A Bio-Inspired System for Human-Like Contour Following

ALFRED SYDNEY BROWN
Supervisor: Prof. Nathan Lepora

Department of Engineering Mathematics
UNIVERSITY OF BRISTOL

A dissertation submitted in support of the degree of MASTER IN ENGINEERING.

APRIL 2020

Word count: 12255

Abstract

In this paper a bio-inspired system is presented that mimics the interaction between a human's brain, arm and fingers in order to trace (follow) the contours of unknown objects in its environment. A dendritic neural network [1], a UR5 robotic arm [2], and a biomimetic tactile sensor (the TacTip [3]) were used to model a human brain, arm and finger, respectively. The interactions between brain (dendritic neural network), arm (robotic arm) and finger (TacTip) was governed by a pose-based servo (PBTS) control policy [4]. The bio-inspired system was tested on four different objects, and the results show that the system was able to trace the contours of these objects robustly. A system such as this is a step towards anthropomorphizing robotics and could aid in the development and usage of human-like robots in daily life and large scale manufacture.

Additional Resources:

Link to contour following and data collection videos and images: `https://photos.app.goo.gl/fRMGJczfm1vjRdXF7`

Link to code used throughout the project: `https://uob-my.sharepoint.com/:f:/g/personal/ab16230_bristol_ac_uk/Eu3uDPQYK4FIjz5lB41s300Bcl1BszVGttr9Kph8aWPWfA?e=DQOZKy`

Table of Contents

1. Introduction

1.1 A General Overview

The human tactile sense, the ability to feel different textures, pressures, and temperatures through our skin enables us to explore and manipulate our surroundings [4, 5]. To illustrate this, picture a man trying get his car keys out of his pocket; he uses his fingers to navigate down one side of his hip and into his pocket in order to retrieve his car keys. Specifically, when his fingers move along one side of his hips and into his trouser pocket, tiny indentations in the skin of his fingers and hand cause electrical signals to be sent to his central nervous system, where his brain decodes these signals into knowledge of the whereabouts of his fingers relative to his body. If the indentations of his skin deform in such a way that his brain recognises these as coming from his car keys, he knows that he has achieved his goal and can retrieve his keys.

Our tactile sense and dexterity has helped enable us to build the world we see around us today. All of our technology stems ultimately from tools put together by our hands [4], and most manual labour involves using our arms, hands and fingers in one way or another. An artificial tactile sense and the ability to manipulate and explore the surroundings will enable robots to perform manual tasks currently needing human labour, which, for better or worse, has been the goal of tactile robotics for half a century [4, 6]. The development of effective systems for an artificial sense of touch is important for future technology: human-like tactile sense and dexterity will be necessary for advanced manufacturing (e.g. entirely autonomous assembly lines and quality control checks), assisted living (e.g. robots to look after the elderly), food production (e.g. picking and sorting delicate foods) and healthcare (e.g. nursing and surgical robots) [7, 8]. Although robots are increasingly replacing humans in industries that rely on repetitive manual labour [9, 10, 11], most robots have poor tactile capabilities because no one knows how to combine tactile perception and control of action to give a robot a robust sense of touch [7].

Yet, the problem of active tactile perception, in order to explore and manipulate our surroundings, has been solved by the human brain, its arms, hands and fingers. It therefore seems only natural to look to human biology in order solve this problem artificially. Consequently, we use a dendritic neural network, a robotic arm, a tactile sensor and a control policy that connects the network to the tactile sensor, via the robotic arm, to mimic the human system between ‘brain’ and ‘periphery’.

In this study, we focus on the interaction between brain and finger, in order to explore and follow the contours of random objects. Specifically, we design this bio-inspired artificial system in order to perform 2-dimensional ‘contour following tasks’. These tasks are analogous to a human tracing their finger along an edge of an object: similar to the path traced out by the fingers of the man trying to find his car keys.

1.2 Technical Objectives

In technical terms, the main objective of this project was to design a dendritic neural network (DenNN) that worked effectively with 2D pose-based tactile servo (PBTS) control to perform contour following tasks. Dendritic neural networks [1], having only entered the literature in 2018, are a recent alternative to many other bio-inspired neural networks, and their capabilities and limitations are as of now, still fairly unknown. By exploring dendritic neural networks in this setting we hoped to gain insights into the practicality of these networks for artificial tactile sensing and control, and learn more about the network itself. Specifically, we aim to

- build an energy-based learning framework for dendritic neural networks in the Python programming language using equilibrium propagation,
- train and validate three different single-hidden-layer dendritic neural networks,
- choose the best performing network and combine it with 2D PBTS control in order to perform contour following tasks around four different objects.

1.3 Project Plan

The technical report is set-up to have five different chapters, after the introduction:

- The first chapter gives biological and contextual background to the different components of the bio-inspired system: dendritic neural networks, tactile sensors, PBTS control and robotic contour following.
- The second chapter explains how dendritic neural networks work, how they differ from artificial neural networks, and how equilibrium propagation [12] can be used to provide a bio-inspired learning framework for dendritic neural networks in order to model tactile data.
- The third chapter goes into detail as to how tactile data was collected and processed in order to train three different dendritic neural networks to perform contour following tasks.
- The fourth chapter explains how the 2D PBTS control policy works, and presents the contour following results.
- In the final chapter, the results are discussed and ideas for further work are presented.

The report is written with the assumption that the reader may not have any prior-knowledge of neural networks or control theory, and therefore tries to be as self-contained as possible.

1.4 The Novelty of this Work

While other robotic contour following systems exist [4, 13, 14], this is the first time that a dendritic neural network has been used to model tactile data. Our bio-inspired system of human neural processing is the most biologically realistic and human-like system to have been developed for robotic contour following to this day. Furthermore, the dendritic neural networks created in this paper were used in combination with equilibrium propagation [12], which, that we are aware of, is original in the field of neural networks.

2. Background

2.1 Tactile Sensors

Touch is one of the five senses that humans possess. It is the first sense we develop [15], and occurs across the entire body - from the soles of our feet to the top of our scalps. It is considered a multi-modal sense given the different modes of sensation our bodies can experience through touch, be it changes in temperature, pressure, pain or pleasure. There is not a single sense organ for touch, but instead a multitude, each with a particular task. As Aristotle once wrote, touch does not have a single "proper sensible" [15]. What he meant by this is that touch does not have a single sensory feature that is only available in a single modality. Colour, for example, is the proper sensible for vision [16]. Touch, or more formally *somatic sensibility*, has four major modalities: discriminative touch (required to recognise the size, shape, and texture of objects and their movement across the skin), proprioception (the sense of static position and movement of the limbs and body), nociception (the signaling of tissue damage or chemical irritation, typically perceived as pain or itch), and temperature sense (warmth and cold) [17]. The terms "haptic" or "active" touch are often used to combine both discriminative touch and proprioception, implying an engagement of movement, and it is this type of touch sensation that this report focuses on.

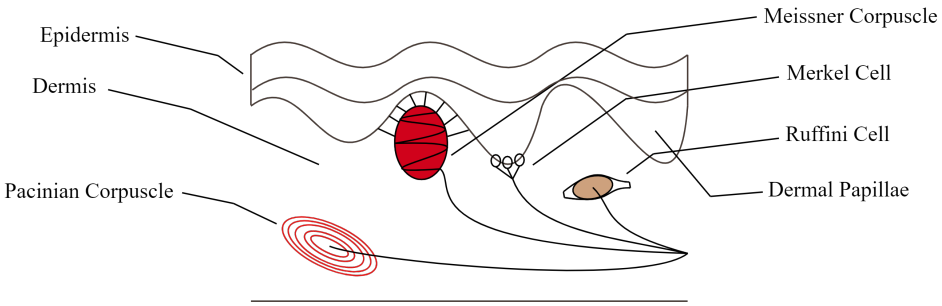


Figure 2.1: A simple schematic of the different receptor organs found in the dermis, below the surface of the skin. Meissner Corpuscles fire action potentials when the dermal papillae move apart or come closer together, allowing for even the subtlest of indentations of the skin to be detected.

It is the incredible diversity in the perception of touch that has motivated in-depth study of this sense. Our tactile sensitivity is greatest on the glabrous (hairless) skin on the fingers, the palm, the sole of the foot and the lips [17]. Touch is mediated by mechanoreceptors that lie below the surface of the skin - the dermis. There are five mechanoreceptors for glabrous skin that each end in four different organs: Merkel receptors, Meissner Corpuscles, Merkel Cells, Pacinian Corpuscles, and Ruffini Cells, each with different purposes [18]. The Meissner Corpuscles lie between epidermal papillae, which are ridge-like structures that form the boundary between the epidermis (skin surface) and dermis. On our fingertips, the ridges are arranged in circular patterns, propagating our individual fingerprints into the dermis [17]. When an object comes into contact with our skin, these papillae displace, moving either closer to neighbouring papillae or further apart. Consequently, collagen fibres that are connected to laminar cells in Meissner Corpuscles, compress or stretch,

causing a sequence of action potentials to fire down the finger, up the arm and into the peripheral nervous system [18]. The peripheral nervous system then further sends these encoded signals to the central nervous system where our brain decode them as touch [19].

Especially in robotics, the ability to mimic human touch has been highly sought after. While robots have been able to grasp objects since the 1950s (like the first industrial robot *Unimate*) [20], it is still an open research challenge for robots to tell how tight their grasp is or whether an object is slipping out of their grippers. It requires a sense of touch to be able to do this - an engineering problem, which to this day, has not been solved to the extent to which humans and animals are able to interpret touch. The diversity of sensors that the human finger possesses, the ability to sense temperature differences, extremely fine textures and shapes, amongst other things, are not an easy thing to replicate mechanically. Nevertheless, rapid developments have been made in recent years, most of which focus on a single sensor that conveys touch-like information. Researchers at the University of Bristol turned their attention to the biology of the human finger, in particular to dermal papillae (described above) in order to create an artificial tactile sensor. The following section will describe their novel approach in detail, as this tactile sensor was used for this study.

The TacTip - A Biomimetic Tactile Sensor

Inspired by the work of Takahashi-Iwanaga and Shimoda on the functional morphology of dermal papillae and other biologically inspired tactile sensors, Bristol researchers Chorley et al. designed a novel tactile sensor - The TacTip - in 2009 [3]. The TacTip uses a camera that captures displaced pins on a flexible rubber hemisphere. The pins mimic dermal papillae and perturb even at the slightest indentations of the rubber skin.

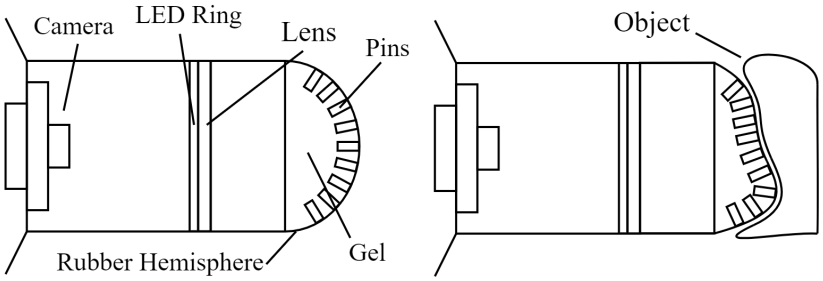


Figure 2.2: The TacTip was developed in the Bristol Robotics Laboratory in 2009 and mimics the behaviour of dermal papillae found underneath human/animal skin. **Left:** A simple schematic showing the basic design of the TacTip. **Right**: When a TacTip comes into contact with an object, the pin displacements are captured by a camera which allows for interpretation of touch in post analysis. To see the exact TacTip used in this paper, please click `here`.

In experiments performed half a decade ago, the TacTip showed great potential in edge detection and edge-encoding of surfaces, and worked at a level on a par with human subjects in active exploration tasks, whereby raised perimeters of different planar objects had to be tracked [22]. The TacTip also showed promising use for industry as a quality control sensor [23] and slip detector [24]. While the sensor lacks the ability to sense changes in temperature like a human finger, it is robust,

cheap and easily adaptable, and can be used in a number of challenges in industrial robotics, medical robotics, and robotic manipulation [25].

Various extensions of the TacTip were also explored (TacTip-GR2, TacTip-M2, TacCylinder), and the TacTip has undergone various subtle hardware upgrades, with the advances in 3D printing technology, for ease of manufacture and affordability [25]. For this study, the most recent TacTip was used. This consisted of 127 pins with white tips in a hexagonal pattern on the inside of a TangoBlack+ rubber skin hemisphere, a silicon gel between an acrylic lens and the rubber hemisphere, and an ELP USB camera module which ran at 120 frames per second.

Other Tactile Sensors

The TacTip is not the only tactile sensor to have been developed [26]. Most tactile sensors comprise of compliant elements and rely on a variety of underlying technologies, such as strain gauges [27], barometric sensors [28], capacitors [29], piezoresistive material [30], or piezoelectric films [31], to convey tactile information. A single capacitor or piece of piezoresistive material, alone, does not offer any positional information as to where the force is being applied, so individual sensors are often tiled together in order to form a whole tactile sensor. A disadvantage of tiling together individual sensors is that this can lead to overly complicated set-ups, making them expensive and hard to manufacture: examples include the TakkTile [32] and BioTac [33] sensors.

Another issue in tactile sensing is spatial resolution and determining the distribution of force, which is why some developers chose to include optical sensors within the tactile sensors. The TacTip, Gelsight [34], and GelForce [35] are all examples of tactile sensors that use optical sensors. They all achieve high spatial resolution, while remaining cheap to manufacture [36, 37, 38]. A downside to having optical sensors is that they limit usable space to have a variety of sensors; changes in temperature, for example, cannot be interpreted. However, these optical-based tactile sensors allow for both recognition of the magnitude of force and the distribution of force, which makes them ideal for solving active tactile exploration tasks like contour following, where a sense of temperature is not needed. While the Gelforce and Gelsight, in principle could be used for contour following tasks, neither are designed to be used as an end effector to a robotic arm (thereby mimicking the connection between arm, hand and finger), which is why the TacTip was chosen for this study.

Tactile Sensors in Industry

Already, tactile sensors have many uses in industry [39]. In the automobile industry, for example, pressure distributions on brakes, clutches, door handles, etc. are often determined using tactile sensors. For medical examinations, tactile sensors can be used to produce images of pressure patterns within soft-tissues and detect cancers and lesions [40, 41]. In robotics, for large scale manufacture, the tactile sensors are used as end-effectors to robotic arms to grip and manoeuvre delicate objects [42].

2.2 Dendritic Neural Networks

Contextualising Dendritic Neural Networks

Artificial neural networks were first proposed in 1943, by McCulloch and Pitts who took inspiration from research in neuroscience. It was proposed that neurons act like gates: they take a set of incoming weighted signals as an input, summate these, and then, depending on whether the sum surpasses a certain threshold (activation function), output another set of weighted signals. Multiple such neurons could then be combined together to form networks with arbitrary depth, inputs and outputs. With the implementation of backpropagation, developed by Werbos and Rumelhart et. al, thirty years later, these artificial neural networks were able to be trained effectively by providing an efficient means of computing how each node (neuron) changes with respect to other nodes further forward in a multi-layer network. With the advancements made in computing, neural networks showed promising applications for machine learning, and the field took off with research dedicated to optimisers, conjugate gradients, optimising initial weights and so forth. These advancements, however, often had nothing to do with the neurobiology of brains. They were purely mathematical means by which to reduce computation costs or speed up optimisation processes. It still remains an open question (often called the synaptic credit problem) as to how the human brain, and mammalian brains in general, optimise their synaptic "weights" to learn.

