. This forms the next training phase where the DQN agent is trained on estimated feature observations. Noise is artificially added to the observations based on the noise profile shown in Figure 4.5, to test the robustness of the DQN agent to more erroneous observations. It is observed that π_{GT} is unable to complete any evaluation wire (Table 4.3 column: Sine Noisy), and is hence not robust to noise in observations, whereas π_{EF} performs considerably better (38 wires completed). This is because the policy is obtained from training the agent on noisy observations, hence feature noise is a part of the dataset, unlike in π_{GT}.

	Environment with Estimated Features	
	Sine	Sine Noisy
π_{GT}	7	0
π_{EF}	45	38

Table 4.3: Performance over 50 sine wires expressed as number of wires completed.

	Environment with Estimated Features	
	Multisine	Multisine Noisy
π_{GT}	0	0
π_{EF}	30	20

Table 4.4: Performance over 50 multisine wires expressed as number of wires completed.

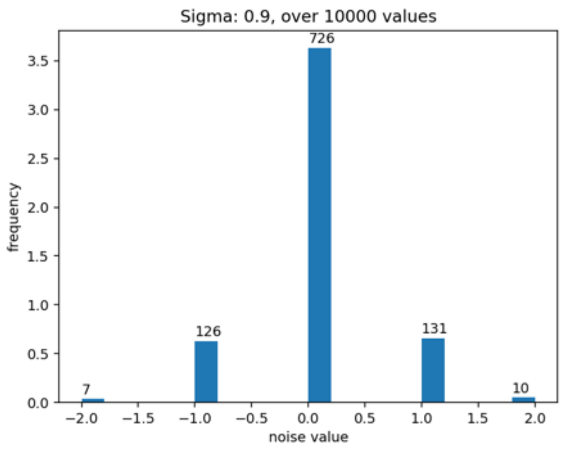


Figure 4.5: Noise profile used in inducing artificial noise.

The generalizability performance of the policies is also tested by evaluating them on multisinusoidal wires, that was not part of the training data. This multisine evaluation set also consists of 50 wires, where each wire is a sum of 10 sine waves with random amplitude, frequency, phase. The resulting multisine wave also has a random global orientation in 2D space. The results of these evaluation experiments are shown in Table 4.4. π_{GT} is unable to complete any of the multisine wires and hence has poor generalizability. π_{EF} on the other hand is able to complete a considerable number of multisine wires, even with artificial noise added to the observations. The following link shows an example of the agent completing a multisine wire under π_{EF}.

$\boxed{\pi_{EF} \text{ evaluation on multisine wire (click to view)}}$

Hence, it can be concluded that the direct coupling case does not provide an effective vision-based reinforcement learning controller in the 2D buzz-wire experiment and further training on noisy observations is necessary. Doing so results in a more robust and generalized policy that can navigate the 2D buzz-wire simulation well.

4.3.1 Noise in Images

So far, the robustness of the policies for feature noise have been discussed. This noise occurs after the feature extraction step of the FENet as seen in Figure 4.1. It is also interesting to observe how the overall vision system performs in case of noise occuring at the camera sensor, causing imperfect image input to the FENet. Many noise models exist for artificially inducing noise into images. In this project, we evaluate the vision-based controller on two particular noise models - Gaussian Blur and Motion Blur. Gaussian Blur simulates blurring of an image due to the camera being out of focus with respect to the region of interest and is characterized by a two dimensional normal gaussian distribution with equal variance in both axes. A bigger kernel results in a bigger variance, and hence more blurring. Motion Blur occurs due to relative motion of the object of interest and the camera. This kernel is characterized by two parameters, the degree of motion (speed) and the angle of motion. The degree dictates the positioning of the mean of the distribution, with a higher degree pushing the mean to the bottom right of the kernel matrix (top left if motion in reverse direction). The angle of motion defines how much the main diagonal of the matrix is rotated, which shows the relative angle of motion between the camera and the object. In the buzz wire experiment, the angle of the kernel is the same as the angle of motion of the agent, or its orientation.

These kernels are applied to the image that is input to the FENet at each step. The policy that is evaluated in this case is π_{EF} since π_{GT} is shown to be not very robust. The following videos show the evaluation of π_{EF} on the blurred image observations for the same wire.

