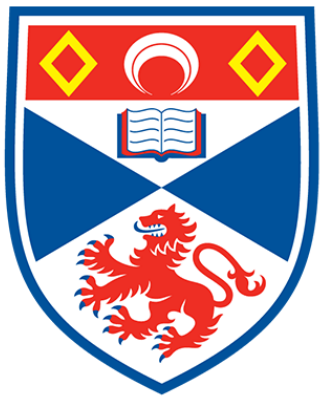
Senior Honours Project



University of
St Andrews

Freeing Neural Training Through Surfing

Georg Wölflein
170011885
Supervisor: Dr. Mike Weir

April 13, 2020

Word count: 7205 words

Abstract

TODO

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 7205 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

Contents

Chapter 1

Introduction

Describe the problem you set out to solve and the extent of your success in solving it. You should include the aims and objectives of the project in order of importance and try to outline key aspects of your project for the reader to look for in the rest of your report. **TODO**

Chapter 2

Context survey

2.1 Neural networks

Backpropagation The backpropagation (BP) algorithm, attributed to Rumelhart et al. [1986], is the classical method of training neural networks. It involves computing the gradient of the loss function with respect to the weights and then using some gradient-based optimisation technique such as gradient descent to update the weights. With the rise in popularity of deep neural networks, methods have been developed to increase the speed of converging to a minimum. The 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 is guaranteed to converge to local (but likely not global) minima.

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 gradient 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 generalized 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[1] [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

[1] 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.

variants of random and local search [Battiti and Tecchiolli 1995; Hirasawa et al. 1998]. These approaches seemed to find better minima 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 are often able to escape some[2] 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) which, while providing valuable practical insight, made it impossible to examine the structure of the loss surface analytically in order to assess issues such as severely suboptimal local minima.

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]. Therefore, it can be said that the local minimum problem is still an active area of research.

The local minimum problems as it relates to neural training has been investigated extensively here in St Andrews. 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 et al. [2000].

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

2.2 Implementation tools

- TensorFlow
- keras

TODO

Chapter 3

Requirements specification

Primary objectives:

1. Design a generic framework that can be used for various neural training algorithms with a clear set of inputs and outputs at each step. This framework should include benchmarking capabilities.
2. For a simple case of this framework (when the dimensionality of the control space and output space are suitably low), implement a visualisation tool that shows the algorithm's steps.
3. Design and implement the neural surfing technique.
4. Evaluate the performance of this and other algorithms on tasks of differing complexity, especially with regard to the local minimum problem and similar issues.

Secondary objectives:

1. Investigate how this approach can be generalized to other numerical optimisation problems.

3.1 Ethics

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

Part I

Theory

Chapter 4

Neural network theory

4.1 Supervised learning

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

$$R(\mathbf{x}) = \hat{y} = y + \epsilon \tag{4.1}$$

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 ϵ 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,

$$\mathcal{I}_A = \mathbb{R}^D. \tag{4.2}$$

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

$$\mathbf{y} = \begin{bmatrix} y_1 & y_2 & \cdots & y_N \end{bmatrix}^\mathsf{T} \tag{4.3}$$

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

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_N \end{bmatrix}^\mathsf{T} \tag{4.4}$$

for representing the corresponding feature vectors.

Definition 4 (Supervised learning)**.** A supervised learning algorithm for a regression task infers the function f given in (4.1) from a set of *labelled training data* of the form explained previously. We use the vector

$$\hat{\mathbf{y}} = \begin{bmatrix} \hat{y}_1 & \hat{y}_2 & \cdots & \hat{y}_N \end{bmatrix}^\mathsf{T} \tag{4.5}$$

to denote the prediction that f produces for each training sample.

4.2 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

$$z_i = \sum_{j=1}^{n} w_{j,i} a_j + b_i \tag{4.6}$$

where $a_j \in \mathbb{R}$ is another unit j's *activation* and $b_i \in \mathbb{R}$ is the ith 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

$$a_i = g(z_i) = g\left(\sum_{j=1}^{n} w_{j,i} a_j + b_i\right). \tag{4.7}$$

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 (4.7), 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[3] as

$$g_{\text{thres}}(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}. \tag{4.8}$$

Commonly used activation functions in modern neural networks include the sigmoid

$$g_{\text{sig}}(x) = \frac{1}{1 + e^{-x}} \tag{4.9}$$

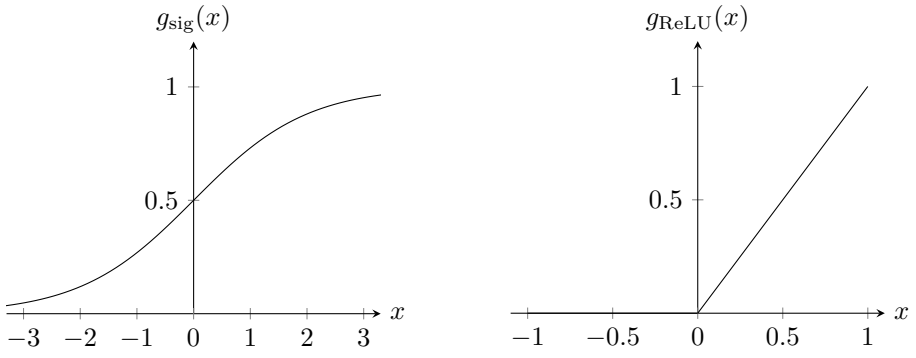
and the rectified linear unit (ReLU)

$$g_{\text{ReLU}} = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases} \tag{4.10}$$

which are depicted in Figure 4.1. Unlike g_{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