However, recent studies have shown that deep artificial neural networks mimic, extremely closely in some cases, the representational transformations in primate perceptual systems, and when trained in tasks similar to those solved by animals, have shown similar behavioural and neurophysiological phenomena to our biological brains [44]. This raised the question as to whether mammalian brains do indeed implement something similar to backpropagation in order to learn. While we know that backpropagation is not biologically feasible (reasons for which will be made apparent further on), something similar may be occurring in our brains. This led to computational neuroscientists proposing models of the brain that are built on similar principles to artificial neural networks, yet have added biological plausibility: dendritic neural networks are a recent addition to such proposed models.

The Biology behind Dendritic Neural Networks

Dendritic neural networks are networks based on the neurobiological structures and interactions between pyramidal cells and interneurons within mammalian cortical microcircuits [1]. They propose that the discrepancy (or error) between a desired network output and an actual network output, can be found to propagate backwards **locally** through apical dendrites of pyramidal cells, and thereby induce learning.

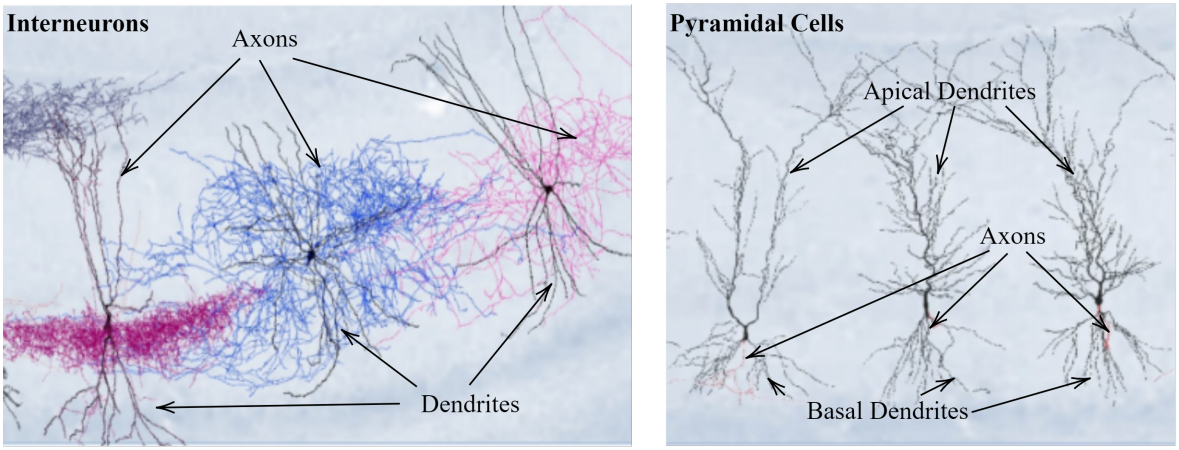


Figure 2.3: 3D reconstructions of CA1 cells in a rat's hippocampus (Source: [45]). (*Left*) **Interneurons**, left to right: axo-axonic cell, bistratified cell, basket cell (dendrites in black, axons in various colours). (*Right*) **Pyramidal Cells**: Apical and Basal (dendrites in black, axons in faint red).

Pyramidal cells are found in various parts of the brain. They consist of a pyramidal-shaped cell body (hence the name), apical and basal dendrites and a highly branched axon. The apical and basal dendrites receive "top-down", and "bottom-up" signals, respectively, from other neurons. These signals then propagate through the cell body (soma) and out of the axon branches. Interneurons are a broad class of neurons (three of which are shown above) found in the brain and throughout the central nervous system. Their primary function is to relay signals between other neurons.

Sacramento et al. suggest that pyramidal cells relay signals to other pyramidal cells in different layers of the brain, and that, via interneurons, pyramidal cells relay signals between themselves within a brain layer, in order for the brain to learn. They showed that a model (the dendritic neural network) of such a biological system can be used for machine learning tasks, such as the MNIST number classification task, achieving accuracies of 97% [1]. Furthermore, they showed that this framework allows for an approximation of the backpropagation algorithm in a localised manner. It is an exciting proposition, and one of the reasons why this neural network was chosen for this task.

Other Bio-Inspired Neural Networks

It is worth mentioning that other computational neuroscientists have also developed other biologically plausible neural networks. For example, Balduzzi et al. proposed an algorithm called 'Kickback' that was based on Spike-Timing Dependent Plasticity (STDP) and the Spike Response Model (SRM) [46] - which are both models used, respectively, to describe synaptic plasticity and action potential generation by neurons. Other examples include random feedback weights with homeostasis, Generalised Recirculation (GR) [47], and spike based Hebbian Learning Rules (HLR)[48].

Similar to dendritic neural networks, which propose that backpropagation-like behaviour is found in apical dendrites, other ideas on potential implementations of backpropagation have been proposed too. Lillicrap et al., for example, suggested that random synaptic feedback weights support backpropagation [49]. They show that even by propagating error backward through a fixed random matrix during training, an artificial neural network can learn effectively. Their work is beneficial in that it shows that neuron-specific teaching signals can enhance learning. However, their work also assumes that the error is not necessarily propagated locally, and in a single computational circuit, which is likely not biologically plausible [50]. Dendritic neural networks allow for localised error propagation and learning with single computational circuit, which is why they were chosen for this study.

2.3 Contour Following Tasks and PBTS Control

Contour following tasks are analogous to robotic vision tasks but for touch [13]. These allow a robot to 'feel' its way around objects or move along certain paths by only responding to the environment that surrounds it. Contour following tasks are the robot analogue of tracing our fingertips over objects [4]; a very human thing to do. The field is still in its infancy, but reliable contour following and tactile manipulation could result in increased use of robotics in industries that currently still rely on human dexterity; quality control in the aerospace and car manufacture industry, to name a few [23].

Previous Work

Work in robotic tactile contour following began thirty years ago [51, 52], with a more recent approach applying a control framework [53]. In 2017, the TacTip was used for the first time, in combination with a probabilistic model and a proportional gain controller, to perform contour following tasks [14]. They applied pin tracking[1] on the pins inside the TacTip for edge perception, allowing them to compute where the TacTip was relative to an edge. Since then, the TacTip has been used with various other models and servo control policies. For example, two deep convolutional neural networks were used on images captured by the camera inside the TacTip, in combination with the 2017-developed proportional gain controller [13]. These deep convolutional neural networks gave robust edge perception, and were, in combination with the proportional controller, able to effectively follow the contours of unknown objects. While the networks showed to be ideal for modelling tactile edge perception, the control policy needed improving. Recently, a new control policy was developed that used both a proportional and integral controller allowing for contour following of 2D edges, 3D surfaces and 3D edges [4]. Their control policy was termed pose-based tactile servo (PBTS) control. In their paper, the control policy was used in combination with a deep convolutional network (PoseNet) and showed effective contour following capabilities, which is why we adjusted it for this study.

[1] Pin tracking is explained in depth in the 'Training and Validation of Dendritic Neural Networks' chapter, but it essentially just means that pins inside the TacTip are tracked/followed and their coordinates are recorded, allowing for pin displacements to be determined, thereby conveying tactile information.

3. Modelling Touch Perception Using Dendritic Neural Networks

In this chapter we explore the mathematical mechanisms behind dendritic neural networks. The key purpose of this chapter is to understand that dendritic neural networks behave very similarly to artificial neural networks, but also allow for local error representation. Furthermore, we show why artificial neural networks, and in particular backpropagation, are not considered biologically plausible by neuroscientists, thereby justifying the choice of using dendritic neural networks. Lastly, the training methodology for dendritic neural networks, using equilibrium propagation, is explained.

3.1 Terminology and Concepts of Fully Connected Artificial Neural Networks

In order to better understand dendritic neural networks, the mathematics and terminology of fully connected artificial neural networks (ANNs) will be presented first. An artificial neural network is, in simple terms, just a highly non-linear composite function. Like all other models it has parameters, inputs and outputs; however, unlike most classical models, the number of parameters can easily reach into the thousands and millions, sometimes even billions. These parameters need to be "learned" by "training" the network with data. Once the parameters are learned the network can formulate accurate predictions given its inputs - thereby encapsulating the often extremely complicated relationship between inputs and outputs.

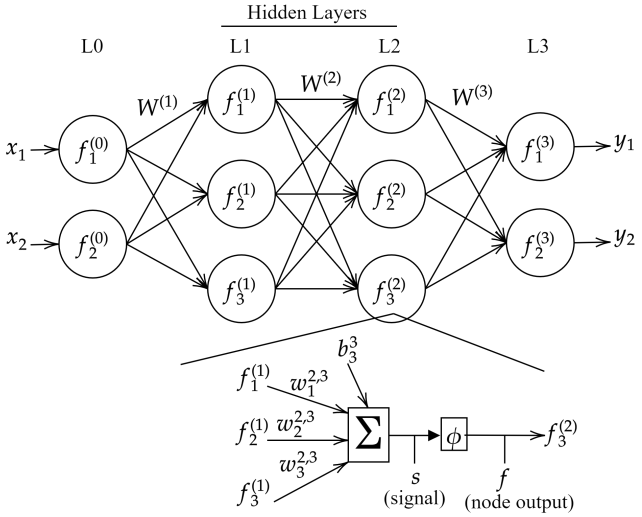


Figure 3.1: An example of a four layer neural network, to help visualise the different terminology described in this subsection. The computation inside a node/neuron is also shown. Notice that there are additional biases for each neuron in the hidden layers, that are omitted in typical diagrams like this.

Terminology of ANNs

ANNs consist of nodes within layers, connected to other nodes via weighted edges to form a network. All nodes in a hidden layer (that is a layer that is not the first nor the last layer) act as functions.

Let the variable $f_j^{(l)}$ represent the output of node j in the l^{th} layer, and $f_i^{(l-1)}$ the output of node i in the $(l-1)^{th}$ layer. An edge between two nodes visualises the operation of multiplying the output $f_i^{(l-1)}$ by a weight w_{ij}^l, where the subscript ij indicates that this weight connects the i^{th} neuron in the $(l-1)^{th}$ layer to the j^{th} neuron in the l^{th} layer. The input into the nodes of the first layer will be represented by the vector $\mathbf{x}$ and the output of the nodes in the last layer will be represented by the vector $\mathbf{y}$. Note that $f_j^{(N)} = y_j$ and $f_j^{(0)} = x_j$, where $N+1$ is the number of layers in the network (because we are starting at $N = 0$ for the first layer).

The *signal* or *activation* $s_j^{(l)}$ represents the weighted sum of inputs into node j in layer l coming from nodes in layer $l-1$, and the bias of the node j. These signals are passed through an *activation function* ϕ. The output of a node $f_j^{(l)}$ is thus,

$$f_j^{(l)} = \phi\left(s_j^{(l)}\right) = \phi\left(\left(\sum_{i=1}^{d(l-1)} w_{ij}^{(l)} f_i^{(l-1)}\right) + b_j^{(l)}\right), \tag{3.1}$$

where d is the depth (number of nodes in a layer) and $b_j^{(l)}$ is a bias associated with the j^{th} neuron in the l^{th} layer.

We define the desired outputs (*targets*) of a neural network with the vector $\mathbf{t}$. We refer to the *training* of a neural network as indicating that it is learning to adjust its weights so that its outputs start to align with the desired target values. In other words, we feed the network sets of input data and compare the outputs it generates to those of the target values and if these do not match well, then we readjust the weights in the network, and continue doing so until the output values match those of the target values as best as possible, or up to a certain stopping criterion. To do this we first forward propagate the inputs to get an output and then *backpropagate the error* between the outputs and targets, to adjust the weights. The rate at which the weights are adjusted is referred to as the *learning rate*.

In order to adjust the weights appropriately the network aims to minimise an *objective function* J (which is also sometimes referred to as the cost or energy function). This objective function encapsulates the error or cost between the outputs of the network and the target values. More generally, J is a mapping from the input configurations and weights relative to the desired output to a positive real number. The aim of the weight update is to minimise the objective function as this ensures that the outputs are matching the desired targets better.

3.2 Learning in an ANN

Once the input has been fed forward through the network (using Equation 3.1 for all nodes and layers) and output vector $\mathbf{y}$ is achieved, we can compare this output with the desired output vector $\mathbf{t}$. It should be noted here that $\mathbf{y}$ and $\mathbf{t}$ come in different forms depending on the task the neural

network is aiming to solve. For classification tasks, for example, the targets $\mathbf{t}$ are set to be the desired classes, and the so-called *SoftMax* function is often used to turn the output of the network into probabilities:

$$y_j = f_j^{(N)} = \frac{e^{s_j^{(N)}}}{\sum_{j=1}^{d(N)} e^{s_j^{(N)}}}. \tag{3.2}$$

Each node in the output layer corresponds to a particular class, and a prediction for a class is generated by simply selecting the output node with the highest probability, so learning can occur.

For regression tasks, the output vector $\mathbf{y}$ is often generated by using a linear activation function:

$$y_j = f_j^{(N)} = c s_j^{(N)}, \tag{3.3}$$

as this allows for a prediction of real numbers (c is just a constant). Note that if c is set to 1, this is the same as not applying any activation function in the last layer and just using the signals to compare with the targets. This form of activation (or lack of) means output $y_j \in \mathbb{R}$ is in the range $(-\infty, \infty)$, and thus target values that are also $t_j \in \mathbb{R}$ can be learned.

Next we need a measure of how close the network outputs are to the desired targets. Using a commonly applied mean-squared objective function,

$$J = \frac{1}{2} \sum_{j=1}^{d(N)} (y_j - t_j)^2, \tag{3.4}$$

we have a measure of cost or error. The aim now is to minimise this function by adjusting the weights. It was not until the 1970s, when Werbos and Rumelhart developed an algorithm, termed backpropagation, that this objective function was able to be effectively minimised. This is also the algorithm that neuroscientists cannot justify as being biologically plausible.

Backpropagation

To best understand how the backpropagation algorithm works, and why it is considered biologically infeasible we start with *computational graphs*. These are graphical trees that describe the dependencies and computations in a neural network, and, towards the end of the network, the relationship to the objective function it is trying to minimise.

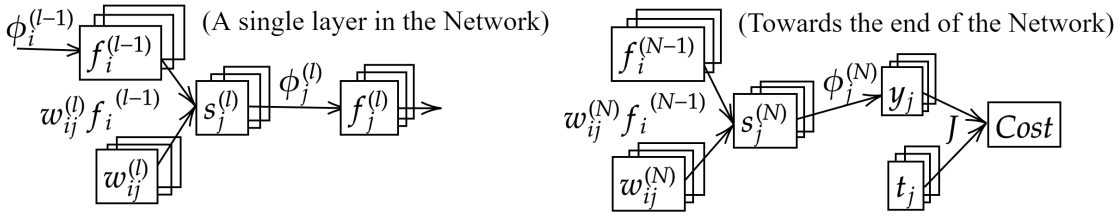


Figure 3.2: Computational Graphs for a typical fully connected artificial neural network. This allows us to easily determine how the weights affect the cost computed by the objective function.