π_{EF} evaluation on images with Gaussian Blur (click to view)

π_{EF} evaluation on images with Motion Blur (click to view)

The same set of sine and multisine evaluation wires as in the previous cases are tested with the blur kernels applied at the image generation stage. Table 4.5 shows the performance of π_{EF} for various gaussian kernels, and Table 4.6 shows the same for various motion kernels. It can be observed that as the degree of blurring increases, the policy is able to complete fewer wires. This is expected as the feature extraction capability of the network becomes poorer due to less distinctly visible features caused by the blurring.

Blur Kernel	Gauss 21x21	Gauss 31x31	Gauss 41x41
Sine	32	36	22
Multisine	12	15	4

Table 4.5: Performance of π_{EF} with Gaussian Blur Kernels expressed as number of wires completed.

Blur Kernel	Motion deg=5	Motion deg=10	Motion deg=20
Sine	34	17	31
Multisine	10	3	10

Table 4.6: Performance of π_{EF} with Motion Blur Kernels expressed as number of wires completed.

4.4 End-To-End RL Model

The previously discussed methods pertain to the vision-based controller with the feature engineering step as part of the design process. However, in case of more complex environments, where the images are cluttered with undesired objects or in higher dimensional simulation, defining such features and creating a dataset for extracting them is difficult. An end-to-end reinforcement learning model can eliminate this step of feature engineering and allow the agent to learn features that are important for completing the buzz wire task. The unified model is as shown in Figure 4.2. Constructing this agent is made by insight obtained from building the vision system and RL agent from the previous training cases. In fact, the convolutional weights in this network are initialised from those in the FENet.

4.4.1 Changes in Training Experiment

Computational Limitations

Several changes were required to be made for this training experiment. Since the agent now takes in observations that are images of size $256 \times 256 \times 3$ (Figure 4.2) as opposed to the 50×1 vector stored in the feature engineering case (Figure 4.1), the size of a single transition stored in the replay buffer is more than 3000 times larger. This imposes serious constraints on memory and size of the replay buffer, thus requiring it to be smaller. A smaller replay buffer leads to only the most recent transitions being stored and makes the training more temporally correlated since it cannot sample batches from past experiences. Due to the limited storage space available (16GB RAM), a maximum of 20000 transitions can be stored in the replay buffer when compared to the previous case where the capacity of the replay buffer was 10^7 transitions (without reaching maximum memory on 16GB RAM). Similarly, the batch size used to train the network was also reduced due to the same reasons from 256 examples to 16.

Additionally, with the previous reward formulation, the agent would hit the wire and the next steps up to the 400^{th} step are kept constant to give a large negative reward. This results in transitions that are repeated many times, leading to inefficient use of the replay buffer. Hence, a new scheme for providing a large negative reward on wire hit is required to be made, where a reward of -10000 is given on wire hit, and the episode is terminated. This leads to efficient use of the replay buffer where the dataset used for training the DQN is more diverse.

To further increase the size of the replay buffer, so that more history of the agent's trial and error experience can be stored, the images stored in the replay buffer were shrunk to 100×100 from 256×256, and a resize preprocessing layer was added to the above network to bring the dimensions of the image back to 256×256. With this the replay buffer size could be increased to 150000 transitions from the previously calculated 20000, and the batch size was increased from 16 to 64. While the effects of blurring caused by shrinking and resizing on the network has not been investigated like in the previous evaluation experiments with gaussian and motion blur, such an analysis may not be necessary since the network architecture of the current model is drastically different than the previous cases. The network is allowed to learn the features from image observations that are resized.

Network Architecture

The architecture of the new DQN agent needed to be changed since simply combining the FENet and the earlier DQN was not possible due to lack of support for branching networks (as in Figure 3.7) in the TF-Agents RL development toolkit [42]. This needed redefinition of some intermediate layers and the weights of the earlier networks could not be utilized due to this. Since a series of dense layers increases the number of parameters many-fold, and this causes slower rate of learning, the number of such layers in the new architecture was reduced. A simpler architecture as seen in Figure 4.6 was used where the weights of the convolutional layers were initialized to those of the trained FENet model and frozen during the new training process. The size of the hidden dense layers was reduced to just two layers since experiments with more layers showed very slow rate of convergence.

