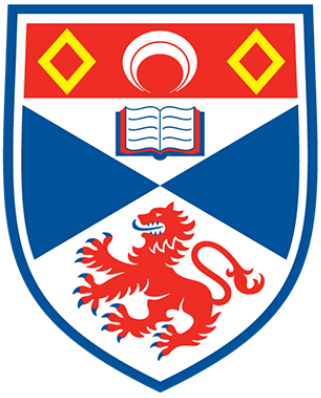
Senior Honours Project



University of
St Andrews

# Freeing Neural Training Through Surfing

*Author:*
Georg Wölflein

*Supervisor:*
Dr. Michael Weir

April 27, 2020

# Abstract

Gradient methods based on backpropagation are widely used in training multilayer feedforward neural networks. However, such algorithms often converge to suboptimal weight configurations known as local minima. This report presents a novel minimal example of the local minimum problem with only three training samples and demonstrates its suitability for investigating and resolving said problem by analysing its mathematical properties and conditions leading to the failure of conventional training regimes. A different perspective for training neural networks is introduced that concerns itself with neural spaces and is applied to study the local minimum example. This gives rise to the concept of setting intermediate subgoals during training which is demonstrated to be a viable and effective means of overcoming the local minimum problem. The versatility of subgoal-based approaches is highlighted by showing their potential for training more generally. An example of a subgoal-based training regime using sampling and an adaptive clothoid for establishing a goal-connecting path is suggested as a proof of concept for further research. In addition, this project includes the design and implementation of a software framework for monitoring the performance of different neural training algorithms on a given problem simultaneously and in real time. This framework can be used to reproduce the findings of how classical algorithms fail to find the global minimum in the aforementioned example.

# Declaration

"I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated.

The main text of this project report is 19960 words long, including project specification and plan.

In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work."

*Georg Wölflein*

# Acknowledgements

TODO

# Contents

# List of figures

# Notation

| Notation | Description |
|---|---|
| $N$ | Number of training samples |
| $D$ | Number of input features |
| $x_i$ | The $i$th input feature ($1 \leq i \leq D$) |
| $\mathbf{x}_i$ | Input feature vector of the $i$th training sample ($1 \leq i \leq N$) |
| $\mathbf{X}$ | Input matrix ($N \times D$) |
| $y_i$ | Output target for the $i$th training sample ($1 \leq i \leq N$) |
| $\hat{y}_i$ | Output prediction of the $i$th training sample ($1 \leq i \leq N$) |
| $\mathbf{y}$ | Output target vector ($N$-dimensional) |
| $\hat{\mathbf{y}}$ | Output prediction vector ($N$-dimensional) |
| $w_i$ | The $i$th weight |
| $\mathbf{w}$ | Weight vector |
| $\mathbf{W}$ | Weight matrix |
| $b$ | Bias term |
| $\mathbf{b}$ | Bias vector |
| $\mathscr{P}$ | Trainable parameters (weights and biases) |
| $z$ | Excitation |
| $g(\cdot)$ | Activation function |
| $\phi(\cdot)$ | Radial basis function |
| $R(\cdot)$ | Regression model |
| $S(\cdot)$ | Single-layer network with one output |
| $\mathbf{S}(\cdot)$ | Single-layer network with multiple outputs |
| $M(\cdot)$ | Multi-layer perceptron |
| $\mathcal{I}$ | Input space |
| $\mathcal{W}$ | Weight space |
| $\mathcal{O}$ | Output space |
| $\mathcal{O}(\cdot)$ | Big O notation |
| $L$ | Loss |
| $\mathbf{J}$ | Jacobian |
| $\mathbf{H}$ | Hessian |
| $\lvert \cdot \rvert$ | Absolute value |
| $\lVert \cdot \rVert$ | Euclidean (L2) norm |
| $\mathbb{R}$ | The set of real numbers |
| $\measuredangle$ | Angle |
| $\triangle$ | Triangle |

Scalar values are generally denoted $v$, vectors $\mathbf{v}$, and matrices $\mathbf{V}$. Vector-valued functions are denoted in bold font.

# Chapter 1

# Introduction

This project is primarily a research-oriented project. That is, it is a project investigating how research on the neural manifestation of a classical problem in numerical optimisation and heuristic search may be set up and progressed. More specifically, its main focus lies in contriving and investigating an example of the local minimum problem in neural networks[1] and designing a training regime that tackles this issue. Of auxiliary importance is a software framework that should be developed to not only to compare different neural training techniques, but also to ensure that the main theoretical results obtained regarding the local minimum problem are empirically verified and reproducible.

## 1.1 Motivation

Imagine you are hiking up a mountain. The trail in front of you leads gradually upwards, and in the distance, you can spot a peak. After hiking for a while, you reach the top that peak, but then you realise it was not in fact the summit because now (from this new vantage point) you can see a small basin in front of you that leads to an even higher peak. This process repeats until at some point the peak up ahead will have the summit cross – to your great relief. Then it is just a matter of hiking across that final valley and up to the summit.

You could only ever see the next peak because it obstructed the view of subsequent peaks until you climbed it. This is an example of *local information*: without prior knowledge or other aid there is no way of knowing if the peak in front is the penultimate one. In fact, in the absence of trails and signposts, how would you know which way to go, if you cannot see the summit cross?

Let us continue with this thought experiment. Assume you could only see a distance of one metre around you, and that is the only local information available to you. How would you find the summit? The most natural approach is just repeatedly taking a step in the direction of the steepest upwards slope. This method is called *gradient ascent* (the gradient is just the slope).

At some point, you will reach the highest point in the one-metre vicinity which means that you are at the top of a peak. We call this a *local optimum* because all points in the local neighbourhood are of lower elevation. If you

[1] Throughout this report, the terms *neural network* and *artifical neural network* (ANN) are used interchangeably.

are standing at the summit cross, then you are very lucky and have found the highest point of the mountain. In this case, your position marks the *global optimum* because you are higher than all the other peaks this mountain has.

How do you proceed once you are at a local optimum? When you have already reached it, your only option is to randomly move in any direction because you have no useful information. A more principled approach might be to throughout the process sometimes allow steps that go downwards in hopes of increasing the likelihood of finding a higher peak. This is essentially the idea behind *simulated annealing* and related techniques. However, while this technique might find a better local optimum, it is still quite unlikely to find the global optimum. This is what we call the *local optimum problem*.

Another point to consider is that if the summit did not have a cross, it would be impossible to know whether a given peak is actually the global optimum using only the local information.

To relate this thought experiment to numerical optimisation problems, let us turn it upside down: instead of trying to locate the highest peak, the goal is now to find the deepest valley, i.e. the lowest point in the landscape. The same logic still applies, except that not we think of an optimum as being a *minimum* instead of a *maximum*, and the technique is called *gradient descent*. The reason for applying this transformation is that aim of a numerical optimisation problem is to minimise a *cost function* for a set of parameters. The surface of this function is like a mountainous landscape, and in finding its global minimum, we run into the aforementioned problem of local optima which we in this context is called the *local minimum problem*.

This project will develop a technique that differs from gradient descent and simulated annealing (and other probabilistic techniques) by using a chain of intermediate targets (subgoals) along the cost surface to pull the process forwards analogous to a *surfer* using a travelling wave to drive their motion.

This report will begin with a context survey identifying not only the classical neural training algorithms, but also the efforts that have been made to deal with the local minimum problem and their limitations. The classical theory of neural networks is presented from the ground up in Chapter 2 which naturally leads to the analysis of popular neural training techniques in Chapter 3. In Chapter 4, a different paradigm of viewing neural training – 'neural surfing theory' – is introduced. Chapter 5 provides a precise definition of the local minimum problem, thereafter facilitating the design of an instance of said problem which is then analysed from both viewpoints described in Chapters 2 and 4. The subsequent chapter lays out the neural surfing technique for undertaking the local minimum problem. Finally, an outlook is provided how this type of technique can be generalised to other problems before finally presenting the software framework and an evaluation of the project.

## 1.2 Context survey

### 1.2.1 Neural training

The first mathematical model representing neurons in the human brain, so-called *perceptrons*, was formulated by McCulloch and Pitts [1943] (see Section 2.2). In 1958, the psychologist Frank Rosenblatt published the first perceptron learning

algorithm [Rosenblatt 1958], but this type of network lacked the ability to learn mappings that were not *linearly separable*. It was not until the 1980s with the introduction of the backpropagation algorithm capable of training networks with hidden layers, that neural networks experienced a substantial rise in popularity.

**Backpropagation** The backpropagation (BP) algorithm – discovered independently by multiple researchers in the 1960s and popularised for neural networks by Rumelhart et al. [1986] – remains the prominent method of training neural networks to this date. It involves computing the derivative of the loss function with respect to the weights and then using some gradient-based optimisation technique (gradient descent or an approximation[2] thereof) to update the weights. With the rise in popularity of deep neural networks, methods have been developed to accelerate training and allow more complex networks to feasibly learn more complex problems. Two main approaches are parallelising the computation and using adaptive learning rates like in the 'Adam optimizer' [Kingma and Ba 2014]. It is well-established that BP, provided a suitable choice of hyperparameters, is guaranteed to converge to a local (but likely not global) minimum. A common technique to subdue the effect of this issue is to run BP multiple times with different random initialisations.

**Derivative-free optimisation** The class of derivative-free optimisation (DFO) algorithms are optimisation techniques that attempt to find a global optimum, requiring only an objective function, but no derivative information. One example of such an algorithm is simulated annealing (SA), proposed by Kirkpatrick et al., that mimics the motion of atoms in the physical process of a slowly cooling material [1983]. Originally employed in discrete optimisation problems such as the combinatorial travelling salesman problem [Černý 1985], it was later generalised and applied to problems with continuous domains [Bélisle et al. 1993]. However, in a comparative study of derivative-free optimisation algorithms, Rios and Sahinidis found that SA performed relatively poorly in comparison to more modern DFOs on general optimisation problems[3] [2009].

The concept of applying DFO as a means of training neural networks is not unique to this project. In the 1990s, several training regimes for neural networks were proposed that did not rely on derivative calculations, employing variants of random and local search [Battiti and Tecchiolli 1995; Hirasawa et al. 1998]. These approaches seemed to find better minima in some settings and did not get stuck in local minima as BP did. More recently, a particular random search approach was affirmed in outperforming BP in the context of deep neural networks for reinforcement learning, although a different family of DFO algorithms, so-called genetic algorithms were proposed as a superior alternative [Such et al. 2017].

A very recent work presents a DFO technique for neural networks that uses a variant of local search belonging in the family of random search algorithms [Aly et al. 2019]. This technique parallels the finding from other works that DFOs

[2]Stochastic gradient descent (SGD) is often used as an approximation for gradient descent whereby the gradient is calculated on a random subset of the data (instead of the whole dataset) for computational efficiency.

[3]It is important to note that Rios and Sahinidis did not assess DFOs for the purpose of neural network optimisation, but rather compared their performances on general convex and non-convex optimisation problems.

are often able to escape some[4] local minima and thus produce better training results; however, they require more iterations and computational resources than BP.

Aly et al., Such et al., and similar works studied the performance of their respective DFO algorithms for training neural networks with a large parameter space (in the order of $10^6$ parameters). Although providing valuable practical insight, this made it infeasible to examine the structure of the loss surface analytically in order to assess issues such as severely suboptimal local minima.

A common theme underlying DFO algorithms is the promise that near-optimal solutions can be found given enough resources. To achieve this, they rely on stochasticity in some form or another: Kirkpatrick et al.'s SA algorithm tolerates suboptimal moves with a certain probability, and the others employ variants of random search as part of their algorithms [Battiti and Tecchiolli 1995; Hirasawa et al. 1998; Such et al. 2017; Aly et al. 2019].

### 1.2.2 The local minimum problem

The local minimum problem, which arises when an algorithm converges to a suboptimal local minimum with a comparatively high loss value, has been extensively studied as a phenomenon in optimisation problems. However, with regards to neural networks, there seems to be differing opinions on the severity of this issue. One frequently cited article claims that "In practice, poor local minima are rarely a problem with large networks" [LeCun et al. 2015]. This is underpinned in theory by other works which proved the nonexistence of suboptimal local minima, although they make varying assumptions on the structure of the underlying neural networks [Kawaguchi 2016; Laurent and von Brecht 2018; Nguyen et al. 2018]. On the other hand, a recent article asserts that "The apparent scarcity of poor local minima has lead practitioners to develop the intuition that bad local minima [...] are practically non-existent" [Goldblum et al. 2019]. The neural local minimum problem remains an active area of research.

There have been various approaches attempting to overcome the local minimum problem as it relates to neural training. Choi et al. presented a method whereby the network is split into two separate parts that are trained separately, but this technique works only on networks with one hidden layer [2008]. Lo et al. followed a different approach through which the mean squared error function is modified in order to 'convexify' the error-weight surface [2012; 2017]. This is achieved using a "risk-averting criterion" that should decrease the likelihood of training samples being underrepresented, but the claim is only to find better local minima as opposed to global ones.

The local minimum problems has been investigated here in St Andrews as well. One particularly promising approach seems to be setting subgoals on the goal path. However, setting these subgoals requires some finesse. Lewis and Weir [1999] show that simply employing a linear chain of subgoals (such as in Gorse et al. [1997]) does not suffice in reliably finding the global minimum, but instead a non-linear chain of subgoals is required. A technique of setting and achieving subgoals that does not rely on BP has been explored in Weir, Lewis et al. [2000].

[4]Guaranteed convergence to a global minimum in every scenario is not asserted, although the results indicate that the local minima are not as 'poor'.

### 1.2.3 Implementation tools

This project will adopt Python as the main programming language. In both acedemia and industry, Python is the de facto standard programming language for machine learning. A 2019 analysis on the world-leading software development platform GitHub found that Python is the most popular language for open source machine learning repositories [Elliott 2019]. Python is a simple yet versatile language that natively supports different programming paradigms (imperative, functional, object-oriented, and more). It is often called an interpreted language[5] because it is dynamically typed and performs automatic memory management (garbage collection) which generally facilitates shorter code than compiled languages such as C or Java, but also means that pure-Python implementations of data-intensive algorithms will usually not be as efficient. One of the most fundamental packages, NumPy, implements very efficient array manipulation operations that, although specified in Python, are carried out at a lower level for performance.

NumPy is just one piece of Python's rich ecosystem of packages that are maintained by open-source contributers in the scientific and engineering community. The two main frameworks for machine learning are TensorFlow by Google and PyTorch by Facebook. At their core, both frameworks facilitate the computation of mathematical operations on tensors, offering support for hardware acceleration via *graphics processing units* (GPUs) and providing parallelisation strategies for distributed computing which is especially potent in the context of machine learning where many operations fit the *single instruction, multple data* (SIMD) pattern. A TensorFlow program is specified as a directed *computational graph* where nodes represent operations and edges represent their inputs and outputs (data tensors) [Martín Abadi et al. 2015]. In the new TensorFlow 2, this graph does not need to be explicitly constructed anymore but is created on the fly which is known as *eager execution*, thereby providing the user with a simpler API similar to NumPy. The slightly younger PyTorch framework provided dynamic computation graphs and a NumPy-like interface since its initial release in 2016, and more recently added support for static computational graphs. Hence the newest versions of both frameworks provide similar computational capabilities. They also facilitate the automatic computation of gradients which is useful for training neural networks. TensorFlow is one of the most popular repositories on GitHub, and PyTorch's popularity is rapidly growing [GitHub 2019].

Keras is a neural network library for Python which is conceived as a high-level interface rather than a framework. It provides implementations of, and abstractions over, common components of neural networks such as layers, optimisers, and activation functions. TensorFlow 2 adopted Keras as part of its core library so that the abstractions provided by Keras can easily be used with the TensorFlow backend.

One should not overlook the concept of interactive notebooks made possible by Python's interpreted nature – that is, mixing rich text (markdown), Python code, and its output. Not only does this allow the programmer to make changes to the code without having to rerun the program, but it also provides a means of presenting Python code in a more engaging way than just comments. Any

[5] There is nuance associated with this statement, but Python certainly exhibits more traits of an interpreted than a compiled language.

type of Python output, including data visualisations, can be presented in such notebooks which makes it attractive for machine learning. These notebooks can be created using the Jupyter package or even run online in the with services such as Google Colab.

## 1.3 Objectives

The initial objectives formulated in the DOER document evolved significantly which is owed to the research-heavy nature of the project and expected. Throughout the course of this project, as a better understanding of the theoretical aspect of neural surfing was attained by research and experimentation, the objectives were adapted. The refined primary objectives are enumerated below in order of decreasing importance.

1. Contrive a minimalist version of the stripe problem and show that it provides a strong basis for investigating and resolving the suboptimal local minimum problem for neural networks.
2. Investigate goal-connecting paths for this problem and design a "neural surfer" that attempts to find such a goal-connecting path.
3. Design a generic framework with a well-defined interface for implementing different gradient-based and derivative-free neural training algorithms and implement such algorithms.
4. For this framework, implement a tool that facilitates the comparison of neural training algorithms on a given problem (dataset) by visualising arbitrary user-specified metrics (including weight and output trajectories) during training in real time.

In addition, a secondary objective for this project was identified:

1. Investigate how the neural surfer can be generalised to more complex problems.

## 1.4 Requirements specification

The requirements below were formulated for the software framework aspect of the project based mainly on the third and fourth objectives from the previous section. It is assumed that the user of the framework is familiar with Python, TensorFlow, and Keras.

1. The framework should be written in Python 3 and use the TensorFlow 2 library for neural computation.
2. The user should be able to specify a neural *problem* as a Keras model with either a custom or random weight initialisation.
3. Each problem may record *metrics* during training, and the user can implement custom metrics.
4. The user may implement a custom *agent* that can be used to train on any problem specified using the framework.

5. The user should be able to specify an *experiment* that runs specified agents simultaneously, reporting and visualising the associated problem metrics for each agent in real time.

6. The framework should provide some implementations of agents (both gradient-based and derivative-free), problems (the stripe problem and others), and experiments for demonstrational purposes.

## 1.5 Software engineering process

A very agile approach was adopted due to the primarily research-oriented character of the project. Weekly supervisor meetings were held where progress was discussed, and tasks were set for the next week. This ensured that changes could be made quickly depending on the outcomes of the experiments that were conducted. The most important aspects were written up in a LaTeX document on a week-by-week basis[6] and sent to the supervisor before each meeting, so the new content could be discussed. This facilitated a disciplined and agile approach to developing experiments and analysing their results. Apart from providing a structured set of notes for reference later on, some of the text and figures could be reused for this report as well.

The experiments themselves were conducted in interactive Python notebooks[7]. This approach is commonplace for machine learning projects, and as a side benefit this made it easy to use the notebooks in the Google Colab service to leverage free GPU acceleration. Important statistics and results were persisted in data files[8] so they could be used for analysis and plotting later.

The first version of the software framework[9] was developed over the inter-semester break and presented to the supervisor. This version included only one agent and one problem as a proof of concept. Initially, visualisations were achieved using the Python library `matplotlib`, but as additional requirements unfolded (such as that the user should be able to interact with the live-updating graphs to toggle the visibility of agents), a web-based front-end was developed over the second semster using the `bokeh` library. At the same time, more agents were implemented, and some of the experiments from the interactive notebooks were ported to the framework, too.

All code related to this project (including this report's LaTeX markup itself) was maintained using the version control system Git in a single repository hosted on GitHub[10]. This approach was undertaken for the reason of avoiding file and code duplication: the data files produced using the interactive notebooks could be used directly to generate the plots for the weekly notes as well as this report. Due to the fact there was only one developer, all commits were made to the `master` branch; adopting a more sophisticated model utilising feature branches or pull requests would cause more overhead than benefit in the single-developer scenario. Furthermore, no continuous integration system with automated testing was employed because that, too, would be excessive for a research project of this scale.

[6] This document is available at `research/progress/main.pdf` in the code submission.
[7] The interactive notebooks are found in the `research/notebooks` folder.
[8] The data files are available in the `*.dat` format in the `research/data` folder.
[9] The software framework is available in the `framework` folder.
[10] The repository can be found at `https://github.com/georgw777/neural-surfing`.

## 1.6 Ethics

There are no ethical considerations. All questions on the preliminary self-assessment form were answered with "NO" and hence no ethics form was completed.

# Chapter 2

# Neural network theory

Before tackling the local minimum problem, we must first understand the motivation and function of neural networks. We will begin by introducing the general concept of supervised learning algorithms which neural networks are an example of. This lays the foundation for a more rigorous definition of artificial neural networks in Section 2.2 as regression models which we widen in the following section to include binary classification tasks as well. The definitions, examples, and lemmata from this chapter are the groundwork for the stripe problem later on.

## 2.1 Supervised learning

At its core, the purpose of a neural network is to infer a function that maps some input to some output, based on sample input-output pairs. In machine learning, we call this a *supervised learning* task. Let us define that more precisely below.

**Definition 1** (Regression model)**.** In machine learning, a regression model $R$ is defined as a mathematical function of the form $R : \mathbb{R}^D \to \mathbb{R}$ given by

$$R(\mathbf{x}) = \hat{y} = y + \epsilon \tag{2.1}$$

that models the relationship between a $D$-dimensional feature vector $\mathbf{x} \in \mathbb{R}^D$ of independent (*input*) variables and the dependent (*output*) variable $y \in \mathbb{R}$. Given a particular $\mathbf{x}$, the model will produce a *prediction* for $y$ which we denote $\hat{y}$. Here, the additive error term $\epsilon$ represents the discrepancy between $y$ and $\hat{y}$.

**Definition 2** (Input space)**.** The input space $\mathcal{I}_R$ of a regression model $R$ is the set of all possible assignments to the feature vector $\mathbf{x}$. For a feature vector of $D$ dimensions,

$$\mathcal{I}_A = \mathbb{R}^D. \tag{2.2}$$

**Definition 3** (Labelled dataset)**.** A labelled dataset consists of $N$ tuples of the form $\langle \mathbf{x}_i, y_i \rangle$ for $i = 1, \ldots, N$. For each feature vector $\mathbf{x}_i$ (a row vector), the corresponding $y_i$ represents the observed output, or *label* [Burkov 2019]. We use the vector

$$\mathbf{y} = \begin{bmatrix} y_1 & y_2 & \cdots & y_N \end{bmatrix}^\mathsf{T} \tag{2.3}$$

to denote all the labelled outputs in the dataset, and the $N \times D$ matrix

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_N \end{bmatrix}^\mathsf{T} \tag{2.4}$$

for representing the corresponding feature vectors.

**Definition 4** (Supervised learning)**.** A supervised learning algorithm for a regression task infers the function $R$ given in Equation (2.1) from a set of *labelled training data* of the form explained previously. We use the vector

$$\hat{\mathbf{y}} = \begin{bmatrix} \hat{y}_1 & \hat{y}_2 & \cdots & \hat{y}_N \end{bmatrix}^\mathsf{T} \tag{2.5}$$

to denote the prediction that $R$ produces for each training sample.

## 2.2 Artifical neural networks

Artifical neural networks (ANNs) take inspiration from the human brain and can be regarded as a set of interconnected neurons. More formally, an ANN is a directed graph of $n$ neurons (referred to as *nodes* or *units*) with weighted edges (*links*). Each link connecting two units $i$ and $j$ is directed and associated with a real-valued weight $w_{i,j}$.

A particular unit $i$'s *excitation*, denoted $z_i$, is calculated as the weighted sum

$$z_i = \sum_{j=1}^{n} w_{j,i} a_j + b_i \tag{2.6}$$

where $a_j \in \mathbb{R}$ is another unit $j$'s *activation* and $b_i \in \mathbb{R}$ is the $i$th unit's *bias*. Notice that in this model, if there exists no link between unit $i$ and a particular $j$ then simply $w_{i,j} = 0$ and therefore $j$ will not contribute to $i$'s excitation.

The unit $i$'s activation is its excitation applied to a non-linear *activation function*, $g : \mathbb{R} \to \mathbb{R}$. We have

$$a_i = g(z_i) = g\left( \sum_{j=1}^{n} w_{j,i} a_j + b_i \right). \tag{2.7}$$

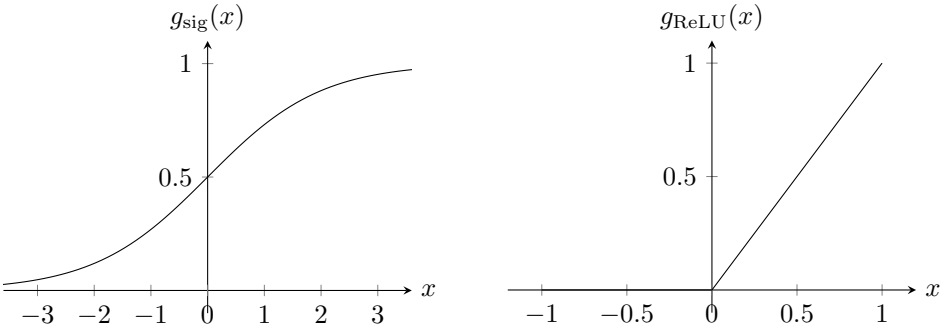
**Activation functions** In its original form, McCulloch and Pitts defined the neuron as having only binary activation [1943]. This means that in our model from Equation (2.7), we would require $a_i \in \{0, 1\}$ and hence an activation function of the form $g_{\text{thres}} : \mathbb{R} \to \{0, 1\}$ which would be defined[11] as

$$g_{\text{thres}}(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}. \tag{2.8}$$

Commonly used activation functions in modern neural networks include the sigmoid

$$g_{\text{sig}}(x) = \frac{1}{1 + e^{-x}} \tag{2.9}$$

[11] In fact, McCulloch and Pitts defined the activation to be zero when $x < \theta$ for a threshold parameter $\theta \in \mathbb{R}$ and one otherwise, but in our model the bias term $b_i$ acts as the threshold.