The backpropagation algorithm uses reverse auto-differentiation to compute the gradients of the objective function with respect to the weights, and then uses gradient descent (or other optimisation techniques) to update the weights so as to minimise the objective function. To perform reverse auto-differentiation, the computational trees come in handy. We introduce the variable δ which encompasses the error between target and output (considered here as the $N^{th}+1$ layer), and the variational effects of the signals on the objective function for all other layers. Note that from now on the biases are incorporated into the weights.

For the $N^{th}+1$ layer,

$$\delta_j^{(N+1)} = \frac{\partial J}{\partial y_j} = \frac{\partial}{\partial y_j}\left(\frac{1}{2}\sum_{j=1}^{d(N)}(y_j - t_j)^2\right) = y_j - t_j. \tag{3.5}$$

For the last (N^{th}) layer of the network and assuming the network is used for regression,[1]

$$\delta_j^{(N)} = \frac{\partial J}{\partial s_j^{(N)}} = \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial s_j^{(N)}} = \delta_j^{(N+1)}\frac{\partial}{\partial s_j^{(N)}}\left(\phi_j^{(N)}\left(s_j^{(N)}\right)\right) = \delta_j^{(N+1)}\phi_j^{\prime(N)}\left(s_j^{(N)}\right) \tag{3.6}$$

where the chain rule was applied once. See the computational tree to see the relationship between the objective function J and signal $s_j^{(N)}$. Notice that the 'delta' of the last layer is a function of the delta of the $N^{th}+1$ layer, and that the derivative of the activation function is needed. Now that the top two deltas have been been found, all other deltas can be found too. This is why the algorithm is called backpropagation or more specifically error-backpropagation, because the top-most error (delta) is propagated 'backwards' - each delta determined by previously computed deltas. Mathematically,

$$\begin{aligned}\delta_i^{(l-1)} = \frac{\partial J}{\partial s_i^{(l-1)}} &= \sum_{j=1}^{d(l)}\frac{\partial J}{\partial s_j^{(l)}}\frac{\partial s_j^{(l)}}{\partial f_i^{(l-1)}}\frac{\partial f_i^{(l-1)}}{\partial s_i^{(l-1)}} \\ &= \sum_{j=1}^{d(l)}\delta_j^{(l)} w_{ij}^{(l)}\phi_i^{\prime(l-1)}\left(s_i^{(l-1)}\right) \\ \delta_i^{(l-1)} &= \phi_i^{\prime(l-1)}\left(s_i^{(l-1)}\right)\sum_{j=1}^{d(l)} w_{ij}^{(l)}\delta_j^{(l)}. \end{aligned} \tag{3.7}$$

This procedure of back-propagating the error is essential for learning to take place in an artificial neural network. However, it is worth noting at this point that back-propagating the error in this way is not biologically feasible. These deltas are computed externally from the network; they are not inherent to the structure of the ANN. The backpropagation algorithm can be seen as forming its own computational tree separate to the computational tree of the neural network. Neurons, with biological synapses, however, can only change their connection strength solely on local signals, i.e. the activity of the neurons they connect to [50]. This **lack of local error representation** is the underlying reason why backpropagation is considered biologically implausible.

[1] For classification tasks, if using the *SoftMax* equation we would need to include a sum, as y_j would also depend on $s_k^{(N)}$, $k \neq j$, i.e. $\delta_j^{(N)} = \frac{\partial J}{\partial s_j^{(N)}} = \sum_{i=1}^{d(N)}\frac{\partial J}{\partial y_i}\frac{\partial y_i}{\partial s_j^{(N)}}$

Weight Update

Next we need to compute how variations in the weights between each layer influences the objective function, so that we can optimise the weights in order to minimise the objective function. Since the weights influence the signals for each neuron in a layer, and these signals in turn influence the value of the objective function, we find

$$\frac{\partial J}{\partial w_{ij}^{(l)}} = \frac{\partial J}{\partial s_j^{(l)}} \frac{\partial s_j^{(l)}}{\partial w_{ij}^{(l)}} = \delta_j^{(l)} f_i^{(l-1)}. \tag{3.8}$$

We now know how the weights affect the objective function, and can use this knowledge to minimise it. Using gradient descent, the update for each weight connecting two neurons becomes

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \delta_j^{(l)} f_i^{(l-1)}. \tag{3.9}$$

This completes the backpropagation algorithm. To reiterate, we see from the equation above that the weight changes are dependent on the back-propagated deltas, indicating that the weights, which are *local* to the network, are being influenced by *externally* computed deltas - which is one of the reasons why this algorithm is considered biologically infeasible.

Another reason for biological implausibility is the symmetry of the forward and backward weights. As we can see from Equation 3.7, the back-propagation of errors (deltas) uses the same weights to those that propagated the information forward in the network. This would imply, biologically, that identical connections exist in both directions between connected neurons. Despite the fact that bidirectional connections are apparently more common than expected, especially in cortical networks, these are not always present [54]. Even if bidirectional connections were always present, backward and forward weights would have to align themselves correctly, which seemingly is not the case in mammalian brains [50]. It is worth mentioning that the general model of the neuron in an artificial neural network is also not true to the biology. The neuron model used by ANNs sends continuous numerical output (assuming the activation function is continuous), which corresponds to the *firing rate* of biological neurons. Neurons, in reality, however, spike.

3.3 The Mathematics of Dendritic Neural Networks

Dendritic neural networks are, in essence, artificial neural networks that are more biologically realistic. Specifically, this type of network addresses the lack of local error representation that the backpropagation algorithm provides. There are two approaches to understanding dendritic neural networks. The first approach is a classical approach, whereby we derive the physical equations that describe electrical currents within neurons (using Leaky Integrate and Fire models [55]), and then piece these together to form a network to perform the task at hand. The second approach is an *energy* based approach, whereby we ignore the fundamental, intricate mechanisms of biological neurons, and instead argue that a dendritic neural network, much like an artificial neural network, is ultimately trying to optimise a certain objective. We will briefly delve into the first approach,

without deriving any equations, to achieve a physical understanding of what these networks look like and how different layers interact. Then we will move on to the second approach, and derive, using mathematics similar to those introduced for ANNs, the equations necessary to get a biologically plausible (or rather, a more biologically plausible) model for predicting tactile information.

Information Propagation in Dendritic Neural Networks

The information flow of dendritic neural networks, in comparison to artificial neural networks, is more complex. Figure 3.3 acts as a visual reference, to help guide the reader with the following explanation.

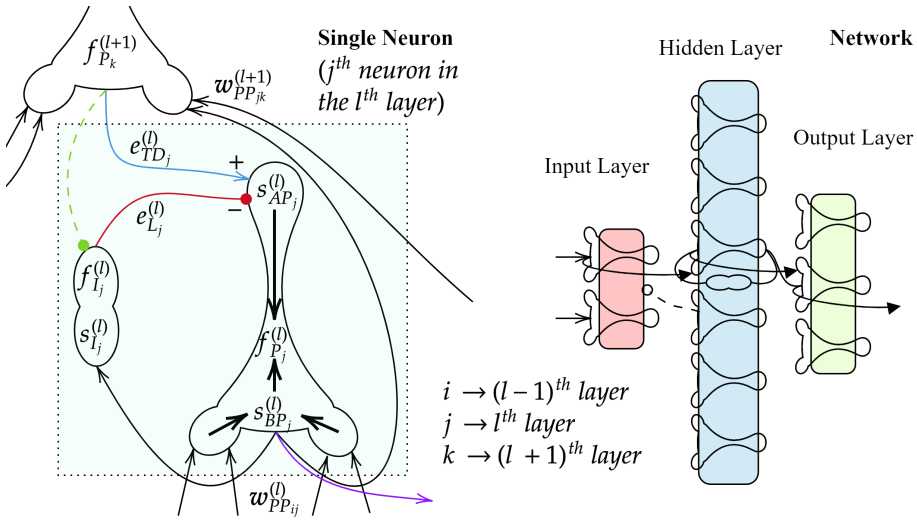


Figure 3.3: **Left**: A pyramidal cell and an interneuron are shown for one layer. The pyramidal cell receives bottom-up (black) signals from pyramidal cells in the previous layer, and top-down (light blue) from pyramidal cells in subsequent layers and lateral (red) error signals from interneurons in the same layer. The interneuron receives bottom-up signals from pyramidal cells within its own layer and top-down signals from pyramidal cells in the layer above. The purple line represents the axon, showing that the pyramidal cell can connect to other pyramidal cells in its own layer or to the layer below, either directly as an error or via an interneuron (dotted green line) **Right**: An example of a three layer dendritic neural network. Note that for clarity most of the connections have been omitted.

On a large scale, imagine (rather gruesomely) unfolding a human brain so that different layers in the brain become apparent. Within each layer, pyramidal cells are present, and dispersed between them, a couple of interneurons. Each brain layer communicates (via electrical signals) with *another* brain layer via the pyramidal cells, and the interneurons allow pyramidal cells *within* a layer to communicate with each other. Furthermore, a pyramidal cell in one layer can communicate with another pyramidal cell in another layer via an interneuron. The dendritic neural network is analogous to a multilayer (deep) fully connected neural network, with additional within-layer communication.

In a single neuron layer (the l^{th} layer), pyramidal cells receive 'bottom up' weighted signals from pyramidal cells in the layer below via their basal dendrites. These inputs together form signals $\mathbf{s}_{BP}^{(l)}$, where BP stands for 'basal pyramidal'.[2] Pyramidal cells also receive 'top-down' signals from pyramidal cells in the layer above **and** 'lateral' signals from interneurons within their own layer, via their apical dendrites. These are error signals, and will be referred to as $\mathbf{e}_{L}^{(l)}$ (lateral error) and $\mathbf{e}_{TD}^{(l)}$ (top-down error) from now on. The signals that these error signals generate within the apical dendrites are described by the vector $\mathbf{s}_{AP}^{(l)}$, where AP stands for 'apical pyramidal'. The lateral error signals are considered 'inhibitory' and the top-down error signals are considered 'excitatory', meaning the former reduces $\mathbf{s}_{AP}^{(l)}$, while the latter increases it. It is this difference, that creates a localised error, allowing for learning to occur [1]. The lateral error signals are a function of both the output of the pyramidal cells in the current layer and the output of pyramidal cells in the layer above, where P stands for 'pyramidal'. The outputs of the pyramidal cells in the l^{th} layer are $\mathbf{f}_{P}^{(l)} = \boldsymbol{\phi}^{(l)}\left(\mathbf{s}_{BP}^{(l)} + \mathbf{s}_{AP}^{(l)}\right)$, where $\boldsymbol{\phi}(\cdot)$ means the activation function ϕ acts on all elements of its input vector and returns a vector of the same size as its input vector.[3] The weights between layers are described by the matrix $\mathbf{w}_{PP}^{(l)}$, where PP indicates the weights are between pyramidal cells. One could add weights between the pyramidal cells and the interneurons and vice versa, but for simplicity these have been omitted.

Using these variables, the Leaky Integrate and Fire models can be applied (see [1]), however, in the following subsection we will see how these dendritic neural networks can also be derived using an energy-based framework which is similar to the derivation used for ANNs above and therefore requires no additional neuroscientific knowledge.

Learning in a Dendritic Neural Network using Energy-based Methods

Since we are only learning the weights between pyramidal cells, the subscripts are omitted from now on. Furthermore, for simplicity, the apical and basal pyramidal signals will be combined into one. The notation will thus be identical to those used in the section above for artificial neural networks, except that we will be using vectors and matrices instead of scalars to represent signals, biases and weights. The following is an extension (we consider biases) on the work developed by [56] in an unpublished paper on an energy formulation of their work on dendritic neural networks. It was written by Joao Sacramento and Walter Senn, two of the co-authors that developed dendritic neural networks, for Yoshua Bengio and Ben Scellier - two, well-known experts in computational neuroscience. Their energy formulation is analogous to Hamiltonian mechanics but for neural networks.

Energy formulation

Let $\mathbf{w}_{PP}^{(l)} = \mathbf{W}^{(l)}$ ($\mathbf{W}^{(l)} \in \mathbb{R}^{d(l-1)\times d(l)}$) be the matrix of weights between two layers of pyramidal cells, and let $\mathbf{s}_{BP}^{(l)} + \mathbf{s}_{AP}^{(l)} = \mathbf{s}^{(l)}$ ($\mathbf{s}^{(l)} \in \mathbb{R}^{d(l)\times 1}$) to simplify the following mathematics. Furthermore, let each

[2] Note we are using vectors and matrices from now on: $\mathbf{s}_{BP}^{(l)}$ is supposed to represent all basal pyramidal signals in one layer ($s_{BP_j}^{(l)} \in \mathbf{s}_{BP}^{(l)}$). Vector $\mathbf{s}_{BP}^{(l)}$ has dimensions $\mathbb{R}^{d(l)\times 1}$, where $d(l)$ is the number of pyramidal cells in the l^{th} layer.

[3] In other words, $\boldsymbol{\phi}(\mathbf{x}) = [\phi(x_1), \phi(x_2), ..., \phi(x_n)]^T$. If $\mathbf{x} \in \mathbb{R}^{n\times 1}$ then $\boldsymbol{\phi}(\mathbf{x}) \in \mathbb{R}^{n\times 1}$.

pyramidal cell have a bias $b_j^{(l)}$ associated with it, and therefore let the biases of all pyramidal cells in the l^{th} layer be described by the vector $\mathbf{b}^{(l)}$ ($\mathbf{b} \in \mathbb{R}^{d(l)\times 1}$). Consider a network of $N+1$ layers, as for the ANN presented in the section above. We now present an energy function, similar to the objective function for ANNs, except that this assigns a scalar value to the network itself and not to the discrepancy between the output of the network and a target value. We refer to it as the 'free energy' of the system, and, using vector notation, it takes the form

$$E^f = \frac{1}{2}\sum_{l=0}^{N}\left\|\mathbf{s}^{(l)} - \left(\mathbf{W}^{(l)}\boldsymbol{\phi}^{(l)}\left(\mathbf{s}^{(l-1)}\right) + \mathbf{b}^{(l)}\right)\right\|^2, \tag{3.10}$$

where $\|\cdot\|^2$ is used to represent the square vector norm. Equation 3.10 tells us that if the signals in layer l start to match the output signals of the layer $l-1$ below, the energy of the system decreases. This energy formulation is analogous to a Hopfield energy used in Hopfield neural networks [57] but for non-discrete states. The energy equation also gives us an indication as to how feed-forward the network is.