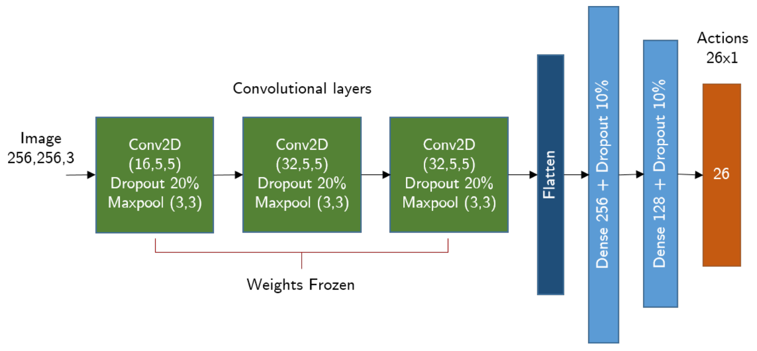


Figure 4.6: End-to-end DQN model for visual control.

Reward Scheme

The reward scheme was also changed to accommodate large negative reward on wire hit and an equally large positive reward on reaching the goal. It was observed empirically that this sparse reward scheme was insufficient to promote the motion of the agent along the forward direction of the wire, and the agent would stay in place and get zero reward. This behaviour is likely taken to overcome an immediate large negative reward by performing actions such that the wire is hit. The reason for such behaviour is likely because the large positive reward is well into the horizon from the starting point of the agent, and is hence diminished in the initial discounted returns. This scenario is also promoted by the fact that the replay buffer stores a smaller number of recent transitions and it is likely that in these transitions the agent makes actions where it has not progressed much along the wire. Thus, a denser reward scheme was required to be made that rewards forward motion of the agent towards the goal and punishes motion in the backward direction. This was made by using the arc length of the wire function and the agent's location on the wire. The reward scheme is then given by

$$r_k = \begin{cases} r_{\text{hit}} & \text{if agent hits wire} \\ r_{\text{finish}} & \text{if agent reaches finish point} \\ r_{progress} & \text{for progress along wire} \end{cases} \tag{4.1}$$

where $r_{hit} = -10000$, $r_{finish} = 10000$ and $r_{progress} = s \cdot \pm \sum_{i=1}^{n} \sqrt{1 + [f'(x_i)]^2}\Delta x_i$, where s is a scaling factor, n is the number of discrete wire elements traversed by the agent, x_i is the i^{th} element along the local x-axis of the wire, δx_i is the difference in the x distance between the i^{th} and $(i-1)^{th}$ wire element, and $f'(x)$ is the first derivative of the wave function used to represent the wire, and $scale = 100$. If the agent moves in the forward direction, towards the goal, this arc length is positive ($r_{progress} = s \times \sum(.)$) and is negative if it moves backwards, away from the goal ($r_{progress} = -s \times \sum(.)$). The drawback with such a reward is prior knowledge of the wave function/trajectory is required to calculate this arc length.

Action Space

To further improve the action selection mechanism, the action 13 in Table A.1 was removed from the action space in the end-to-end training phase. This prevents the agent's policy from being stuck in a scenario of inaction, where it may be so that the agent believes obtaining no reward by staying in place without moving is better than moving and hitting the wire. The final layer of the DQN is then only 26 actions accordingly.

4.4.2 Training Results For End-To-End Model

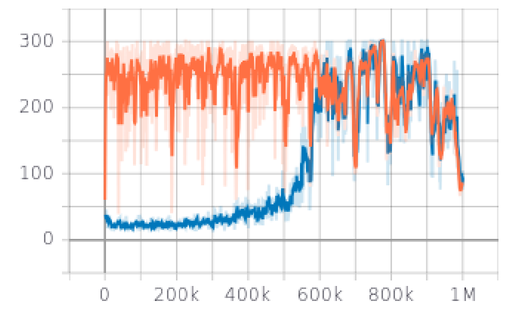
Table 4.7 shows the important hyperparameters used in generating the end-to-end policy. The starting value for ϵ (exploration parameter) is reduced to 0.5 to promote faster learning. A high value of ϵ results in a high probability of random actions chosen, which can result in slower learning of the DQN agent. The replay buffer capacity and batch size were reduced due to the reasons explained above. The experiment is conducted for a longer period of time with number of epochs at 10000 due to the slow learning progression.