[3] 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 4.1: Plots of the the two most common activation functions.

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 prevelant in shallow[4] networks so the sigmoid function still remains popular [Neal 1992].

Particularly in deep neural networks, different neurons (grouped in *layers*, see Section 4.2.2) 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 generalized 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 (4.1). 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 ith unit to the value the ith element in $\mathbf{x}$ for $i = 1, \ldots, D$. Then, we propagate the activations using (4.7) until finally 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].

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

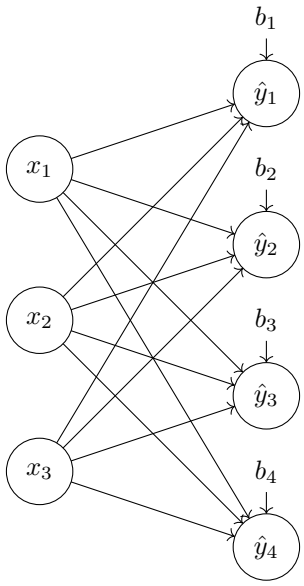


Figure 4.2: A single-layer perceptron with three input and four output neurons.

4.2.1 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 4.2. 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 (4.8) [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 (4.7) to give the activation of a particular output neuron i as

$$a_i = y_i = g\left(z_i\right) = g\left(\sum_{j=1}^{m} w_{j,i} x_j + b_i\right) = g\left(\mathbf{w}_i^{\mathsf{T}} \mathbf{x}_i + b_i\right) \tag{4.11}$$

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

$$\mathbf{S}(\mathbf{x}; \mathbf{W}, \mathbf{b}) = \mathbf{g}\left(\mathbf{W}^\mathsf{T}\mathbf{x} + \mathbf{b}\right) \tag{4.12}$$

where the $m \times n$ matrix

$$\mathbf{W} = \begin{bmatrix} \mathbf{w}_1 & \mathbf{w}_2 & \cdots & \mathbf{w}_n \end{bmatrix} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \cdots & w_{m,n} \end{bmatrix} \tag{4.13}$$

captures all weights and the vector

$$\mathbf{b} = \begin{bmatrix} b_1 & b_2 & \cdots & b_n \end{bmatrix}^\mathsf{T} \tag{4.14}$$

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 (4.1). Moreover, if we additionally use the threshold activation function from (4.8), we arrive at the SLP model given by Rosenblatt [1958].

4.2.2 Multi-layer perceptron

A multi-layer perceptron[5] (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

$$M(\mathbf{x}; \mathscr{P}) = \hat{y} = f_L\left(\mathbf{f}_{L-1}\left(\ldots\left(\mathbf{f}_1\left(\mathbf{x}\right)\right)\right)\right) \tag{4.15}$$

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 $f_L(\mathbf{x}) = S(\mathbf{x}; \mathbf{W}_L, \mathbf{b}_L)$.

[5] Unlike SLPs, the activation function in a MLP as defined in literature does not necessarily need to be the binary threshold function g_{thres}; in fact, it is often one of the more modern activation functions explained in Section 4.2 [Burkov 2019; Hastie et al. 2017]. Hence we can use the term 'multi-layer perceptron'.