(a) Sigmoid (b) Rectified linear unit

Figure 2.1: Plots of the the two most common activation functions.

and the rectified linear unit (ReLU)

$$g_{\text{ReLU}} = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases} \tag{2.10}$$

which are depicted in Figure 2.1. Unlike $g_{\text{step}}$, these activation functions are differentiable which is an advantage for being able to use gradient descent [Russell and Norvig 2010, p. 729].

Rectified units do not suffer from the *vanishing gradient effect* [Glorot et al. 2011]. This phenomenon occurs with sigmoid activation functions when they reach high saturation, i.e. when the input is significantly far from zero such that the gradient is almost horizontal. However, the vanishing gradient problem is usually not prevalant in shallow[12] networks so the sigmoid function still remains popular [Neal 1992].

Particularly in deep neural networks, different neurons (grouped in *layers*, see Section 2.2.2) often have different activation functions [Burkov 2019], but for the purposes of this report it is more convenient (in terms of notation) to have the activation function be the same for all neurons, so it does not need to be supplied as a parameter to the function describing the particular neural network. Much of the work in this report can easily be generalised to designs with multple activation functions. This is because the algorithms explained in this report do not concern themselves with the specifics of the activation functions, as long as they are non-linear.

**ANNs as regression models** We can employ an ANN to model a regression problem of the form given in Equation (2.1). To do so, we need at least $D + 1$ neurons in the network. We consider the first $D$ units to be the *input* neurons, and the last neuron, $n$, is the output unit. Furthermore, we require $w_{j,k} = 0$ for $j, k \in \mathbb{Z}^+$ where $j \leq n$ and $k \leq D$ to ensure that there are no links feeding into the input neurons.

To obtain the prediction $\hat{y}$ given the $D$-dimensional feature vector $\mathbf{x}$, we set the activation of the $i$th unit to the value the $i$th element in $\mathbf{x}$ for $i = 1, \ldots, D$. Then, we propagate the activations using Equation (2.7) until finally

[12]Shallow networks refer to ANNs with few layers.

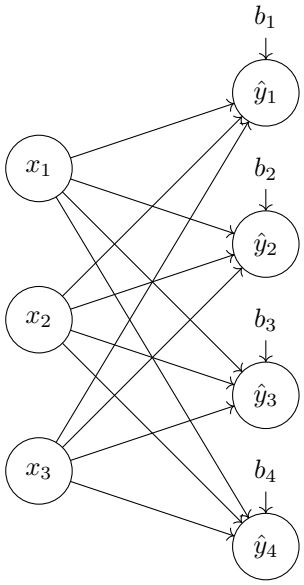



Figure 2.2: A single-layer perceptron with three input and four output neurons.

the prediction is the activation of the last neuron, $\hat{y} = a_n$. This process is often called *forward propagation* or *forward pass* [Burkov 2019].

### 2.2.1 Single-layer network

We introduce a single-layer network (SLN) as a type of ANN which consists of two conceptual layers, an input and an output layer. Every input node is connected to every output node, but there are no intra-layer links (i.e. there are no links between any two input nodes or any two output nodes), as shown in Figure 2.2. This is what we call a *fully-connected feedforward* architecture. SLN architectures will always form a *directed acyclic graph* (DAG) because there are no intra-layer or backwards connections.

We purposefully use the term SLN instead of single-layer perceptron (SLP) to avoid confusion. A SLP has only one output unit and uses the threshold activation function given in Equation (2.8) [Rosenblatt 1958]. In our definition of a SLN we allow more than one output and impose no restrictions on $g$, except that the same activation function is used for every output neuron. We still use the term 'single layer' because the input layer, lacking any incoming weight or bias connections, is not considered to be a 'proper' layer.

Let us consider a SLN with $m$ inputs and $n$ outputs. Since every output unit $i$ only has connections from every input unit $j$, we can adapt Equation (2.7) to give the activation of a particular output neuron $i$ as

$$a_i = y_i = g\left(z_i\right) = g\left(\sum_{j=1}^{m} w_{j,i} x_j + b_i\right) = g\left(\mathbf{w}_i^{\mathsf{T}} \mathbf{x}_i + b_i\right) \tag{2.11}$$

where $\mathbf{w}_i = \begin{bmatrix} w_{1,i} & w_{2,i} & \cdots & w_{m,i} \end{bmatrix}^{\mathsf{T}}$ represents the weights of all the edges

that connect to output unit $i$. This is all we need to formally define a SLN.

**Definition 5** (Single-layer network)**.** A SLN with $m$ inputs and $m$ outputs is the vector-valued function $\mathbf{S} : \mathbb{R}^m \to \mathbb{R}^n$ defined as

$$\mathbf{S}(\mathbf{x}; \mathbf{W}, \mathbf{b}) = \mathbf{g}\left(\mathbf{W}^\mathsf{T}\mathbf{x} + \mathbf{b}\right) \tag{2.12}$$

where the $m \times n$ matrix

$$\mathbf{W} = \begin{bmatrix}\mathbf{w}_1 & \mathbf{w}_2 & \cdots & \mathbf{w}_n\end{bmatrix} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \cdots & w_{m,n} \end{bmatrix} \tag{2.13}$$

captures all weights and the vector

$$\mathbf{b} = \begin{bmatrix} b_1 & b_2 & \cdots & b_n \end{bmatrix}^\mathsf{T} \tag{2.14}$$

represents the biases. The vector-valued activation function $\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^n$ is simply the activation function $g : \mathbb{R} \to \mathbb{R}$ applied pointwise to a vector, i.e.

$$\mathbf{g}(\mathbf{z}) = \begin{bmatrix} g(z_1) & g(z_2) & \cdots & g(z_n) \end{bmatrix}^\mathsf{T}$$

for the vector of excitations $\mathbf{z} = \begin{bmatrix} z_1 & z_2 & \cdots & z_n \end{bmatrix}^\mathsf{T}$.

Unlike the formula for a regression model, a SLN is a vector-valued function, due to the fact that there are multiple outputs. Note that when $n = 1$, we reach the same form as in Equation (2.1). Moreover, if we additionally use the threshold activation function from Equation (2.8), we arrive at the SLP model given by Rosenblatt [1958].

### 2.2.2 Multi-layer perceptron

A multi-layer perceptron[13] (MLP) is a fully-connected feedforward ANN architecture with multiple layers which we will define in terms of multiple nested functions as in Burkov [2019].

**Definition 6** (Multi-layer perceptron)**.** A MLP $M$ with $m$ inputs and $L$ layers is the mathematical function $M : \mathbb{R}^m \to \mathbb{R}$ defined as the nested function

$$M(\mathbf{x}; \mathscr{P}) = \hat{y} = f_L\left(\mathbf{f}_{L-1}\left(\ldots\left(\mathbf{f}_1\left(\mathbf{x}\right)\right)\right)\right) \tag{2.15}$$

for the trainable parameters $\mathscr{P} = \langle \mathscr{W}, \mathscr{B} \rangle$ consisting of the weight matrices $\mathscr{W} = \mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_L$ and bias vectors $\mathscr{B} = \mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_L$ such that the nested functions are given by $\mathbf{f}_l(\mathbf{x}) = \mathbf{S}(\mathbf{x}; \mathbf{W}_l, \mathbf{b}_l)$ for $l = 1, \ldots, L-1$. The outermost function $f_L$ represents a SLN with only one output unit and is hence the scalar-valued function $f_L(\mathbf{x}) = S(\mathbf{x}; \mathbf{W}_L, \mathbf{b}_L)$.

[13] Unlike SLPs, the activation function in a MLP as defined in literature does not necessarily need to be the binary threshold function $g_{\text{thres}}$; in fact, it is often one of the more modern activation functions explained in Section 2.2 [Hastie et al. 2017; Burkov 2019]. Hence we can use the term 'multi-layer perceptron'.

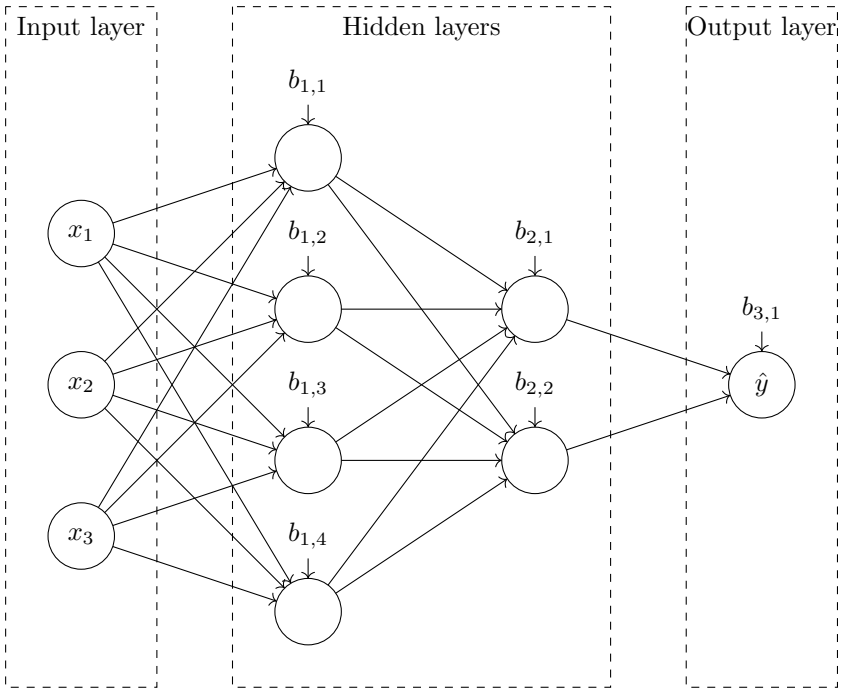



Figure 2.3: A multi-layer perceptron with three inputs and two hidden layers.

Notice that for every $l < L$, $\mathbf{W}_l$ is a $n_l \times m_l$ matrix such that $n_l = m_{l+1}$ to esnure that the number of outputs of layer $l$ is the number of inputs to layer $l + 1$. This means that the MLP has $m_1$ input neurons. Since the final layer has only one output unit, $\mathbf{W}_L$ has only one row, and finally $n_L = 1$

The graph representing this type of network consists of connecting the outputs of the SLN representing layer $l$ with the inputs of the SLN representing layer $l + 1$, as shown in Figure 2.3. The layers between the input and output layers are referred to as *hidden* layers.

Since MLPs are simply nested SLNs, it follows that MLPs retain the DAG property and are therefore *feedforward* networks as well. In the forward pass, the activations are propagated from layer to layer (i.e. nested function to nested function) as in Equation (2.12).

## 2.3 The decision boundary in input space

We will briefly introduce the concept of binary classification and show how it fits in the framework of the already defined regression model. This will allow us to examine the so-called decision boundary in input space which will be useful for formulating the stripe problem (Section 5.2).

**Definition 7** (Binary classification model)**.** A binary classification model $C$ is defined as a mathematical function of the form

$$C(\mathbf{x}) = \hat{y} = y + \epsilon \tag{2.16}$$

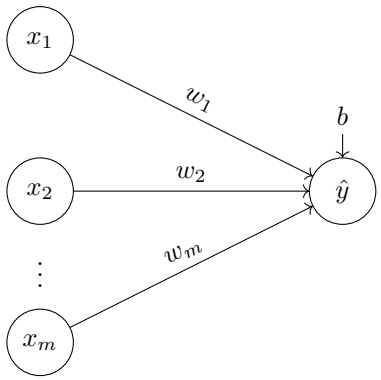



Figure 2.4: The DAG representing a SLN with $m$ inputs and one output unit.

with the same notation as in Definition 1 except that we impose the additional restriction that $y, \hat{y} \in \{0, 1\}$ such that the signature of the function becomes $C : \mathbb{R}^D \to \{0, 1\}$.

**Definition 8** (Decision boundary)**.** Given a binary classification model $C$, the decision boundary is the hypersurface[14] in input space $\mathcal{I}_C$ that separates the two output classes [Russell and Norvig 2010, p. 723]. We will also use the term 'hyperplane' to loosely refer to the decision boundary if it is flat/linear.

**Lemma 1.** *Given a decision threshold $t$, we can use a regression model $R$ to solve any binary classification problem.*

*Proof.* We are looking define an equivalent classification model $C$ that outputs 0 if $R(\mathbf{x}) < t$ and 1 otherwise. This can be achieved using the threshold activation function $g_{\text{thres}}$ from Equation (2.8) in the form

$$C(\mathbf{x}) = g_{\text{thres}}(R(\mathbf{x}) - t). \tag{2.17}$$

□

**Remark.** What we have shown is that we can repurpose any regression model for a binary classification task, including for example MLPs. When using a MLP, the sigmoid activation function Equation (2.9) naturally lends itself to be used on the output unit because its range, the interval $(0, 1)$, can be interpreted as a probability. In this case we would set the decision threshold $t = \frac{1}{2}$.

**Lemma 2** (Single-layer sigmoidal decision boundary)**.** *A single-layer sigmoidal MLP with a decision threshold $t \in (0, 1)$ will have only one hyperplane in input space.*

*Proof.* Consider a single-layer MLP $M$ with $m$ inputs. By Definition 6, this is equivalent to SLN $S$ with $m$ inputs and one output as shown in Figure 2.4. The equation of decision boundary can be obtained by setting the output

[14] A hypersurface is a manifold with one fewer dimension. Since the input space is $D$-dimensional, the hypersurface representing the decision boundary will have $D - 1$ dimensions.

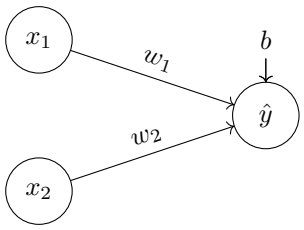



Figure 2.5: A simple MLP with one layer and two inputs (equivalently, a SLN with two inputs and one ouput).

equal to the decision threshold, so $t = S(\mathbf{x})$ for the input feature vector $\mathbf{x} = \begin{bmatrix} x_1 & x_2 & \cdots & x_m \end{bmatrix}^\mathsf{T}$. We have

$$\begin{aligned} t &= S(\mathbf{x}) \\ &= g_{\text{sig}}\left(\mathbf{w}^\mathsf{T}\mathbf{x} + b\right) \\ &= \frac{1}{1 + e^{-\mathbf{w}^\mathsf{T}\mathbf{x} - b}} \\ \frac{1}{t} - 1 &= e^{-\mathbf{w}^\mathsf{T}\mathbf{x} - b} \\ \ln\left(\frac{1}{t} - 1\right) &= -\mathbf{w}^\mathsf{T}\mathbf{x} - b. \end{aligned}$$

For $\ln\left(\frac{1}{t} - 1\right)$ to be real-valued, we must ensure that $\frac{1}{t} - 1 > 0$ which is the case because $0 < t < 1$.

We obtain only one linear equation of the form

$$0 = w_1 x_2 + w_2 x_2 + \cdots + w_m x_m + b + \ln\left(\frac{1}{t} - 1\right) \tag{2.18}$$

which means that there is only one hyperplane. □

**Example 1.** Let us consider a MLP $M$ with only one layer and two inputs, as depicted in Figure 2.5. We will consider the configuration where $w_1 = w_2 = 1$ and $b = 0$. The output unit will have the sigmoid activation function, and we will choose the decision threshold $t = \frac{1}{2}$.

The decision boundary will be a line because $\mathcal{I}_M$ has two dimensions, and a hyperplane in a two-dimensional space is simply a line. The equation of this line can be obtained from Equation (2.18) as

$$\begin{aligned} 0 &= w_1 x_1 + w_2 x_2 + b + \ln\left(\frac{1}{\left(\frac{1}{2}\right)} - 1\right) \\ &= x_1 + x_2 + \ln 1 \\ x_2 &= -x_1. \end{aligned}$$

Figure 2.6 depicts this hyperplane along with some samples in input space to show how they would be classified. The arrow on the hyperplane shows the direction of increasing output, i.e. what would be classified as 1.

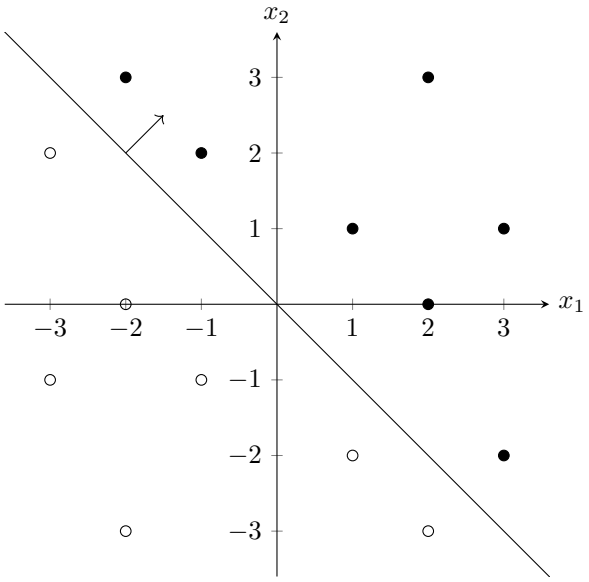



Figure 2.6: Plot of the input space of the MLP from Figure 2.5 with sigmoid activation where $w_1 = w_2 = 1$, $b = 0$, and some sample data. Filled in dots represent a prediction of $\hat{y} > \frac{1}{2}$ whereas the empty circular dots represent $\hat{y} < \frac{1}{2}$.

# Chapter 3

# Neural network training

We will now introduce the two neural training techniques outlined in Section 1.1 (gradient descent and simulated annealing) more formally in the context of how we defined neural networks in the previous chapter. For gradient descent, we will derive the famous backpropagation algorithm, but we will also look at a derivative-free notion of gradient descent which we will call *greedy probing*. Simulated annealing is of course derivative-free in nature. We also define some issues related to these methods more precisely as a means of setting the scene for the neural surfing technique later on.

*Training* with regard to neural networks refers to the process of altering a network's weights and biases with the goal of achieving an optimal configuration that reduces the error of the predictions, i.e. how far they are 'off'. This how the network facilitates *learning* the input-output function from Definition 4. We will use a simple loss function that uses mean squared error for this purpose.

**Definition 9** (Mean squared error)**.** Let $\{\langle \mathbf{x}_i, y_i \rangle\}_{i=1}^{N}$ be a labelled dataset (see Definition 3). The mean squared error of a set of predictions $\hat{\mathbf{y}}$ is given as an average over the sum of squared differences,

$$E\left(\mathbf{y}, \hat{\mathbf{y}}\right) = \frac{1}{N} \sum_{i=1}^{N} \left(\hat{y}_i - y_i\right)^2. \tag{3.1}$$

**Definition 10** (Loss function)**.** Given an MLP $M$ with $P$ trainable parameters (weights and biases), the loss function $L : \mathbb{R}^P \to \mathbb{R}$ is a function that maps weight (and bias) configurations to their associated error values. Let $\mathbf{y} \in \mathbb{R}^N$ be the target outputs for training. Then the loss function is defined as

$$L(\mathbf{p}) = \sum_{i=1}^{N} \left(M\left(\mathbf{x}_i; \mathbf{p}\right) - y_i\right)^2. \tag{3.2}$$

Notice the similarity to Equation (3.1); we have only omitted the factor $\frac{1}{N}$ since the actual loss values are not as important as their relationship to each other, and multiplying by $N$ will retain that relationship. We use the term *error-weight surface* to refer to the graph of this function.

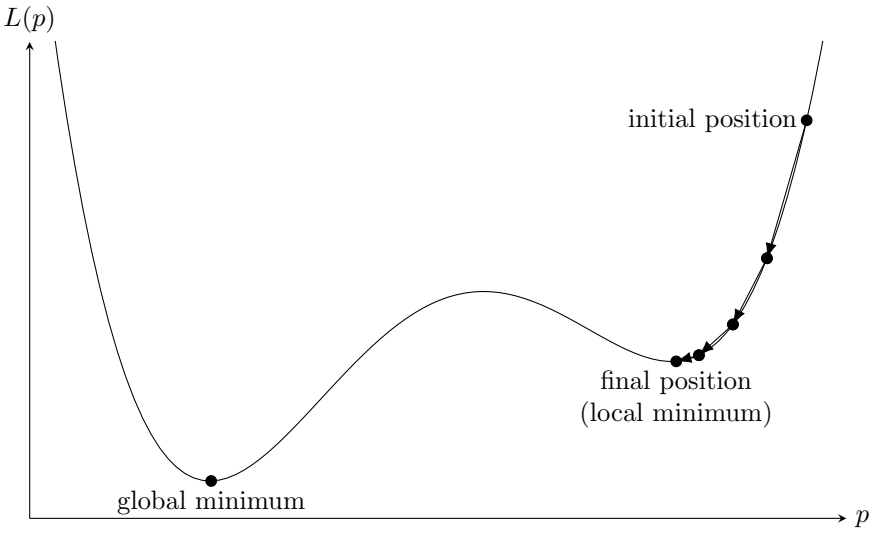



Figure 3.1: An illustration gradient descent training on an error-weight surface with only one parameter (not drawn to scale).

## 3.1 Backpropagation with gradient descent

Backpropagation (BP) with gradient descent is an iterative algorithm for training neural networks that, provided a suitable learning rate $\alpha$, is guaranteed to converge to a *local minimum*. The main idea is as follows:

1. Calculate the derivative of the loss function with respect to the current trainable parameters $\mathbf{p}$ as $\boldsymbol{\Delta}\mathbf{p} = \frac{\delta L}{\delta \mathbf{p}}(\mathbf{p})$.
2. Take a step in the negative direction of this gradient, i.e. update the trainable parameters $\mathbf{p} \leftarrow \mathbf{p} - \alpha\boldsymbol{\Delta}\mathbf{p}$ where $\alpha \in \mathbb{R}$ is the learning rate.
3. Repeat steps 1 and 2 until a predefined convergence criterion is met.

Figure 3.1 shows the steps that this algorithm would make on a simple error-weight surface with only one parameter.

Calculating the derivative of the loss function with respect to each of the trainable parameters is a core part of the gradient descent algorithm. Let us look at calculating this gradient for the example of a single-layer MLP. We will come back to these results in Section 5.2.

**Example 2** (Gradient in a single-layer MLP)**.** Let us revisit the SLN with $m$ inputs and one output from Figure 2.4. The loss function will be in terms of the trainable parameters, i.e. the weights $\mathbf{w}$ and bias $b$, so

$$\begin{aligned} L = L(\mathbf{w}, b) &= \sum_{i=1}^{N} \left(S\left(\mathbf{x}_i; \mathbf{w}, b\right) - y_i\right)^2 \\ &= \sum_{i=1}^{N} \left(g\left(\mathbf{w}^\mathsf{T}\mathbf{x}_i + b\right) - y_i\right)^2. \end{aligned} \tag{3.3}$$

We obtain the partial derivative of the loss with respect to the bias as

$$\begin{aligned}\frac{\delta L}{\delta b} &= 2\sum_{i=1}^{N}\left(g\left(\mathbf{w}^{\mathsf{T}}\mathbf{x}_i+b\right)-y_i\right)\frac{\delta}{\delta b}\left(g\left(\mathbf{w}^{\mathsf{T}}\mathbf{x}_i+b\right)-y_i\right)\\ &= 2\sum_{i=1}^{N}\left(g\left(\mathbf{w}^{\mathsf{T}}\mathbf{x}_i+b\right)-y_i\right)g'\left(\mathbf{w}^{\mathsf{T}}\mathbf{x}_i+b\right),\end{aligned} \tag{3.4}$$

and similarly we can differentiate with respect to the weights

$$\begin{aligned}\frac{\delta L}{\delta \mathbf{w}} &= 2\sum_{i=1}^{N}\left(g\left(\mathbf{w}^{\mathsf{T}}\mathbf{x}_i+b\right)-y_i\right)\frac{\delta}{\delta \mathbf{w}}\left(g\left(\mathbf{w}^{\mathsf{T}}\mathbf{x}_i+b\right)-y_i\right)\\ &= 2\sum_{i=1}^{N}\left(g\left(\mathbf{w}^{\mathsf{T}}\mathbf{x}_i+b\right)-y_i\right)g'\left(\mathbf{w}^{\mathsf{T}}\mathbf{x}_i+b\right)\mathbf{x}_i.\end{aligned} \tag{3.5}$$

Now we would denote the gradient of the loss with respect to the trainable parameters $\mathbf{p}$ as the row vector

$$\frac{\delta L}{\delta \mathbf{p}} = \begin{bmatrix}\frac{\delta L}{\delta w_1} & \frac{\delta L}{\delta w_2} & \cdots & \frac{\delta L}{\delta w_m} & \frac{\delta L}{\delta b}\end{bmatrix}.$$

## 3.2 Greedy probing

Unlike BP, greedy probing is a simple derivative-free optimisation (DFO) technique. At each iteration, the algorithm will sample a predefined number of configurations in the local neighbourhood of the current parameter configuration $\mathbf{p}$. The loss is calculated at each of these samples, and the best (i.e. lowest value) is chosen as the new parameter configuration. In this sense, greedy probing is similar to gradient descent, except that the gradient is manually calculated instead of using the partial derivative. As a result, greedy probing will suffer similar issues as gradient descent, but depending on the sampling radius $r \in \mathbb{R}$ it is conjectured that it might be less sensitive to local pertubations ('noise') on the error-weight surface.