The training progression for the end-to-end model training is shown in Figure 4.7. It is observed that the agent still does not receive positive average rewards at the end of 10000 epochs (Figure 4.7(b)). However, there is a steady trend in the maximum returns (Figure 4.7(c)) that show more and more positive rewards being obtained as the epochs progress. If the experiment is run for longer, it is likely that a policy is learnt that is able to complete the buzz wire. As the experiment currently stands, the best policy is still unable to complete the wire, but is able to make a turn as seen in the following evaluation video.

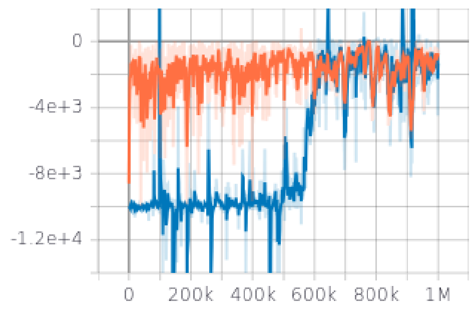
End-to-end policy evaluated on a sine wire (click to view)

Hyperparameter	Value
ϵ - Probability of random actions	$0.5 \rightarrow 0.0001$ (linear decay)
α - learning rate	$10^{-3} \rightarrow 10^{-6}$ (linear decay)
γ - Discount factor	0.4
Replay buffer capacity	$150,000$
Batch Size	64
Target network update frequency	50
Experience collection steps per epoch	2000
Training steps per epoch	100
Number of training epochs	10000
Evaluation examples	50
Number of training epochs after which evaluation occurs	10

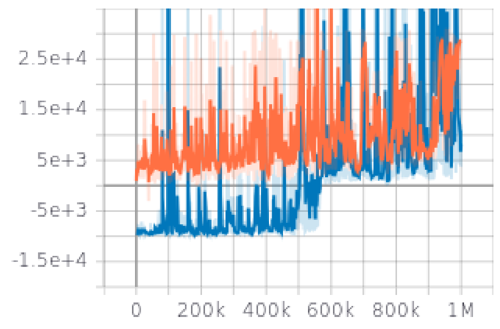
Table 4.7: Hyperparameters chosen for the training experiment in generating end-to-end model.



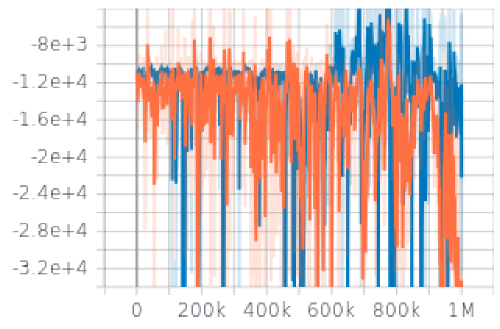
(a) Episode Length.



(b) Average Returns.



(c) Maximum Returns.



(d) Minimum Returns.

Figure 4.7: Training progression for end-to-end policy. Blue shows the training curve and orange shows the evaluation curve (plots from TensorBoard).

The progress of learning is still quite sluggish in spite of the improvements made in promoting the agent's forward motion. This is likely because of the smaller capacity of the replay buffer and the agent is not trained on data that is sufficiently in the past, i.e., catastrophic forgetting is likely occurring due to the dataset still being quite recent. More crucially, in the end-to-end case, the observations are of much higher dimension than in the previous training cases (50×1 feature vector previously, compared to $256 \times 256 \times 3$ here). The previous observations were also information rich, containing specific information like the agent position with respect to the wire, it's orientation, as well as the local goal to be reached (future point orientation). With the image observations, the DQN agent must learn the specific information on it's own and then make action decisions based on these learnt features. While the convolutional filters are initialised from the previously trained feature extractor network, much of the feature extraction capability is lost in removing the branched dense layers used for classification, and thus the agent must still spend a lot of effort to learn to extract relevant features.