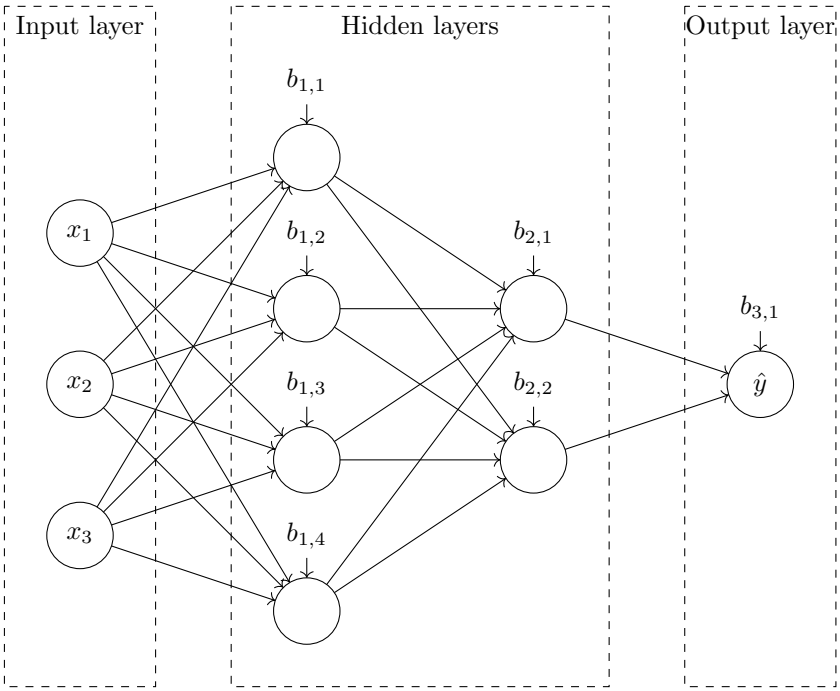


Figure 4.3: A multi-layer perceptron with three inputs and two hidden layers.

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 4.3. 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 (4.12).

4.3 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 8.1).

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

$$C(\mathbf{x}) = \hat{y} = y + \epsilon \tag{4.16}$$

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[6] 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_{thres} from (4.8) in the form

$$C(\mathbf{x}) = g_{\text{thres}}(R(\mathbf{x}) - t). \tag{4.17}$$

□

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 (8.1) 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 4.4. The equation of decision boundary can be obtained by setting the output 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

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

[6] 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.

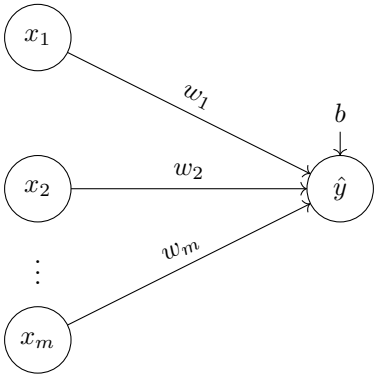


Figure 4.4: The DAG representing a SLN with m inputs and one output unit.

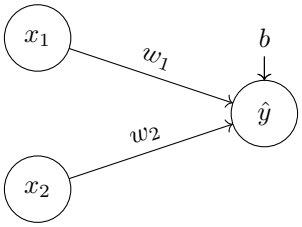


Figure 4.5: A simple MLP with one layer and two inputs (equivalently, a SLN with two inputs and one ouput).

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

$$0 = w_1x_2 + w_2x_2 + \cdots + w_mx_m + b + \ln\left(\frac{1}{t}-1\right) \tag{4.18}$$

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 4.5. 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

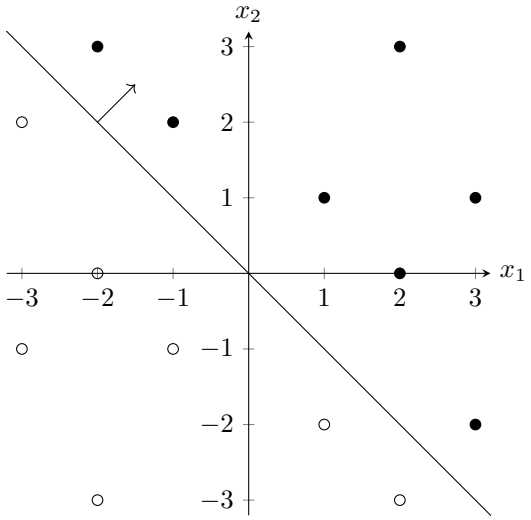


Figure 4.6: Plot of the input space of the MLP from Figure 4.5 with sigmoid activation where $w_1 = w_2 = 1$, $b = 0$, and some sample data. Filled in dots represent a prediction of $\hat{y} > \frac{1}{2}$ whereas the empty circular dots represent $\hat{y} < \frac{1}{2}$.

line can be obtained from (4.18) 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 4.6 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.

Chapter 5

Neural network training

This chapter will introduce two methods of training neural networks to provide an intuition on how this can be achieved. Later in this report, we will look at some issues related to these methods as a means of setting the scene for the neural surfing technique.

In this context, *training* refers to the process of changing the 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'. 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,

$$E\left(\mathbf{y}, \hat{\mathbf{y}}\right) = \frac{1}{N} \sum_{i=1}^{N} \left(\hat{y}_i - y_i\right)^2. \tag{5.1}$$

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

$$L(\mathbf{p}) = \sum_{i=1}^{N} \left(M\left(\mathbf{x}_i; \mathbf{p}\right) - y_i\right)^2. \tag{5.2}$$

Notice the similarity to (5.1). However, we have 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.