The sampling technique may either be *exhaustive* or *random*. In the exhaustive case, the samples around point $\mathbf{p}_i$ are given by

$$\left\{\mathbf{p}_i + r\frac{\begin{bmatrix}p_1 & p_2 & \cdots & p_P\end{bmatrix}^{\mathsf{T}}}{\left\|\begin{bmatrix}p_1 & p_2 & \cdots & p_P\end{bmatrix}\right\|} : \begin{array}{l} p_1, p_2, \ldots, p_P \in \{-1, 0, 1\} \text{ and} \\ \left\|\begin{bmatrix}p_1 & p_2 & \cdots & p_P\end{bmatrix}\right\| \neq 0\end{array}\right\} \tag{3.6}$$

which means that $3^P - 1$ samples are generated at each iteration. On the other hand, the random sampling technique will generate a predefined number of random samples in weight space with the condition that given the current configuration $\mathbf{p}_i$, for every candidate sample $\hat{\mathbf{p}}$, it must be true that $\|\mathbf{p}_i - \hat{\mathbf{p}}\| = r$.

## 3.3 Simulated annealing

The rationale of the simulated annealing (SA) algorithm lies in its analogy to the behaviour that atoms exhibit in a substance that is slowly cooling down

[Kirkpatrick et al. 1983]. At high temperatures, the atoms move around with high kinetic energy, but as the temperature cools down, they begin to move more slowly until completely losing thermal mobility. When this process is carried out sufficiently slowly, the atoms will settle in a perfectly aligned crystal structure with minimum energy. However, if the cooling process is too fast, the final structure will be chaotic and hence not be at the minimum energy state [Press et al. 1992, p. 444].

At each iteration, the algorithm will explore random sample points one after another in the local neighbourhood of the current configuration $\mathbf{p}_i$ until one is accepted. The probability of accepting a candidate sample point $\hat{\mathbf{p}}$ at the $i$th iteration is given by the probability distribution

$$P(\hat{\mathbf{p}}|\mathbf{p}_i) = \begin{cases} \exp\left(-\frac{k}{T_i}\left(L(\hat{\mathbf{p}}) - L(\mathbf{p}_i)\right)\right) & L(\hat{\mathbf{p}}) > L(\mathbf{p}_i) \\ 1 & L(\hat{\mathbf{p}}) \leq L(\mathbf{p}_i) \end{cases} \tag{3.7}$$

for an energy coefficient $k \in \mathbb{R}$ [Rios and Sahinidis 2009]. This means that the SA algorithm will always accept a better location, but, with a certain probability, might take a suboptimal step.

The temperature at the $i$ iteration is determined according to a *cooling schedule*. We will employ a simple approach as a proof of concept that calculates $T_i$ as the geometric sequence

$$T_i = (1 - c)\, T_{i-1} = (1 - c)^i\, T_0 \tag{3.8}$$

where $c$ is a cooling rate typically of the order of $10^{-1}$ or $10^{-2}$. Already, this simple SA algorithm has three hyperparameters: the energy coefficient $k$, the initial temperature $T_0$, and the cooling rate $c$. More sophisticated implementations will require even more hyperparameters (such as the algorithm proposed by Press et al. [1992] which will be remarked upon in Section 9.2) that makes it increasingly difficult to design SA in a generic fashion to suit the training of neural networks.

## 3.4 Issues and outlook

In section Section 1.1, we have intuitively identified one significant issue with these techniques: the local minimum problem. With the knowledge of the previous sections, we are now able to appreciate more profoundly how this problem affects the training techniques. Moreover, we can examine some of their issues of efficiency.

**Suboptimal local minima** It is clear that a prevalant problem these optimisation techniques face is that of suboptimal local minima (known simply as the 'local minimum problem'). Figure 3.1, which was used to explain the process of gradient descent with BP, actually provides a simple example of said problem. In fact, this example is quite similar to the analogy to hiking in the mountains drawn in Section 1.1: imagine the figure depicts the side-view of the mountain landscape, and you are at the position marked 'local minimum'. Due to the fact that you cannot see behind the slight incline to the left, you would not know that there is a global minimum in that direction.

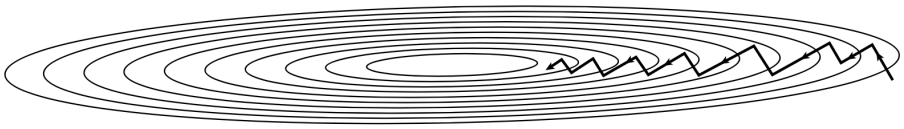


Figure 3.2: Contour plot of the method of steepest gradient descent in a long, narrow valley [Press et al. 1992, p. 421]. Travel proceeds in near right angles due to the gradients.

The local minimum problem is not unique to gradient descent with BP. It is easy to see that greedy probing, being a form of gradient descent itself, will suffer from the same problem. SA tries to circumvent the local minimum problem by allowing intermediate states that are worse than their predecessors, but even this technique is not guaranteed to find the global minimum in a finite amount of time. In the hiking analogy, the SA algorithm would likely need to take a lot of suboptimal steps when starting off at the wrong peak.

Due to the fact that the neural training techniques described above are guaranteed to converge to local minima, one could come to the conclusion to simply choose any of the techniques and run them multiple times using different starting points because this will increase the likelihood of finding a better local minimum. This is, in fact, a widely used approach known in practice as the *random initialisation* technique. However, it is important to note that a finite number of initialisations is still not guaranteed to find a global optimum, and especially if the global minimum basin is quite small, it is quite unlikely to do so.

**Efficiency** In the context of computation time per epoch, BP will outperform the derivative-free techniques because it computes only the gradient of the loss function (along with the loss value itself). Both of these computations are in the order of the complexity of computing a forward pass. On the other hand, derivative-free techniques rely on probing, so they must compute as many forward passes as samples they require per epoch. In that sense, the DFO algorithms will be slower.

However, BP has other efficiency drawbacks as a result of relying solely on derivatives. Firstly, it is more sensitive to local 'noise' in the loss function. For the second point, consider the direction of the gradients at the edges of ravines, or "long, narrow valleys" [Press et al. 1992], in the cost surface. When following the gradients with a specific step size, travel will becomes quite slow because it will proceed almost at right angles, as shown in Figure 3.2. Furthermore, if the step size is too large, the algorithm might 'jump' over the ravine which poses an issue if the ravine contains the global minimum. Per contra, DFO algorithms are not as sensitive to local noise, and their probing approach will facilitate more efficient progress along narrow valleys, although they will be more likely to accidentally jump over ravines if the step size is too large.

# Chapter 4

# Neural surfing theory

In the analogy of Section 1.1 we found that relying solely on our limited local information of the mountain landscape is not sufficient to effectively find the global minimum. Luckily, in the case of neural networks, we actually have more information than just the error-weight surface. This is the main idea behind the neural surfing technique. Instead of only considering the error-weight space, we will also look at the so-called output space. In doing so, we are viewing the neural training problem from a different perspective than it is classically studied. This chapter will discuss the concepts of weight and output space, as well as issues related to these concepts such as unrealisable regions and what is meant by a goal-connecting path.

## 4.1 Weight and output spaces

In Definition 6 we established that the tuple $\langle \mathscr{W}, \mathscr{B} \rangle$ along with the activation function is sufficient to fully define a MLP. Most importantly, we have $\mathscr{W} = \mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_L$ and $\mathscr{B} = \mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_L$ representing each layer's weight matrices and bias vectors, respectively. These parameters will be useful for defining the weight and ouput spaces.

**Definition 11** (Weight space)**.** The weight space $\mathcal{W}_A$ of an artificial neural network $A$ is the set of all possible assignments to its *trainable parameters*. The trainable parameters are its weights $\mathscr{W}$ and biases $\mathscr{B}$. If $A$ has $P$ trainable parameters then its weight space is defined as

$$\mathcal{W}_A = \mathbb{R}^P. \tag{4.1}$$

**Definition 12** (Output space)**.** The output space $\mathcal{O}_A$ of an artificial neural network $A$ with one output neuron spans the space of all possible output predictions on the training set. From Equation (2.5), the vector $\hat{\mathbf{y}}$ represents the prediction $\hat{y}$ for all $N$ training samples. The output space spans all possible assignments of $\hat{\mathbf{y}}$, so

$$\mathcal{O}_{\mathcal{A}} = \mathbb{R}^N. \tag{4.2}$$

**Lemma 3.** *The weight space for a SLN $S$ with $m$ inputs and $n$ outputs is* $\mathcal{W}_S = \mathbb{R}^{n(m+1)}$.

*Proof.* $S$'s trainable parameters are the weight matrix $\mathbf{W} \in \mathbb{R}^{m \times n}$ from Equation (2.13) and bias vector $\mathbf{b} \in \mathbb{R}^n$ from Equation (2.14). By Definition 11, the weight space encompasses all values of $\mathbf{W}$ and $\mathbf{b}$, so

$$\mathcal{W}_S = \mathbb{R}^{m \times n} \times \mathbb{R}^n = \mathbb{R}^{mn+n} = \mathbb{R}^{(m+1)n}.$$

□

**Lemma 4.** *A MLP $M$ with $L$ layers where the number of inputs to layer $l$ is given as $m_l$ will have the weight space $\mathcal{W}_M = \mathbb{R}^P$ where $P = \sum_{l=1}^{L-1} (m_{l+1}(m_l + 1))+ m_L + 1$.*

*Proof.* By Definition 6, $M$ is comprised of $L$ SLNs which we will denote $\mathbf{S}_1, \mathbf{S}_2, \ldots, S_L$. This allows us to express the weight space of $M$ as the product of the weight spaces of each of the SLNs,

$$\mathcal{W}_M = \prod_{l=1}^{L} \mathcal{W}_{S_l}.$$

For every layer $l$, the number of inputs to $S_l$ will be the number of inputs to the $l$th layer, $m_l$. Let $n_l$ denote the number outputs for each layer $l$. Then, by Lemma 3,

$$\mathcal{W}_{S_l} = \mathbb{R}^{n_l(m_l+1)}.$$

By splitting of the last factor in the product of weight spaces, we obtain

$$\mathcal{W}_M = \prod_{l=1}^{L-1} \mathcal{W}_{S_l} \times \mathcal{W}_{S_L} = \prod_{l=1}^{L-1} \mathbb{R}^{n_l(m_l+1)} \times \mathbb{R}^{n_L(m_L+1)}.$$

Notice that for any layer $l$, the number of outputs is equal to the number of inputs to the next layer, so $n_l = m_{l+1}$ except for the last layer where there is only one output unit leaving $n_L = 1$. This leaves

$$\begin{aligned} \mathcal{W}_M &= \prod_{l=1}^{L-1} \mathbb{R}^{m_{l+1}(m_l+1)} \times \mathbb{R}^{m_L+1} \\ &= \mathbb{R}^{\sum_{l=1}^{L-1} m_{l+1}(m_l+1)} \times \mathbb{R}^{m_L+1} \\ &= \mathbb{R}^{\sum_{l=1}^{L-1} m_{l+1}(m_l+1)+m_L+1}, \end{aligned}$$

so $\mathcal{W}_M = \mathbb{R}^P$ with

$$P = \sum_{l=1}^{L-1} m_{l+1}(m_l + 1) + m_L + 1.$$

□

**Remark.** The significance of Lemma 4 is that we obtain a formula for the number of trainable parameters $P$ in a MLP. By Definition 11, $P$ determines the dimensionality of the weight space. On other other hand, Definition 12 states that the number of samples in the training set $N$ determines the dimensionality of the output space. There is no relationship between $P$ and $N$ since the number of samples in the training set can be arbitrarily chosen. It follows that there is no relationship between the dimensionalities of $\mathcal{W}$ and $\mathcal{O}$.

### 4.1.1 Relationship between weight and output space

We will now examine the nature of the mapping between the two spaces, and whether there exists a linear mapping. Note that linear mappings can exist between spaces of different dimensionalities [Rudin 2006].

**Definition 13** (Weight-output mapping)**.** Given an artifical neural network $A : \mathbb{R}^m \to \mathbb{R}^n$ with $m$ inputs and $n$ outputs parameterised by a set of trainable parameters $\mathbf{w} \in \mathcal{W}_A$, the weight-to-output-space mapping $h_A : \mathcal{W}_A \to \mathcal{O}_A$ for a dataset with $N$ $m$-dimensional feature vectors given by the matrix $\mathbf{X} = \begin{bmatrix}\mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_N\end{bmatrix}^\mathsf{T} \in \mathbb{R}^{N\times m}$ is

$$h_A(\mathbf{w}) = \begin{bmatrix} A(\mathbf{x}_1;\mathbf{w}) \\ A(\mathbf{x}_2;\mathbf{w}) \\ \vdots \\ A(\mathbf{x}_N;\mathbf{w}) \end{bmatrix}.$$

Note that we use the term 'weight-output mapping' to refer to the 'weight-to-output-space mapping' which should be confused with the mapping from weight space to a particular output prediction.

**Theorem 5.** *For a SLN $S$ with one output unit, the function $h_S : \mathcal{W}_S \to \mathcal{O}_S$ is not a linear mapping.*

*Proof.* Let $S$ have $m$ inputs, as depicted in Figure 2.4. Modifying the formula for a SLN given in Definition 5 Equation (2.12) for the case where there is only one output unit, we obtain $\hat{y} = f_{\text{SLP}}(\mathbf{x};\mathbf{w}^\mathsf{T},b) = g(\mathbf{w}\mathbf{x}+b)$ where $\mathbf{x} = \begin{bmatrix}x_1 & x_2 & \cdots & x_m\end{bmatrix}^\mathsf{T}$ is the input feature vector.

We will consider a dataset with $N$ samples where the input is given by the $N\times m$ matrix $\mathbf{X} = \begin{bmatrix}\mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_N\end{bmatrix}^\mathsf{T}$ as in Equation (2.4). By Definition 13, the mapping from weight to output space $h_S : \mathcal{W}_S \to \mathcal{O}_S$ is

$$h_S(\mathbf{w}) = \begin{bmatrix} g\left(\mathbf{w}^\mathsf{T}\mathbf{x}_1 + b\right) \\ g\left(\mathbf{w}^\mathsf{T}\mathbf{x}_2 + b\right) \\ \vdots \\ g\left(\mathbf{w}^\mathsf{T}\mathbf{x}_N + b\right) \end{bmatrix}.$$

We will assume, by way of contradiction, that $h$ is a linear mapping. From the definition of linear mappings, it must be true that $h(\mathbf{u}+\mathbf{v}) = h(\mathbf{u}) + h(\mathbf{v})$ for $\mathbf{u},\mathbf{v} \in \mathcal{W}_S$ [Rudin 2006]. On the LHS we have

$$h(\mathbf{u}+\mathbf{v}) = \begin{bmatrix} g((\mathbf{u}+\mathbf{v})^\mathsf{T}\mathbf{x}_1 + b) \\ g((\mathbf{u}+\mathbf{v})^\mathsf{T}\mathbf{x}_2 + b) \\ \vdots \\ g((\mathbf{u}+\mathbf{v})^\mathsf{T}\mathbf{x}_N + b) \end{bmatrix} = \begin{bmatrix} g(\mathbf{u}^\mathsf{T}\mathbf{x}_1 + \mathbf{v}^\mathsf{T}\mathbf{x}_1 + b) \\ g(\mathbf{u}^\mathsf{T}\mathbf{x}_2 + \mathbf{v}^\mathsf{T}\mathbf{x}_2 + b) \\ \vdots \\ g(\mathbf{u}^\mathsf{T}\mathbf{x}_N + \mathbf{v}^\mathsf{T}\mathbf{x}_N + b) \end{bmatrix}$$

and on the RHS we get

$$h(\mathbf{u}) + h(\mathbf{v}) = \begin{bmatrix} g(\mathbf{u}^\mathsf{T}\mathbf{x}_1 + b) \\ g(\mathbf{u}^\mathsf{T}\mathbf{x}_2 + b) \\ \vdots \\ g(\mathbf{u}^\mathsf{T}\mathbf{x}_N + b) \end{bmatrix} + \begin{bmatrix} g(\mathbf{v}^\mathsf{T}\mathbf{x}_1 + b) \\ g(\mathbf{v}^\mathsf{T}\mathbf{x}_2 + b) \\ \vdots \\ g(\mathbf{v}^\mathsf{T}\mathbf{x}_N + b) \end{bmatrix} = \begin{bmatrix} g(\mathbf{u}^\mathsf{T}\mathbf{x}_1 + b) + g(\mathbf{v}^\mathsf{T}\mathbf{x}_1 + b) \\ g(\mathbf{u}^\mathsf{T}\mathbf{x}_2 + b) + g(\mathbf{v}^\mathsf{T}\mathbf{x}_2 + b) \\ \vdots \\ g(\mathbf{u}^\mathsf{T}\mathbf{x}_N + b) + g(\mathbf{v}^\mathsf{T}\mathbf{x}_N + b) \end{bmatrix},$$

leaving

$$\begin{bmatrix} g(\mathbf{u}^\mathsf{T}\mathbf{x}_1 + \mathbf{v}^\mathsf{T}\mathbf{x}_1 + b) \\ g(\mathbf{u}^\mathsf{T}\mathbf{x}_2 + \mathbf{v}^\mathsf{T}\mathbf{x}_2 + b) \\ \vdots \\ g(\mathbf{u}^\mathsf{T}\mathbf{x}_N + \mathbf{v}^\mathsf{T}\mathbf{x}_N + b) \end{bmatrix} = \begin{bmatrix} g(\mathbf{u}^\mathsf{T}\mathbf{x}_1 + b) + g(\mathbf{v}^\mathsf{T}\mathbf{x}_1 + b) \\ g(\mathbf{u}^\mathsf{T}\mathbf{x}_2 + b) + g(\mathbf{v}^\mathsf{T}\mathbf{x}_2 + b) \\ \vdots \\ g(\mathbf{u}^\mathsf{T}\mathbf{x}_N + b) + g(\mathbf{v}^\mathsf{T}\mathbf{x}_N + b) \end{bmatrix}.$$

Let $\alpha_i = \mathbf{u}^\mathsf{T}\mathbf{x}_i$ and $\beta_i = \mathbf{v}^\mathsf{T}\mathbf{x}_i$ for all $i$. Since $\mathbf{u}, \mathbf{v} \in \mathbb{R}^m$ and all $\mathbf{x}_i \in \mathbb{R}^m$, it follows that $\alpha_i, \beta_i \in \mathbb{R}$ for all $i$. Hence $g(\alpha + \beta + b) = g(\alpha + b) + g(\beta + b)$.

The only functions that satisfy $g$ are functions that satisfy Cauchy's functional equation[15], but these solutions only apply when $b = 0$ and furthermore are linear, whereas the activation function $g$ is non-linear. We arrived at a contradiction, thus disproving our initial assumption that $h$ is a linear mapping, so it must be a non-linear mapping. □

**Corollary 5.1.** *For any SLN $S$, the function $h_S : \mathcal{W}_S \to \mathcal{O}_S$ is not a linear mapping.*

*Proof.* We will generalise the results from Theorem Theorem 5 to SLNs with multiple outputs. Let $S$ have $m$ inputs and $n$ outputs. We construct $n$ smaller SLNs, $S_1, S_2, \ldots, S_n$ where each $S_i$ has all $m$ input units, but only the $i$th output unit. The DAG representing $S_i$ will only contain links from the input nodes to output node $\hat{y}_i$ (and, of course, the associated bias term $b_i$) as depicted in Figure 4.1.

Now, we can simulate the function of $S$ by the construction

$$S(\mathbf{x}) = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_n \end{bmatrix} = \begin{bmatrix} S_1(\mathbf{x}) \\ S_2(\mathbf{x}) \\ \vdots \\ S_n(\mathbf{x}) \end{bmatrix}.$$

By Theorem Theorem 5, each $S_i$ does not have a linear mapping from weight space to output space, so $S$ cannot have a linear mapping either. □

**Corollary 5.2** (Weight-output mapping in general)**.** *For any MLP $M$, $h_M : \mathcal{W}_M \to \mathcal{O}_M$ is not a linear mapping.*

*Proof.* Let $M$ have $L$ layers. By Definition 6, $M$ is a nested function of $L$ SLNs. Corollary 5.1 states that each of these SLNs does not have a linear mapping from weight to output space. Hence the composition of $L$ SLNs that forms $M$ does not have a linear mapping from weight to output space. □

**Remark.** The findings from Corollary 5.2 are very significant. They show that there is no apparent relationship between weight and output space that we can easily determine analytically. If there were a straightforward mapping between weight and output space, we would be able to simply determine the ideal weight configuration that would achieve our target $\mathbf{y}$ in output space.

However, since this is not possible, the findings above set the scene for the neural surfing technique. One of the core assumptions is that at a small enough scale, the mapping between weight and output space is *locally linear*, or at least close enough.

[15] Cauchy's functional equation is $f(a + b) = f(a) + f(b)$. For $a, b \in \mathbb{Q}$, the only solutions are linear functions of the form $f(x) = cx$ for some $c \in \mathbb{Q}$ [Reem 2017].

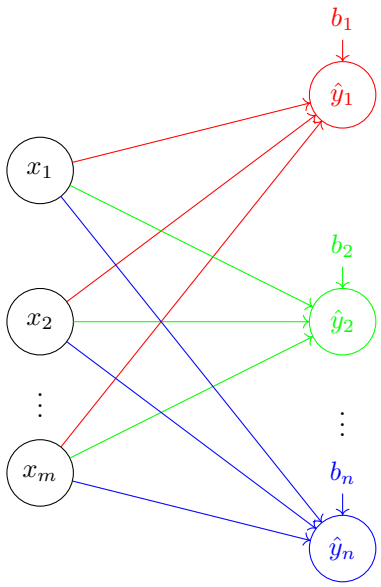



Figure 4.1: The DAGs representing the constructions of $n$ SLNs with one output from a SLN with $m$ inputs and $n$ outputs. Each color represents one of the constructed smaller SLNs.

### 4.1.2 Gradient descent from the perspective of weight and output space

Gradient descent with mean squared error is classically viewed from the perspective of the error-weight surface. In this regard, its objective is to descend the error-weight surface. Now that we have examined the relationship between the weight and output spaces, it follows naturally to determine how gradient descent fits in this frame of reference. It becomes apparent that in output space, MSE can be thought of as greedily trying to reduce the Euclidean distance from $\hat{\mathbf{y}}$ to $\mathbf{y}$, i.e. from the prediction to the target. To see this, recall Definition 9 and substitute

$$\|\hat{\mathbf{y}} - \mathbf{y}\| = \sqrt{\sum_{i=1}^{N} (\hat{y}_i - y_i)^2}$$

in Equation (3.1) to obtain $E(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{N} \|\hat{\mathbf{y}} - \mathbf{y}\|^2$. Since the number of samples $N$ is constant during training, there exists a linear relationship between the mean squared error and the Euclidean distance of the current output to the target output (goal).

## 4.2 Unrealisable regions

**Definition 14** (Strongly unrealisable point)**.** Given an artificial neural network $A$, a point $\mathbf{p} \in \mathcal{O}_A$ in output space is *strongly unrealisable* if and only if there exists no weight configuration $\mathbf{w} \in \mathcal{W}_A$ such that $h_A(\mathbf{w}) = \mathbf{p}$. In order words, it is impossible to attain $\mathbf{p}$.

**Definition 15** (Strongly unrealisable region)**.** Given an artificial neural network $A$, a *strongly* unrealisable region $\mathcal{U} \subset \mathcal{O}_A$ is a subspace of the output space where every point $\mathbf{p} \in \mathcal{U}$ is strongly unrealisable.

It is apparent that there exists no neural learning algorithm that can elicit a change in weight space that will attain a point in a strongly unrealisable region in output space. Hence we define a *weakly* unrealisable region for a particular neural learning algorithm as a region in output space that cannot be attained by a particular algorithm.

**Lemma 6.** *A strongly unrealisable region cannot encompass the whole output space.*

*Proof.* Let us consider an artifical neural network $A$. We will show that for every unrealisable region, $\mathcal{U} \subsetneq \mathcal{O}_A$. By Definition 15, $\mathcal{U} \subset \mathcal{O}_A$, so it remains to prove that every unrealisable region $\mathcal{U} \neq \mathcal{O}_A$.

Choose any weight configuration $\mathbf{w} \in \mathcal{W}_A$. Let the point $\mathbf{p} = h_A(\mathbf{w})$. We know that $\mathbf{p}$ is *not* strongly unrealisable because $\mathbf{w}$ achieves $\mathbf{p}$. Hence no unrealisable region can contain $\mathbf{p}$, so $\mathbf{p} \notin \mathcal{U}$ but $\mathbf{p} \in \mathcal{O}_A$. It follows that $\mathcal{U} \neq \mathcal{O}_A$. □

Let us look at a couple of examples of unrealisable regions.

**Example 3.** A trivial example of an unrealisable region is predicting two different outputs for the same training sample. Consider again an MLP $M$ with one layer and two inputs, as shown in Figure 2.5. Let $\mathbf{x} \in \mathbb{R}^2$ be any point in input space. For this example, let the training data be the matrix $\mathbf{X} = \begin{bmatrix}\mathbf{x} & \mathbf{x}\end{bmatrix}^\mathsf{T}$. Now we can define an unrealisable region

$$\mathcal{U} = \left\{ \begin{bmatrix} p_1 \\ p_2 \end{bmatrix} : p_1, p_2 \in \mathbb{R}, p_1 \neq p_2 \right\}$$

because $h_M(\mathbf{x})$ cannot produce two different outputs for the same value of $\mathbf{x}$.