Chapter 5

Conclusion and Recommendations

5.1 Conclusion

The objective of this project was to build a vision-based reinforcement learning controller for the 2D Buzz Wire simulation where the agent is able to navigate a given wire to the goal point without coming in contact with the wire. The two sub-tasks of extracting specific features from images and the reinforcement learning agent making action decisions based on these features have been discussed. The architecture of the neural networks in both cases has been explained in Chapter 3. It is observed that the feature extractor network shows reasonably good accuracy in estimating the selected features from a given local observation in the 2D environment. The DQN agent is able to navigate and complete a given wire, with a policy trained from ground truth observations. A policy trained with the feature extractor network output as the observations has been obtained and the two policies have been compared for their performance.

The performance of the policies have been discussed in the following terms:

- Number of wires completed in an evaluation wire set – The more number of such wires completed in the environment, the better the policy of the DQN agent.
- Robustness to noisy observations – The policy is considered robust if it is able to navigate considerable numbers of evaluation wires under noisy conditions, where observations are imperfect due to the accuracy of the FENet and externally added noise.
- Generalizability – Whether the policy is able to navigate a set of unseen multisine wires. A policy is more generalizable if it is able to complete more of these multisine wires.

Policies are compared with these factors and a better candidate for the 2D buzz-wire task is chosen.

Different training cases have been considered and the policies generated for the feature extraction training cases are compared. It was observed that the policy generated by training the agent on noisy observations due to the FENet (π_{EF}) performed better than the policy generated by training on ground-truth features (π_{GT}) in terms of robustness to external noise and generalizability to new wire shapes. Thus, π_{EF} is a better candidate policy for the 2D buzz-wire task for the given architecture of the agent and observer. This policy was also tested for the case of noisy images, with two noise models considered, and shows considerably good robustness in these cases.

An end-to-end model that makes action decisions based on image observations was constructed and a training experiment was carried out. Many challenges were faced in terms of efficiently handling replay buffer data since the observation size is now much larger than in previous cases, thus allowing only fewer and more temporally correlated examples to be stored and used for training the DQN agent. The effects of such a change have been discussed with respect to the slow training progression, and the policy generated is unable to complete the given example of the buzz wire. Changes in the reward scheme to promote the agent to move towards the goal have been discussed and show better results than with a sparse reward.

5.2 Recommendations and Future Work

While progress has been made in building a vision-based controller in the 2D Simulation, additional analyses in the same has to be made before moving to more complex environments.

1. In continuing with the developments of this project, constructing a unified end-to-end controller model for the 2D simulation environment is a natural first step. The problems faced during training such a model were mainly with respect to memory management, where the replay buffer capacity had to be reduced by many orders of magnitude. While obtaining better memory resources by increasing storage space may not be feasible for this problem, a deeper look into managing the replay buffer more efficiently by storing experience data in a more compact way can be beneficial.

2. The current problem of storing image data in the replay buffer stems from the fact that the image size used is too large. Smaller image observation sizes should be explored and their effect on the performance of the vision-based controller with feature engineering must be investigated and compared with the performance of the end-to-end model.

3. The Deep-Q learning algorithm provides a simple method for extending Q-learning into larger observation and action spaces. However, it is also data hungry, requiring large replay buffer sizes that store historical data from many environment steps in the past. Sample efficient reinforcement learning must be explored in literature that can overcome the need for such large replay buffer sizes, and also for faster convergence of the policy. Model-based RL algorithms or Model-free RL with Self-Predictive Representations [44] can provide such alternatives.

4. From the insight gained in the 2D simulation environment, the 3D environment can be used to develop a controller for the buzz-wire task. The 3D platform can be used to create a very close representation of the real world buzz wire setup, with the robot arm included. However, constructing such an environment with a large number of example wires for the robot arm to train on can become difficult since modelling these wires in a 3D modelling software can become quite cumbersome. A method to automate this process for generating randomly shaped wires is necessary to obtain a generalizable control policy for the robot arm. While this can be overcome by sequentially training the arm to complete a specific wire shape before moving to the next, the random wire generation is necessary to reduce experiment time in obtaining such a generalized policy.