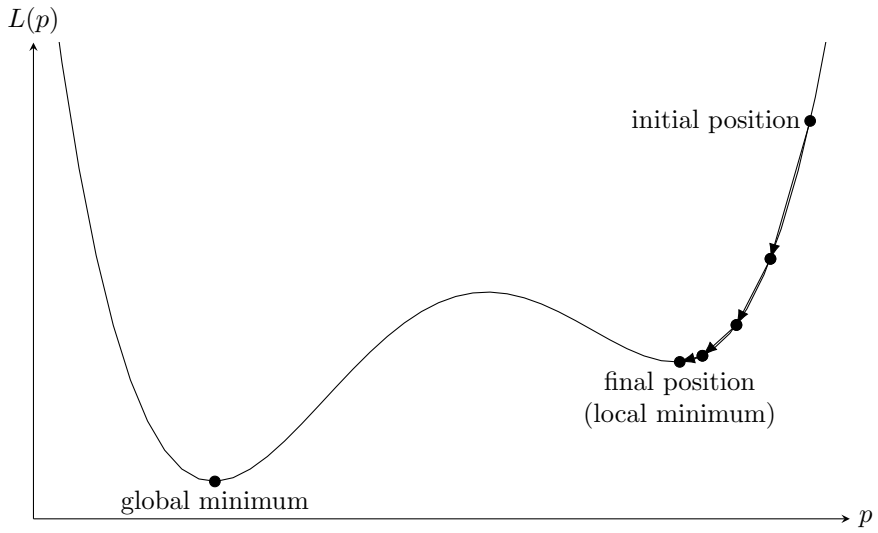


Figure 5.1: An illustration gradient descent training on an error-weight surface with only one parameter (not drawn to scale).

5.1 Backpropagation

Backpropagation (BP) via gradient descent is an iterative algorithm for training neural networks that, provided a suitable learning rate α, is guaranteed to converge to a *local minimum* (see Section 6). The main idea is as follows:

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

Figure 5.1 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 8.1.

Example 2 (Gradient in a single-layer MLP)**.** Let us revisit the SLN with m inputs and one output from Figure 4.4. The loss function will be in terms of

the trainable parameters, i.e. the weights $\mathbf{w}$ and bias b, so

$$\begin{aligned} L = L(\mathbf{w}, b) &= \sum_{i=1}^{N} \left(S\left(\mathbf{x}_i; \mathbf{w}, b\right) - y_i\right)^2 \\ &= \sum_{i=1}^{N} \left(g\left(\mathbf{w}^\mathsf{T}\mathbf{x}_i + b\right) - y_i\right)^2. \end{aligned} \tag{5.3}$$

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

$$\begin{aligned} \frac{\delta L}{\delta b} &= 2\sum_{i=1}^{N} \left(g\left(\mathbf{w}^\mathsf{T}\mathbf{x}_i + b\right) - y_i\right) \frac{\delta}{\delta b}\left(g\left(\mathbf{w}^\mathsf{T}\mathbf{x}_i + b\right) - y_i\right) \\ &= 2\sum_{i=1}^{N} \left(g\left(\mathbf{w}^\mathsf{T}\mathbf{x}_i + b\right) - y_i\right) g'\left(\mathbf{w}^\mathsf{T}\mathbf{x}_i + b\right), \end{aligned} \tag{5.4}$$

and similarly we can differentiate with respect to the weights

$$\begin{aligned} \frac{\delta L}{\delta \mathbf{w}} &= 2\sum_{i=1}^{N} \left(g\left(\mathbf{w}^\mathsf{T}\mathbf{x}_i + b\right) - y_i\right) \frac{\delta}{\delta \mathbf{w}}\left(g\left(\mathbf{w}^\mathsf{T}\mathbf{x}_i + b\right) - y_i\right) \\ &= 2\sum_{i=1}^{N} \left(g\left(\mathbf{w}^\mathsf{T}\mathbf{x}_i + b\right) - y_i\right) g'\left(\mathbf{w}^\mathsf{T}\mathbf{x}_i + b\right) \mathbf{x}_i. \end{aligned} \tag{5.5}$$

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

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

5.2 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

$$\left\{ \mathbf{p}_i + r\frac{\begin{bmatrix} p_1 & p_2 & \cdots & p_P \end{bmatrix}^\mathsf{T}}{\left\| \begin{bmatrix} p_1 & p_2 & \cdots & p_P \end{bmatrix} \right\|} : \begin{array}{l} p_1, p_2, \ldots, p_P \in \{-1, 0, 1\} \text{ and} \\ \left\| \begin{bmatrix} p_1 & p_2 & \cdots & p_P \end{bmatrix} \right\| \neq 0 \end{array} \right\} \tag{5.6}$$

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 $\|\mathbf{p}_i - \hat{\mathbf{p}}\| = r$.

5.3 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 ith iteration is given by the probability distribution

$$P(\hat{\mathbf{p}}|\mathbf{p}_i) = \begin{cases} \exp\left(-\frac{k}{T_i}\left(L(\hat{\mathbf{p}}) - L(\mathbf{p}_i)\right)\right) & L(\hat{\mathbf{p}}) > L(\mathbf{p}_i) \\ 1 & L(\hat{\mathbf{p}}) \leq L(\mathbf{p}_i) \end{cases} \tag{5.7}$$

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

$$T_i = (1 - c)\, T_{i-1} = (1 - c)^i\, T_0 \tag{5.8}$$

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 Press et al. [1992] as remarked in Section 13.0.1) that makes it increasingly difficult to design SA in a generic fashion to suit the training of neural networks.