**Example 4** (XOR mapping)**.** Let us look at a less contrived example. While it "is well known that any Boolean function [...] can be approximated by a suitable two-layer feed-forward network" [Blum 1989], single-layer networks with non-decreasing activation functions can only learn a Boolean mapping that is *linearly separable* [Russell and Norvig 2010, p. 723]. A linearly separable mapping has a linear decision boundary which means that there is only one linear hyperplane separating the two classes (true and false).

The XOR function is not linearly separable because it requires at least two linear decision boundaries, as shown in Figure 4.2. We will consider the same single-layer architecture from Figure 2.5 again, using the sigmoid activation function. The sigmoid is a non-decreasing function. Since we only have one unit (the output neuron) with this non-decreasing non-linear activation function, it follows that we can only have one decision boundary. We just showed that the XOR mapping requires two decision boundaries. Therefore, given the input matrix

$$\mathbf{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix},$$

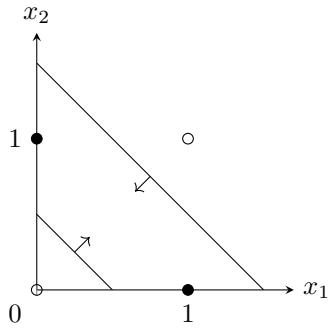



Figure 4.2: The locations of the two hyperplanes in input space acting as decision boundaries required to learn an XOR mapping. The filled-in dot represents an activation of 1 (true) and the circle represents 0 activation (false).

the point $\mathbf{p} = \begin{bmatrix} 0 & 1 & 1 & 0 \end{bmatrix}^\mathsf{T}$ in output space is strongly unrealisable. So $\mathcal{U} = \{\mathbf{p}\}$ is an example of an unrealisable region.

**Remark.** Example 3 demonstrate that unrealisable regions can arise even in very simple scenarios, so it is vital to take this phenomenon into account when designing a path-finding technique in output space, such as the neural surfer. Furthermore, one must also consider what happens if the target $\mathbf{y}$ lies in an unrealisable region (which means that $\mathbf{y}$ itself is unrealisable). In this case, the global minimum of the error-weight surface (using mean squared error) would be the weight configuration that produces the point closest to $\mathbf{y}$ (in terms of Euclidean distance) which is realisable.

## 4.3 Goal-connecting paths

**Definition 16** (Goal-connecting path)**.** For an artificial neural network $A$ with current weight configuration $\mathbf{w}_0 \in \mathcal{W}_A$, a goal-connecting path in output space to the goal $\mathbf{g} \in \mathcal{O}_A$ is a sequence of points $\mathbf{s}_0, \mathbf{s}_1, \ldots, \mathbf{s}_S \in \mathcal{O}_A$ where the initial state $\mathbf{s}_0 = h_A(\mathbf{w}_0)$ and final state $\mathbf{s}_S = \mathbf{g}$. The points $\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_S$ are referred to as *subgoals*, hence $S$ is the number of subgoals in the goal-connecting path.

The goal-connecting path is *realizable* if and only if no subgoal is a strongly unrealisable point (Definition 14). This means that a realizable goal-connecting path can equivalently be defined by the weight configurations $\mathbf{w}_0, \mathbf{w_1}, \ldots, \mathbf{w}_S \in \mathcal{W}_A$ such that $h_A(\mathbf{w}_i) = \mathbf{s}_i$ for $i \leq S$.

**Definition 17** (Ideal goal-connecting path)**.** The ideal goal-connecting path of $S$ subgoals for an artifical neural network $A$ is, in terms of weight space, the *shortest* realizable goal-connecting path with equidistant subgoals.

**Lemma 7.** *Given $\mathbf{w}_0$ and $\mathbf{w}_S$, the ith subgoal of the ideal goal-connecting path in weight space is given by $\mathbf{w}_i = \mathbf{w}_0 + \frac{i}{S}(\mathbf{w}_S - \mathbf{w}_0)$.*

*Proof.* First, we will show that the points are equidistant, i.e. $\|\mathbf{w}_0 - \mathbf{w}_1\| = \|\mathbf{w}_1 - \mathbf{w}_2\| = \cdots = \|\mathbf{w}_{S-1} - \mathbf{w}_S\|$. This is equivalent to asserting that $\|\mathbf{w}_{i-1} - \mathbf{w}_i\| =$

$\|\mathbf{w}_i - \mathbf{w}_{i+1}\|$ for $0 < i < S$. Substituting on the LHS,

$$\begin{aligned}\|\mathbf{w}_{i-1} - \mathbf{w}_i\| &= \left\|\mathbf{w}_0 + \frac{i-1}{S}(\mathbf{w}_S - \mathbf{w}_0) - \mathbf{w}_0 - \frac{i}{S}(\mathbf{w}_S - \mathbf{w}_0)\right\| \\ &= \left\|-\frac{1}{S}(\mathbf{w}_S - \mathbf{w}_0)\right\|,\end{aligned}$$

and on the RHS,

$$\begin{aligned}\|\mathbf{w}_i - \mathbf{w}_{i+1}\| &= \left\|\mathbf{w}_0 + \frac{i}{S}(\mathbf{w}_S - \mathbf{w}_0) - \mathbf{w}_0 - \frac{i+1}{S}(\mathbf{w}_S - \mathbf{w}_0)\right\| \\ &= \left\|-\frac{1}{S}(\mathbf{w}_S - \mathbf{w}_0)\right\|.\end{aligned}$$

The shortest path between two points is a straight line, so the ideal goal-connecting path in weight space must form a straight line from $\mathbf{w}_0$ to $\mathbf{w}_S$, and all subgoals must lie on this line. The equation of the line $l : \mathbb{R} \to \mathcal{W}_A$ from $\mathbf{w}_0$ to $\mathbf{w}_S$ is $l(\lambda) = \mathbf{w}_0 + \lambda(\mathbf{w}_S - \mathbf{w}_0)$. It is easy to see that $l\left(\frac{i}{S}\right) = \mathbf{w}_i$ for $0 \leq i \leq S$, so the subgoals are collinear. □

**Remark.** Lemma 7 shows a construction for achieving the ideal goal-connecting path, given knowledge of the target weight configuration $\mathbf{w}_S$. Of course, this is not known to the neural learning algorithm while it is learning, only once it finished and was able to actually achieve the goal. However, we can use the ideal goal-connecting path in retrospect to evaluate the performance of the neural learning algorithm in comparison to the ideal path. Furthermore, we can plot the ideal goal-connecting path in output space in order to find unrealisable regions.

## 4.4 A 'cheat' technique for evaluating subgoal trajectories

The neural surfer must be able to perform two main tasks: (i) set the subgoals in output space; and (ii) achieve these subgoals. This 'cheat' technique pertains only to the latter task, ignoring the former. It can be used in order to determine whether and to what extent a training regime[16] is *capable* of realising each subgoal. The idea is that splitting the problem into two parts will facilitate a better analysis of the neural surfing technique by isolating pain points and performance issues. Two main questions can be evaluated:

- How well is the approach getting to the subgoals?
- How good is the state the approach ends up in when it achieves the subgoal?

[16]This technique can be used for *any* regime that provides a mechanism for achieving (sub)goals. As such, this technique not only provides a means of testing the neural surfing algorithm, but also provides a means of using gradient to realise a subgoal trajectory.

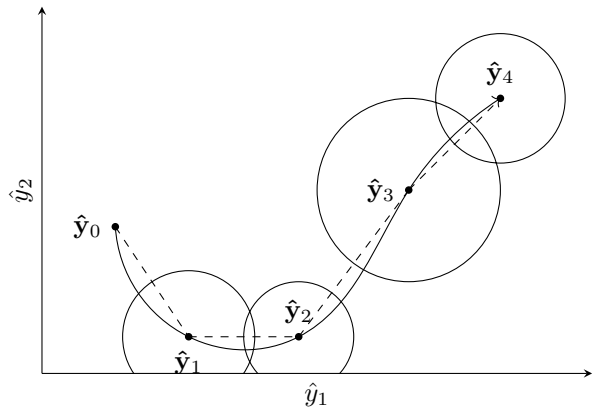



Figure 4.3: Output space for the 'cheat' technique for $S = 4$ subgoals and $\mu = \frac{1}{2}$. The circles around each point indicate the threshold that constitutes 'achieving' a particular point. The radius of the circle around point $\hat{\mathbf{y}}_i$ is $(1-\mu)\left\|\hat{\mathbf{y}}_i - \hat{\mathbf{y}}_{i-1}\right\|$ for $i = 1, \ldots, S$; hence the circles are not necessarily identical in size. While the weight configurations $\mathbf{w}_0, \ldots, \mathbf{w}_S$ are equidistant and collinear in weight space, it is important to note that the corresponding points in output space $\mathbf{y}_1, \ldots, \mathbf{y}_{n+1}$ will usually neither be collinear nor equidistant due to the inherent non-linearity of the network. In practice, $\mu$ will be set to a larger value such as 0.9 (which will decrease the circles' radii) to ensure that enough progress was made in achieving a subgoal before proceding to the next.

**Achieving a subgoal** One important question that must be answered is what constitutes realising a subgoal. For the purpose of this technique, we will assume that achieving some fraction $\mu \in (0, 1)$ of the distance to the subgoal constitutes as 'achieving' it. In more sophisticated scenarios, this technique could be extended to use functions that give a specific threshold for each subgoal.

**Procedure** The steps below explain the methodology of this technique, for $S$ subgoals (where the last subgoal represents the goal). Figure 4.3 illustrates this technique in output space.

1. Determine the point in weight space $\mathbf{w}_S$ that corresponds to the target in output space (i.e. the global minimum on the error-weight surface).

   This can be done either by (i) analytically finding the global minimum of the error-weight space if the network architecture is simple enough (see Theorem 14); or (ii) running gradient descent from many random initial configurations in order to find the (likely) global minimum.

2. Determine the ideal goal line $\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_S$ in weight space using Lemma 7 and the corresponding goal-connecting path $\hat{\mathbf{y}}_1, \hat{\mathbf{y}}_2, \ldots, \hat{\mathbf{y}}_S$ in output space.

3. Let $i = 0$. Starting at weight configuration $\mathbf{w}_i$, perform the particular neural network training technique until achieving a point $\mathbf{p}$ in output space where $\|\mathbf{p} - \hat{\mathbf{y}}_{i+1}\| \leq (1-\mu)\left\|\hat{\mathbf{y}}_i - \hat{\mathbf{y}}_{i+1}\right\|$, i.e. at least a fraction $\mu$ of the distance to the next subgoal is achieved.

4. Repeat the previous step for $i = 1, 2, \ldots, S$.

We will employ this technique in Sections 5.3.3 and 5.3.4 to evaluate the feasability of the ideal goal line in a suboptimal local minimum setting using both gradient-based and derivative-free optimisation regimes.

# Chapter 5

# The local minimum problem

The previous chapters explained the theory behind neural networks and their classical training algorithms. We looked at some of these algorithms' shortcomings in Section 3.4 and hypothesised why they would potentially fail to find global minima in the presence of suboptimal local minima. Now we can define the local minimum problem more accurately, contrive an example of said problem, and finally not only analyse it from the classical viewpoint of Chapter 2, but also relate it to the paradigm of Chapter 4.

## 5.1 The mathematics of local and global minima

In order to contrive and analyse an instance of the local minimum problem, we must formally define what is meant by 'local' and 'global' minima and identify the necessary and sufficient conditions that prove their existence. We will do this by returning to the running analogy of hiking in the mountains to provide some intuition. Let us begin with the global minimum, which is simply a point with an elevation lower than any other point.

**Definition 18** (Global minimum)**.** The function $f : \mathbb{R}^n \to \mathbb{R}$ has a *global minimum* at point $\mathbf{p} \in \mathbb{R}^n$ if and only if for all $\mathbf{x} \in \mathbb{R}^n$ it is true that $f(\mathbf{p}) \leq f(\mathbf{x})$.

We claimed in Section 1.1 to have a local minimum if the surroundings of our one-metre radius of vision are of higher elevation than the ground beneath our feet. To define local minima, we will use the same concept, just allowing that the radius of the circle describing the visible surroundings can be arbitrarily small.

**Definition 19** (Local minimum)**.** The function $f : \mathbb{R}^n \to \mathbb{R}$ has a *local minimum* at point $\mathbf{p} \in \mathbb{R}^n$ if there exists a ball with centre $\mathbf{p}$ where $f(\mathbf{p}) \leq f(\mathbf{x})$ for all points $\mathbf{x}$ in that ball. A *suboptimal local minimum* is a local minimum that is not a global minimum.

Now it is possible to define the local minimum problem using these terms.

**Problem 1** (Local minimum problem)**.** A training algorithm exhibits the local minimum problem if there exists a weight initialisation that leads to training converging to a suboptimal local minimum on the error-weight surface.

In order to later analyse error-weight surfaces analytically, we must derive the conditions that are necessary and sufficient for local minima to exist. We must first realise that if our immediate surroundings are of higher elevation, then the infinitesimal slope beneath our feet is completely flat (because moving just a bit in any direction will lead to a higher elevation). In formal terms, this infinitesimal slope is the gradient, or vector of first partial derivatives, which is given by the *Jacobian*.

**Definition 20** (Jacobian)**.** Let $f : \mathbb{R}^n \to \mathbb{R}$ be a continuously differentiable function of the form $f = f(\mathbf{x})$ where $\mathbf{x} = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}^\mathsf{T}$. The Jacobian of $f$ (or simply *derivative* of $f$) is a row vector of its first-order partial derivatives,

$$\mathbf{J}_f = \frac{\delta f}{\delta \mathbf{x}} = \begin{bmatrix} \frac{\delta f}{\delta x_1} & \frac{\delta f}{\delta x_2} & \cdots & \frac{\delta f}{\delta x_n} \end{bmatrix}.$$

Now that we have defined the Jacobian as representing the infinitesimal slope of the mountain, we must realise an important fact. Knowing that the slope of the mountain is flat at a specific point is not a sufficient condition for there being a local minimum; after all, we could be talking about a global maximum (or a saddle point). We will call these points *critical points*. To determine whether a critical point constitutes a local minimum, we must examine its immediate surrounding, to see if these points are of higher elevation. This will be achieved by looking at the second partial derivatives given by the *Hessian*, more specifically the existence of a local minima is predicated on whether the Hessian is *positive definite*. Let us define these terms.

**Definition 21** (Hessian)**.** Let $f : \mathbb{R}^n \to \mathbb{R}$ be a function of the form $f = f(\mathbf{x})$ where $\mathbf{x} = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}^\mathsf{T}$ for which all second partial derivatives exist and are continuous over $\mathbb{R}^n$. The Hessian of $f$ is a $n \times n$ matrix of the second-order partial derivatives, given by

$$\mathbf{H}_f = \begin{bmatrix} \frac{\delta^2 f}{\delta x_1^2} & \frac{\delta^2 f}{\delta x_1 x_2} & \cdots & \frac{\delta^2 f}{\delta x_1 x_n} \\ \frac{\delta^2 f}{\delta x_2 x_1} & \frac{\delta^2 f}{\delta x_2^2} & \cdots & \frac{\delta^2 f}{\delta x_2 x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\delta^2 f}{\delta x_n x_1} & \frac{\delta^2 f}{\delta x_n x_2} & \cdots & \frac{\delta^2 f}{\delta x_n^2} \end{bmatrix}.$$

**Definition 22** (Positive definite matrix)**.** A symmetric $n \times n$ matrix $\mathbf{M} \in \mathbb{R}^{n \times n}$ is said to be *positive definite* if and only if $\mathbf{x}^\mathsf{T}\mathbf{M}\mathbf{x} > 0$ for all $\mathbf{x} \in \mathbb{R}^n \setminus \mathbf{0}$.

We will prove a simple technique for showing that a $2 \times 2$ matrix is positive definite, as this will help us prove the existence of local minima later.

**Theorem 8** ($2 \times 2$ positive definite matrix)**.** *The matrix* $\mathbf{M} = \begin{bmatrix} a & b \\ b & c \end{bmatrix}$ *is positive definite if* $a > 0$ *and* $ac - b^2 > 0$.

*Proof.* Let $\mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}^\mathsf{T}$ and $f(x_1, x_2) = \mathbf{x}^\mathsf{T}\mathbf{M}\mathbf{x}$. Performing the multiplication,

$$\begin{aligned} f(x_1, x_2) &= \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} a & b \\ b & c \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \\ &= \begin{bmatrix} x_1 a + x_2 b & x_1 b + x_2 c \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \\ &= x_1^2 a + 2x_1 x_2 b + x_2^2 c. \end{aligned}$$

We will now discuss under what conditions $f(x_1, x_2) > 0$ holds. When $x_2 = 0$, it must necessarily be the case that $a > 0$. Otherwise, dividing by $x_2^2$, we can write

$$\left(\frac{x_1}{x_2}\right)^2 a + \frac{2x_1 b}{x_2} + c > 0,$$

and substituting $p = \frac{x_1}{x_2}$ we get

$$p^2 a + 2pb + c > 0.$$

Treating this as a quadratic function in terms of $p$, we realise that since $a > 0$, the parabola opens up. The discriminant $D = 4b^2 - 4ac$ is negative if $ac - b^2 > 0$ which means that the parabola has no roots. Thus its range is positive.

We have shown that $f(x_1, x_2) > 0$ if $a > 0$ and $ac - b^2 > 0$ for all $x_1, x_2 \in \mathbb{R}$ except $x_1 = x_2 = 0$. By Definition 22, $\mathbf{M}$ must be positive definite. □

Let us finally introduce a theorem that we can later use as a tool to prove the existence of local minima.

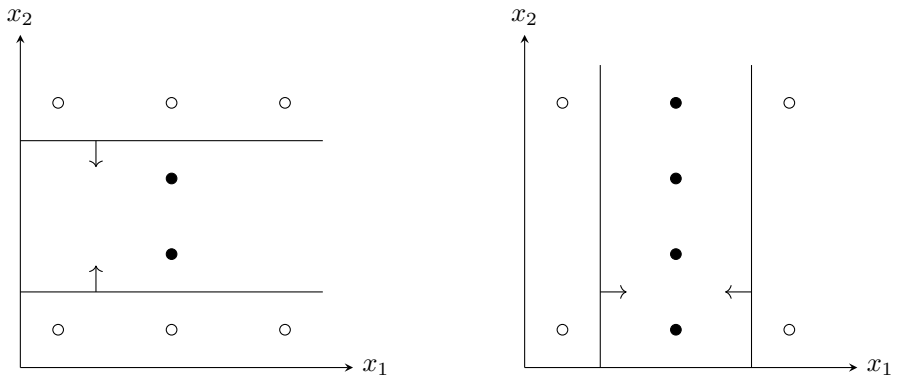
**Theorem 9** (Local minimum)**.** *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a continuously differentiable function, and let the point $\mathbf{p} \in \mathbb{R}^n$ be such that $\mathbf{J}_f(\mathbf{p}) = \mathbf{0}$ and $\mathbf{H}_f(\mathbf{p})$ is positive definite. Then $\mathbf{p}$ is a local minimum of $f$.*

We will require this theorem to justify that gradient-based training converges to a local minimum in Section 5.2. However, the proof of this theorem exceeds the scope of this report. The interested reader may consult Loomis and Sternberg [1990, p. 190].

## 5.2 The stripe problem

The stripe problem provides a practical example of the local minimum problem. We will consider a two-dimensional input space with two (initially) parallel hyperplanes that form a stripe. The initial configuration will have the hyperplanes arranged horizontally and misclassify some of the samples. Changing the weights in the neural network will allow the hyperplanes to rotate, but when they do so, they are forced to misclassify even more samples (thereby increasing the mean squared error) until eventually reaching the target configuration with zero error.

The stripe problem can be achieved by a 2-2-1 sigmoidal MLP [Weir 2019]. Figure 5.1 shows the initial and target configurations of the hyperplanes for this




(a) Initial configuration (b) Target configuration

Figure 5.1: Hyperplanes in input space for the stripe problem with eight samples (adapted from Weir [2019]).

type of network. The hidden layer is required because without it, the sigmoid activation function will produce only one hyperplane, as proved in Lemma 2.

In this section, we will formulate a version of the stripe problem that requires no hidden layers and fewer input samples by using a different kind of activation function. The reduced number of parameters will lend itself better for analysis.

### 5.2.1 Radial basis activation functions

We will first introduce the concept of radial basis functions. When used as the activation function, they exhibit some advantageous properties that will aid us to contrive a simple version of the stripe problem.

**Definition 23** (Radial basis function)**.** A radial basis function (RBF) is a smooth continuous real-valued[17] function $\phi : \mathbb{R} \to \mathbb{R}$ that satisfies the property $\phi(x) = \phi(\|x\|)$ [Buhmann 2000]. For RBFs to be useful activation functions, we three additional restrictions: (i) $\phi(0) = 1$; (ii) $\phi(x)$ is strictly decreasing for the domain $[0, \infty)$ when $\phi(x) \neq 0$; and (iii) that

$$\lim_{x \to -\infty} \phi(x) = \lim_{x \to \infty} \phi(x) = 0.$$

Two commonly used RBFs, both infinitely differentiable, are the Gaussian function

$$\phi(x) = e^{-x^2} \tag{5.1}$$

and the bump function[18]

$$\phi(x) = \begin{cases} e^{1 - \frac{1}{1-x^2}} & \text{for } -1 < x < 1 \\ 0 & \text{otherwise} \end{cases}. \tag{5.2}$$

These functions, graphed in Figure 5.2, exhibit slightly different properties. The

[17] We define RBFs has having a scalar domain and range because this suffices for our purposes. In actual fact, RBFs are defined more generally to map between suitable vector spaces [Buhmann 2000].

[18] We give a slightly modified version of the well-known $C_\infty$ "bump" function [Johnson 2015] that is vertically scaled such that $\phi(0) = 1$ for convenience.

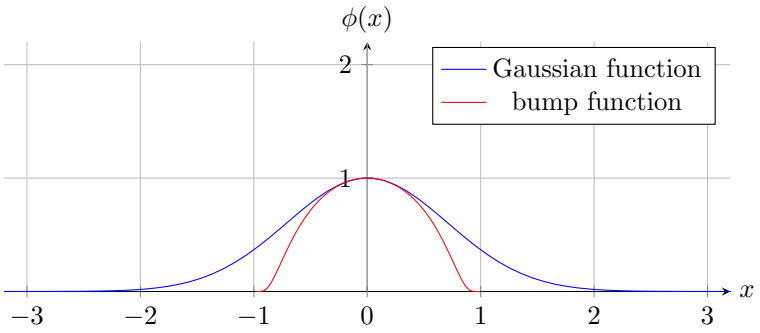



Figure 5.2: Plots of the two most common radial basis functions.

Gaussian function never actually reaches zero and its derivative is never zero (except at the peak, i.e. $x = 0$). On the other hand, for the bump function, we have $\phi(x) = 0$ and $\frac{d\phi}{dx} = 0$ for $x \notin (-1, 1)$.

**Lemma 10** (Single-layer RBF decision boundaries)**.** *A single-layer Gaussian RBF MLP with decision threshold $t \in (0, 1)$ will have two hyperplanes in input space.*

*Proof.* The proof is similar in method as in Lemma 2. Consider an equivalent SLN $S$ with $m$ inputs and one output as shown in Figure 2.4. To obtain the decision boundaries, we set the output equal to the decision threshold, so

$$\begin{aligned} t &= S(\mathbf{x}) \\ &= \phi\left(\mathbf{w}^{\mathsf{T}}\mathbf{x} + b\right) \\ &= e^{-\left(\mathbf{w}^{\mathsf{T}}\mathbf{x}+b\right)^2} \\ -\ln t &= \left(\mathbf{w}^{\mathsf{T}}\mathbf{x} + b\right)^2 \\ \pm\sqrt{-\ln t} &= \mathbf{w}^{\mathsf{T}}\mathbf{x} + b. \end{aligned}$$

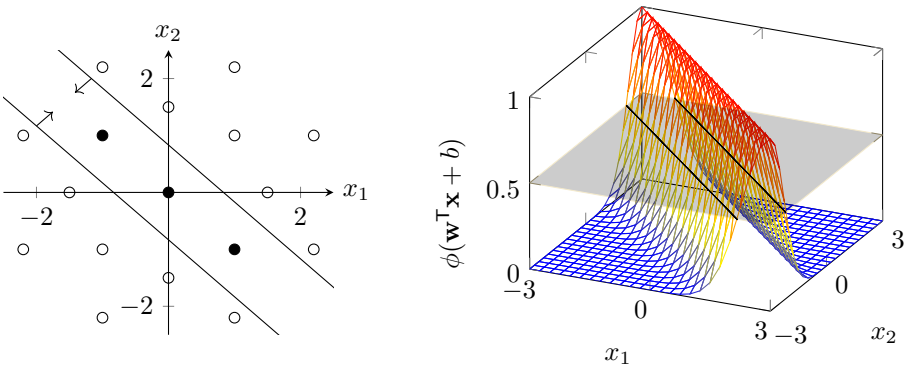
Since $t \in (0, 1)$, it follows that $\ln t < 0$ and thus $\sqrt{-\ln t} > 0$ which of course means that $\sqrt{-\ln t} \neq 0$. Hence

$$\mathbf{w}^{\mathsf{T}}\mathbf{x} + b \pm \sqrt{-\ln t} = 0 \tag{5.3}$$

has two distinct solutions, no matter the values of $\mathbf{w}$, $\mathbf{x}$, and $b$. Thus there will always be two hyperplanes. □

**Remark.** Although Lemma 10 proves the existence of two hyperplane decision boundaries for Gaussian RBFs, it is trivial to modify this proof for any other type of RBF, such as the bump function. As a consequence, we know that unlike single-layer sigmoidal networks (see Lemma 2), we can use single-layer RBF networks to generate a decision boundary in the form of a stripe which will allow us to use this type of network to provide a more simple example of the stripe problem.