5. An attempt has been made into creating the 3D environment using Pybullet as the physics and rendering engine. It was observed that some objects, especially the wire object, did not scale in the same way as other objects. This caused problems in defining the trajectory of the wire, in the offsets required in placing the robot arm near the wire, and in placing the loop around the wire in an accurate manner. Extreme care should be taken in making such offsets, since it results in calculating ground truth features like distance between the loop and the wire, and future local goal point considerations.

6. Finally, insight from the 2D simulation in constructing vision and RL models can be utilized in constructing the same for the 3D environment. Transfer learning must first be explored by applying the same pipeline in the new environment and the simulation gap between the 2D and 3D environments must be judged.

Appendix A

Actions and Indices

List of agent actions in x, y and θ degrees of freedom, and corresponding indices in the output of the Deep-Q Network.

	Degree of Freedom		
Action Index	x	y	θ
0	-1	-1	-1
1	-1	-1	0
2	-1	-1	1
3	0	-1	-1
4	0	-1	0
5	0	-1	1
6	1	-1	-1
7	1	-1	0
8	1	-1	1
9	-1	0	-1
10	-1	0	0
11	-1	0	1
12	0	0	-1
13	0	0	0
14	0	0	1
15	1	0	-1
16	1	0	0
17	1	0	1
18	-1	1	-1
19	-1	1	0
20	-1	1	1
21	0	1	-1
22	0	1	0
23	0	1	1
24	1	1	-1
25	1	1	0
26	1	1	1

Table A.1: Agent actions in x, y, and θ.

Bibliography

[1] D. Nguyen-Tuong and J. Peters, “Model learning for robot control: a survey,” *Cognitive processing*, vol. 12, no. 4, pp. 319–340, 2011.

[2] R. Bars, P. Colaneri, L. Dugard, F. Allgöwer, A. Kleimenov, and C. Scherer, “Trends in theory of control system design status report prepared by the ifac coordinating committee on design methods,” *IFAC Proceedings Volumes*, vol. 41, no. 2, pp. 2144–2155, 2008. 17th IFAC World Congress.

[3] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[4] J. Sarangapani, *Neural network control of nonlinear discrete-time systems.* CRC press, 2018.

[5] 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,” 2018.

[6] “The rubion fruit picking robot by octinion.” `http://octinion.com/products/agricultural-robotics/rubion`. Accessed: 2021-08-23.

[7] K. Narendra and K. Parthasarathy, “Identification and control of dynamical systems using neural networks,” *IEEE Transactions on Neural Networks*, vol. 1, no. 1, pp. 4–27, 1990.

[8] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction.* Cambridge, MA, USA: A Bradford Book, 2018.

[9] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, “Deep reinforcement learning: A brief survey,” *IEEE Signal Processing Magazine*, vol. 34, p. 26–38, Nov 2017.

[10] 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,” *The International Journal of Robotics Research*, vol. 40, p. 698–721, Jan 2021.

[11] G. Dulac-Arnold, D. Mankowitz, and T. Hester, “Challenges of real-world reinforcement learning,” 2019.

[12] D. Silver, S. Singh, D. Precup, and R. S. Sutton, “Reward is enough,” *Artificial Intelligence*, vol. 299, p. 103535, 2021.

[13] L. Buşoniu, T. de Bruin, D. Tolić, J. Kober, and I. Palunko, “Reinforcement learning for control: Performance, stability, and deep approximators,” *Annual Reviews in Control*, vol. 46, pp. 8 – 28, 2018.

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, “Playing atari with deep reinforcement learning,” *CoRR*, vol. abs/1312.5602, 2013.

[15] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, “Continuous control with deep reinforcement learning,” 2019.

[16] A. Franceschetti, E. Tosello, N. Castaman, and S. Ghidoni, “Robotic arm control and task training through deep reinforcement learning,” 2020.

[17] P. Zapotezny-Anderson and C. Lehnert, “Towards active robotic vision in agriculture: A deep learning approach to visual servoing in occluded and unstructured protected cropping environments,” *IFAC-PapersOnLine*, vol. 52, no. 30, pp. 120–125, 2019. 6th IFAC Conference on Sensing, Control and Automation Technologies for Agriculture AGRICONTROL 2019.

[18] J. Ruiz-del-Solar, P. Loncomilla, and N. Soto, “A survey on deep learning methods for robot vision,” *CoRR*, vol. abs/1803.10862, 2018.