Chapter 6

The local minimum problem

6.1 The mathematics of local and global minima

Definition 11 (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 $f(\mathbf{p}) \leq f(\mathbf{x})$.

Definition 12 (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.

Definition 13 (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,

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

Definition 14 (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 15 (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 $\mathbf{x} \in \mathbb{R}^n \setminus \mathbf{0}$.

Theorem 3 (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 prove that training converges to a local minimum in Section 8.1. However, the proof of this theorem exceeds the scope of this report. The interested reader may consult Loomis and Sternberg [1990, p. 190].

6.2 The suboptimal local minimum problem

TODO : explain local minimum problem with respect to gradient descent, greedy probing, and SA (+ random initialisation)

Chapter 7

Neural surfing theory

7.1 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 16 (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

$$\mathcal{W}_A = \mathbb{R}^P. \tag{7.1}$$

Definition 17 (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 (4.5), 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

$$\mathcal{O}_\mathcal{A} = \mathbb{R}^N. \tag{7.2}$$

Lemma 4. *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 (4.13) and bias vector $\mathbf{b} \in \mathbb{R}^n$ from (4.14). By Definition 16, 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 5. *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 lth layer, m_l. Let n_l denote the number outputs for each layer l. Then, by Lemma 4,

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

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 5 is that we obtain a formula for the number of trainable parameters P in a MLP. By Definition 16, P determines the dimensionality of the weight space. On other other hand, Definition 17 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 $\mathcal{O}$.

7.1.1 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 18 (Weight-output mapping)**.** Given an artifical neural network $A : \mathbb{R}^m \to \mathbb{R}^n$ with m inputs and n outputs parameterized by a set of trainable parameters $\mathbf{w} \in \mathcal{W}_\mathcal{A}$, 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 6. *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 4.4. Modifying the formula for a SLN given in Definition 5 (4.12) 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 (4.4). By Definition 18, 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[7], 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 6.1. *For any SLN S, the function $h_S : \mathcal{W}_S \to \mathcal{O}_S$ is not a linear mapping.*

Proof. We will generalize the results from Theorem 6 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 ith 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 7.1.

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

Corollary 6.2 (Weight-output mapping in general)**.** *For any MLP M, $h_M : \mathcal{W}_M \to \mathcal{O}_M$ is not a linear mapping.*

[7] 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].

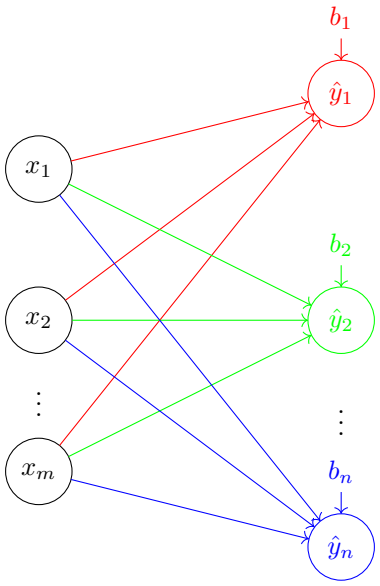


Figure 7.1: The DAGs representing the constructions of n SLNs with one output from a SLN with m inputs and n outputs. Each color represents one of the constructed smaller SLNs.

Proof. Let M have L layers. By Definition 6, M is a nested function of L SLNs. Corollary 6.1 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 6.2 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.

7.1.2 Gradient descent from the perspective of weight and output space

TODO : Write about how SGD with MSE usually gets viewed from the perspective of error-weight surface. However, it is interesting to look at the perspective of weight and output space. It becomes apparent that MSE can be

thought of as greedily trying to reduce the Euclidean distance from $\hat{\mathbf{y}}$ to $\mathbf{y}$ in output space.

7.2 Unrealizable regions

Definition 19 (Strongly unrealizable point)**.** Given an artificial neural network A, a point $\mathbf{p} \in \mathcal{O}_A$ in output space is *strongly unrealizable* 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 20 (Strongly unrealizable region)**.** Given an artificial neural network A, a *strongly* unrealizable region $\mathcal{U} \subset \mathcal{O}_A$ is a subspace of the output space where every point $\mathbf{p} \in \mathcal{U}$ is strongly unrealizable.

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 unrealizable region in output space. Hence we define a *weakly* unrealizable region for a particular neural learning algorithm as a region in output space that cannot be attained by a particular algorithm.

Lemma 7. *A strongly unrealizable region cannot encompass the whole output space.*

Proof. Let us consider an artifical neural network A. We will show that for every unrealizable region, $\mathcal{U} \subsetneq \mathcal{O}_A$. By Definition 20, $\mathcal{U} \subset \mathcal{O}_A$, so it remains to prove that every unrealizable 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 unrealizable because $\mathbf{w}$ achieves $\mathbf{p}$. Hence no unrealizable 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 unrealizable regions.