**Example 5.** Like in Example 1, let us consider once more a single-layered MLP $M$ with two inputs, depicted in Figure 2.5, where $w_1 = w_2 = 1$ and $b = 0$. Let




(a) Input space with sample data

(b) Input-activation space

Figure 5.3: Plots of the hyperplanes of the MLP from Figure 2.5 with Gaussian RBF activation where $w_1 = w_2 = 1$, $b = 0$.

the threshold be $t = \frac{1}{2}$ again, but this time, we will use the Gaussian RBF as the activation function.

From Equation (5.3), we obtain the equations of the hyperplanes as

$$\begin{aligned} \mathbf{w}^\mathsf{T}\mathbf{x} + b \pm \sqrt{-\ln\frac{1}{2}} &= 0 \\ w_1x_1 + w_2x_2 + b &= \pm\sqrt{-\ln\frac{1}{2}} \\ x_1 + x_2 &= \pm\sqrt{-\ln\frac{1}{2}}, \end{aligned}$$

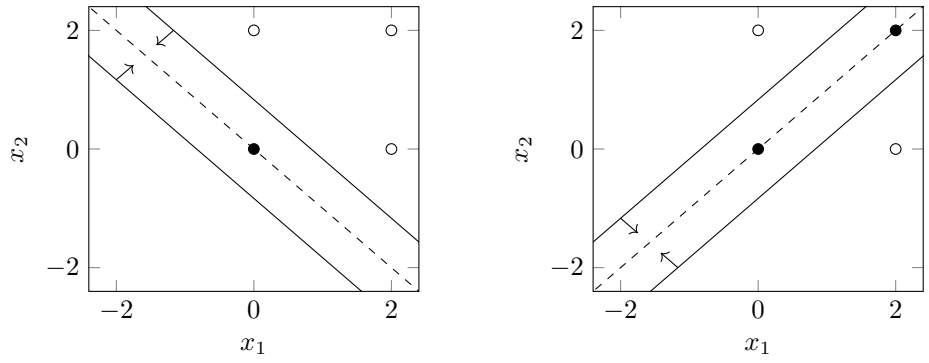
so the hyperplanes are at $x_2 = -x_1 - 0.8325\ldots$ and $x_2 = -x_1 + 0.8325\ldots$, as shown in Figure 5.3.

**Remark.** One key realisation is that when $w_1$ is fixed, changing the value of $w_2$ will result in both hyperplanes being rotated around their respective $x_1$-intercepts (the hyperplanes remain parallel). The same is true vice-versa, except that the hyperplanes are rotated around their $x_2$-intercepts. Changing the value of $b$ simply translates the hyperplanes linearly in input space.

### 5.2.2 Formulating the problem

Example 5 showed that a simple 2-1 network with radial basis activation constructs a scenario where the hyperplanes form a stripe that can be rotated by adjusting the weights. This means that we could easily contrive the stripe problem from Figure 5.1. However, since we established that the hyperplanes will always remain parallel in our RBF network and since they rotate around the origin[19] when $b$ is fixed, we can create a much simplified version of the stripe problem using only four samples.

[19]More precisely, when changing $w_1$, the hyperplanes rotate around their respective $x_2$ intercepts, and similarly when altering $w_2$ the centre of rotation are the $x_1$-intercepts in input space.




(a) Initial configuration

(b) Target configuration

Figure 5.4: The hyperplanes in the initial and target configurations of the RBF stripe problem. The dashed line represents the zero-excitation line.

Figure 5.4 depicts the initial and target configurations of this simplified version. It is obvious that whichever direction the stripe rotates, it will need to misclassify one of the samples before achieving the target configuration. On the other hand, the zero-excitation line must always pass through the origin (because we have no bias term) which means that the sample at the origin will always have maximal activation. Hence we can discard this sample, leaving only three samples to consider. This will lend itself comfortably for analysis later, as the output space can be visualised in a three-dimensional plot.

| input ($\mathbf{x}$) | target output ($y$) | initial output ($\hat{y}$) |
|---|---|---|
| $\begin{bmatrix}2 & 2\end{bmatrix}$ | 1 | $\phi(4)$ |
| $\begin{bmatrix}0 & 2\end{bmatrix}$ | $\phi(2)$ | $\phi(2)$ |
| $\begin{bmatrix}2 & 0\end{bmatrix}$ | $\phi(2)$ | $\phi(2)$ |

Table 5.1: The dataset for the RBF stripe problem.

We give the dataset for the stripe problem in Table 5.1. Notice that the second and third samples do not have a target output of zero, but rather $\phi(2)$ which is close to zero (for the Gaussian RBF, $\phi(2) \approx 0.018$, and for the bump function $\phi(2) = 0$). This will make some calculations later more convenient because it guarantees that there exists at least one weight configuration with a MSE of zero.

Let us examine the error-weight surface of the stripe problem. It is depicted in Figures 5.5 and 5.6 for the Gaussian and bump activation functions, respectively. The graphs are quite similar, showing that the initial weight configuration $\mathbf{w}_0$ has a MSE of around 1. Most importantly, we see that in order to get to any of the goal weight states $\mathbf{w}_S$ in the region where the MSE is close to zero, we must first overcome a 'hill'.

For the remainder of this project, we will consider only the Gaussian RBF function, but let it be noted that in principle, any type of RBF that satisfies the criteria given in Definition 23 is suitable. The Gaussian RBF, however, lends itself better for the purposes of analysis because it is not a piecewise defined function, and it does not have a derivative of zero anywhere except at $x = 0$.

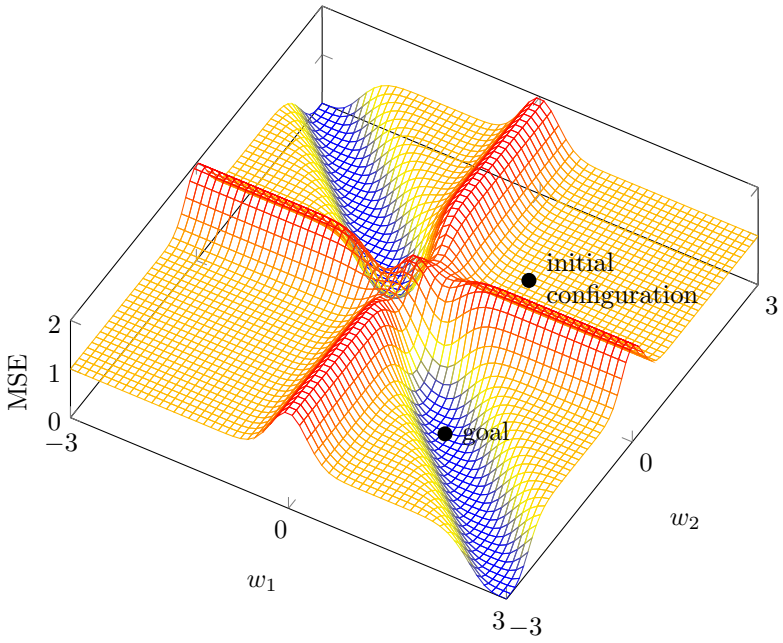



Figure 5.5: Error-weight surface of the stripe problem with Gaussian activation.

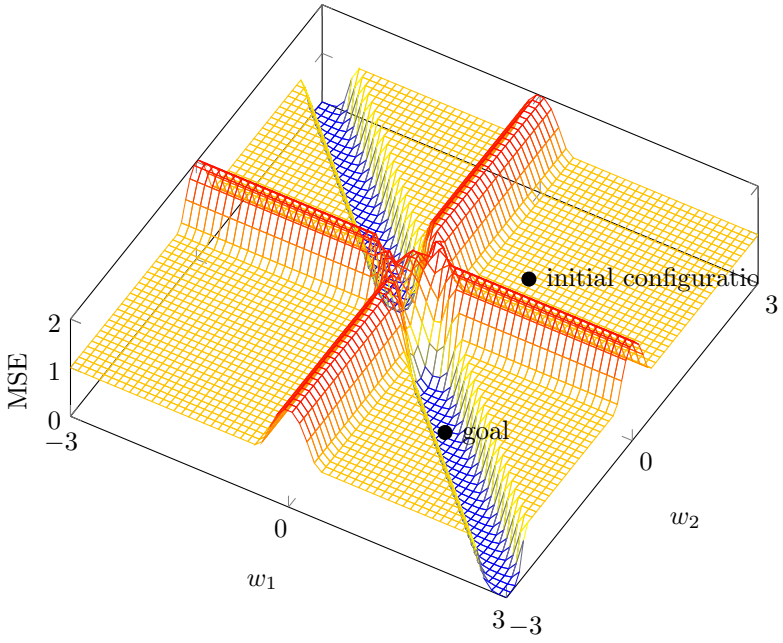



Figure 5.6: Error-weight surface of the stripe problem with bump activation.

In addition, gradient descent fails immediately with the bump function because the initial weight configuration is at an area where the gradient is exactly zero. Henceforth the term 'stripe problem' shall be used to refer to the Gaussian RBF version of the stripe problem.

### 5.2.3 Critical points

We will now attempt to find the critical of the error-weight surface analytically, so we can subsequently determine the local minima in the next step. From Table 5.1 we obtain the input matrix

$$\mathbf{X} = \begin{bmatrix} 2 & 2 \\ 0 & 2 \\ 2 & 0 \end{bmatrix}$$

and target output vector $\mathbf{y} = \begin{bmatrix} 1 & \phi(2) & \phi(2) \end{bmatrix}^\mathsf{T}$. Modifying the SLN function from Definition 5 for the case of our 2-1 RBF network without bias, we have $S(\mathbf{x}) = \phi(\mathbf{w}^\mathsf{T}\mathbf{x})$. This means that our loss function is given by

$$\begin{aligned} L(\mathbf{w}) = L &= \sum_{i=1}^{3} \left(\phi\left(\mathbf{w}^\mathsf{T}\mathbf{x}_i\right) - y_i\right)^2 \\ &= \left(\phi(\mathbf{w}^\mathsf{T}\mathbf{x}_1) - 1\right)^2 + \left(\phi(\mathbf{w}^\mathsf{T}\mathbf{x}_2) - \phi(2)\right)^2 + \left(\phi(\mathbf{w}^\mathsf{T}\mathbf{x}_3) - \phi(2)\right)^2 \\ &= \left(\phi(2w_1 + 2w_2) - 1\right)^2 + \left(\phi(2w_2) - \phi(2)\right)^2 + \left(\phi(2w_1) - \phi(2)\right)^2. \end{aligned}$$

Substituting Equation (5.1), we obtain

$$L = \left(e^{-4(w_1+w_2)^2} - 1\right)^2 + \left(e^{-4w_2^2} - e^{-4}\right)^2 + \left(e^{-4w_1^2} - e^{-4}\right)^2. \tag{5.4}$$

We will now identify and examine the critical points of this function.

**Lemma 11.** *Three critical points of the error-weight surface given by $L$ are $\mathbf{p}_1 = \begin{bmatrix} -1 & 1 \end{bmatrix}^\mathsf{T}$, $\mathbf{p}_2 = \begin{bmatrix} 1 & -1 \end{bmatrix}^\mathsf{T}$, and $\mathbf{p}_3 = \begin{bmatrix} 0 & 0 \end{bmatrix}^\mathsf{T}$.*

*Proof.* We need to find the Jacobian, $\mathbf{J}_L$ like in Example 2. Differentiating $L$ with respect to $w_1$ gives

$$\begin{aligned} \frac{\delta L}{\delta w_1} &= 2\left(e^{-4(w_1+w_2)^2} - 1\right)\frac{\delta}{\delta w_1}e^{-4(w_1+w_2)^2} \\ &\quad + 2\left(e^{-4w_1^2} - e^{-4}\right)\frac{\delta}{\delta w_1}e^{-4w_1^2} \\ &= -16\left(e^{-(2w_1+2w_2)^2} - 1\right)(w_1 + w_2)\, e^{-4(w_1+w_2)^2} \\ &\quad - 16w_1\left(e^{-4w_1^2} - e^{-4}\right)e^{-4w_1^2}, \end{aligned}$$

and for $w_2$ we have

$$\begin{aligned}\frac{\delta L}{\delta w_2} &= 2\left(e^{-4(w_1+w_2)^2}-1\right)\frac{\delta}{\delta w_2}e^{-4(w_1+w_2)^2} \\ &\quad + 2\left(e^{-4w_2^2}-e^{-4}\right)\frac{\delta}{\delta w_2}e^{-4w_2^2} \\ &= -16\left(e^{-4(w_1+w_2)^2}-1\right)(w_1+w_2)\,e^{-4(w_1+w_2)^2} \\ &\quad - 16w_2\left(e^{-4w_2^2}-e^{-4}\right)e^{-4w_2^2}.\end{aligned}$$

This allows us to express the Jacobian as

$$\begin{aligned}\mathbf{J}_L &= \begin{bmatrix}\frac{\delta L}{\delta w_1} & \frac{\delta L}{\delta w_2}\end{bmatrix} \\ &= -16\left(e^{-4(w_1+w_2)^2}-1\right)(w_1+w_2)\,e^{-4(w_1+w_2)^2} \\ &\quad - 16\begin{bmatrix}w_1\left(e^{-4w_1^2}-e^{-4}\right)e^{-4w_1^2} \\ w_2\left(e^{-4w_2^2}-e^{-4}\right)e^{-4w_2^2}\end{bmatrix}^{\mathsf{T}}. \end{aligned} \tag{5.5}$$

To find the critical points, we set $\mathbf{J}_L = \mathbf{0}$. It is trivial to see that $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_3$ are solutions to this equation. □

At first, it seems like these three points are the only solutions to $\mathbf{J}_L = \mathbf{0}$. However, upon examining some trial runs of gradient descent, there seemed to be two more local minima around $\begin{bmatrix}1 & 1\end{bmatrix}^{\mathsf{T}}$ and $\begin{bmatrix}-1 & -1\end{bmatrix}^{\mathsf{T}}$.

**Lemma 12.** *Two more critical points are at* $\mathbf{p}_4 = \begin{bmatrix}\eta & \eta\end{bmatrix}^{\mathsf{T}}$ *and* $\mathbf{p}_5 = \begin{bmatrix}-\eta & -\eta\end{bmatrix}^{\mathsf{T}}$ *where* $\eta$ *can be numerically approximated to* 0.999916 *(6 s.f.).*

*Proof.* We can find these critical points by setting $w_1 = w_2$ in the Jacobian; due to the symmetry of the Jacobian this results in both components becoming. Setting $\mathbf{J}_L = 0$ with $w_1 = w_2$, we obtain

$$\begin{aligned}-32\left(e^{-16w_1^2}-1\right)w_1e^{-16w_1^2} - 16w_1\left(e^{-4w_1^2}-e^{-4}\right)e^{-4w_1^2} &= 0 \\ 2\left(e^{-16w_1^2}-1\right)w_1e^{-16w_1^2} + w_1\left(e^{-4w_1^2}-e^{-4}\right)e^{-4w_1^2} &= 0.\end{aligned}$$

Noting that $w_1 = 0$ is a solution (which we have already found in Lemma 11), we can divide by $w_1$ to obtain

$$\begin{aligned}0 &= e^{-32w_1^2}\left(2+e^{28w_1^2}\right) - 2e^{-16w_1^2} - e^{-4} \\ &= 2e^{-32w_1^2} + e^{-4w_1^2} - 2e^{-16w_1^2} - e^{-4}.\end{aligned}$$

Substituting $x = e^{-4w_1^2}$, we get the equation

$$2x^8 - 2x^4 + x - e^{-4} = 0$$

which cannot be simplified further. Using a numerical solver, we can find the two roots of this equation and then find the value of $w_1 = \pm\frac{1}{2}\sqrt{-\log x}$ (discarding the non-real solutions) as $w_1 \approx 0.999916$ and $w_1 \approx -0.999916$. Since $w_1 = w_2$ we get the solutions $\mathbf{p}_4$ and $\mathbf{p}_5$. □

For the remainder of this project, we will assume that these five critical points are the only ones. Analysing the graph from Figure 5.5 does suggest so.

### 5.2.4 Local minima

**Lemma 13.** *The local minima on the error-weight surface given by $L$ are $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$.*

*Proof.* We have previously shown that $\mathbf{J}_L = \mathbf{0}$ for the three critical points $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_3$. Let us now compute the Hessian. We will express the Jacobian from Equation (5.5) as

$$\mathbf{J}_L = -16\left(r + \begin{bmatrix} s(w_1) \\ s(w_2) \end{bmatrix}^{\mathsf{T}}\right) \tag{5.6}$$

where

$$\begin{aligned} q &= e^{-4(w_1+w_2)^2} \\ r &= (q^2 - q)(w_1 + w_2) \\ s(x) &= x\left(e^{-8x^2} - e^{-4-4x^2}\right). \end{aligned}$$

Let us first compute the derivatives of $q$ with respect to $w_1$ and $w_2$, which, interestingly enough, are equal:

$$\frac{\delta q}{\delta w_1} = \frac{\delta q}{\delta w_2} = -8q(w_1 + w_2).$$

The derivative of $r$ with respect to $w_1$ is

$$\begin{aligned} \frac{\delta r}{\delta w_1} &= q^2 - q + (w_1 + w_2)\frac{\delta}{\delta w_1}(q^2 - q) \\ &= q^2 - q + (w_1 + w_2)(2q - 1)\frac{\delta q}{\delta w_1} \\ &= q^2 - q - 8q(w_1 + w_2)^2(2q - 1). \end{aligned}$$

Here, it can be shown that $\frac{\delta r}{\delta w_1} = \frac{\delta r}{\delta w_2}$. The exact derivation is left as an exercise to the reader.

It remains to find the derivative of $s$,

$$\begin{aligned} s'(x) &= e^{-8x^2} - e^{-4-4x^2} + x\frac{\delta}{\delta x}\left(e^{-8x^2} - e^{-4-4x^2}\right) \\ &= e^{-8x^2} - e^{-4-4x^2} - 8x^2\left(2e^{-8x^2} - e^{-4-4x^2}\right). \end{aligned}$$

Calculating all the second derivatives of Equation (5.6), we get

$$\begin{aligned} \frac{\delta^2 L}{\delta w_1^2} &= -16\frac{\delta}{\delta w_1}\left(r + s(w_1)\right) \\ &= -16\left(\frac{\delta r}{\delta w_1} + s'(w_1)\right), \end{aligned}$$

$$\begin{aligned} \frac{\delta^2 L}{\delta w_1 w_2} &= -16\frac{\delta}{\delta w_2}\left(r + s(w_1)\right) \\ &= -16\frac{\delta r}{\delta w_2} = -16\frac{\delta r}{\delta w_1}, \end{aligned}$$

$$\begin{aligned}\frac{\delta^2 L}{\delta w_2 w_1} &= -16\frac{\delta}{\delta w_1}\left(r + s(w_2)\right)\\ &= -16\frac{\delta r}{\delta w_1},\end{aligned}$$

and

$$\begin{aligned}\frac{\delta^2 L}{\delta w_2^2} &= -16\frac{\delta}{\delta w_2}\left(r + s(w_2)\right)\\ &= -16\left(\frac{\delta r}{\delta w_2} + s'(w_2)\right) = -16\left(\frac{\delta r}{\delta w_1} + s'(w_2)\right).\end{aligned}$$

This allows us to write an expression for the Hessian matrix in the form

$$\mathbf{H}_L = \begin{bmatrix}-16s'(w_1) & 0\\ 0 & -16s'(w_2)\end{bmatrix} - 16\frac{\delta r}{\delta w_1}. \tag{5.7}$$

**First critical point** $(w_1 = -1, w_2 = 1)$ By Theorem Theorem 8, $\mathbf{H}_L$ is positive definite if and only if $a > 0$ and $ac - b^2 > 0$ when expressing the matrix in the form $\mathbf{H}_L = \begin{bmatrix}a & b\\ b & c\end{bmatrix}$. We will begin by showing $a = \frac{\delta^2 L}{\delta w_1^2} > 0$. To do that, we must first evaluate $q$,$s'(w_1)$, and $\frac{\delta r}{\delta w_1}$ for the current weight configuration.

$$\begin{aligned}q &= e^{-4(w_1+w_2)^2} = 1\\ s'(w_1) &= e^{-8w_1^2} - e^{-4-4w_1^2} - 8w_1^2\left(2e^{-8w_1^2} - e^{-4-4w_1^2}\right) = -8e^{-8}\\ \frac{\delta r}{\delta w_1} &= q^2 - q - 8q(w_1+w_2)^2(2q-1) = 0.\end{aligned}$$

We get

$$\begin{aligned}a &= -16s'(w_1) - 16\frac{\delta r}{\delta w_1}\\ &= 128e^{-8} > 0.\end{aligned}$$

It remains to show that $ac - b^2 > 0$. Noticing that $s'(1) = s'(-1)$, we realise that in fact $a = c$. So,

$$\begin{aligned}ac - b^2 &= a^2 - b^2\\ &= 128^2 e^{-16} - \left(-16\frac{\delta r}{\delta w_1}\right)^2\\ &= 128^2 e^{-16} > 0.\end{aligned}$$

Hence, we showed that the point $\mathbf{p}_1$ forms a local minimum.

**Second critical point** $(w_1 = 1, w_2 = -1)$ Notice that this point can be obtained simply by switching $w_1$ and $w_2$ from the previous critical point. We have showed previously that $s'(-1) = s'(1)$ and furthermore we established earlier that $\frac{\delta r}{\delta w_1} = \frac{\delta r}{\delta w_2}$. Hence the Hessian from Equation (5.7) will be positive definite too, thus proving that the second critical point is a local minimum.

**Third critical point** ($w_1 = w_2 = 0$) Using the same notation as Equation (5.7), we evaluate $q$,$s'(w_1)$, and $\frac{\delta r}{\delta w_1}$ for this weight configuration.

$$\begin{aligned} q &= e^{-4(w_1+w_2)^2} = 1 \\ s'(w_1) &= e^{-8w_1^2} - e^{-4-4w_1^2} - 8w_1^2\left(2e^{-8w_1^2} - e^{-4-4w_1^2}\right) = 1 - e^{-4} \\ \frac{\delta r}{\delta w_1} &= q^2 - q - 8q(w_1+w_2)^2(2q-1) = 0. \end{aligned}$$

We get

$$\begin{aligned} a &= -16s'(w_1) - 16\frac{\delta r}{\delta w_1} \\ &= 16e^{-4} - 16 \not> 0. \end{aligned}$$

This violates the first condition of Theorem Theorem 8, so this point is not a local minimum.

**Fourth critical point** ($w_1 = w_2 \approx 0.999916$) Calculating the Hessian for $\mathbf{p}_4$ numerically, we obtain

$$\mathbf{H}_L \approx \begin{bmatrix} 0.04295905890 & -0.00005595797825 \\ -0.00005595797825 & 0.04295905890 \end{bmatrix}$$

and it is easy to see that the determinant of $\mathbf{H}_L$ as well as its top left cell are positive, so we have a local minimum.

**Fifth critical point** ($w_1 = w_2 \approx -0.999916$) The Hessian obtained at $\mathbf{p}_5$ is equal to the Hessian at $\mathbf{p}_4$. It follows that this point is a local minimum as well. In summary, we have shown that $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_4$, $\mathbf{p}_5$ are local minima, but not $\mathbf{p}_3$. □

### 5.2.5 Global minima

In Section 5.2.3 we identified five critical points on the error-weight surface for the stripe problem, and in Section 5.2.4 we have shown that all but one of these points are local minima. Now we will determine which of these local minima are also global minima. Figure 5.7 shows the critical points which allows the hypothesis that if we have identified all local minima, then $\mathbf{p}_1$ and $\mathbf{p}_2$ should be the global minima. While we were not able to prove that the five critical points we identified are the only ones, we *can* prove that $\mathbf{p}_1$ and $\mathbf{p}_2$ are in fact global minima because their loss values minimal.

**Theorem 14.** *The points* $\mathbf{p}_1 = \begin{bmatrix} -1 & 1 \end{bmatrix}^\mathsf{T}$ *and* $\mathbf{p}_2 = \begin{bmatrix} 1 & -1 \end{bmatrix}^\mathsf{T}$ *are global minima of the stripe problem's error-weight surface* $L(\mathbf{w})$.