Gradient Dynamics DenNN Signals

How do the signals affect this energy? By expansion of the sum,

$$E^f = \frac{1}{2}\left(\ldots + \left\|\mathbf{s}^{(l)} - \left(\mathbf{W}^{(l)T}\boldsymbol{\phi}^{(l)}\left(\mathbf{s}^{(l-1)}\right) + \mathbf{b}^{(l)}\right)\right\|^2 + \left\|\mathbf{s}^{(l+1)} - \left(\mathbf{W}^{(l+1)T}\boldsymbol{\phi}^{(l)}\left(\mathbf{s}^{(l)}\right) + \mathbf{b}^{(l+1)}\right)\right\|^2 + \ldots\right),$$

we define (analogous to the Hamiltonian equation) the gradient dynamics of the signals for each neuron in any layer (l) as,

$$\dot{\mathbf{s}}^{(l)} = -\frac{\partial E^f}{\partial \mathbf{s}^{(l)}}, \tag{3.11}$$

which is computed as

$$\dot{\mathbf{s}}^{(l)} = -\mathbf{s}^{(l)} + \mathbf{W}^{(l)T}\boldsymbol{\phi}^{(l)}\left(\mathbf{s}^{(l-1)}\right) + \mathbf{b}^{(l)} + \mathbf{e}_{TD}^{(l)} + \mathbf{e}_{L}^{(l)}, \tag{3.12}$$

where,

$$\mathbf{e}_{TD}^{(l)} = \boldsymbol{\phi}^{\prime(l)}\left(\mathbf{s}^{(l)}\right) \odot \mathbf{W}^{(l+1)}\mathbf{s}^{(l+1)}, \tag{3.13}$$

which is interpreted as the top-down error signal, and

$$\mathbf{e}_{L}^{(l)} = -\boldsymbol{\phi}^{\prime(l)}\left(\mathbf{s}^{(l)}\right) \odot \left(\mathbf{W}^{(l+1)}\left(\mathbf{W}^{(l+1)T}\boldsymbol{\phi}^{(l+1)}\left(\mathbf{s}^{(l)}\right) + \mathbf{b}^{(l+1)}\right)\right), \tag{3.14}$$

which is interpreted as the lateral error signal, as shown visually in Figure 3.3. The $-\mathbf{s}^{(l)}$ term is interpreted as neuronal leak (axon output) and $\mathbf{W}^{(l)T}\boldsymbol{\phi}^{(l)}\left(\mathbf{s}^{(l-1)}\right)$ is the feed-forward input (from pyramidal cells in the previous layer). The biases are interpreted as defects/differences between individual neurons, that can adjust to synaptic changes, and the derivative of the action potential ϕ can be thought of as dendritic transfer conductance. The symbol $\odot$ is used to represent the element-wise product, and note that for the last layer $\mathbf{e}_{TD}^{(l)} = \mathbf{e}_{L}^{(l)} = \mathbf{0}$, because $\mathbf{W}^{(l+1)} = \mathbf{0}$. Further note that given input $\mathbf{x}$ into the network, for the first hidden layer $\boldsymbol{\phi}^{(1)}(\mathbf{x}) = \mathbf{x}$.

Comparison of DenNN Signals to ANN Signals

Consider now the feed-forward signal generated in an artificial neural network,

$$\mathbf{s}_{ff}^{(l)} = \mathbf{W}^{(l)T}\boldsymbol{\phi}^{(l)}\left(\mathbf{s}^{(l-1)}\right) + \mathbf{b}^{(l)}.$$

If we add the lateral and top-down errors into a single error signal $\mathbf{e}^{(l)}$, then we observe that in steady state ($\dot{\mathbf{s}}^{(l)} = \mathbf{0}$) the error,

$$\begin{aligned}\mathbf{e}^{(l)} = \mathbf{s}^{(l)} - \mathbf{s}_{ff}^{(l)} &= \boldsymbol{\phi}'^{(l)}\left(\mathbf{s}^{(l)}\right) \odot \left(\mathbf{W}^{(l+1)}\left(\mathbf{s}^{(l+1)} - \left(\mathbf{W}^{(l+1)T}\boldsymbol{\phi}^{(l+1)}\left(\mathbf{s}^{(l)}\right) + \mathbf{b}^{(l+1)}\right)\right)\right) \\ &= \boldsymbol{\phi}'^{(l)}\left(\mathbf{s}^{(l)}\right) \odot \left(\mathbf{W}^{(l+1)}\left(\mathbf{s}^{(l+1)} - \mathbf{s}_{ff}^{(l+1)}\right)\right) = \boldsymbol{\phi}'^{(l)}\left(\mathbf{s}^{(l)}\right) \odot \mathbf{W}^{(l+1)}\mathbf{e}^{(l+1)},\end{aligned} \tag{3.15}$$

which is identical to the error backpropagation presented above for ANNs (in vector notation). In steady state, the free energy is equivalent to (half) the sum of the errors $\mathbf{e}^{(l)}$, and, if no target value is presented in the final layer (i.e. $\mathbf{e}^{(N)} = \mathbf{0}$), then this forces all errors to zero, and thus $\mathbf{s}^{(l)} = \mathbf{s}_{ff}^{(l)}$. If this steady state is not entirely reached, however, the behaviour is not entirely feed-forward.

Learning Targets

For the network to learn particular targets, we require an additional term. Similar to ANNs, we define an objective function as,

$$J = \frac{\beta}{2}\sum_{j=1}^{d(N)}(y_j - t_j)^2 = \frac{\beta}{2}(\mathbf{y}-\mathbf{t})^T(\mathbf{y}-\mathbf{t}) = \frac{\beta}{2}\|\mathbf{y}-\mathbf{t}\|^2, \tag{3.16}$$

where $\mathbf{s}^{(N)} = \mathbf{y}$ and $\mathbf{t}$, as before, are the output and target values, respectively. The hyperparameter β, referred to in the paper [56] as the nudging coefficient, is simply a means by which to control how much the error gets scaled when it propagates backwards. We thus define the 'total energy' as

$$E = E^f + J = \frac{1}{2}\sum_{l=1}^{N}\left\|\mathbf{s}^{(l)} - \left(\mathbf{W}^{(l)T}\boldsymbol{\phi}^{(l)}\left(\mathbf{s}^{(l-1)}\right) + \mathbf{b}^{(l)}\right)\right\|^2 + \frac{\beta}{2}\|\mathbf{y}-\mathbf{t}\|^2. \tag{3.17}$$

The gradient dynamics are identical to the 'free energy' system except that now, for the last layer,

$$\dot{\mathbf{s}}^{(N)} = \dot{\mathbf{y}} = -\mathbf{y} + \mathbf{W}^{(N)T}\boldsymbol{\phi}^{(l)}\left(\mathbf{s}^{(N-1)}\right) + \mathbf{b}^{(l)} + \beta(\mathbf{y}-\mathbf{t}). \tag{3.18}$$

Thus, $\mathbf{e}^{(N)}$ is no longer zero, but instead $\mathbf{e}^{(N)} = \beta(\mathbf{y}-\mathbf{t})$. In steady state, we can propagate this error backwards (scaled by β) to find the other errors and update the weights.

Weight Update

By differentiation of the total energy function, the gradient dynamics of the weights are

$$\dot{\mathbf{W}}^{(l)} = -\frac{\partial E}{\partial \mathbf{W}^{(l)}} = -\left(\mathbf{s}^{(l)} - \left(\mathbf{W}^{(l)T}\boldsymbol{\phi}^{(l)}\left(\mathbf{s}^{(l-1)}\right) + \mathbf{b}^{(l)}\right)\right)\boldsymbol{\phi}^{(l)T}\left(\mathbf{s}^{(l-1)}\right). \tag{3.19}$$

To aid understanding, we present a comparison to ANNs. In steady-state, the gradient dynamics of the weights for a dendritic neural network becomes

$$\dot{\mathbf{W}}^{(l)} = -\frac{\partial E}{\partial \mathbf{W}^{(l)}} = -\mathbf{e}^{(l)}\boldsymbol{\phi}^{(l)T}\left(\mathbf{s}^{(l-1)}\right), \tag{3.20}$$

which, under integration by the Euler method, is identical to the gradient descent weight update we showed above used by the backpropagation algorithm for ANNs; in other words,

$$\mathbf{W}^{(l)} = \mathbf{W}^{(l)} - \eta\mathbf{e}^{(l)}\boldsymbol{\phi}^{(l)T}\left(\mathbf{s}^{(l-1)}\right), \tag{3.21}$$

where η is the learning rate, which is identical to the step-size used for the Euler method. Note that, during the weight update, we are assuming that $\mathbf{y} \neq f\left(\mathbf{W}^{(N)}\right)$, i.e. that $\mathbf{y}$ has already been fixed before the weights are updated. The exact methodology for learning will be described in the next section.

Bias Update

Similarly, for the biases,

$$\dot{\mathbf{b}}^{(l)} = -\frac{\partial E}{\partial \mathbf{b}^{(l)}} = -\left(\mathbf{s}^{(l)} - \left(\mathbf{W}^{(l)T}\boldsymbol{\phi}^{(l)}\left(\mathbf{s}^{(l-1)}\right) + \mathbf{b}^{(l)}\right)\right). \tag{3.22}$$

That concludes all the necessary equations for dendritic neural networks. Since the errors are functions of previous and subsequent weights, biases and signals, dendritic neural networks allow for a localised propagation of error, thereby making it more biologically plausible, while still approximating the effective approach of back-propagating the errors used by ANNs. Other than this, however, dendritic neural networks make the same, biologically false, assumptions that ANNs make (see end of the previous section).

Note that the equations above can be used both for regression and classification purposes, as for ANNs. For regression purposes nothing has to be adjusted, but for classification purposes the output signals would need to be transformed into probabilities. This could be achieved using the *Softmax* equation presented above for ANNs, for example. However, for this study, we only used dendritic neural networks for regression purposes.

3.4 Energy-based Methodology for Learning

Dendritic neural networks provide a means by which to propagate the errors locally. In order for such a network to learn target values, given a set of input values, an algorithm, or methodology is needed. For this study, the 'equilibrium propagation' framework, proposed by Scellier and Bengio in 2017, was used [12]. Similar to the way backpropagation can be used for networks that have computational graphs, equilibrium propagation can be used for energy-based models. The equilibrium propagation framework requires only one 'computational circuit', whereas learning in

artificial neural networks requires two - one for the forward pass and one for the backpropagation of errors - before the weights are updated. In equilibrium propagation the same computation is used twice before the weights are updated, making it more biologically plausible [12].

There are three phases to equilibrium propagation. Note that before any of these phases are computed, all the weights and signals are either initialised to random values or zeros. The first phase, the 'free energy' phase, uses equation 3.12 to find a local minimum of the free energy network, and generates an output prediction $\mathbf{y}$. Specifically, an input $\mathbf{x}$ is fed into the network, and then all signals are determined by integrating equation 3.12 using the Euler method. The step-size Δt, and the number of iterations n_I are set manually. The second phase, referred to as the 'nudging' phase, moves the predicted output towards the desired target - using equation 3.18. The third phase, the 'weakly clamped' phase, performs the same computation as the first phase, except that the signals for the output layer are not updated, instead they are fixed by the values determined by the nudging phase, and their values seep backwards through the network, readjusting all the signals in the hidden layers to a new and different energy minimum.

Finally, the weights are adjusted, using equation 3.21. Their job is in essence to unlearn the fixed point generated in the first phase and learn the fixed point generated by the third phase, thereby reducing the total energy of the system (Equation 3.17). Note that the weight update is only dependent on the signals of the nodes in the network, which are determined by the same computation (adjusted only slightly), which is why it is considered a local computation.

These phases are repeated for each newly presented input $\mathbf{x}$. The input $\mathbf{x}$ belong to the training set $\mathbf{X}_t$. For multiple epochs (repetitions of presenting the training set), the dendritic neural network learns to predict the training targets. Finally, after a certain number of epochs, the weights and biases will be such that any input $\mathbf{x} \in \mathbf{X}_t$ will result in an output that is close to the desired target values. The weights and biases are then saved, and new, unseen data that is presented to the network can be predicted.

This method for learning weights and biases, albeit more biologically plausible, is computationally inefficient [12]. This is due to the fact that the free energy phase and the weakly clamped phase require integration, which, if performed using the Euler method, requires additional *for* loops slowing the process of learning down. All of the above is also summarised by the pseudo-code presented in the next section.

3.5 Implementation of a Dendritic Neural Network Learning Framework in Python

The dendritic neural networks designed for this project were implemented in the Python programming language using the freely available *NumPy* math module [58]. To help navigate the following description, the main functions and classes that were created are colour coded. A *class* (the *DenNN class*) was created for single-hidden layer dendritic neural networks, which allowed

for network parameters to be adjusted as desired. This included being able to change the input, hidden-layer, and output dimensions, as well as control other parameters, such as the nudging parameter beta, the number of integrations (n_I) and the step-size (δt) used in the Euler method, necessary for equilibrium propagation. Within the *class*, various functions were defined. The four main functions necessary for learning were defined as; firstly, the *free-energy* function, which computed the signals (Equation 3.12) that minimised the free energy function (Equation 3.10); secondly, the *nudging* function, which moved the output layer activations towards the target values (Equation 3.18); thirdly, the *weakly clamped phase* function which performed the same computation as the *free-energy* function, except this time without computing the output layer activations; and finally the *weight-update* function, which updated the weights and biases according to Equation 3.21. Other functions in the class included the *activation function* (sigmoid) and its derivative (necessary for the other functions to compute the equations mentioned above). After the Python class had been prepared, the equilibrium propagation method for learning was implemented in a separate Python file; the dendritic neural network class described above was imported, and the training data was loaded into the file as well. The following shows pseudo-code for what was implemented in the separate Python file in order to induce learning in a dendritic neural network. Please note, the code for this can be found in the link presented in the Abstract.

Algorithm 1 Dendritic Neural Network Learning Framework

Import *DenNN class* and *Training Data* $\mathbf{X}_t$
Initialise: *Randomise all weights and biases, set activations to zero.*
for epochs **do**
    for $\mathbf{X}_t$ **do**
        for 1 to n_I **do**
            Minimise *Free Energy [Apply free-energy function]*
        Nudge *Outputs Towards Desired Targets [Apply nudging function]*
        for 1 to n_I **do**
            Minimise *Weakly Clamped Phase [Apply weakly clamped phase function]*
        Update *Weights and Biases [Apply weight-update function]*
Save Weights and Biases

3.6 Chapter Summary

The following summarizes some of the key observations we made in this chapter. Dendritic neural networks,

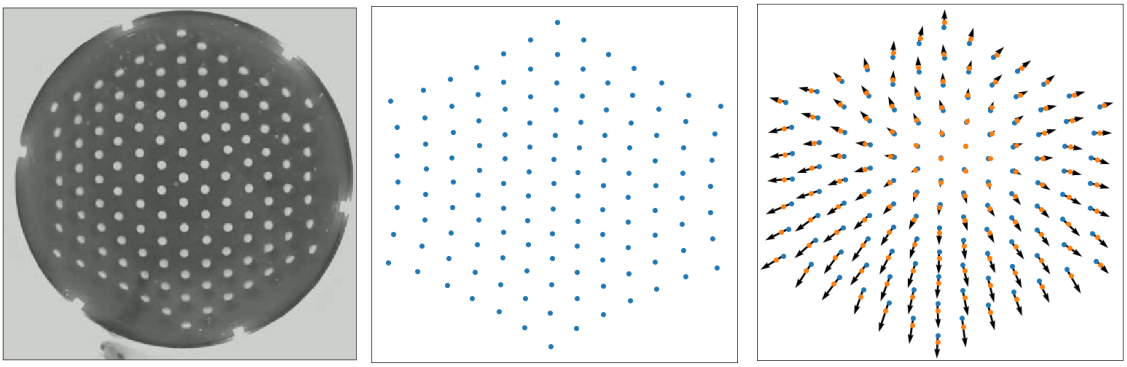
- in steady state, behave identical to artificial neural networks;
- allow for localised error propagation, whereas artificial neural networks do not;
- only require one computational circuit, whereas artificial neural networks require two;
- can use equilibrium propagation to learn, whereas artificial neural networks use reverse auto-differentiation on computational trees (backpropagation);
- are, given the above, considered more biologically plausible than artificial neural networks by neuroscientists [1].