Example 3. A trivial example of an unrealizable 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 4.5. 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 unrealizable 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

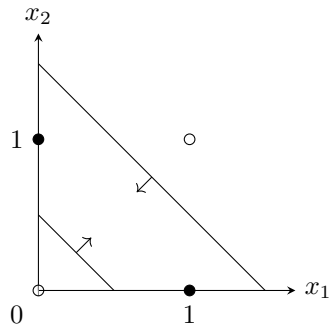


Figure 7.2: The locations of the two hyperplanes in input space acting as decision boundaries required to learn an XOR mapping. The filled-in dot represents an activation of 1 (true) and the circle represents 0 activation (false).

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 7.2. We will consider the same single-layer architecture from Figure 4.5 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

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

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

TODO : Explain significance if target is unrealizable (XOR example demonstrates that)

7.3 Goal-connecting paths

Definition 21 (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 unrealizable point (Definition 19). 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 22 (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 8. *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 8 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 unrealizable regions.

Chapter 8

Problems

8.1 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 8.1 shows the initial and target configurations of the hyperplanes for this 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.

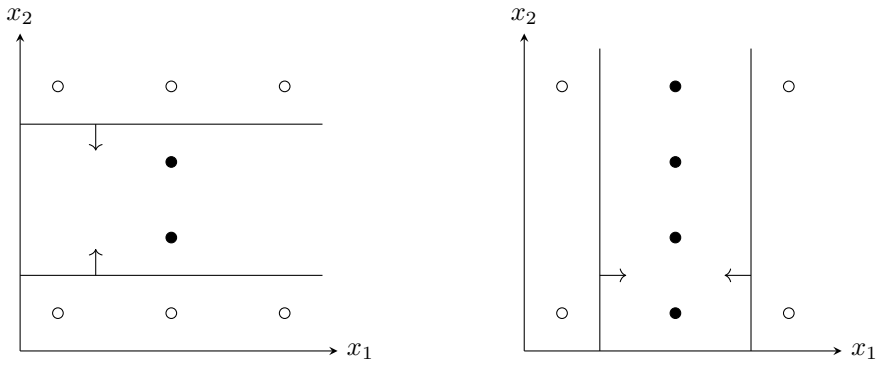
In this section, we will formulate a version of the stripe problem that requires no hidden layers and fewer input samples by using a different kind of activation function. The reduced number of parameters will lend itself better for analysis.

8.1.1 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[8] function $\phi : \mathbb{R} \to \mathbb{R}$ that satisfies the property $\phi(x) = \phi(\|x\|)$ [Buhmann 2000]. For RBFs to be useful activation functions,

[8] 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].



(a) Initial configuration (b) Target configuration

Figure 8.1: Hyperplanes in input space for the stripe problem with eight samples (adapted from Weir [2019]).

we three additional restrictions: (i) $\phi(0) = 1$; (ii) $\phi(x)$ is strictly increasing for $x < 0$ 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

$$\phi(x) = e^{-x^2} \tag{8.1}$$

and the bump function[9]

$$\phi(x) = \begin{cases} e^{1 - \frac{1}{1-x^2}} & \text{for } -1 < x < 1 \\ 0 & \text{otherwise} \end{cases}. \tag{8.2}$$

These functions, graphed in Figure 8.2, exhibit slightly different properties. The 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 9 (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 4.4. To obtain the

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

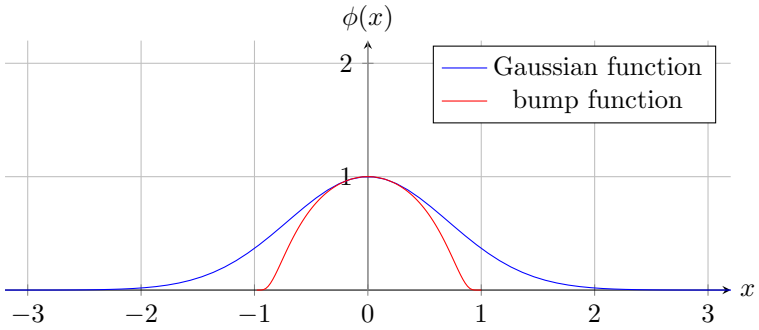


Figure 8.2: Plots of the two most common radial basis functions.