*Proof.* The loss function $L(\mathbf{w})$ from Equation (5.4) is a sum of squared terms whose arguments are real-valued. This means that if we can show that $L(\mathbf{w}) = 0$ for some $\mathbf{w}$, then we have proved, by Definition 18 that $L$ has a global minimum at $\mathbf{w}$. Let us look at the value of the loss function at $\mathbf{p}_1$,

$$\begin{aligned} L(\mathbf{p}_1) &= \left(e^{-4(w_1+w_2)^2} - 1\right)^2 + \left(e^{-4w_2^2} - e^{-4}\right)^2 + \left(e^{-4w_1^2} - e^{-4}\right)^2 \\ &= \left(e^0 - 1\right)^2 + \left(e^{-4} - e^{-4}\right)^2 + \left(e^{-4} - e^{-4}\right)^2 = 0. \end{aligned}$$

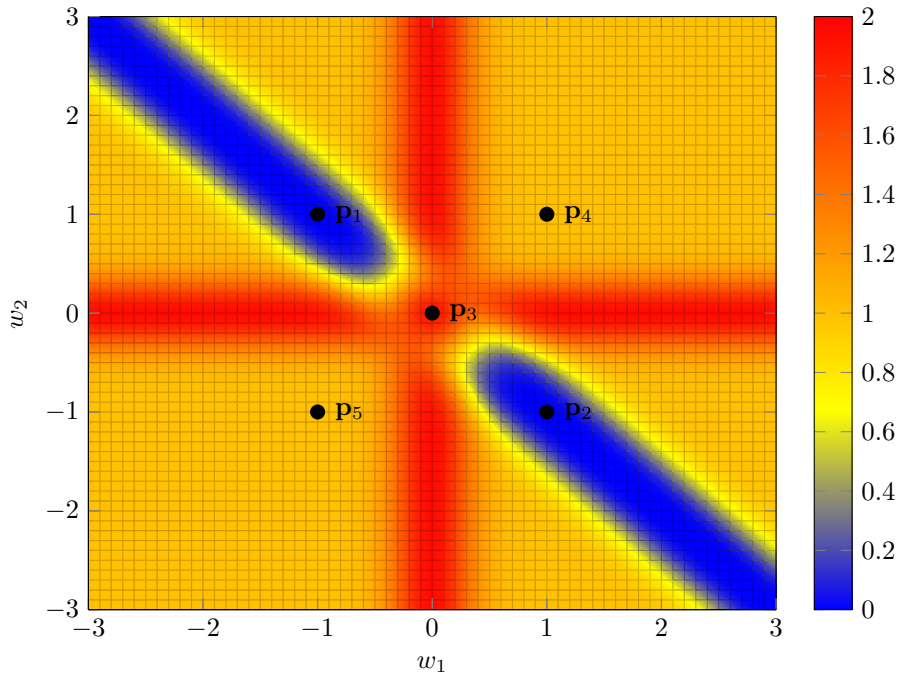



Figure 5.7: Heat map of the stripe problem's error-weight surface showing the critical points. It provides a birds-eye perspective of Figure 5.5.

Similarly, we obtain $L(\mathbf{p}_2) = 0$. It follows that both $\mathbf{p}_1$ and $\mathbf{p}_2$ are global minima. □

**Corollary 14.1.** *The points* $\mathbf{p}_4$ *and* $\mathbf{p}_5$ *are suboptimal local minima.*

*Proof.* We have already shown in Lemma 13 that $\mathbf{p}_4$ and $\mathbf{p}_5$ are local minima. Let us look at the loss values at $\mathbf{p}_4$.

$$L(\mathbf{p}_4) = \left(e^{-16\eta^2} - 1\right)^2 + \left(e^{-4\eta^2} - e^{-4}\right)^2 + \left(e^{-4\eta^2} - e^{-4}\right)^2$$

Since $\eta > 0$, we conclude that $L(\mathbf{p}_4) > 0$. In Theorem 14 we have shown that the minimum of $L(\mathbf{w})$ is zero. Thus, by Definition 18, it follows that $\mathbf{p}_4$ is *not* a global minimum, so it is a suboptimal local minimum. By extension, since $L(\mathbf{w}) = L(-\mathbf{w})$ and $\mathbf{p}_4 = -\mathbf{p}_5$, it must be the case that $\mathbf{p}_5$ represents a suboptimal local minimum as well. □

### 5.2.6 On the convergence of gradient descent

Theorem 14 and Corollary 14.1 allow us to characterise the nature of the error-weight space and provide a rigorous argument that the stripe problem indeed has suboptimal local minima. In fact, if $w_1$ and $w_2$ are either both positive or both negative, gradient descent will converge to a suboptimal local minimum, whereas if $w_1$ and $w_2$ have different signs, gradient descent training will converge to a global minimum. To see this, consider the vector field of the directions of the negative gradients depicted in Figure 5.8. When starting at a point with

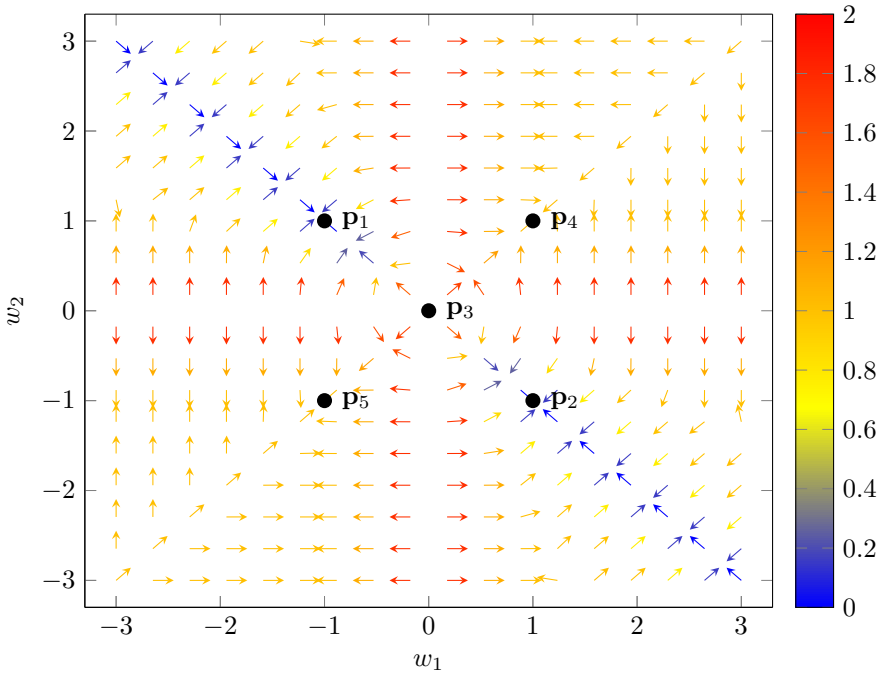
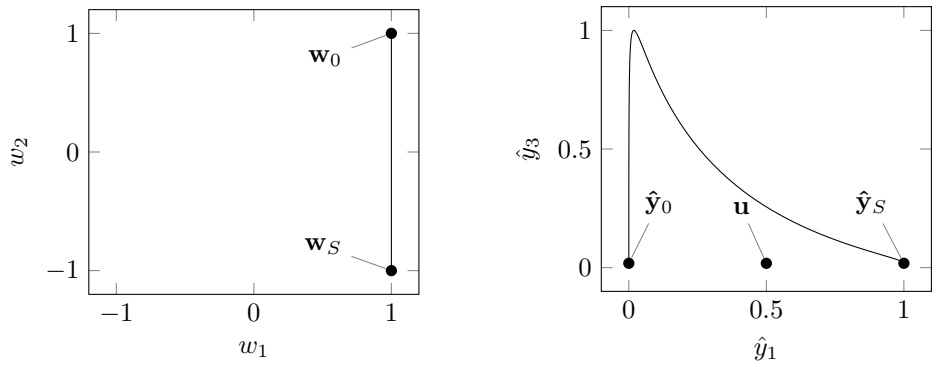



Figure 5.8: Vector field of the negative gradients of the error-weight function. The vectors' magnitudes are normalised and their colour represents the loss values at their respective origins.




(a) Weight space.

(b) Output space.

Figure 5.9: Ideal goal-connecting path in weight and ouput spaces. It suffices to show the first and third samples of the output space as the second stays constant. Note also that $\hat{\mathbf{y}}_S = \mathbf{y}$ because the target is in fact realisable. The point labelled $\mathbf{u}$ is the example of an unrealisable point referred to in Section 5.2.7.

$w_1 > 0$, $w_2 > 0$ and following the arrows, one will end up at the suboptimal local minimum $\mathbf{p}_4$ eventually. On the other hand, when starting at a point $w_1 < 0$, $w_2 > 0$, it is easy to see that travel will converge to $\mathbf{p}_1$. The same logic can be applied to the other two points due to symmetry. Note that would be possible to formally prove this observation by considering the directions of the Jacobian in different regions of the plot. However, this proof would be very long and exceed the scope of this report. Furthermore, the experimental evidence in Section 5.3.1 verifies this claim empirically.

### 5.2.7 Ideal goal-connecting path

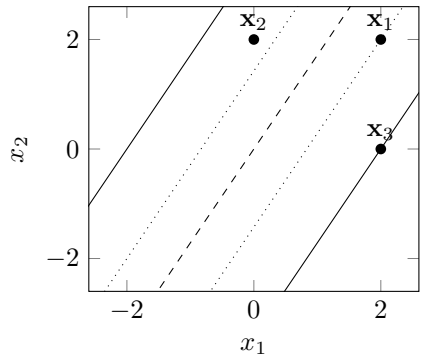
In Lemma 7 we showed that the ideal goal-connecting path is a straight line in weight space that starts at the initial configuration $\mathbf{w}_0$ and ends at the goal $\mathbf{w}_S$. We established that the initial weight configuration is $\mathbf{w}_0 = \begin{bmatrix}1 & 1\end{bmatrix}^\mathsf{T}$ which is near a suboptimal local minimum (see Corollary 14.1) and the target is $\mathbf{w}_S \begin{bmatrix}1 & -1\end{bmatrix}^\mathsf{T}$ which we proved in Theorem 14 is a global minimum.

Figure 5.9 depicts the ideal goal line in weight space as well as how this line corresponds to output space. It is interesting to see the extreme deformation of this path in output space. In fact, a straight line from $\hat{\mathbf{y}}_0$ to $\hat{\mathbf{y}}_S$ would *not* be a valid goal-connecting path because it would pass through an unrealisable region. To see this, consider for example the point $\mathbf{u}$ lying midway on the straight line between the initial and target configurations,

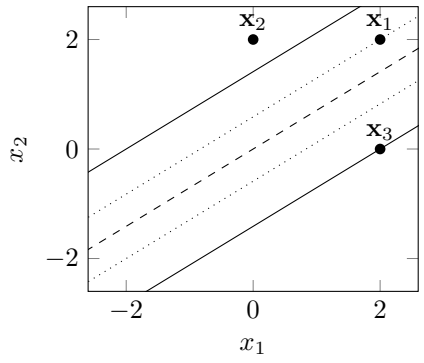
$$\mathbf{u} = \frac{\mathbf{y} - \hat{\mathbf{y}}_0}{2} = \begin{bmatrix}\frac{1-\phi(4)}{2} & \phi(2) & \phi(2)\end{bmatrix}^\mathsf{T}.$$

**Lemma 15.** *The point* $\mathbf{u}$ *is strongly unrealisable.*

*Proof.* We need to find a weight configuration $\mathbf{w} = \begin{bmatrix}w_1 & w_2\end{bmatrix}^\mathsf{T}$ that satisfies $\phi\left(\mathbf{w}^\mathsf{T}\mathbf{x}_i\right) = u_i$ for $i = 1, 2, 3$ where $u_i$ is the $i$th component of $\mathbf{u}$. The first input




(a) Input space for $w_1 = 1$ and $w_2 = \frac{1}{2}\sqrt{-\ln\frac{1-e^{-16}}{2}} - 1$.




(b) Input space for $w_1 = 1$ and $w_2 = -\frac{1}{2}\sqrt{-\ln\frac{1-e^{-16}}{2}} - 1$.

Figure 5.10: Input space with hyperplanes at zero excitation (dashed), $\frac{1-e^{-16}}{2}$ activation (dotted) and $\phi(2)$ activation (solid) for two possible input configurations. It can be seen that although $\mathbf{x}_1$ always intersects with the dotted line, only one of the two other input points will intersect with the solid line.

$\mathbf{x}_1$ is given by $\mathbf{x}_1 = \begin{bmatrix}2 & 2\end{bmatrix}^\mathsf{T}$ and so we obtain the following equation:

$$\begin{aligned}\phi(\mathbf{w}^\mathsf{T}\mathbf{x}_1) &= \frac{1-\phi(4)}{2}\\ e^{-4(w_1+w_2)^2} &= \frac{1-e^{-16}}{2}\\ w_1 + w_2 &= \pm\frac{1}{2}\sqrt{-\ln\frac{1-e^{-16}}{2}}.\end{aligned} \tag{5.8}$$

Similarly, for the second input $\mathbf{x}_2 = \begin{bmatrix}0 & 2\end{bmatrix}^\mathsf{T}$ we obtain

$$\begin{aligned}\phi(\mathbf{w}^\mathsf{T}\mathbf{x}_2) &= \phi(2)\\ e^{-4w_2^2} &= e^{-4}\\ w_2 &= \pm 1,\end{aligned} \tag{5.9}$$

and finally for $\mathbf{x}_3 = \begin{bmatrix}2 & 0\end{bmatrix}^\mathsf{T}$ we get

$$\begin{aligned}\phi(\mathbf{w}^\mathsf{T}\mathbf{x}_3) &= \phi(2)\\ e^{-4w_1^2} &= e^{-4}\\ w_1 &= \pm 1.\end{aligned} \tag{5.10}$$

There exist no solutions for $w_1$ and $w_2$ that satisfy Equations (5.8) to (5.10). Thus, by Definition 14, we conclude that $\mathbf{u}$ is a strongly unrealisable point. □

**Remark.** Lemma 15 proves the existence of one specific unrealisable point, but this methodology could be repeated for other points in order to prove that there is actually a larger unrealisable region consisting of more than that one point on the goal line. Although this will not be carried in this report, it is an important

aspect of the stripe problem to keep in mind. The intuition in this scenario with three outputs is that it will often be possible to satisfy two of the three output targets, but not the third. Figure 5.10 illustrates this point by considering the hyperplanes of the target output values for the point **u** discussed above.

## 5.3 Experimental results and analysis

A variety of experiments have been carried out on the stripe problem using the different optimisation techniques from Chapter 3 with and without subgoals. This section will summarise the findings from some of these experiments.

### 5.3.1 Gradient descent

In order to test the theoretical findings from Section 5.2.6, gradient descent was used to train on the stripe problem from a dozen different starting configurations. When considering $w_1$ and $w_2$ as the $x$ and $y$ axes of the Cartesian plane, a total of three initial weight configurations per quadrant were tested.

| trial | initial weights $\mathbf{w}_0$ | learning rate $\alpha$ | momentum $\beta$ |
|---|---|---|---|
| 1 | $\begin{bmatrix}1.8 & 1.5\end{bmatrix}$ | 10 | 0.9 |
| 2 | $\begin{bmatrix}1.5 & 1.8\end{bmatrix}$ | 10 | 0.9 |
| 3 | $\begin{bmatrix}2.0 & 2.0\end{bmatrix}$ | 100 | 0.7 |
| 4 | $\begin{bmatrix}1.8 & -1.5\end{bmatrix}$ | 1 | 0.9 |
| 5 | $\begin{bmatrix}1.5 & -1.8\end{bmatrix}$ | 1 | 0.9 |
| 6 | $\begin{bmatrix}2.0 & -2.0\end{bmatrix}$ | 100 | 0.7 |
| 7 | $\begin{bmatrix}-1.8 & -1.5\end{bmatrix}$ | 10 | 0.9 |
| 8 | $\begin{bmatrix}-1.5 & -1.8\end{bmatrix}$ | 10 | 0.9 |
| 9 | $\begin{bmatrix}-2.0 & -2.0\end{bmatrix}$ | 100 | 0.7 |
| 10 | $\begin{bmatrix}-1.8 & 1.5\end{bmatrix}$ | 1 | 0.9 |
| 11 | $\begin{bmatrix}-1.5 & 1.8\end{bmatrix}$ | 1 | 0.9 |
| 12 | $\begin{bmatrix}-2.0 & 2.0\end{bmatrix}$ | 100 | 0.7 |

Table 5.2: The dataset for the RBF stripe problem.

Table 5.2 shows the initial configurations along with the learning rate $\alpha$ and momentum[20] $\beta$ of the BP via gradient descent algorithm. For each trial, training was carried out for 10000 epochs. The hyperparameters had to be adjusted depending on the initial weight configuration in order to ensure convergence within the epoch limit. In Figure 5.11, the trajectories of each of the trials in weight space is shown. It is evident that each trial converges to the local minimum point in the quadrant of its initial weight configuration. This means

[20] As explained in Section 3.1, the learning rate $\alpha$ controls the step size. However, due to the fact that training using only a specific step size took too long, it was decided to use BP with momentum. Essentially, instead of determining the weight update based solely on the magnitude of the gradient and value of $\alpha$, we additionally take into account an exponentially weighted average of the past gradients where the hyperparameter $\beta \in [0, 1]$ determines the degree of that weighting. Introducing the momentum term is common practice in this type of scenario, so it will not be explained in further detail. It should not adversely affect the performance of gradient descent, so it is fair to use this technique to aid the speed of the experiments.

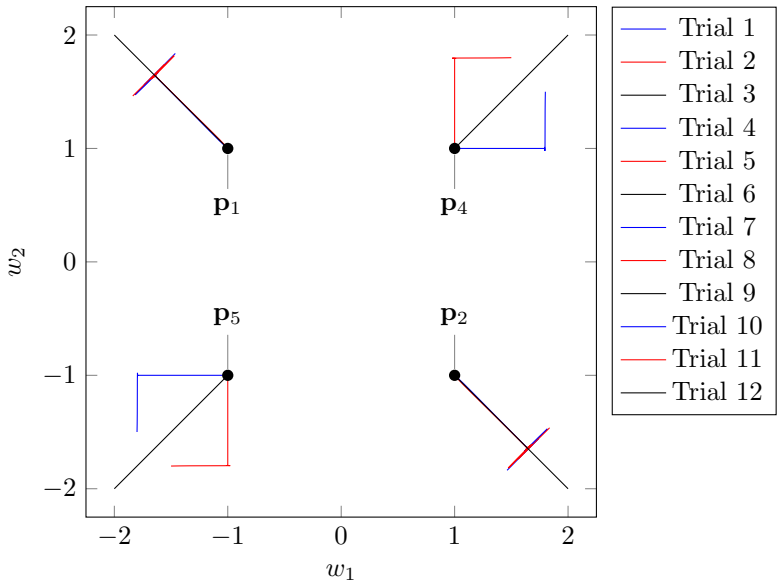


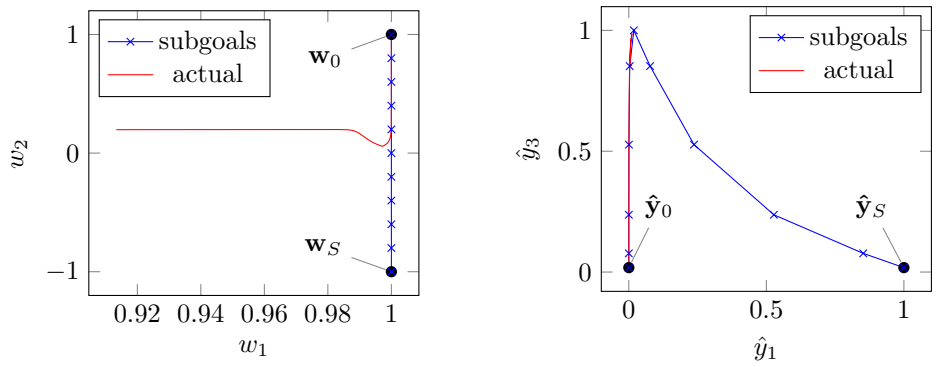

Figure 5.11: Some gradient descent trajectories in weight space. Trials in the same quadrant have different colours to differentiate between them. Trials 1-3 are in the first quadrant (Q1), 4-6 in Q2, 7-9 in Q3 and 10-12 in Q3.

that trials 1-3 and 7-9 converge to a suboptimal local minimum (Q1 and Q3), whereas trials 4-6 and 10-12 converge to a global minimum (Q2 and Q4), thus empirically affirming the claim made in Section 5.2.6.

Even with a relatively large learning rate and despite the use of momentum, the fact that training requires around 10000 epochs to converge highlights one of the core issues of gradient descent. As discussed in Section 3.4, specifically Figure 3.2, the classical BP algorithm becomes quite slow in "long, narrow valleys" [Press et al. 1992, p. 421] due to the gradient alternating in near right-angle directions. Although the motivation of the stripe problem is to highlight another issue (that of suboptimal local minima), it is noteworthy that this example is quite potent in highlighting other issues as well. The error-weight surface in Figure 5.5 shows examples of this phenomenon quite convincingly because the area marked in blue is a long, narrow valley.

### 5.3.2 Derivative-free techniques

Other techniques, namely greedy probing and simulated annealing have been evaluated in the context of the stripe problem, too. They parallel the findings of Section 5.3.1 insofar as initial weight configurations in Q1 and Q3 converging to suboptimal local minima. In fact, the trajectories in weight space are quite similar to Figure 5.11 with the notable difference that training using these derivative-free techniques found the local or global minimum (depending on the quadrant of initialisation) requiring only around 100 epochs where BP required 100000. These findings can be reproduced using the framework (see Chapter 8).




(a) Weight space.

(b) Output space.

Figure 5.12: Gradient descent with $S = 10$ subgoals on the ideal goal-connecting path.

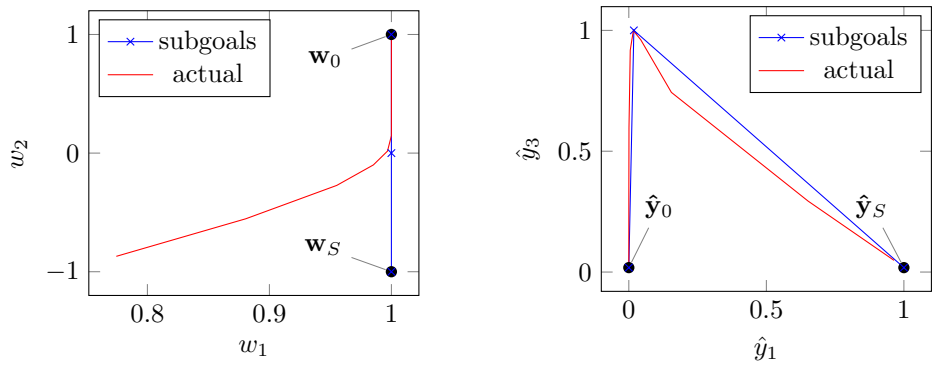
### 5.3.3 Gradient descent with subgoals

After empirically verifying in Section 5.3.1 that the classical BP algorithm fails to converge to a global minium with a weight initialisation in Q1 or Q3, we will now employ the 'cheating' technique introduced in Section 4.4 using gradient descent as a means to test if neural training techniques in general can realise the ideal goal line (or some trajectory close to it).

Recall the inputs and target outputs of the stripe problem presented in Table 5.1. Figure 5.12 shows the ideal goal line from the initial configuration $\mathbf{w}_0 = \begin{bmatrix} 1 & 1 \end{bmatrix}^\mathsf{T}$ to the target $\mathbf{w}_S = \begin{bmatrix} 1 & -1 \end{bmatrix}^\mathsf{T}$ along with the $S = 10$ equidistant subgoals on that line. The red line represents the actual weight and output trajectories achieved by the 'cheat' technique in conjunction with gradient descent for the parameter[21] $\mu = 0.9$. It can be seen in Figure 5.12b that although the fifth subgoal was achieved, the approach failed to realise the sixth subgoal, and hence the remaining path to the goal. The training regime achieves the first four subgoals in weight space (see Figure 5.12a) perfectly, but just shy of achieving the fifth subgoal, $w_2$ is altered such that the subgoal can never be reached.

Notice that the goal line in weight space keeps the weight $w_1$ constant, whereas the actual trajectory violates this principle. Upon achieving the third subgoal, the technique fails to simply continue to decreasing the value of $w_2$ such that it becomes negative. This is due to the nature of the radial basis activation function. As shown in Figure 5.2, the RBF outputs its maximum value when the input is zero, and its output decreases as the absolute value of the input increases. The gradient descent approach is almost at the maximum of the RBF when en route to achieving the fourth subgoal, but it does not know that the function would decrease again if it continued further. Essentially, there is a "blind spot" [Karayiannis 1998] because the algorithm cannot 'see' the other side of the hump. Furthermore, the gradient of the RBF is flat when the input is zero ($\phi'(0) = 0$) which is another reason why gradient descent might have

[21] Recall from Section 4.4 that $\mu$ is the minimum fractional progress that needs to be attained towards the next subgoal before that subgoal is considered 'realised'. The value of $\mu = 0.9$ was chosen after experiments with different values showed that when $\mu$ is too low, the path deformation in output space from the ideal goal line can become very extreme.




(a) Weight space.

(b) Output space.

Figure 5.13: Gradient descent ($\alpha = 0.1$, $\beta = 0$) with $S = 2$ subgoals on the ideal goal-connecting path.

difficulties mounting the top of the hump.

It was conjectured that in order to mitigate the problem of the "blind spot", it might be necessary to increase the step size at the region where the gradient is near-zero, as this would increase the likelihood of the technique 'jumping' over the hump. One method of ensuring a greater step size in that region is introducing momentum. Experiments were conducted with different numbers of subgoals, and different learning rates. The most important finding is that the target could be achieved using only a minimal number of subgoals ($S = 2$, so using only one subgoal and the actual goal) by keeping the learning rate $\alpha = 0.1$ (like in Figure 5.12) but using momentum with $\beta = 0.9$. The relevant trajectories are depicted in Figure 5.13. While the subgoal is achieved reasonably well in the two dimensions of output space portrayed in Figure 5.13b, the final prediction of $\hat{y}_2$ changes by around 0.07 albeit it should stay constant. This is because the step size due to momentum increased too much after overcoming the 'hump'. One mechanism to mitigate this problem would be to increase a learning rate schedule; since the final weight configuration is already in the right quadrant, gradient descent is guaranteed to converge to the target (as discussed theoretically in Section 5.2.6 and confirmed empirically in Section 5.3.1), given an adequate learning rate.

### 5.3.4 Derivative-free techniques with subgoals

The greedy probing approach was also tested with $S = 2$ subgoals using a sampling radius of $r = 0.1$. To ensure that the trajectory in output space gets close enough to the first subgoal such that it passes the 'hump', the fractional progress parameter $\mu$ had to be increased to 0.99.