4. Training and Validation of the Dendritic Neural Networks

In this chapter, we prepare three dendritic neural networks for contour following tasks presented in the next chapter. The dendritic neural networks aim to accurately predict the distance from an edge to the centre of a TacTip and the angle of rotation of a TacTip on that edge. These two 'targets' (position and angle) are necessary to perform contour following tasks in the subsequent chapter. Specifically, we explore how data was collected and processed in order to train the networks, the architecture of the networks, and the selection of the parameters used. Finally, we present the training and validation results.

4.1 Data Collection and Processing

Similar to the human brain encoding touch by sending electrical signals produced by mechanoreceptors underneath the skin, we can artificially encode tactile information using tactile sensors. The tactile information captured by a TacTip sensor can be encoded in various ways; however, this report only focused on the pin tracking method. This approach determines the **x** and **y** pin position coordinates inside a TacTip using computer vision. Specifically, the pins are extracted from the images captured by the camera inside the TacTip using OpenCV's *CVBlobDetector* module. This module converts the detected 'blobs' (the pins) into so-called 'keypoints' which can then be further converted into an array of **x**,**y** coordinates. A 'keypoint' is a generic computer vision term used to describe the location and dimensions of a feature - in this case the pins of a TacTip. The pin tracking method allows for a 1-dimensional representation of the tactile data by concatenating the **x** coordinates with the **y** coordinates. There are 127 pins in a TacTip and using the pin tracking method the locations of the pins were converted into a 1×254 array (first **x**, then **y** positions) to allow for a 1-dimensional input into the neural network.



(a) Image Data (b) Pin Tracking Data (c) An example of pin displacement

Figure 4.1: *Data Processing* **a)** The pins of the TacTip captured by the camera **b)** The extracted pin positions using OpenCV's *CVBlobDetector* computer vision module **c)** Initial pin positions, and a second set of pin positions later in time and vectors showing the direction of displacement.

To ensure that the pins were always collected in the same order, initial pin keypoints were extracted while the TacTip was not contorted. By 'same order' we mean that the keypoints extracted by the

blob-detection module were always converted into the same locations within the array describing the pin locations. This was relevant for both the data collection and the contour following tasks, as any change in order would affect the prediction that the neural network makes, thereby invalidating the network's prediction. Furthermore, the initial pin position was used to find the pin displacements, by subtracting it from any other array. The pin displacements, not the pin positions, were used as input to the neural network.

Data Collection Methodology

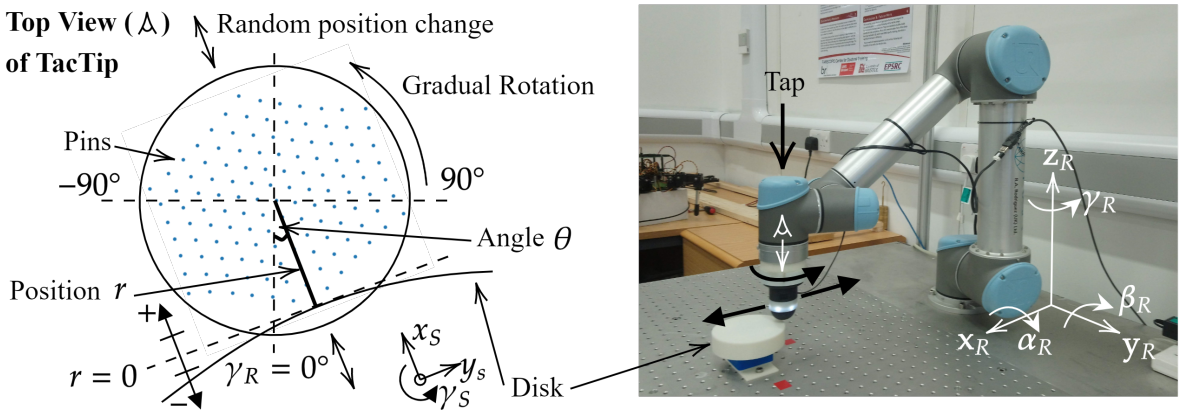


Figure 4.2: *Data Collection* **Left**: A diagram of a top view of a TacTip (as if the camera inside the TacTip mount would not rotate with the TacTip, see view symbol on robot head). The position r is the distance from the centre of the TacTip to the edge of the disk. The positions were changed randomly along the x_R axis (subscript R being the robot's frame of reference), ranging from -5 mm to +5 mm, where 0 mm is the centre of the TacTip on the edge of the disk. The angle θ describes the clockwise or anti-clockwise rotation of the TacTip around its own axis (γ_S, in sensor's frame of reference 'S') relative to the robot's frame of reference angle γ_R. This angle was gradually changed from $-45°$ to $+45°$. **Right**: The UR5 robot used for the data collection and black arrows indicating the motion of changing r and θ. The robot was programmed to keep the bottom of the TacTip 2 mm above the surface of the disk and then move the TacTip down by 5 mm, thereby causing a 3 mm indentation of the rubber skin, and then move back to being 2 mm above the disk, before changing position and angle. We refer to this down-up action as a tap. For robustness a little random variation ($\pm$1 mm) in the tapping distance was given, resulting in actual tapping distances between 4 to 6 mm.

Data was collected using a UR5 robotic arm that was programmed to tap the TacTip on the edge of a disk. After every tap, the position normal to the disk was set to change randomly and the rotation of the TacTip around its own axis was programmed to rotate gradually from -45 to +45 degrees (see Figure 4.2). For the training and validation sets, 1000 and 500 samples were taken, respectively. The validation set was not extracted from the same data used for the training set, but instead was collected on a different run of the robot. In sequence, the data was collected as follows:

1. Align the center of the TacTip with the edge of the disk.
2. Change the position r randomly and set the angle θ to -45 degrees.
3. Tap the TacTip onto the disk and take an image of the displaced pins.

4. Using *CVBlobDetector* extract keypoints of pins, subtract this from the initial keypoints and convert them into a 1-dimensional array of pin displacements.
5. Repeat steps 2 to 4, but increase the angle θ. When $\theta = +45$ degrees, the desired number of samples will have been reached.

Experimental Issues

In order to collect data at an efficient rate, the robot was programmed to move at linear and rotational speeds of 30 mm/s. This would result in the table, on which the robot was mounted, to vibrate slightly while collecting data, which may have resulted in some minimal loss of accuracy.

4.2 Three Dendritic Neural Networks

Three different dendritic neural networks were trained on the collected data. One of the networks used pin displacements as input (DisDenNN), another applied principle component analysis to the input (PCADenNN), and the third split the network into two parallel networks and applied principle component analysis to each of their inputs (PDenNN). Doing this, we were able to observe if single hidden-layer dendritic neural networks perform better with smaller input dimensions and whether the angle and position targets can be inferred independently. In the end, the parallel dendritic neural network performed the best overall, and was thus used for the contour following tasks. The three networks are summarised in Table 4.1.[1]

Table 4.1: **Comparison of Network Parameters**

	Networks			
	DisDenNN	PCADenNN	PDenNN	
Parameters	-	-	Angle	Position
PCA Applied To Input	False	True	True	True
Number of Principle Components	-	35	35	30
Input Dimension	254×1	35×1	35×1	30×1
Single Hidden Layer Dimension	500×1	1000×1	1000×1	500×1
Output Dimension	2×1	2×1	1×1	1×1
Learning Rate	0.001	0.001	0.001	0.001
Nudging Parameter (Beta)	0.3	1	0.5	1.2
Epochs Trained	200	200	200	200
Average Angle Prediction Error (degrees)	6.55	5.89	5.36	-
Average Position Prediction Error (mm)	1.34	0.53	-	0.38

DisDenNN

This dendritic neural network used pin displacements as input, a single hidden layer of 500 neurons and had two outputs for angle and position. The network produced an average error of 6.55 degrees in predicting the angle, and had an average error of 1.34 mm in predicting the

[1] While it is difficult to make a fair comparison between different neural networks, the average accuracies obtained by the PCADenNN and PDenNN are within the range of average accuracies obtained by state-of-the-art convolutional artificial neural networks on a similar task [59]. These convolutional networks achieved average accuracies in the range of $0.2 - 0.6$ mm for position and $1.6 - 6.4$ degrees for angle.

position. We observed that a lower nudging parameter value produced better results for predicting the angle, but worse results for predicting the position. We also observed that a small learning rate of 0.001 worked better than larger learning rates, and that a hidden layer with 500 neurons produced better results than a hidden layer with less or more neurons.

PCADenNN

This network applied principle component analysis (PCA) to its input data. In 2019, it was shown that PCA can effectively generalise pin displacements captured by the TacTip for contour following tasks [60]. PCA, therefore, was a good means of reducing the input dimension of a neural network (given the kind of data being collected).

PCA is a statistical method for extracting the most important features in a data set. It does so by transforming the data into linearly uncorrelated variables called principle components. Specifically, principle components ($\mathbf{w}_k$) are the eigenvectors of the covariance matrix of the standardised data set $\mathbf{X}^T\mathbf{X}$. Each eigenvector has a corresponding eigenvalue, and those eigenvectors with the largest eigenvalues contain the most information (variance) in the data. The training data set was standardised by subtracting the mean $\bar{x}_j = \frac{1}{n}\sum_{i=1}^{n} x_{ij}$ from each data point x_{ij} and then dividing by the variance $\sigma_j^2 = \frac{1}{n}\sum_{i=1}^{n}(x_{ij} - \bar{x_j})^2$, i.e. $x_{ij} = \frac{x_{ij}-\bar{x}_j}{\sigma_j}$, where $x_{ij} \in \mathbf{X}$.

The PCADenNN used 35 principle components (thereby maintaining 99.47% of the variance in the data) as input into the network. This network achieved average errors of 5.89 degrees and 0.53 mm for angle and position, respectively, showing mainly an improvement in position accuracy. We observed that this network needed more nodes in its hidden layer, and a higher beta value, compared to the DisDenNN. Generally, PCA and a reduced network-input-dimension, showed the ability to enhance learning for single hidden-layer dendritic neural networks (given this tactile data).

PDenNN

This parallel dendritic neural network achieved the task of learning the two targets by using two different networks to learn each target separately and then combined their outputs. PCA was applied twice to the training data $\mathbf{X}$, extracting two different sets of principle components ($\mathbf{w}_k^r$ for position and $\mathbf{w}_k^\theta$ for angle). Principle components $\mathbf{w}_k^r$ were then used as input to the network learning the position, and $\mathbf{w}_k^\theta$ were used as input into the network learning the angle. The hidden layer of the position network used 500 neurons, and the hidden layer of the angle network used 1000 neurons. Finally, the outputs were combined to produce a 2-dimensional output, as generated by the networks implemented above. A diagram of the architecture of this network can be found in Figure 4.3.

For the angle network, 35 principle components (saving 99.47% of the variance) were used as input, 1000 neurons in a single hidden layer, a learning rate of 0.001, and a beta value of 0.5 were a good

combination. For the position network, 30 principle components (saving 99.35% of the variance) were used as input, 500 hidden layer neurons, a learning rate of 0.001 and a beta value of 1.2, also worked well. This parallel network reduced the average errors to 5.36 degrees and 0.38 mm for angle and position, respectively, for the validation set. We observed that learning the targets separately had a beneficial effect on target predictions, showing that the angle and position targets were able to be inferred independently using dendritic neural networks and PCA.

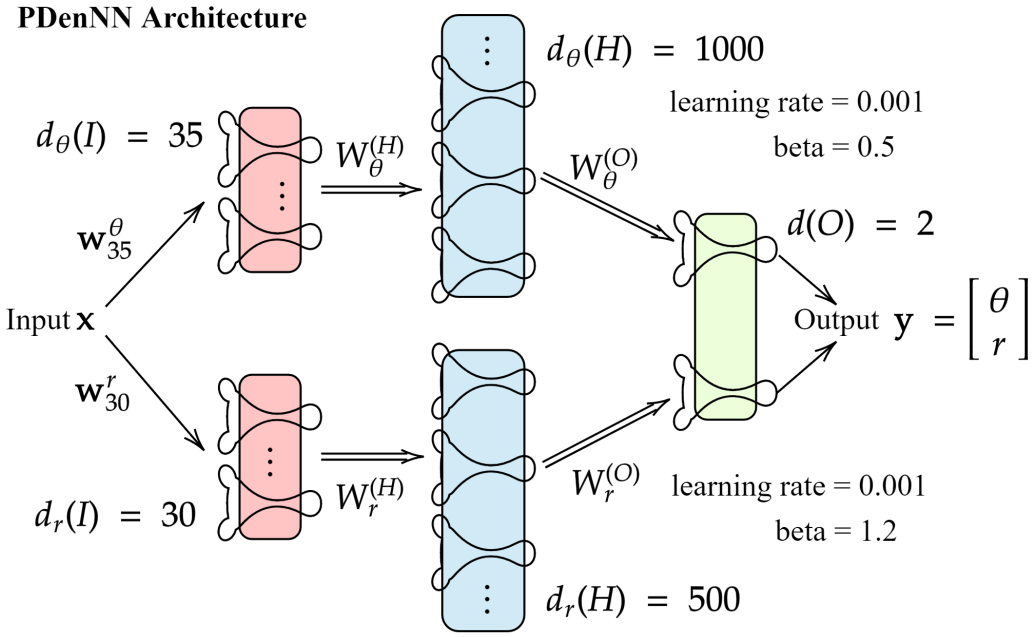


Figure 4.3: A diagram of the parallel dendritic neural network (PDenNN) used to perform the contour following tasks in the next section of this report. It consisted of two separate networks that used a different number of principle components as input and had different hidden layer sizes and beta (nudging parameter) values.

Preparing Real-Time Input Data into PDenNN for Contour Following

In preparation for the contour following tasks, where data was fed into the parallel dendritic neural network in individual samples, the test data was split into its individual samples as well. Each sample was then matrix-multiplied by the projection matrix (the matrix consisting of the chosen principle components). The projection matrices for the position network and angle network were $\mathbb{R}^{(254\times30)}$ and $\mathbb{R}^{(254\times35)}$, respectively. A sample of dimension $\mathbb{R}^{(1\times254)}$ would thus become dimension $\mathbb{R}^{(1\times30)}$ for the position network and $\mathbb{R}^{(1\times35)}$ for the angle network. Every sample was standardised first before projection.

4.3 Validation Results

In this section we show the validation results attained by each of the networks. We found that the DisDenNN generally predicted positions that were too large and angles that were too small. The PCADenNN was more accurate in its predictions, but struggled to predict negative angles below

-20 degrees. The PDenNN behaved similarly to the PCADenNN (also struggling to predict negative angles below -20 degrees), but was generally more accurate in predicting the position.