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

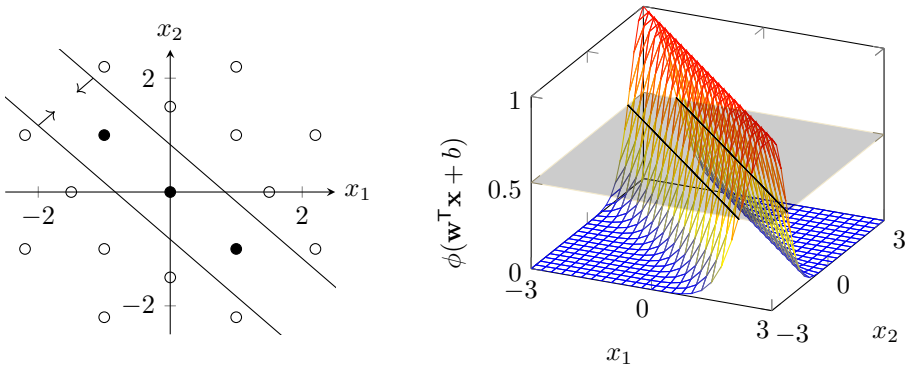
$$\begin{aligned} 0 < t &< 1 \\ \ln t &< \ln 1 \\ -\ln t &> 0 \\ \sqrt{-\ln t} &> 0 \end{aligned}$$

which means that $\sqrt{-\ln t} \neq 0$. Hence

$$\mathbf{w}^{\mathsf{T}}\mathbf{x} + b \pm \sqrt{-\ln t} = 0 \tag{8.3}$$

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 9 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.



(a) Input space with sample data

(b) Input-activation space

Figure 8.3: Plots of the hyperplanes of the MLP from Figure 4.5 with Gaussian RBF activation where $w_1 = w_2 = 1$, $b = 0$.

Example 5. Like in Example 1, let us consider once more a single-layered MLP M with two inputs, depicted in Figure 4.5, where $w_1 = w_2 = 1$ and $b = 0$. Let the threshold be $t = \frac{1}{2}$ again, but this time, we will use the Gaussian RBF as the activation function.

From (8.3), we obtain the equations of the hyperplanes as

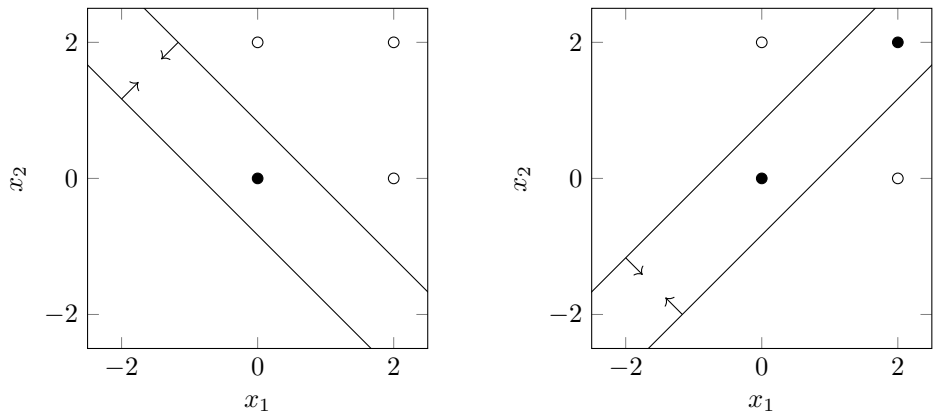
$$\begin{aligned} \mathbf{w}^\mathsf{T}\mathbf{x} + b \pm \sqrt{-\ln\frac{1}{2}} &= 0 \\ w_1x_1 + w_2x_2 + b &= \pm\sqrt{-\ln\frac{1}{2}} \\ x_1 + x_2 &= \pm\sqrt{-\ln\frac{1}{2}}, \end{aligned}$$

so the hyperplanes are at $x_2 = -x_1 - 0.8325\ldots$ and $x_2 = -x_1 + 0.8325\ldots$, as shown in Figure 8.3.

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.

8.1.2 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 8.1. However, since we established that the hyperplanes will always remain parallel in our RBF network and since they rotate around the



(a) Initial configuration

(b) Target configuration

Figure 8.4: The hyperplanes in the initial and target configurations of the RBF stripe problem.

origin when b is fixed, we can create a much simplified version of the stripe problem using only four samples.

Figure 8.4 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.