One of the drawbacks of the greedy probing technique is that per epoch, it needs to compute the forward pass of the network for every weight sample and every output sample. Hence if the sampling technique generates $s$ samples per epoch and the dataset contains $N$ records, a total of $sN$ forward passes must be computed[22]. In the case of the exhaustive sampling technique, Sec-

[22]In practice, since the weights are the same for each sample, the forward passes are batched

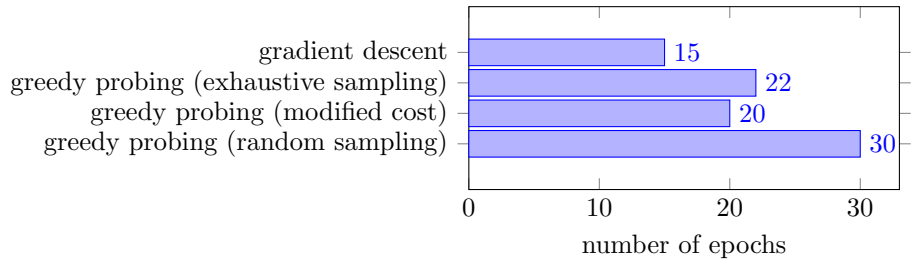
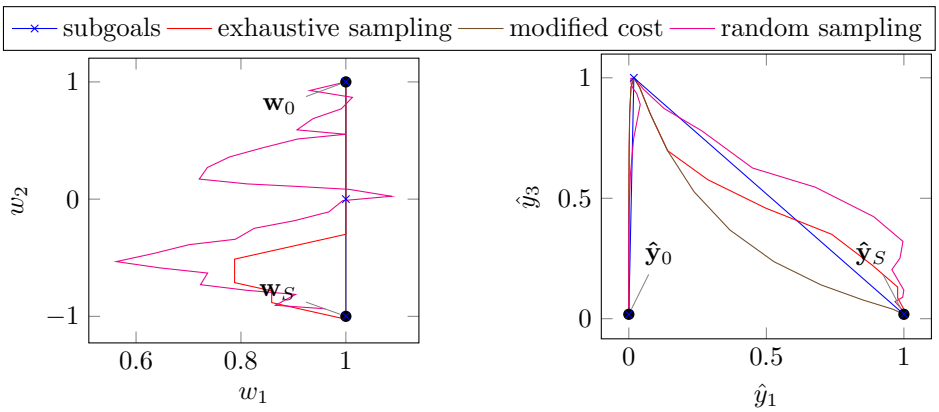



Figure 5.14: Number of epochs until achieving the final subgoal by training regime.

tion 3.2 explains that $s = 3^P - 1$ samples are generated for a $P$-dimensional weight space. So, for the stripe problem, the exhaustive sampling technique will generate 8 weight samples which means that a total of 24 forward passes must be carried out per epoch. Note that gradient descent must only perform one derivative calculation which is comparable in computational complexity to a forward pass (assuming the loss value itself is not calculated, only its derivative). The advantage of the random sampling technique is twofold: the user may specify the number of samples to calculate, and the directions are not as constrained in weight space. We chose a value of $s = 6$ for the random sampling technique which means that it only requires 18 forward passes per epoch. However, in Figure 5.15 it becomes apparent that there is a drawback: the random sampling technique needs more epochs to achieve the final goal. In this regard, gradient descent actually outperformed all of the tested derivative-free schemes because it requires fewer epochs and fewer forward passes per epoch. However, it must be noted that finding the optimal hyperparameters that actually allowed gradient descent to converge with only one subgoal was not easy and required a lot of trial and error, so these results are highly customised towards the stripe problem and will likely perform poorly in other settings.

An alternative cost function was also tested for the greedy probing approach, calculating the cost as the sum of the trajectory's strain in both weight space and output space in order to penalise large turns in both spaces. Figure 5.15 provides a comparison of the trajectories of all three approaches. It can be seen that the modified cost function exactly produces the ideal goal line in weight space; however, this is only because the initial direction of the trajectory in weight space was correct. If this were not the case, this modified cost function would fail, so in that regard it is not fair to claim that the modified cost function is superior. Another observation is that due to the randomness of the random sampling technique, its trajectory is chaotic in both the weight and output spaces. For the exhaustive sampling technique, it is interesting to see that like gradient descent (Figure 5.13), the trajectory in weight space begins to deviate from the goal line in weight space after achieving the subgoal, but unlike gradient

for each sample which means that $s$ forward passes are computed, each with a batch dimension of $N$. This is possible because the neural layers in essence just represent a matrix multiplication (see Equation (2.12)), but keep in mind that this can only achieve a speedup if the matrix multiplication is parallelised (on a GPU for example), and even then, this argument only applies to large values of $N$, whereas for the stripe problem $N = 3$. Therefore, let us just say for argument's sake that $sN$ forward passes must be computed.




(a) Weight space.

(b) Output space.

Figure 5.15: Greedy probing ($r = 0.1$) with $S = 2$ subgoals on the ideal goal-connecting path.

descent, it is actually able to find back to the goal line. A possible explanation for this could be the increased value of $\mu$ used for greedy probing.

# Chapter 6

# The neural surfing technique

## 6.1 Motivation

## 6.2 Clothoids

Consider a scenario in two-dimensional output space, as is shown in Figure 6.1. The current weight configuration produces a point $C$ in output space. How do we evaluate the effectiveness of a candidate sample point $B$ in achieving the goal (or subgoal), $A$? We would like to analyse the properties of a curve that originates at $C$, passes through $B$, and ends at $A$. This curve should undergo a linear decrease in curvature until reaching a curvature of zero at the goal. We call this type of curve *clothoid*, and there can only be one clothoid that passes through all three points and has a curvature of zero at $A$. In fact, this clothoid will be a segment of the *Euler spiral* (up to some affine transformation).

For the purposes of the neural surfer, we are interested in two specific properties of that clothoid: the angle that the goal line makes with the tangent of the clothoid at $C$, and that angle at $A$. In Figure 6.4, these angles are marked $\alpha$ and $\beta$, respectively. This leads to the following problem which we will discuss

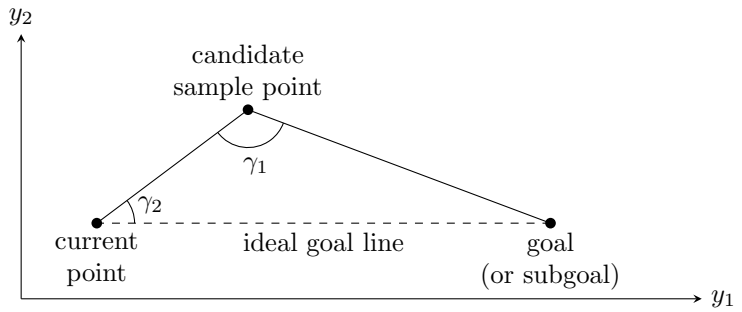



Figure 6.1: A secenario in two-dimensional output space. Note that the ideal goal line need not be parallel to the $y_1$ axis.

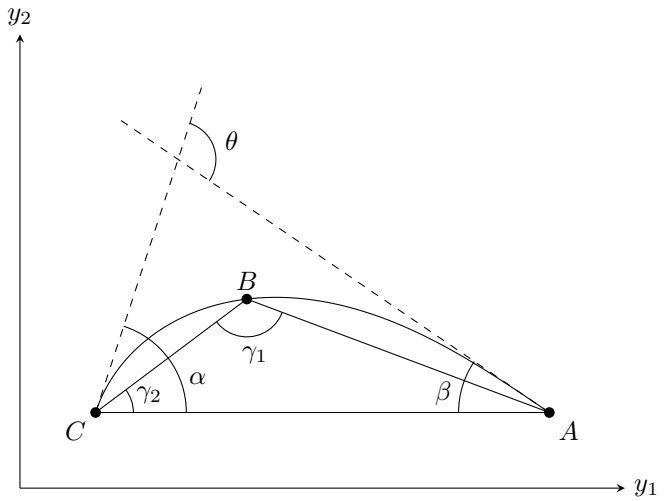



Figure 6.2: The scenario from Figure 6.1 with the goal-connecting clothoid.

in this section:

**Problem 2** (Clothoid construction)**.** Given the angles $\gamma_1$ and $\gamma_2$, determine the angles $\alpha$ and $\beta$ of the resulting clothoid, i.e. find a function that evaluates the mapping $\langle \gamma_1, \gamma_2 \rangle \to \langle \alpha, \beta \rangle$.

### 6.2.1 Euler spiral

The Euler spiral is a special curve with an interesting property: its curvature increases linearly with the curve length from the origin. As a consequence, the Euler spiral never forms a complete loop (or circle), a trait that will become useful for the neural surfing technique. Figure 6.3 depicts the normalised Euler spiral. It can be defined using the Fresnel sine and cosine integrals

$$F_S(t) = \int_0^t \sin \frac{\pi s^2}{2}\, ds \tag{6.1}$$

and

$$F_C(t) = \int_0^t \cos \frac{\pi s^2}{2}\, ds \tag{6.2}$$

with the parametric equations $x = F_C(t)$ and $y = F_S(t)$. The derivative at a specific point $(x, y)$ in terms of $t$ is calculated as

$$\frac{dy}{dx} = \frac{\left(\frac{dy}{dt}\right)}{\left(\frac{dx}{dt}\right)} = \frac{\sin \frac{\pi t^2}{2}}{\cos \frac{\pi t^2}{2}} = \tan \frac{\pi t^2}{2},$$

so the angle $\omega$ of the clothoid's tangent at $(F_C(t), F_S(t))$ can be expressed in terms of $t$ as

$$\omega(t) = \tan^{-1}\left(\tan \frac{\pi t^2}{2}\right) = \frac{\pi t^2}{2}. \tag{6.3}$$

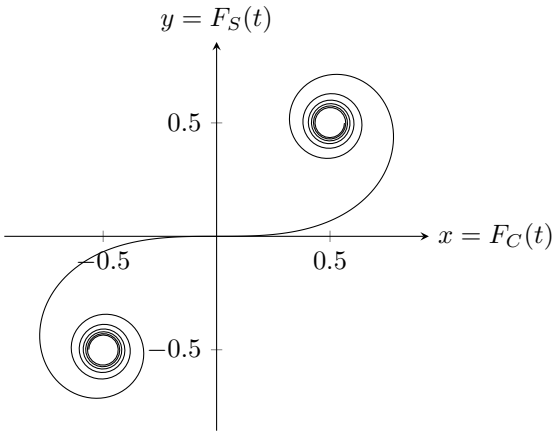



Figure 6.3: Plot of the Euler spiral.

### 6.2.2 Construction

Figure 6.4 shows how the three points $A, B, C$ from the scenario in Figure 6.1 can be mapped to the Euler spiral. Let us use the notation $\mathcal{A}, \mathcal{B}, \mathcal{C}$ to denote the corresponding points in two-dimensional Cartesian coordinate plane of the Euler spiral. $\mathcal{A}$ must be at the origin of the spiral in order to ensure that the curvature is zero at the goal point. Then the two points $\mathcal{B}$ and $\mathcal{C}$ must be found along the curve such that the triangle formed by these points, $\triangle\mathcal{ABC}$, has angles $\measuredangle\mathcal{ABC} = \gamma_1$ and $\measuredangle\mathcal{ACB} = \gamma_2$.

The problem of finding $\triangle\mathcal{ABC}$ is equivalent to finding the values $t_\mathcal{A}, t_\mathcal{B}, t_\mathcal{C}$ that produce the points $\mathcal{A}, \mathcal{B}, \mathcal{C}$ when plugged into Equations (6.1) and (6.2). We have established previously that $\mathcal{A} = (0, 0)$; thus $t_\mathcal{A} = 0$ but it remains to find $t_\mathcal{B}$ and $t_\mathcal{C}$.

Let us constrain $t_\mathcal{A}$ and $t_\mathcal{B}$ to be in the interval $(0, \sqrt{3}]$ to ensure that the resulting clothoid will not wind too tight[23]. Furthermore, it must be the case that $t_\mathcal{A} < t_\mathcal{B} < t_\mathcal{C}$ which is also why the interval above does not include zero. The steps below can be used to find $t_\mathcal{B}$ and $t_\mathcal{C}$.

1. Create a sequence of $n$ sample values of $t$ such that $t_i = \frac{i}{n}\sqrt{3}$ for $i = 1, 2, \ldots, n$.

2. Calculate the points $P_1, P_2, \ldots, P_n$ along the Euler spiral where the $i$th point is given by $P_i = (F_C(t_i), F_S(t_i))$.

3. For $b, c \in \mathbb{Z}^+$ subject to $b < c \leq n$, find the pair of values $b, c$ that minimises
$$(\measuredangle AP_bP_c - \gamma_1)^2 + (\measuredangle AP_cP_b - \gamma_2)^2 . \tag{6.4}$$
In other words, find the triangle formed by the points $P_b$, $P_c$, and $\mathcal{A}$ which is the most similar[24] to $\triangle ABC$. When $n \to \infty$, we find $t_b = t_\mathcal{B}$ and $t_c = t_\mathcal{C}$

[23]In fact, the clothoid from Figure 6.4 was plotted for $0 \leq t \leq \sqrt{3}$. At $t = \sqrt{3}$, the clothoid 'points' downwards.

[24]Two triangles are *similar* if their side lengths are proportional and hence angles are identical.

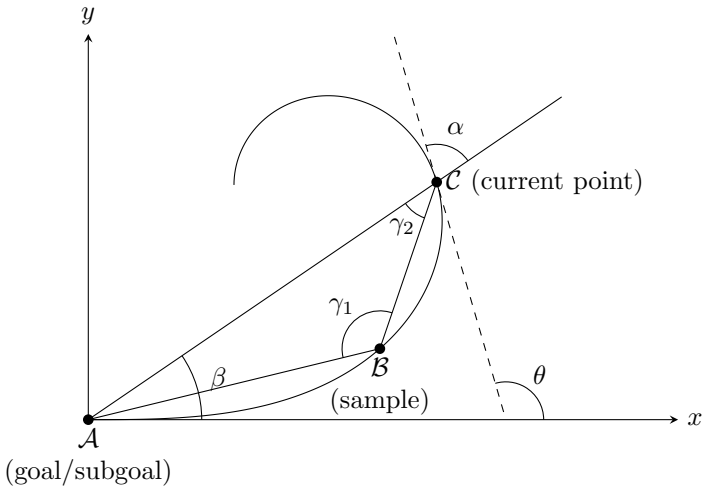



Figure 6.4: Plot of a clothoid segment with three points of interest: $\mathcal{C}$ is the initial point, $\mathcal{B}$ is the sample point and $\mathcal{A}$ is the goal/subgoal (at the origin of the clothoid). The dashed line is tangential to the clothoid at $\mathcal{C}$.

(equivalently, $P_b = \mathcal{B}$ and $P_c = \mathcal{C}$), but values of $n$ in the order of 1000 will be suitable for practical purposes.

Once we have determined $t_\mathcal{B}$ and $t_\mathcal{C}$ and thus know the coordinates of $\mathcal{B}$ and $\mathcal{C}$, we can find $\beta$ as the angle that $\mathcal{C}$ makes with the $x$-axis, giving

$$\beta = \tan^{-1}\left(\frac{\mathcal{C}_y}{\mathcal{C}_x}\right). \tag{6.5}$$

Since $\theta$ is the angle of the tangent at $\mathcal{C}$, we obtain from Equation (6.3) that $\theta = \frac{\pi}{2}\left(t_\mathcal{C}\right)^2$. Furthermore, we can deduce from Figure 6.4 that $\theta = \alpha + \beta$, so

$$\alpha = \theta - \beta = \frac{\pi}{2}\left(t_\mathcal{C}\right)^2 - \beta. \tag{6.6}$$

This completes the construction of the clothoid; we have found a naïve algorithm for computing Problem 2.

### 6.2.3 Lookup table

A disadvantage of the algorithm from Section 6.2.2 is that it must compute $n$ points on the Euler spiral and then perform the angle calculations for $\mathcal{O}(n^2)$ triangles, *for each sample point* and this is carried out *at every training step.* Instead, we could sample $n$ points along the Euler spiral for values of $t$ in the interval $(0, \sqrt{3}]$, use these to construct the triangles as explained in step 2 of the algorithm, compute the angles $\gamma_1, \gamma_2, \alpha, \beta$ for each triangle and record these values in a table. This table will only need to be constructed once, thus the expensive calculations are not repeated. Treating $\langle \gamma_1, \gamma_2 \rangle$ as the (multi-dimensional) index to the table, our table will represent exactly the mapping $\langle \gamma_1, \gamma_2 \rangle \to \langle \alpha, \beta \rangle$ from Problem 2.

However, this table will only have a total of $(n-1)(n-2)$ entries which means that the mapping would only be defined for certain values of $\langle \gamma_1, \gamma_2 \rangle$. When querying using some pair $\langle \overline{\gamma}_1, \overline{\gamma}_2 \rangle$ that is not in the table, we will find the pair $\langle \gamma_1, \gamma_2 \rangle$ in the table which is closest in terms of Euclidean distance (its *nearest neighbour*) and return that entry's $\langle \alpha, \beta \rangle$. In fact, this is equivalent to minimising Equation (6.4) when setting $\overline{\gamma_1} = \measuredangle AP_bP_c$ and $\overline{\gamma_2} = \measuredangle AP_cP_b$.

The nearest neighbour can be found quite efficiently using a so-called $k$-d tree data structure with an average time complexity logarithmic to the number of points [Friedman et al. 1977]. Hence, using this technique, we can find the clothoid parameters for Problem 2 in $\mathcal{O}(\log n)$ time[25] on average.

## 6.3 Adaptive clothoid technique

**TODO**

[25] This is because we have $n^2$ points and $\mathcal{O}(\log n^2) = \mathcal{O}(\log n)$.

# Chapter 7

# Generalising neural surfing

In this chapter we will consider how any neural surfing technique can be generalised to different neural network architectures and problems. For this purpose, we will define a 'neural surfer' to be an algorithm that samples states in weight space $\mathcal{W}$ that are in the local neighbourhood, producing predictions in output space $\mathcal{O}$ and deciding which weight state to transition to based on the associated predictions. The motivation for this analysis is to demonstrate the potential for research in neural surfing algorithms: if a good neural surfer for the stripe problem is devised, it will have applications beyond that simple problem, and potentially even neural networks.

## 7.1 Multiple layers

Up until now, in the context of the stripe problem, we have looked at a single-layer network with one output unit. However, what happens when we increase the number of layers? Recall that the definitions of weight and output space in Section 4.1 are completely decoupled. In fact, we proved in Theorem 5 that there exists no linear mapping between these spaces for SLNs with one output unit which in Corollary 5.2 we generalised to arbitrary MLPs. In a sense, the hidden layers are thus hidden behind an abstraction: the neural training technique itself has no knowledge whether or not the underlying network had hidden layers; indeed it not aware of the neural network's architecture at all because its only information is the generated prediction $\hat{\mathbf{y}}$ in output space $\mathcal{O}$ associated with each sample $\mathbf{w}$ in weight space $\mathcal{W}$. Both $\mathcal{W}$ and $\mathcal{O}$ can be of arbitrary dimensionality, and so the technique can be used for any MLP.

## 7.2 Multiple outputs

In Definition 6, we characterised MLPs as multi-layer feedforward networks with one output unit, meaning that the last layer has only one neuron. We have shown this to be suitable and sufficient for regression tasks as well as binary classification problems. Yet in practice, we can also have multi-layer networks with multiple outputs. For example, multi-class classification tasks require $c$ output units for a one-hot-encoded problem with $c$ classes.

Let us now show that any neural surfing technique could also train this type of network. The nïve approach would be to perform a construction like in Figure 4.1 that was used to prove Corollary 5.1. However, especially for multi-class problems, it is often the case that the activation function on the last layer depends on other nodes in that layer. For example, the softmax activation function, defined for the $i$th output unit given that layer's excitation vector $\mathbf{z}$ as

$$\sigma_i(\mathbf{z}) = \frac{e^{z_i}}{\sum_j e^{z_j}}, \tag{7.1}$$

is commonly used on the final layer of a neural network for multi-class classification because the outputs can roughly be interpreted as probabilities.

Instead, we must follow a different approach. In Definition 4 we assumed that the neural network will output a scalar value because it only has one output node. We can generalise Equation (2.5) which represented the output vector (the scalar prediction for each input sample) to instead be the concatenation of all prediction vectors. This will not interfere with the notion of output space from Definition 12 and thus the methodology of the neural surfer can also be used for any type of feedforward neural network architecture.

Notice that now we can apply the same reasoning as in the previous section claiming that the weight and output spaces are decoupled to assert that theoretically the neural surfing approach could be generalised to networks with arbitrary inter-layer and intra-layer connections such as recurrent networks.

## 7.3 Potential field navigation

A common task in robotics is to smoothly navigate an obstacle course to reach a goal. It is convenient to consider this obstacle course in a two-dimensional space, but the approach can be generalised to arbitrary dimensionality. Assume that the robot knows the coordinates of the goal and that it has some number of distance sensors oriented at equidistant angles in the interval $[\frac{\pi}{2}, \frac{\pi}{2}]$ relative to the current heading.

To prevent the robot from crashing into obstacles and encourage smooth motion, a potential field is defined over the state space that should 'repel' the robot from obstacles but 'attract' it to the goal [Russell and Norvig 2010, pp. 999–1000]. The obtacle potential is a function that decays exponentially with the distance to the nearest obstacle (measured by the distance sensor) that could be of a form similar to

$$O(d_i) = \begin{cases} \frac{1}{d_i} e^{-\frac{1}{d_i^2 - S^2}} & 0 \leq d_i < S \\ 0 & d_i \geq S \end{cases} \tag{7.2}$$

where $d_i$ is the distance to the obstacle measured by the $i$th distance sensor and $S$ is the so-called fall-off range [Weir, Buck et al. 2006]. The goal potential could be of the form $G(d_g) = d^2$ for a distance $d_g$ to the goal. The combined potential is then given by

$$U = G(d_a) + \sum_i O(d_i). \tag{7.3}$$

The idea of neural surfing actually originated from its analogy to robot navigation using potential field techniques, so it could also be applied backwards

from neural surfing to robot path planning. The control space for robots is its wheels, and for neural networks that is the weights and biases (i.e. the weight space $\mathcal{W}$). The output space for robots is the movement achieved in the two-dimensional obstacle space, and for neural nets that is the activation of the output layer (i.e. the output space $\mathcal{O}$). Furthermore, the robot's sensors measure distances to obstacles which is comparable to the neural surfer probing samples in the local neighbourhood of its current configuration. Finally, the potential field from Equation (7.3) is like a neural network's loss function. Therefore, it is conjectured that a neural surfer could be converted to become a potential field agent. Notice that potential fields also possess a local minimum problem which however for the two-dimensional case can be overcome [Weir, Buck et al. 2006].

# Chapter 8

# The framework

## 8.1 Design

The requirements specification in Section 1.4 makes it clear that the framework must be made up of the following components:

1. an interface to specify a neural *problem* (as a labelled dataset with an initial weight configuration);
2. a mechanism to calculate problem-specific *metrics* during training;
3. an interface for implementing custom *visualisations* that plot metrics in real time;
4. an interface for implementing an *agent* that can be used to train on a problem; and
5. the notion of an *experiment* which coordinates the training and reporting (visualisation) aspects.

Figure 8.1 shows a high-level view of these components and their associations. The user should be able to provide their own implementations of problems, visualisations, agents, and metrics. All but the latter are provided in the framework as abstract Python classes, such that the user can implement their own functionality.

The design decision was made to provide the user with a different means of specifying metrics that is more user-friendly than having to inherit from an abstract base class. Due to the fact that metrics are defined on a per-problem basis and a metric is a function that can be evaluated on a problem in order to produce some data (tensor), it was decided to allow the user to specify these problems as *decorated functions* in their implementation of the abstract `Problem` class, as shown in Listing 1.

Finally, the `Experiment` class is not abstract because that is the class responsible for coordinating how all the user-defined components work together. The user would simply instantiate this class with a set of agents (each defined on a problem), and then call the `run_server()` method with a list of visualisations to display. Listing 2 provides a minimal example of this.

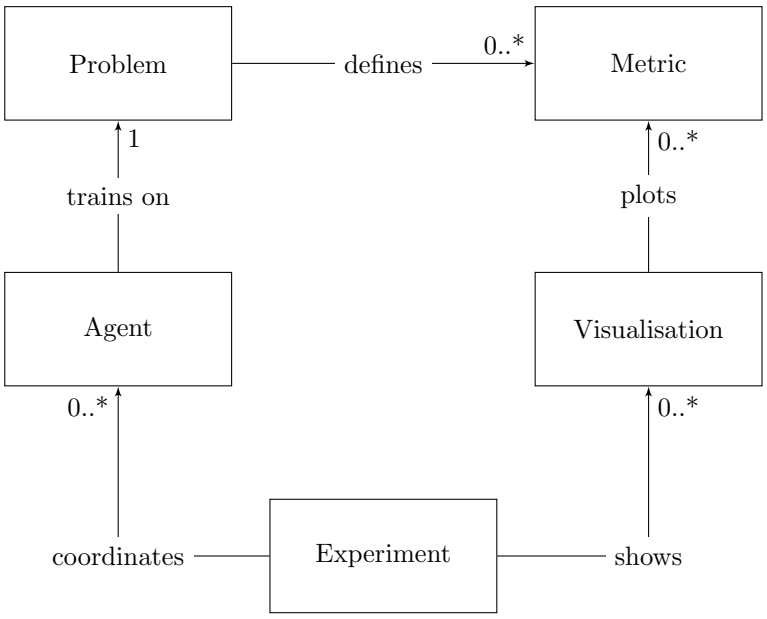



Figure 8.1: High-level view of the relationships between components of the `nf` framework.

```
class MyNeuralProblem(Problem):
    # ...

    @Problem.metric
    def weights(self):
        # Code to obtain the current weight state...
        return weight_state
```

Listing 1: Example of how a metric can be defined for a problem using a decorated function. Here, the metric will be called `weights`.