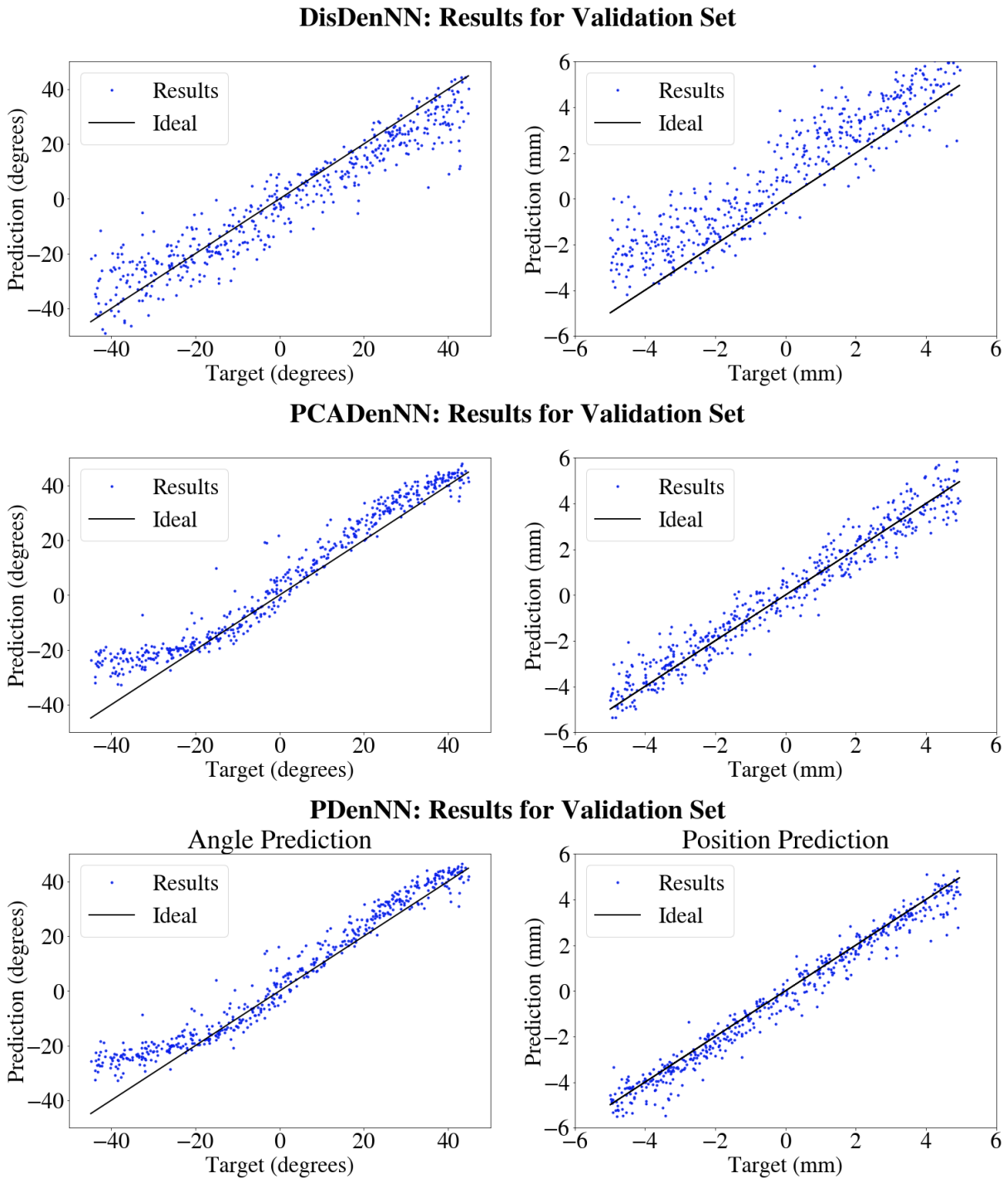


Figure 4.4: **Top**: Validation Results using pin displacements to train a dendritic neural network with one hidden layer of 500 neurons, 2 outputs (angle and position), a learning rate of 0.001, and a beta value of 0.3. **Middle**: Validation results using PCA. The network consisted of 35 inputs (number of principle components), a hidden layer with 1000 neurons, 2 outputs (angle and position), a learning rate of 0.001, and a beta value of 1. Both networks were trained for 200 epochs. **Bottom**: The validation results for the PDenNN.. The PDenNN was trained for 200 epochs, and the parameters can be found in Table 4.1 or in Figure 4.3 above.

Loss and Total Energy Curves

The loss and total energy curves for the PDenNN network are shown in Figure 4.5. The loss was computed as the square of the difference between the predicted output and target (see equation 3.16 in Chapter 2, with $\beta = 1$), as is usually done for ANNs. The total energy was computed using equation 3.17 (also in Chapter 2), giving a measure of both the discrepancy between network outputs and targets, and how feed-forward the network behaved. Interestingly, while the general shape of the total energy curves and the angle loss curve are similar to curves that one might typically find using ANNs, the position loss curve has quite an unusual shape. It is hard to know what caused it: it could be that this was caused by the data, but it could also be something inherent to do with dendritic neural networks learning in a different manner to artificial neural networks.

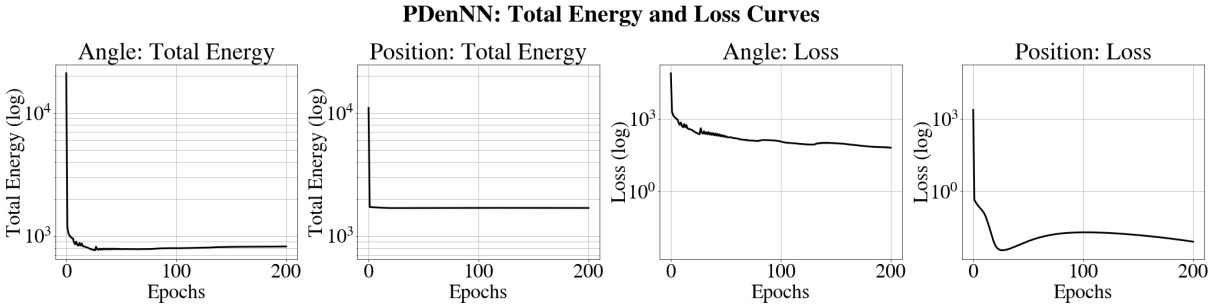


Figure 4.5: The total energy and loss curves for the angle and position during training. A log scale is used for the y axes to see the changes in total energy and loss more clearly.

4.4 Chapter Summary

In this chapter we showed:

- how data was collected in order to train and validate three different dendritic neural networks in preparation for the contour following tasks;
- three different network architectures and parameter choices;
- how the networks performed on the validation set.

We discovered (given the context of this report) that:

- reducing the input size by applying PCA can help improve learning in single hidden-layer dendritic neural networks;
- a larger nudging parameter value improves learning the position target, while a lower nudging parameter improves learning the angle target.
- the angle and position targets can be learned separately when PCA is applied to the tactile data.

5. Contour Following Tasks

In this chapter we explore how well the PDenNN (see previous chapter) performed on various contour following (2D servoing) tasks. Specifically, we explain the control policy, experimental set up, results and conclusions. To see the contour following tasks in action click `here`.

5.1 The Contour Following Policy

In 2D servoing (contour following) a robot moves around an unknown object using just a control policy and pose predictions. Explicitly, the TacTip is tapped onto an edge of an unknown object, then returns to its set height and moves to another point along which it predicts the contour of the object to lie beneath and then taps again and repeats the process for a set number of steps. (Note that the continuous edge of an object is considered the contour here.) The pose predictions are made by the PDenNN model, which uses the tactile information it receives from an object's edge. Next, these predictions or 'model outputs' are used to infer an error. This error encompasses the discrepancy between where the sensor currently is and where it should be, so that the centre of the sensor is precisely on an edge and at an angle normal to that edge. The purpose of the control policy is to minimise this error. This type of control policy is referred to in the literature as pose-based tactile servo (PBTS) control, and the following explanation is based on the work by Lepora and Lloyd [4], but tailored specifically for 2D contour following.

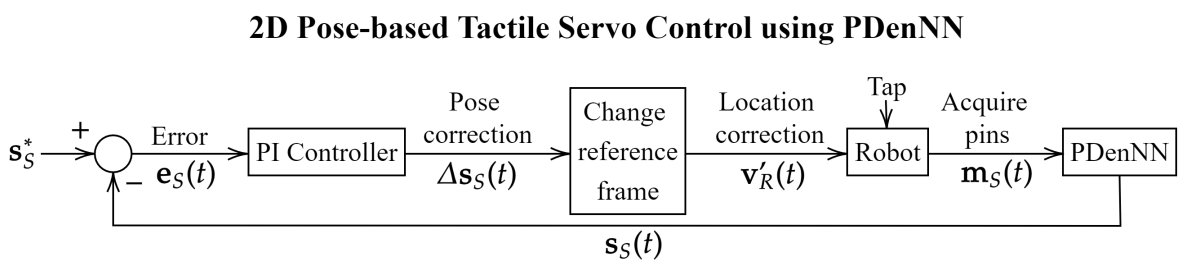


Figure 5.1: The control loop used for the contour following tasks. An error is generated in the sensor's frame of reference, which causes a PI controller to correct the pose and location of the sensor. The sensor is then tapped onto the unknown object to acquire the pin data, which is then passed into the PDenNN model to return a pose estimate of the sensor to produce a new error.

Consider two reference frames. Let the first reference frame belong to that of robot (R) and the second to that of the tactile sensor (S). The robot used in this study had six degrees of freedom with which it could move the tactile sensor. (See Figure 4.2, showing the TacTip attached to the robotic arm as an end effector.) Any location, within the (acceptably) movable range, that a robot can move the centre-bottom of a TacTip, within its own reference frame, can be described by the 6-dimensional vector,

$$\mathbf{v}_R = (x_R, y_R, z_R, \alpha_R, \beta_R, \gamma_R) = (\boldsymbol{\tau}_R, \boldsymbol{\theta}_R), \tag{5.1}$$

where $\boldsymbol{\tau}_R$ contains the positional components and $\boldsymbol{\theta}_R$ contains the rotational components (Euler angles). The UR5 robot used for the contour following tasks was programmed so that if such a

coordinate vector was passed into its work frame it would automatically move the TacTip to the location described by vector $\boldsymbol{\tau}$, at angles described by the vector $\boldsymbol{\theta}$ in 3D space, irrespective of where the TacTip was being held previously. For 2D servoing tasks, the angles α_R and β_R remain fixed and so does the height coordinate z (except for when the TacTip taps onto the object).
Consider now the reference frame of the sensor. The error we wish to minimise is

$$\mathbf{e}_S(t) = \mathbf{s}_S^* - \mathbf{s}_S(\mathbf{m}(t)), \tag{5.2}$$

where the vector $\mathbf{m}(t)$ is a set of tactile measurements at discrete times $t = 1, 2, \ldots$. In this case $\mathbf{m}(t)$ was the set of extracted pins after the robot had tapped the TacTip onto the object's edge. The pose estimate $\mathbf{s_S}(\mathbf{m}(t))$ is a vector of k tactile features that the PDenNN model returns, given the input $\mathbf{m}(t)$. Since the model was trained for displacements away from an edge along the x-axis of the robots frame of reference (x_R), and rotations around the sensor's local z-axis (γ_S) relative to the robots frame of reference (γ_R), the outputs r and θ (see previous chapter, in the data collection section), imply that

$$\mathbf{s}_S(\mathbf{m}(t)) = (r(t), 0, 0, 0, 0, \theta(t)). \tag{5.3}$$

In other words, $\mathbf{s}_S(\mathbf{m}(t))$ indicates how far the centre of the sensor is from being exactly on and normal to the edge of an object. The vector $\mathbf{s}_S^*$ is the reference pose that the controller drives the sensor towards. The reference pose is essential for moving the sensor around the object. In this study the reference pose was set to

$$\mathbf{s_S^*} = (0, 1, 0, 0, 0, 0), \tag{5.4}$$

as this ensured that an error will always exist (and as will become apparent shortly, if an error exists, then the sensor is forced to move). Next, a control law is needed to minimise the error $\mathbf{e}_S$. A proportional-integral (PI) controller in discrete time was used to control the sensor pose, and the desired change of pose in the sensor's reference frame can thus be expressed as

$$\Delta\mathbf{s}_S(t) = K_p\mathbf{e}_S + K_i F\left(\sum_{t'=0}^{t} \mathbf{e}_S(t')\right). \tag{5.5}$$

The integral (sum) term is passed through an anti-windup function F that bounds it to ± 5 and ± 45 in the x and γ components. These boundaries were set because the PDenNN model was only trained on data that was collected between these ranges, and ensured that the integral term could not become too dominant in the control. The constant diagonal matrices K_p and K_i (both $\mathbb{R}^{(6\times 6)}$), represent the proportional and integral gains, respectively, and were set to

$$K_p = \mathrm{diag}(0.5, 1, 0, 0, 0, 0.5), \qquad K_i = \mathrm{diag}(0.3, 1, 0, 0, 0, 0.1). \tag{5.6}$$

Next, we need to convert the proposed displacement in the sensor reference frame $\Delta\mathbf{s}_S(t)$ given by the control law, into a movement by the robot; to get the sensor into its desired pose. To do this, quaternions were used, as these allow for easy transformations between rotating frames. Let

$$\Delta\mathbf{s}_S = (\Delta x_S, \Delta y_S, \Delta z_S, \Delta\alpha_S, \Delta\beta_S, \Delta\gamma_S) = (\Delta\boldsymbol{\tau}_S, \Delta\boldsymbol{\theta}_S). \tag{5.7}$$

We transform the Euler angles $\Delta\boldsymbol{\theta}_S$ into the quaternion $\Delta Q\boldsymbol{\theta}_S$ using the standard transformation matrices. We do the same for $\boldsymbol{\theta}_R$ (the Euler angles for the robot's frame of reference) which now becomes $Q\boldsymbol{\theta}_R$. Then, the new location in the robot's frame of reference is

$$\mathbf{v}'_R = (\boldsymbol{\tau}'_R, \boldsymbol{\theta}'_R) = \left((Q\theta_R)(\Delta\boldsymbol{\tau}_S)(Q\theta_R)^* + \boldsymbol{\tau}_R,\ E(H(Q\boldsymbol{\theta}_R, \Delta Q\boldsymbol{\theta}_S))\right), \tag{5.8}$$

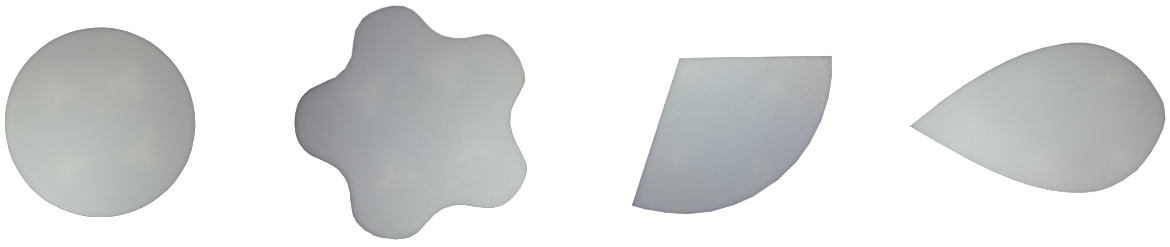
where $(Q\theta_R)^*$ is the conjugate of $Q\theta_R$, $H(\cdot,\cdot)$ is the Hamilton product and E transforms the resulting quaternion back into Euler angles.[1] The time factor was neglected for simplicity in the equation above, but adding in a time component, we see that

$$\mathbf{v}'_R(t) = \mathbf{v}_R(t+1) = f(\mathbf{v}_R(t)) \tag{5.9}$$

which defines the policy for moving the sensor around the unknown object ($f(\cdot)$ is just to show that location $\mathbf{v}_R$ is a function of its previous self). Once a new location was reached, the robot was programmed to tap the TacTip onto the object (or rather where it predicted the edge of the object to be) in order to get the pin data, so that the PDenNN model could make a prediction for x_S and γ_S.