8.1.3 Global minima

8.1.4 A suboptimal local minimum

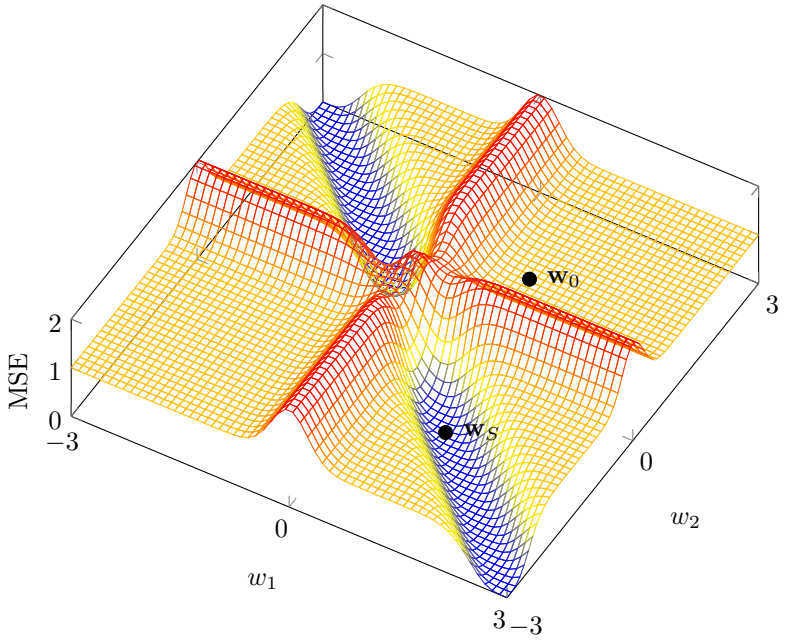


Figure 8.5: Error-weight surface of the stripe problem with Gaussian activation.

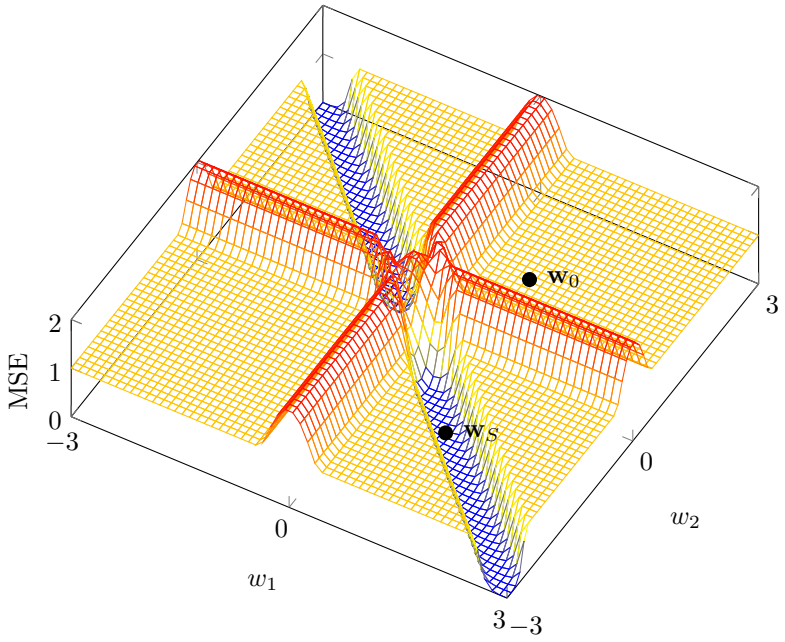


Figure 8.6: Error-weight surface of the stripe problem with bump activation.

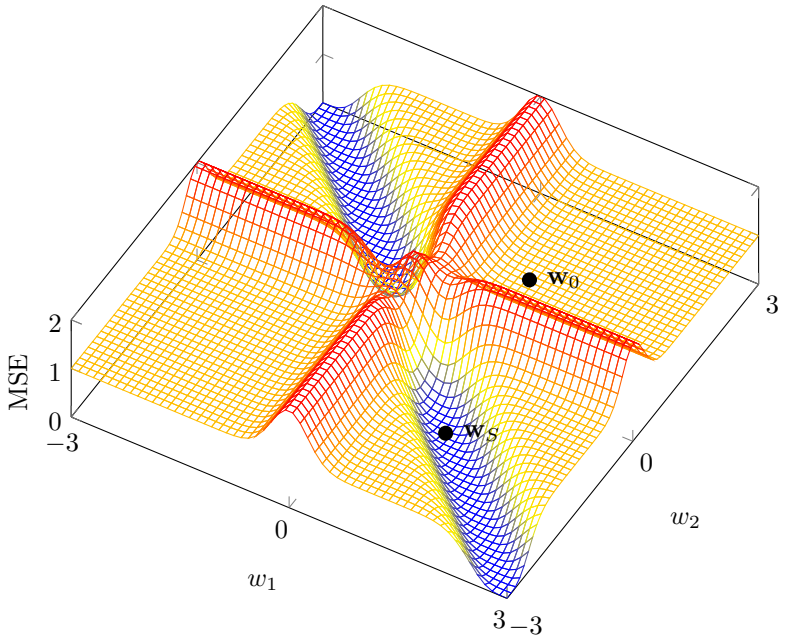


Figure 8.7: Error-weight surface of the stripe problem with Gaussian activation.

Chapter 9

Generalising neural surfing

Generalize to classification as regression with multiple output variables

Part II

Framework

Chapter 10

Design

TODO

Chapter 11

Implementation

TODO

Chapter 12

Experimental results

TODO

Part III

End

Chapter 13

Evaluation and critical appraisal

13.0.1 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]. **TODO** : 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?

Chapter 14

Conclusions and future work

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

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_∞ "bump" functions. eprint: arXiv:1508.04376. Retrieved from https://arxiv.org/abs/1508.04376

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

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

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