## 8.2 Implementation

The top-level package that constitutes this framework is named `nf` which is short for "neural framework". The framework code is provided in three main modules inside the `nf` package, which will be explained below. The main base classes are implemented in the respective `__init__.py` files so that they can be conveniently loaded using for example the syntax `from nf.agents import Agent` instead of the more verbose `from nf.agents.agent import Agent` which would arise when placing the `Agent` class inside a file named `agent.py`. This is common practice for these types of frameworks.

**The `problems` module** This module contains the abstract `Problem` class. Some problem implementations are provided in submodules, such as the `StripeProblem` class in the `stripe_problem` submodule.

```
agents = {
    "Agent A": AgentA(MyNeuralProblem()),
    "Agent B": AgentB(MyNeualProblem())
}

experiment = Experiment(agents)
experiment.run_server([
    Scatter2D("weights"),
    Scatter2D("output:1", "weights:0"),
    Histogram("output:0")
])
```

Listing 2: Minimum example of how an experiment can be specified and run. Here, `AgentA` and `AgentB` are agent implementations that the user has written, and `MyNeuralProblem` is defined like in Listing 1 but with an additional metric named `output`. The visualisations `Scatter2D` and `Histogram` are provided as part of the framework.

**The `agents` module** This module contains the abstract `Agent` class which requires the subclasses to override a `fit()` method of the same form as in the `keras` library. In fact, the agents must support the same lifecycle methods and callback hooks as `keras` models. To take care of some of this for the user `Agent` class is provided alongside two abstract subclasses: `DerivativeBasedAgent` and `DerivativeFreeAgent` that provide more support in implementing each of these techniques. A total of four agent implementations have been provided (two of each category). Figure 8.2 provides a UML diagram of this module, showing how the implementations are related via inheritance.

The `agents` module also contains a submodule, `sampling`, which defines the abstract `SamplingTechnique` class (this is shown in the diagram as well). It provides a common way that derivative-free agents can perform sampling in weight space. A total of three sampling techniques have been implemented: the `ExhaustiveSamplingTechnique` and `RandomSamplingTechnique` have been explained in Section 3.2. The third technique, `RandomSamplingGenerator` is a variant of the random sampling technique that does not require the number of samples to generate up front; instead, random samples are generated on demand using a so-called `generator` function in Python. This leads to efficiency benefits in the Simulated Annealing algorithm where it is not known a priori how many weight states need to be sampled before one is accepted.

**The `experiment` module** This module contains not only the `Experiment` class, but also two major aspects that are linked closely to the experiment itself: visualisations and metrics. Let us briefly look at their implementation before examining the `Experiment` class itself.

The visualisations are achieved using the `bokeh` library for Python which provides a web-based user interface. Originally, the classical plotting library `matplotlib` was used, but issues with regard to facilitating user interaction on plots that were updating in real time ultimately lead to the adoption of `bokeh`. The `visualisations` submodule contains an abstract `Visualisation`

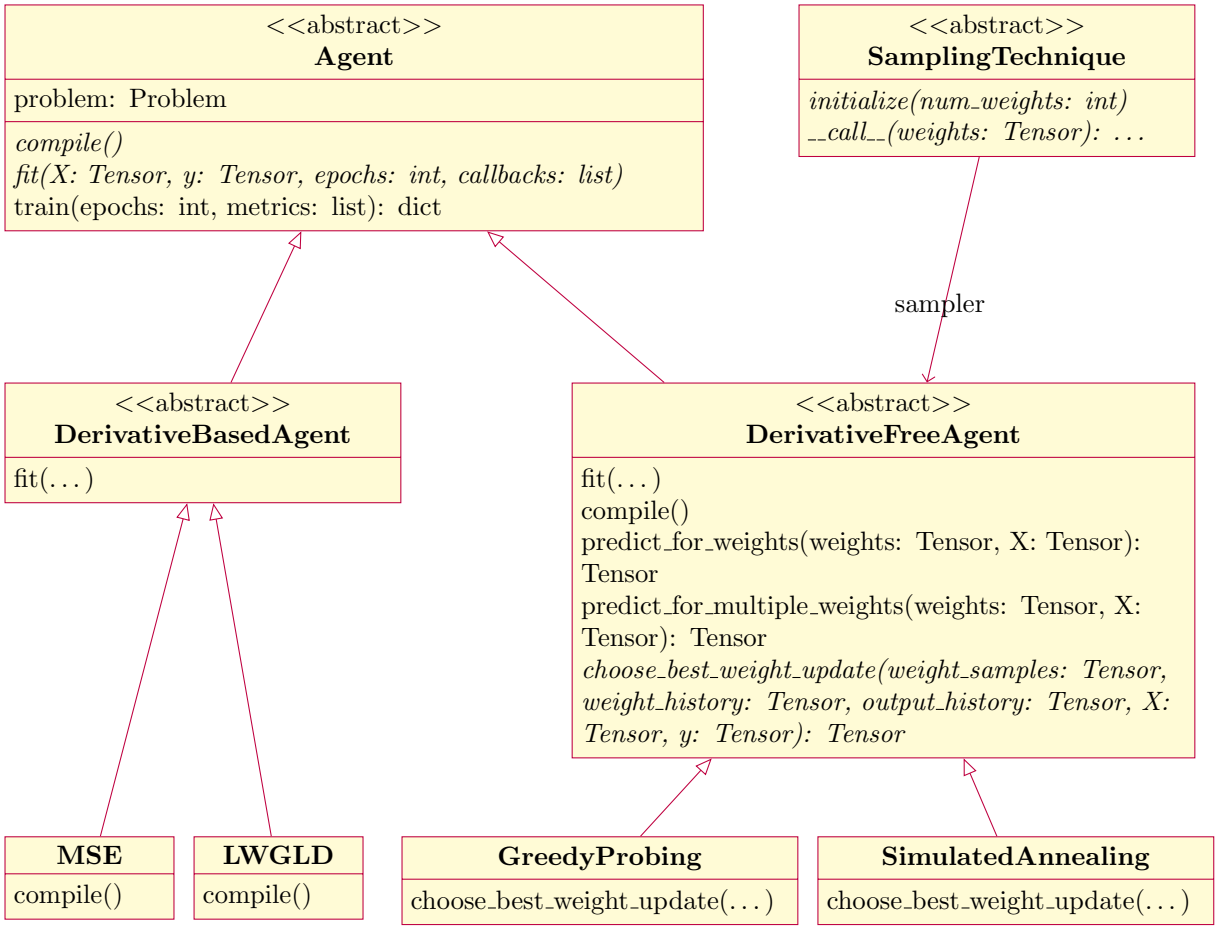



Figure 8.2: UML class diagram of the `agents` module, showing only the (non-underscored) public methods. Note that "LossWithGoalLineDeviation" is abbreviated as LWGLD and the subclasses of `SamplingTechnique` have been omitted.

class which provides a common interface for creating viualisations, so that the user may implement custom ones using `bokeh`. Two useful types of visualisations are provided with the framework: a two-dimensional scatter plot (`Scatter2D`) that can plot arbitrary metrics on each axis, and a histogram (`Histogram`) that will plot data over time (epochs). Each agent's data is plotted in a unique colour, and that colour is consistent across all visualisations. Furthermore, each visualisation supports toggling the visibility of each agent's data[26].

The `Metric` class is contained in the `metrics` submodule which is how metrics are internally represented in the framework. One interesting feature is the specification of metrics: Listing 2 actually showed three different ways that the user may specify metrics for a visualisation. The default syntax is of the form *<name>*:*<dim* 1*>*:*<dim* 2*>*:...:*<dim* $n$*>* where *<name>* is the metric's name and *<dim* $i$*>* is the index into the $i$th dimension of the rank-$n$ tensor representing the metric data. In the case of the `Scatter2D` visualisation which provides a two-dimensional scatter plot, one can also simply omit the last dimension index, in which case the metrics are unfolded among that dimension such that for example `Scatter2D("weights")` is equivalent to `Scatter2D("weights:0", "weights:1")`.

Most importantly, this module contains the eponymous class `Experiment` which is responsible for coordinating all the aforementioned components. When running the experiment, this class uses a form of round-robin scheduling whereby each agent is run for a specific number epochs at a time parameterised by `epoch_batch_size` before moving on to the next agent. Once each agent had their turn, the collected metrics are aggregated and sent to update the front end in real time.

There are two specific considerations that were made with regards to the collection and aggregation of metrics. Firstly, a mechanism was developed that determines the minimal set of metrics that are necessary for the visualisations, and computes only these (instead of all) in order to speed up training. Furthermore, the experiment itself collects some default metrics that can be visualised by the user. These are (i) `epoch` which collects the epoch number at each epoch; and (ii) `run_time`[27] which collects the average run_time per epoch[28]. Both contain a scalar value per epoch.

**Demonstrations** To supplement the framework, three demonstration scripts were written located in the parent directory of the framework. These showcase how the framework can be used to set up and run experiments and are explained in Appendix A. Furthermore, screenshots of the demonstrations in action are provided in the `framework/screenshots` folder of the code submission.

## 8.3 Documentation

The quality of a framework is quite obviously predicated in part on the quality of its documentation. All classes and methods in the framework are doc-

[26] On the front end, the user has a button to hide each of the agents separately. This is demonstrated in Appendix A.

[27] This metric is, in fact, used as the $x$-axis of the `Histogram` visualisation.

[28] Since training is carried out in batches of `epoch_batch_size`, the experiment does not know the run time on a per-epoch basis, so it will assign each epoch in the batch the run time of the entire batch divided by `epoch_batch_size`.

umented using not only *type annotations* in line with the PEP[29] 484 standard, but also *docstrings* according to the PEP 257 standard. This allowed the automatic generation of comprehensive documentation of the `nf` package using the `pdoc` tool. The documentation is available in HTML format at `framework/docs/nf/index.html` and PDF format at `framework/docs.pdf`.

Furthermore, a user manual is provided in Appendix A of this report and the demonstration scripts mentioned in the previous section can be used as starter code.

[29]PEP, short for Python Enhancement Proposal, is a collection of design documents containing best practices for developing Python code.

# Chapter 9

# Discussion

## 9.1 Evaluation

Evaluation against objectives

**Error and weight space**

**Stripe problem**

**Generalisation** Evaluate generalisation potential proved in generalisation section

## 9.2 Critical appraisal

**Gradient descent**

**Greedy probing** Also talk about sampling techniques

**Simulated annealing** More complex implementations of SA may combine the so-called downhill simplex algorithm [Nelder and Mead 1965] with SA such as in Press et al. [1992, p. 444-455], thereby introducing three additional hyperparameters. In fact, Press et al. remark that "there can be quite a lot of problem-dependent subtlety" in choosing the hyperparameters, and that "success or failure is quite often determined by the choice of annealing schedule" [1992, p. 452].

Generic framework, so could not implement custom annealing schedules with restarts, etc. Furthermore, at what point is the algorithm 'adjusted too much' to the problem?

**The framework** Compare with scipy.optimize and nevergrad; also explain that they do not specifically target neural networks

## 9.3 Conclusions

## 9.4 Future work

# Appendix A

# Framework user manual

## A.1 Installing

Before starting the installation, ensure that Python 3.7 or later is available on your machine. The framework uses the `poetry` Python package for dependency management, and it is necessary to first install this package via `python3 install poetry`. Then, the framework's dependencies can be installed by running the following command in the `framework` folder:

```
poetry install
```

## A.2 Documentation

The documentation of the neural framework `nf` package is provided in HTML format at `framework/docs/nf/index.html` and the same information is also available in PDF format at `framework/docs.pdf`. Notice that this documentation is only of the framework itself and not the demonstrational scripts provided alongside it. Please refer to the documentation itself for information on how to develop custom agents, problems, and experiments. Furthermore, the demonstrations explained in Appendix A.3 provide starting points for how you could set up your own experiments.

If you wish to generate the documentation yourself, run

```
./generate_docs_html.sh
```

from within the `framework` folder to generate the HTML documentation, or

```
./generate_docs_pdf.sh
```

to obtain the PDF documentation. Note that the latter requires additional dependencies, namely `pandoc` and `xelatex`.

## A.3 Running the demonstrations

The neural framework package comes with three experiments for demonstration. Each experiment runs all of the implemented agents on a different problem. Notice, however, that the hyperparameters are varied depending on the problem

to ensure that the agents are effective. This was especially important for the simulated annealing agent. Various metrics are visualised including weight and output spaces as well as training loss. The following demonstrations are implemented:

- `demo_stripe_problem.py` of the RBF stripe problem as given in Section 5.2;
- `demo_shallow_problem.py` of a neural problem with a shallow excitation gradient (this problem was not analysed in this report but described in detail in Figure 7 and accompanying text of `research/progress/main.pdf`); and
- `demo_simple_problem.py` of a simple neural problem without local minima.

To run any of the implementations, simply execute the command

```
python3 demo_stripe_problem.py
```

replacing `demo_stripe_problem.py` with one of the scripts listed above. This will start a web server on port 5000 by default and open a web browser to `http://localhost:5000/` where the front end will be running. The folder `framework/screenshots` contains a screenshot of each of the demonstrations.

There are a variety of command line options which are identical for these demonstrational scripts. To obtain this information, simply run the script with the `-h` option appended. Below is an example for the `demo_stripe_problem.py` script.

```
$ python3 demo_stripe_problem.py -h
usage: demo_stripe_problem.py [-h] [--batch-size EPOCH_BATCH_SIZE]
                              [--batches EPOCH_BATCHES] [--columns COLS]
                              [--port PORT]

A demonstrational tool for the neural framework

optional arguments:
  -h, --help            show this help message and exit
  --batch-size EPOCH_BATCH_SIZE
                        Number of epochs to train for per agent per batch
  --batches EPOCH_BATCHES
                        Number of batches of training to perform for each
                        agent
  --columns COLS        The number of columns that the visualisations should
                        be arranged
  --port PORT           The port to run the server on
```

## A.4 Using the front end

As previously mentioned, when starting an experiment, a browser window will open with the front end. The web page will contain a grid of visualisations. These visualisations will usually be line plots where the $x$ and $y$ axes can be

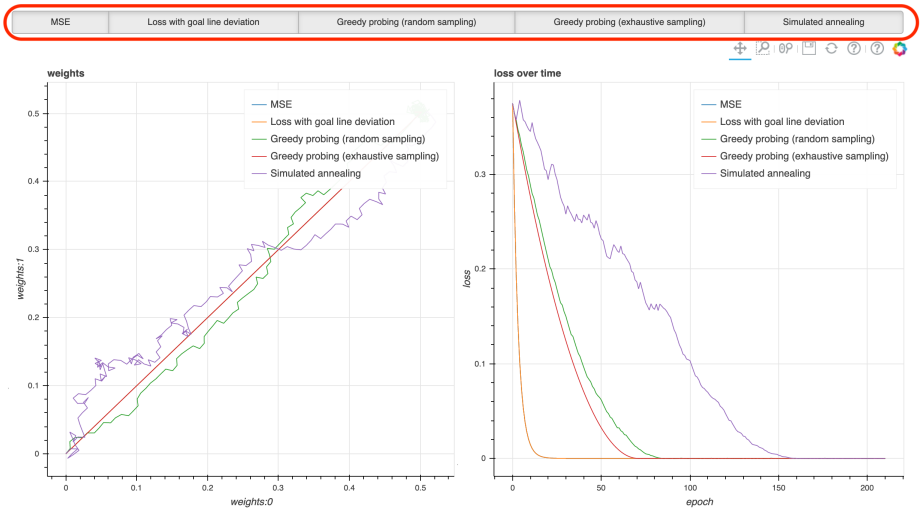



Figure A.1: Screenshot of the top part of the web page that constitutes the front end of the neural framework for the `demo_simple_problem.py` experiment. The area to the top marked in red contains the buttons for toggling each of the agents.

arbitrary metrics. The exact visualisations to be displayed are defined in the setup of the experiment (in an argument to the `run_server()` method).

Initially, the plots will be empty when the page is loaded. This is because the agents will start training and the plot is only updated after a specific number of epochs, defined by the `batch_size` command line argument. These updates happen in real time as training progresses. If the plot is updating too slowly, it is recommended to simply lower the value of `batch_size`.

Each visualisation will include the metrics gathered on all agents, each of a different colour (but the colours are consistent for each plot). You may each agent's visibility using the buttons at the top, as shown in Figure A.1. This will hide or show all plot lines with that agent's data.

# Bibliography


Aly, A., Guadagni, G. & Dugan, J. B. (2019). Derivative-free optimization of neural networks using local search. In *2019 IEEE 10th annual ubiquitous computing, electronics mobile communication conference (uemcon)* (pp. 293–299). doi:10.1109/UEMCON47517.2019.8993007

Battiti, R. & Tecchiolli, G. (1995). Training neural nets with the reactive tabu search. *IEEE Transactions on Neural Networks*, *6*(5), 1185–1200. doi:10.1109/72.410361

Bélisle, C. J. P., Romeijn, H. E. & Smith, R. L. (1993). Hit-and-run algorithms for generating multivariate distributions. *Mathematics of Operations Research*, *18*(2), 255–266. Retrieved from http://www.jstor.org/stable/3690278

Blum, E. K. (1989). Approximation of boolean functions by sigmoidal networks: Part i: Xor and other two-variable functions. *Neural Computation*, *1*(4), 532–540. doi:10.1162/neco.1989.1.4.532

Buhmann, M. D. (2000). Radial basis functions. *Acta Numerica*, *9*, 1–38. doi:10.1017/S0962492900000015

Burkov, A. (2019). *The hundred-page machine learning book*. Andriy Burkov.

Černý, V. (1985). Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, *45*(1), 41–51. doi:10.1007/bf00940812

Choi, B., Lee, J.-H. & Kim, D.-H. (2008). Solving local minima problem with large number of hidden nodes on two-layered feed-forward artificial neural networks. *Neurocomputing*, *71*(16-18), 3640–3643. doi:10.1016/j.neucom.2008.04.004

Elliott, T. (2019). The state of the octoverse. Retrieved from https://github.blog/2019-01-24-the-state-of-the-octoverse-machine-learning/

Friedman, J. H., Bentley, J. L. & Finkel, R. A. (1977). An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.* *3*(3), 209–226. doi:10.1145/355744.355745

GitHub. (2019). The state of the octoverse. Retrieved from https://octoverse.github.com/

Glorot, X., Bordes, A. & Bengio, Y. (2011). Deep sparse rectifier neural networks. In G. Gordon, D. Dunson & M. Dudík (Eds.), *Proceedings of the fourteenth international conference on artificial intelligence and statistics* (Vol. 15, pp. 315–323). Proceedings of Machine Learning Research. Fort Lauderdale, FL, USA: PMLR. Retrieved from http://proceedings.mlr.press/v15/glorot11a.html

Goldblum, M., Geiping, J., Schwarzschild, A., Moeller, M. & Goldstein, T. (2019). Truth or backpropaganda? an empirical investigation of deep learning theory. eprint: arXiv:1910.00359. Retrieved from https://arxiv.org/abs/1809.10749

Gorse, D., Shepherd, A. J. & Taylor, J. G. (1997). The new ERA in supervised learning. *Neural Networks*, *10*(2), 343–352. doi:https://doi.org/10.1016/S0893-6080(96)00090-1

Hastie, T., Friedman, J. & Tisbshirani, R. (2017). *The elements of statistical learning: Data mining, inference, and prediction* (2nd). Springer.

Hirasawa, K., Togo, K., Murata, J., Ohbayashi, M., Shao, N. & Hu, J. (1998). A new random search method for neural networks learning-random search with variable search length (rasval). In *1998 IEEE international joint conference on neural networks proceedings. IEEE world congress on computational intelligence (cat. no.98ch36227)* (Vol. 2, 1602–1607 vol.2). doi:10.1109/IJCNN.1998.686017

Johnson, S. G. (2015). Saddle-point integration of $C_\infty$ "bump" functions. eprint: arXiv:1508.04376. Retrieved from https://arxiv.org/abs/1508.04376

Karayiannis, N. (1998). Learning algorithms for reformulated radial basis neural networks. (Vol. 3, 2230–2235 vol.3). doi:10.1109/IJCNN.1998.687207

Kawaguchi, K. (2016). Deep learning without poor local minima. eprint: arXiv:1605.07110. Retrieved from https://arxiv.org/abs/1605.07110

Kingma, D. P. & Ba, J. (2014). Adam: A method for stochastic optimization. eprint: arXiv:1412.6980. Retrieved from https://arxiv.org/abs/1412.6980

Kirkpatrick, S., Gelatt, C. D. & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, *220*(4598), 671–680. doi:10.1126/science.220.4598.671. eprint: https://science.sciencemag.org/content/220/4598/671.full.pdf

Laurent, T. & von Brecht, J. (2018). Deep linear networks with arbitrary loss: All local minima are global. In J. Dy & A. Krause (Eds.), *Proceedings of the 35th international conference on machine learning* (Vol. 80, pp. 2902–2907). Proceedings of Machine Learning Research. Stockholmsmässan, Stockholm Sweden: PMLR. Retrieved from http://proceedings.mlr.press/v80/laurent18a.html

LeCun, Y., Bengio, Y. & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444. doi:10.1038/nature14539

Lewis, J. P. & Weir, M. K. (1999). Subgoal chaining and the local minimum problem. In *IJCNN'99. international joint conference on neural networks. proceedings (cat. no.99ch36339)* (Vol. 3, pp. 1844–1849). doi:10.1109/IJCNN.1999.832660

Lo, J. T.-H., Gui, Y. & Peng, Y. (2012). Overcoming the local-minimum problem in training multilayer perceptrons with the NRAE training method. In *Advances in neural networks – ISNN 2012* (pp. 440–447). doi:10.1007/978-3-642-31346-2_50

Lo, J. T.-H., Gui, Y. & Peng, Y. (2017). Solving the local-minimum problem in training deep learning machines. In *Neural information processing* (pp. 166–174). doi:10.1007/978-3-319-70087-8_18

Loomis, L. & Sternberg, S. (1990). *Advanced calculus*. Math Series. Jones and Bartlett Publishers.

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, ... Xiaoqiang Zheng. (2015). TensorFlow: Large-scale ma-

chine learning on heterogeneous systems. Retrieved from https://www.tensorflow.org/

McCulloch, W. S. & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, *5*(4), 115–133. doi:10.1007/bf02478259

Neal, R. M. (1992). Connectionist learning of belief networks. *Artificial Intelligence*, *56*(1), 71–113. doi:10.1016/0004-3702(92)90065-6

Nelder, J. A. & Mead, R. (1965). A Simplex Method for Function Minimization. *The Computer Journal*, *7*(4), 308–313. doi:10.1093/comjnl/7.4.308. eprint: https://academic.oup.com/comjnl/article-pdf/7/4/308/1013182/7-4-308.pdf

Nguyen, Q., Mukkamala, M. C. & Hein, M. (2018). On the loss landscape of a class of deep neural networks with no bad local valleys. eprint: arXiv: 1809.10749. Retrieved from https://arxiv.org/abs/1809.10749

Press, W. H., Teukolsky, S. A., Vetterling, W. T. & Flannery, B. P. (1992). *Numerical recipes in c (2nd ed.): The art of scientific computing*. USA: Cambridge University Press.

Reem, D. (2017). Remarks on the cauchy functional equation and variations of it. *Aequationes mathematicae*, *91*(2), 237–264. doi:10.1007/s00010-016-0463-6

Rios, L. & Sahinidis, N. (2009). Derivative-free optimization: A review of algorithms and comparison of software implementations. *Journal of Global Optimization*, *56*. doi:10.1007/s10898-012-9951-y

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, *65*(6), 386–408. doi:10.1037/h0042519

Rudin, W. (2006). *Functional analysis*. International series in pure and applied mathematics. McGraw-Hill.

Rumelhart, D. E., Hinton, G. E. & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536. doi:10.1038/323533a0

Russell, S. & Norvig, P. (2010). *Artificial intelligence: A modern approach* (3rd). Pearson.

Such, F. P., Madhavan, V., Conti, E., Lehman, J., Stanley, K. O. & Clune, J. (2017). Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *CoRR*, *abs/1712.06567*. arXiv: 1712.06567. Retrieved from http://arxiv.org/abs/1712.06567

Weir, M. K. (2019). CS3105: Cases & approaches 2: Neural local minima. Retrieved from https://studres.cs.st-andrews.ac.uk/2018_2019/CS3105/Lectures/CS3105%20Cases%20&%20Approaches/CS3105%20Case%20&%20Approach%202%20S2_19.pdf

Weir, M. K., Buck, A. & Lewis, J. P. (2006). Potbug: A mind's eye approach to providing bug-like guarantees for adaptive obstacle navigation using dynamic potential fields. In S. Nolfi, G. Baldassarre, R. Calabretta, J. C. T. Hallam, D. Marocco, J.-A. Meyer, ... D. Parisi (Eds.), *From animals to animats 9* (pp. 239–250). Berlin, Heidelberg: Springer Berlin Heidelberg.

Weir, M. K., Lewis, J. P. & Milligan, G. (2000). Using tangent hyperplanes to direct neural training. In *Proceedings of the international ICSC symposium on neural computation*.