Consider briefly the case in which $\mathbf{s}^*_S = (0,0,0,0,0,0)$. Then any error in x_s and γ_s in $\mathbf{s}_S(\mathbf{m}(t))$ would be brought to zero by the control law implemented above. In other words the centre-bottom of the TacTip would end up on and normal to the edge of an object and then stop; it would not move around the object, because if the error is $\mathbf{0}$ then $\Delta\mathbf{s}_S$ is also zero, and consequently no change to the location $\mathbf{v}_R$ occurs. However, if $\mathbf{s}^*_S$ contains a y^*_S component, the error can never be brought to zero, because y^*_S is not affected by the model's prediction. If the control brings Δx_S and $\Delta\gamma_S$ to zero, then the component y^*_S implies a perfect tangential move of size $|y^*_S|$ mm along the unknown object's edge.

5.2 Experimental Set Up



(a) Disk (b) Flower (c) Trine (d) Teardrop

Figure 5.2: Bird's-eye view of the four 3D printed shapes used for the contour following tasks. The 2D surfaces shown here extend uniformly into the page giving them a 3D body. **(a) - (d)** are the different labels given to these shapes.

[1] Note that all the position vectors presented above are in millimeters and the Euler angles are in degrees (except when they were converted into quaternions, in which case radians were used).

Four objects with different shapes were chosen for the contour following tasks. Each object had a continuous, closed-loop 2D contour. Note that while the contours are 2-dimensional, the objects are three dimensional, so as to provide an edge for the sensor to be tapped onto. The objects were 3D printed using VeroWhite resin plastic, and designed so that they could be rigidly mounted to the work desk of the robotic arm.

These shapes were chosen for the following reasons: the disk, because this was the same shape used for training the dendritic neural network, and therefore gives an indication of how well the network was able to learn; the flower, because of its concave and convex nature, gives an indication of how well the control policy can deal with smooth inflection points; the trine, because of straight and sharp corners, gives an indication of how well the control policy can deal with straight edges, corners and sharp transitions between straight and curved edges; and finally, the teardrop, which gives an indication of how well the control policy can deal with varying curvatures and a 300° turn around a very sharp edge. Note that the performance of the control policy is dependent on the performance of the PDenNN model, and can thus also give us an insight into the network's capabilities and limitations.

Each object was mounted onto the work desk of the robotic arm separately, and the sensor was brought down to the object by the robot to be 2 mm above the objects surface and roughly aligned with an edge. The number of steps necessary for the robot to move the sensor around the object was dependent on the size of the object, and was manually set (after some trial and error). The disk required 266 steps, the flower 272 steps, the trine 222 steps and the teardrop 260 steps.

5.3 Results

The contour following results are presented in Figure 5.3 below. The different shapes are clearly distinguishable, and the trajectories are closed loops, indicating that the sensor did not start to wander off the object, but remained fixed to the object's edge. Given the different features of the objects, the analysis of the results will be broken down; first qualitatively and then quantitatively.

Qualitative Results

- **Disk**: The control policy seemed to work well on the disk object, resulting in only small perturbations away from the object's edge here and there. The small perturbations are likely due to the high sensitivity to the slightest changes in pin displacements, that the network suffered from. Another potential, but likely minimal, cause for perturbations could be slight imperfections in the edge of the disk object.

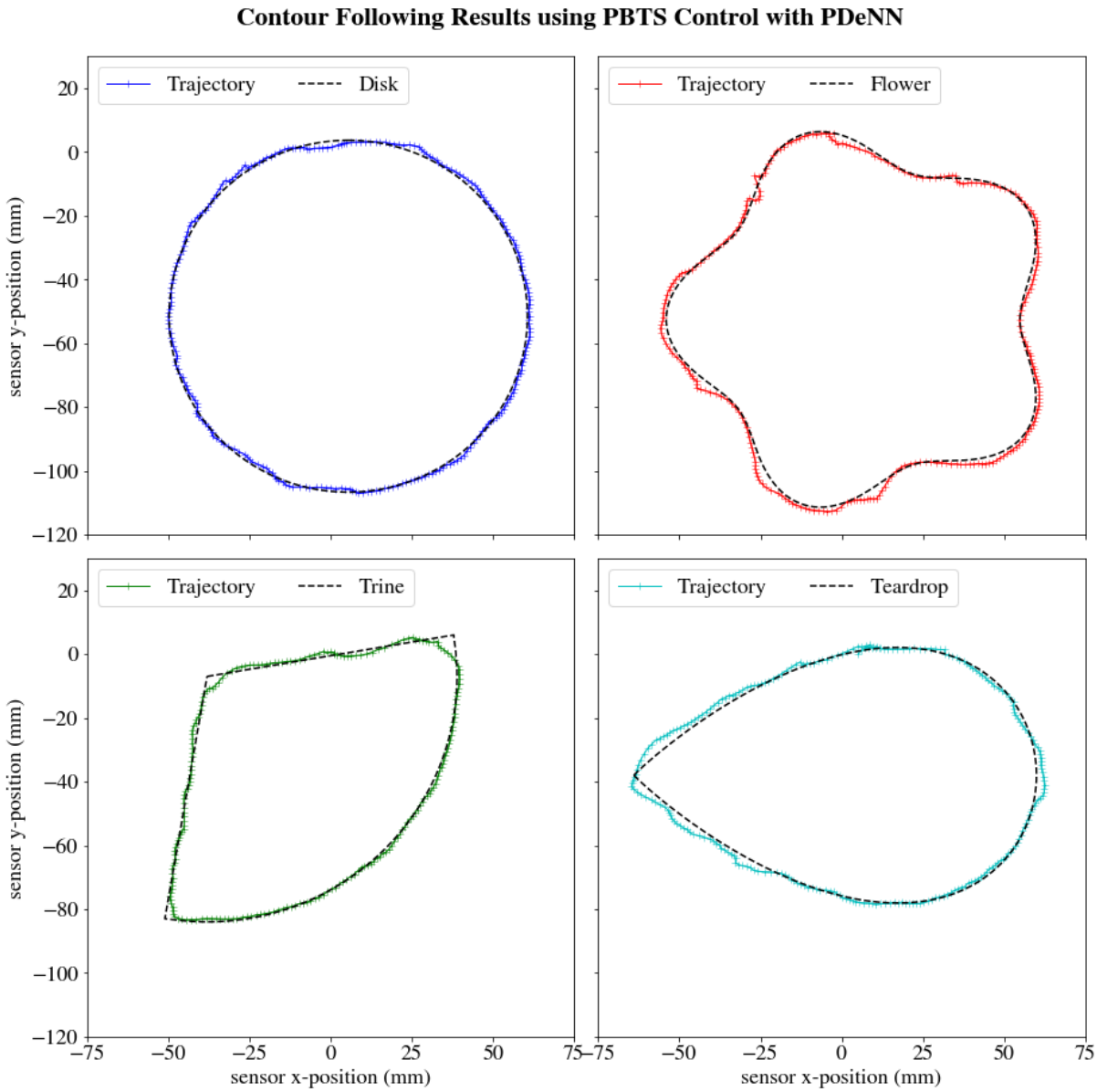


Figure 5.3: The contour following results obtained by extracting the x and y coordinates of the centre-bottom of the sensor (i.e. the x, y components of $v_R(t)$) for each of the different objects using PBTS control with a tailored parallel dendritic neural network (PDenNN). The neural network had been trained on data collected using the disk shape (top left). The trajectories are plotted in colour and the contours of the true shapes are also plotted in striped black. The 'true shapes' were plotted by finding the parametric equations that best fitted the objects: these equations can be found in the Appendix.

- **Flower**: The control policy did not work as well on this shape as it did with the disk. Besides some perturbations, clear deviations away from the object are visible. The deviations occur almost exclusively at the inflection points, where the flower goes from being concave to convex. This could be an indicator that the control policy was causing the sensor to overshoot, because its tangential move was set to be too large, or that the model predicted non-sensible outputs given un-trained pin displacements (different curvatures than the model was trained

for), causing the sensor to deviate. Nevertheless, the sensor always found a way back to the object's edge, showing that the PDenNN model and the control were able to guide the sensor back in the right direction, showing some sense of robustness.

- **Trine**: The control worked well around the curved edge, and fairly well around the straight edges, showing only slight deviations. This indicates that the training of the model on a curved edge can work well for contour following of straight edges. What did not work well however, was moving around the corners; the sensor ended up simply cutting the corners. It is likely that the 'V' shaped imprints on the TacTip's rubber hemisphere, caused by the corners, created pin displacements so unusual to the network, that it did not quite know what to predict.
- **Teardrop**: The control struggled with the curvature of the edge of this object, deviating away from it at almost regular intervals. This too is likely a fault of the tangential step-size, overstepping the change in curvature slightly. As with the trine object, it struggled to remain on the path of the contour around its one, very sharp, 300 degree corner, where it overshot on two occasions.

Quantitative Results

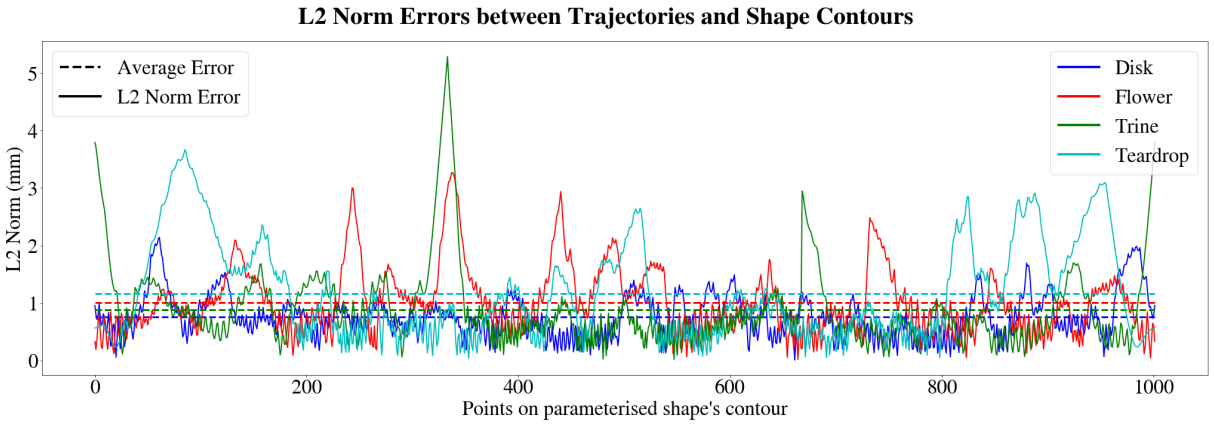


Figure 5.4: The L2 Norm (Euclidean) errors between the trajectories and the parameterised shapes (and their averages). The L2 norm errors were computed by finding the minimum L2 distance from a point on the parameterised shape to all points on its corresponding trajectory. This was done for all, 1000, points that made up the parameterised shape.

Quantitatively, using L2 norms as a measure of discrepancy between the 'true' shape (best-fit parameterised shape) and the trajectories (see Figure 5.4), we can say that, on average, the control policy worked best on the disk object. This is as expected, given that the PDenNN had been trained using data from that exact disk. Next, in order of best to worst, were its performances on the trine object, flower object and teardrop object. According to this measure, averages of 0.75 mm (disk), 0.89 mm (trine), 1.01 mm (flower), 1.16 mm (teardrop) were achieved.

The discrepancy between achieving average accuracies of 0.38mm on the validation set, but average perturbations of 0.75mm for the contour following around the disk is of interest. It could be that the tangential move i.e. $|y_S^*| = 1$ was perhaps slightly too large, causing the sensor to overshoot a little too much. Alternatively, it could also be the sensitivity of the model. The disk changes by 0.51 degrees if one moves tangentially on the edge of the disk (causing an error of 0.001 in position, so not much of an overshoot), and on inspection of the PDenNN results for the validation set (see Figure 4.4 in Chapter 3), we find that the model predicts angles that are on average somewhat higher than they should be around 0.51 degrees, causing the sensor to move into or away from the object slightly more than it should. Another cause could be the choice in controller gain parameters (see Equation 5.6). Perhaps these were set slightly to high.

The quantitative results also give insight into how the system reacted to the different shapes. Consider the width and the number of spikes above 2 mm in Figure 5.4. Looking at these, we can identify the extreme cases where the sensor left a contour, for how long, and how often it occurred for a particular shape. We find that: 1) for the trine, there are a total of three narrow, sharp-looking spikes, two of which are larger than any other spike, 2) for the flower, there are four, smaller, sharp-looking spikes, 3) for the teardrop, there are six, wider-looking spikes, and 4) for the disk, there are no spikes present above 2 mm. From this, we infer that the system found it harder to move the sensor around the flower shape than the trine shape, but in both cases the sensor was able to quickly return to the contour. We further infer that the shape the system found hardest to deal with was the teardrop. The number and width of the spikes clearly indicate that the system wandered off the object's contour for longer periods of time. We believe the reason for this is that the gain parameters and the step-size were nonoptimal for the changing curvature that the teardrop shape presented.

5.4 Summary of Results

We found that the bio-inspired system developed in this paper:

- was able to move the tactile sensor around novel 2D contours, having only been trained on a circular edge;
- struggled with sharp 'V' shaped corners and points of inflection in curvature, which was likely caused by the network not being able to appropriately interpret the x and y displacements of those types of contours;
- slightly overshot the contours of the objects, especially when the curvature changed rapidly (like for the teardrop). We concluded that this was likely caused by the gain parameters and the step-size not being set appropriately.

6. Conclusion

The purpose of this report was to design a bio-inspired system that mimicked the interaction between the human brain and its fingers in order to follow the contours of unknown objects. To achieve this we used a dendritic neural network in combination with a 2D tactile servoing control policy, a tactile sensor and a robotic arm.

Starting with the tactile sensor, we showed how it works, and why it is bio-inspired. In terms of contour following, it was an ideal sensor as it allowed one to determine the distribution of force on the tactile sensor. This in turn allowed for edge perception, which meant we were able to compute where the sensor was relative to an edge at any given moment in time.

Next we looked at the neuroscience behind dendritic neural networks, and then moved onto explaining how they work mathematically. We showed that dendritic neural networks add another dimension of biological realism to fully connected artificial neural networks by allowing for a localised backpropagation of error. We established that in steady state they behave identical to artificial neural networks, and that they allow for a single computational circuit, using equilibrium propagation, whereas artificial neural networks require two, using reverse auto-differentiation.

Three different single hidden-layer dendritic neural networks were designed: the DisDenNN, PCADenNN, and the PDenNN. We found that reducing the input size by applying PCA can help improve learning and that the angle and position targets can be learned separately and effectively when PCA is applied to the tactile data. Since the PDenNN performed the best on average, it was selected as the model to be used in the control policy for the contour following tasks, thereby completing the bio-inspired system.

Despite slight deviations in the contour following, the bio-inspired system managed to get the tactile sensor to move around all objects. Given that three of the shapes had novel curvatures (i.e. curvatures that the PDenNN model had not been trained on), this was a surprisingly good result. It showed that dendritic neural networks can be used for tactile contour following tasks, and that training them on data collected on a disk allows them to work well on un-trained contours, indicating that the bio-inspired system has a sense of adaptability and robustness.