[19] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation,” *CoRR*, vol. abs/1311.2524, 2013.

[20] A. S. Razavian, H. Azizpour, J. Sullivan, and S. Carlsson, “CNN features off-the-shelf: an astounding baseline for recognition,” *CoRR*, vol. abs/1403.6382, 2014.

[21] W. Rawat and Z. Wang, “Deep convolutional neural networks for image classification: A comprehensive review,” *Neural Comput.*, vol. 29, no. 9, 2017.

[22] P. Corke, *Robotics, Vision and Control: Fundamental Algorithms in MATLAB.* Springer Publishing Company, Incorporated, 2nd ed., 2017.

[23] F. Zhang, J. Leitner, M. Milford, B. Upcroft, and P. Corke, “Towards vision-based deep reinforcement learning for robotic motion control,” 2015.

[24] S. Levine, C. Finn, T. Darrell, and P. Abbeel, “End-to-end training of deep visuomotor policies,” 2016.

[25] S. Gu, E. Holly, T. Lillicrap, and S. Levine, “Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates,” 2016.

[26] S. Levine, P. Pastor, A. Krizhevsky, and D. Quillen, “Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection,” 2016.

[27] “Make your own buzz wire game.” `https://www.youtube.com/watch?v=6fmCjeGLe9k&ab_channel=WMDIY`. Accessed: 2021-08-23.

[28] “Niryo one documentation,” Jul 2020. https://niryo.com/docs/niryo-one/.

[29] S. Marx, A. Kanso, and R. Müller, “Unconventional path planning for a serial kinematics robot with reinforcement learning using the example of the wire loop game,” 07 2020.

[30] R. Meyes, H. Tercan, S. Roggendorf, T. Thiele, C. Büscher, M. Obdenbusch, C. Brecher, S. Jeschke, and T. Meisen, “Motion planning for industrial robots using reinforcement learning,” *Procedia CIRP*, vol. 63, pp. 107 – 112, 2017.

[31] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning Representations by Back-Propagating Errors.* Cambridge, MA, USA: MIT Press, 1988.

[32] G. Cybenko, “Approximation by superpositions of a sigmoidal function,” *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

[33] “Learn reinforcement learning (3) - dqn improvement and deep sarsa,” Jul 2019. https://greentec.github.io/reinforcement-learning-third-en/.

[34] H. van Hasselt, A. Guez, and D. Silver, “Deep reinforcement learning with double q-learning,” 2015.

[35] L. Lin, “Reinforcement learning for robots using neural networks,” 1992.

[36] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, “Human-level control through deep reinforcement learning,” *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[37] "Convolutional neural network." https://www.mathworks.com/discovery/convolutional-neural-network-matlab.html. Accessed: 2021-08-29.

[38] Y. LeCun, L. Jackel, L. Bottou, A. Brunot, C. Cortes, J. Denker, H. Drucker, I. Guyon, U. Muller, E. Sackinger, *et al.*, "Comparison of learning algorithms for handwritten digit recognition," in *International conference on artificial neural networks*, vol. 60, pp. 53–60, Perth, Australia, 1995.

[39] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, (Red Hook, NY, USA), p. 1097–1105, Curran Associates Inc., 2012.

[40] R. Dorussen, "Learning how to solve the buzz-wire game with a robot arm," msc thesis report, Eindhoven University of Technology, 2021.

[41] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[42] S. Guadarrama, A. Korattikara, O. Ramirez, P. Castro, E. Holly, S. Fishman, K. Wang, E. Gonina, N. Wu, E. Kokiopoulou, L. Sbaiz, J. Smith, G. Bartók, J. Berent, C. Harris, V. Vanhoucke, and E. Brevdo, "TF-Agents: A library for reinforcement learning in tensorflow." https://github.com/tensorflow/agents, 2018. [Online; accessed 25-June-2019].

[43] T. B. Sriram, "Machine learning based vision-in-the-loop system for automated buzz wire demonstrator," internship report, Eindhoven University of Technology, 2020.

[44] M. Schwarzer, A. Anand, R. Goel, R. D. Hjelm, A. Courville, and P. Bachman, "Data-efficient reinforcement learning with self-predictive representations," 2021.