To conclude: this paper presented the first ever dendritic neural network used to model tactile data, and it has shown that in combination with PBTS control these networks can be used effectively for tactile servoing. By using dendritic neural networks as the 'brain' of the system, we created the most biologically realistic and human-like system for tactile contour following to date. We hope that this paper inspires further work into dendritic neural networks and encourages the field of tactile robotics to continue to look towards human and animal biology to design effective systems in order to advance the field of robotics as a whole.

6.1 Further Work

Dendritic Neural Networks

- It would be interesting to explore deep convolutional dendritic neural networks that, instead of using pin displacements, use the entire image captured by the camera inside the TacTip. This would reduce the number of data processing steps required, as one would no longer need to apply computer vision to extract the pin locations.
- One of the issues we faced in designing the dendritic neural networks in this paper was choosing optimal hyperparameters (learning rate, nudging parameter, input dimension, etc...). We used a trial and error approach and therefore could not have known if we had used the best possible combination of hyperparameters. While the trial and error method resulted in an effective model for contour following, a parameter optimisation policy (such as a Bayesian hyperparameter optimisation method used in [13]) could potentially further improve the results.

PBTS Control Policy

- The controller gain and step-size parameters could be better optimised. One could simulate different contours and then optimise the parameters according to the robot's performance in the simulations.
- A derivative term could be added to the PI controller. This could help prevent the sensor from overshooting on certain contours.

System Advancements

- While the system was bio-inspired, it could be made even more human-like. TacTip sensors could be designed to fit the ends of mechanical fingers on a robotic hand. The robotic hand could then be mounted to a robotic arm, instead of just a single TacTip on its own. This would allow one to perform 3-dimensional contour following tasks around objects that the robotic arm is able to hold itself: such as tennis ball, or a set of keys.

6.2 Acknowledgements

Thank you to Nathan Lepora for supervising my project; to Alex Church for helping me learn how to control the robot; to John Lloyd for helping me understand how to use computer vision on the TacTip; to Rui Ponte Costa for helping me understand dendritic neural networks and Heng Wei Zhu for helping me understand equilibrium propagation.

Appendix

Parametric equations for the various shapes used in the contour following tasks:

Disk:

$$x = 55.2\cos(t) + 5.5, y = 55.2\sin(t) - 51.5, t \in [0, 2\pi]$$

Flower:

$$x = 1.09\left(50 + 6\sin(5t - \frac{\pi}{2})\right)\cos(t) + 7, y = 1.09\left(50 + 6\sin(5t - \frac{\pi}{2})\right)\sin(t) - 52.5, t \in [0, 2\pi]$$

Trine:

$$x = \begin{cases} 76t/1.41 - 37 & 0 \le t < 1.41 \\ 76\sin(t) - 37 & 1.41 \le t \le 3.27 \\ 13t/1.41 - 50 & 0 \le t \le 1.41 \end{cases}, y = \begin{cases} 13t/1.41 - 8 & 0 \le t < 1.41 \\ 76\cos(t) - 8 & 1.41 \le t \le 3.27 \\ 76t/1.41 - 84 & 0 \le t \le 1.41 \end{cases}$$

Teardrop

$$x = -62 * \cos(t) - 2, y = 51\sin(t)\sin\left(\frac{t}{2}\right)^{0.9} - 38, t \in [0, 2\pi]$$

The computer vision and PDenNN parameters used for the contour following tasks. For more information on how CVBlobDetector works, see [61].

Pin Tracking		**Parallel Dendritic Neural Network**	
CVBlobDetector Parameters		Network Parameters	
Min Threshold	82.39	Position r	
Max Threshold	215.98	Input Layer Dimension	30
Filter by Color	True	Hidden Layer Dimension	500
Blob Color	255	Output Layer Dimension	1
Filter by Area	True	Learning Rate	0.001
Min Area	82.39	Beta	1.2
Max Area	190.96	Angle θ	
Filter by Circularity	True	Input Layer Dimension	35
Min Circularity	0.44	Hidden Layer Dimension	1000
Filter by Inertia	True	Output Layer Dimension	1
Min Inertia Ratio	0.27	Learning Rate	0.001
Filter by Convexity	True	Beta	0.5
Min Convexity	0.57	Epochs trained (both)	200

The Equilibrium Propagation Parameters:

Equilibrium Propagation	
Integration Method	Euler
Step-size Δt	1
Number of Integration Loops n_I	5

Bibliography

[1] João Sacramento et al. “Dendritic cortical microcircuits approximate the backpropagation algorithm”. In: *Advances in Neural Information Processing Systems* 2018-Decem.October (2018), pp. 8721–8732.

[2] *Universal Robots UR5 | Cobot Webshop*. URL: https://cobotwebshop.com/ur5?gclid=Cj0KCQjw1Iv0BRDaARIsAGTWD1vgOnaT_btHDEWnvB9cB_HSpIHJ82NC7AjcJvrA9pU6Z4XGbR25X24aAk5eEALw_wcB.

[3] Craig Chorley et al. “Development of a tactile sensor based on biologically inspired edge encoding”. In: *2009 International Conference on Advanced Robotics, ICAR 2009* (2009), pp. 1–6.

[4] Nathan F Lepora and John Lloyd. “Pose-Based Tactile Servo Control using Deep Learning”. In: (2016), pp. 1–16.

[5] Roland S. Johansson and J. Randall Flanagan. “Coding and use of tactile signals from the fingertips in object manipulation tasks”. In: (2009). URL: www.nature.com/reviews/neuro.

[6] Leon D. Harmon. “Automated Tactile Sensing”. In: *The International Journal of Robotics Research* 1.2 (June 1982), pp. 3–32. URL: http://journals.sagepub.com/doi/10.1177/027836498200100201.

[7] *Tactile Robotics*. URL: https://www.bristolroboticslab.com/tactile-robotics.

[8] *Why your new work colleague could be a robot - BBC News*. URL: https://www.bbc.co.uk/news/business-51442445.

[9] *Robots Are Displacing Manual Labor Jobs | Design News*. URL: https://www.designnews.com/automation-motion-control/robots-are-displacing-manual-labor-jobs/179405531258230.

[10] *The future of manual labor: no people, just robots? | ZDNet*. URL: https://www.zdnet.com/article/the-future-of-manual-labor-no-people-just-robots/.

[11] *As Robots Threaten More Jobs, Human Skills Will Save Us*. URL: https://www.forbes.com/sites/mohanbirsawhney/2018/03/10/as-robots-threaten-more-jobs-human-skills-will-save-us/#40087eb83fce.

[12] Benjamin Scellier and Yoshua Bengio. “Equilibrium propagation: Bridging the gap between energy-based models and backpropagation”. In: *Frontiers in Computational Neuroscience* 11 (2017), pp. 1–13.

[13] Nathan F. Lepora et al. “From Pixels to Percepts: Highly Robust Edge Perception and Contour Following Using Deep Learning and an Optical Biomimetic Tactile Sensor”. In: *IEEE Robotics and Automation Letters* 4.2 (2019), pp. 2101–2107.

[14] Nathan F. Lepora et al. “Exploratory Tactile Servoing with Active Touch”. In: *IEEE Robotics and Automation Letters* 2.2 (Apr. 2017), pp. 1156–1163.

[15] Matthew Fulkerson. “Touch”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N Zalta. Spring 2016. Metaphysics Research Lab, Stanford University, 2016.

[16] *Sensibles | Encyclopedia.com*. URL: https://www.encyclopedia.com/religion/encyclopedias-almanacs-transcripts-and-maps/sensibles.

[17] Eric R Kandel et al. *Principles of neural science*. McGraw-Hill, 2000.

[18] Amanda Zimmerman et al. “of Mammalian Skin”. In: *Science* 346.6212 (2014), pp. 950–954.

[19] *How the Peripheral Nervous System Works*. URL: https://www.verywellmind.com/what-is-the-peripheral-nervous-system-2795465.

[20] *Unimate - The First Industrial Robot*. URL: https://www.robotics.org/joseph-engelberger/unimate.cfm.

[21] Hiromi Takahashi-Iwanaga and Hiroshi Shimoda. “The three-dimensional microanatomy of Meissner corpuscles in monkey palmar skin”. In: *Journal of Neurocytology* 32.4 (2003), pp. 363–371.

[22] Tareq Assaf et al. “Seeing by touch: Evaluation of a soft biologically-inspired artificial fingertip in real-time active touch”. In: *Sensors (Switzerland)* 14.2 (2014), pp. 2561–2577.

[23] Nathan F. Lepora and Benjamin Ward-Cherrier. “Tactile Quality Control with Biomimetic Active Touch”. In: *IEEE Robotics and Automation Letters* 1.2 (2016), pp. 646–652.

[24] Jasper Wollaston James et al. “Slip detection with a biomimetic tactile sensor”. In: *IEEE Robotics and Automation Letters* 3.4 (2018), pp. 3340–3346.

[25] Benjamin Ward-Cherrier et al. “The TacTip Family: Soft Optical Tactile Sensors with 3D-Printed Biomimetic Morphologies”. In: *Soft Robotics* 5.2 (2018), pp. 216–227.

[26] Zhanat Kappassov et al. “Tactile sensing in dexterous robot hands-Review”. In: *Robotics and Autonomous Systems* 74 (2015), pp. 195–220. URL: https://hal.uca.fr/hal-01680649.

[27] Xuefeng Wang et al. *Development of polyimide flexible tactile sensor skin*. Tech. rep. 2003, pp. 359–366.

[28] Citation Tenzer et al. “The Feel of MEMS Barometers: Inexpensive and Easily Customized Tactile Array Sensors”. In: *IEEE Robot. Automat. Mag* 21.3 (2014), pp. 89–95. URL: http://nrs.harvard.edu/urn-3:HUL.InstRepos:22088984.

[29] Guanhao Liang et al. “Real-time measurement of the three-axis contact force distribution using a flexible capacitive polymer tactile sensor”. In: *J. Micromech. Microeng* 21 (2011), p. 35010.

[30] Lucia Beccai et al. “Development and experimental analysis of a soft compliant tactile microsensor for anthropomorphic artificial hand”. In: *IEEE/ASME Transactions on Mechatronics* 13.2 (2008), pp. 158–168.

[31] Lucia Seminara et al. “Electromechanical characterization of piezoelectric PVDF polymer films for tactile sensors in robotics applications”. In: *Sensors and Actuators, A: Physical* 169.1 (Sept. 2011), pp. 49–58.

[32] *TakkTile Sensors | Soft Robotics Toolkit*. URL: https://softroboticstoolkit.com/book/takktile-sensors.

[33] *BioTac - ROS Wiki*. URL: http://wiki.ros.org/BioTac.

[34] *GelSight – Portable, non-destructive elastomeric 3D imaging systems*. URL: https://www.gelsight.com/.

[35] Kazuto Kamiyama et al. *Evaluation of a Vision-based Tactile Sensor*. Tech. rep.

[36] Nathan F. Lepora and Benjamin Ward-Cherrier. "Superresolution with an optical tactile sensor". In: *IEEE International Conference on Intelligent Robots and Systems* 2015-Decem (2015), pp. 2686–2691.

[37] Katsunari Sato et al. "Finger-shaped GelForce: Sensor for measuring surface traction fields for robotic hand". In: *IEEE Transactions on Haptics* 3.1 (2010), pp. 37–47.

[38] Micah K. Johnson and Edward H. Adelson. "Retrographic sensing for the measurement of surface texture and shape". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition* (2010), pp. 1070–1077.

[39] Pedro Silva Girão et al. "Tactile sensors for robotic applications". In: *Measurement* 46 (2013), pp. 1257–1271. URL: http://dx.doi.org/10.1016/j.measurement.2012.11.015.

[40] Calum Roke et al. "Lump localisation through a deformation-based tactile feedback system using a biologically inspired finger sensor". In: *Robotics and Autonomous Systems* 60.11 (2012), pp. 1442–1448. URL: http://dx.doi.org/10.1016/j.robot.2012.05.002.

[41] Yousef Al-Handarish et al. "A Survey of Tactile-Sensing Systems and Their Applications in Biomedical Engineering". In: (2020). URL: https://doi.org/10.1155/2020/4047937.

[42] Pedro Silva Girão et al. *Tactile sensors and their use in industrial, robotic and medical applications*. Tech. rep. 2007.

[43] Warren S. McCulloch and Walter H. Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *Systems Research for Behavioral Science: A Sourcebook* 5 (1943), pp. 93–96.

[44] Blake A. Richards et al. "A deep learning framework for neuroscience". In: *Nature Neuroscience* 22.11 (2019), pp. 1761–1770.

[45] MiglioreI Rosanna et al. "The physiological variability of channel density in hippocampal CA1 pyramidal cells and interneurons explored using a unified datadriven modeling workflow". In: (2018), p. 4.

[46] David Balduzzi et al. "Kickback cuts backprop's red-tape: Biologically plausible credit assignment in neural networks". In: *Proceedings of the National Conference on Artificial Intelligence* 1 (2015), pp. 485–491.

[47] Randall C. O'Reilly. "Biologically Plausible Error-Driven Learning Using Local Activation Differences: The Generalized Recirculation Algorithm". In: *Neural Computation* 8.5 (1996), pp. 895–938.

[48] Xiaohui Xie and H Sebastian Seung. "Persistent Neural Activity". In: *Nips* (2000).

[49] Timothy P. Lillicrap et al. "Random synaptic feedback weights support error backpropagation for deep learning". In: *Nature Communications* 7 (2016), pp. 1–10. URL: http://dx.doi.org/10.1038/ncomms13276.

[50] James C R Whittington and Rafal Bogacz. "Theories of Error Back-Propagation in the Brain". In: *Trends in Cognitive Sciences* 23.3 (2019), pp. 235–250. URL: https://doi.org/10.1016/j.tics.2018.12.005.

[51] Alan D Berger. *CARNEGIE MELLON Department of Electrical and Computer Engineering˜ On Using a Tactile Sensor for Real-Time Feature Extraction*. Tech. rep.

[52] N. Chen et al. "Edge tracking using tactile servo". In: *IEEE International Conference on Intelligent Robots and Systems* 2 (1995), pp. 84–89.

[53] *A control framework for tactile servoing*. Tech. rep.

[54] Sen Song et al. "Highly Nonrandom Features of Synaptic Connectivity in Local Cortical Circuits". In: (). URL: www.plosbiology.org.

[55] *Biological neuron model - Wikipedia*. URL: https://en.wikipedia.org/wiki/Biological_neuron_model#Leaky_integrate-and-fire.

[56] For Ben. "Theory of prediction-error propagation in cortical microcircuits Neuronal dynamics". In: (2017), pp. 1–6.

[57] *Hopfield network - Wikipedia*. URL: https://en.wikipedia.org/wiki/Hopfield_network.

[58] *NumPy — NumPy*. URL: https://numpy.org/.

[59] Nathan F Lepora and John Lloyd. *Optimal Deep Learning for Robot Touch*. Tech. rep.

[60] Institute of Electrical and Electronics Engineers. "Shear-invariant Sliding Contact Perception with a Soft Tactile Sensor". In: ().

[61] *Blob Detection Using OpenCV (Python, C++) | Learn OpenCV*. URL: https://www.learnopencv.com/blob-detection-using-opencv-python-c/.