Insert Title Here

by

PARBHU Varun

Project Supervisor: Professor (Dr) BHURUTH MUDDUN OSK

Project submitted to the Department of Mathematics,
Faculty of Science, University of Mauritius,
as a partial fulfillment of the requirement
for the degree of
M.Sc. Mathematical and Scientific Computing (Part time)

December 2022

Contents

List of Figures

List of Tables

Acknowledgement

Placeholder

Abstract

Placeholder

Terms and Definitions

α	Learning Rate
η	Reduced Learning Rate
$\mathbf{I}$	Identity Matrix
$\odot$	Hadamard Product

Placeholder

Placeholder

Chapter 1

Introduction

Introduction

Chapter 2

Deep Learning Networks

Deep learning networks use many hidden layers to learn structure in large datasets and the key aspect is that the learning process is not human engineered. Starting from a simple mathematical model of a neuron (McCulloch & Pitts 1943) and the development of the perceptron (Rosenblatt 1958), deep learning networks were particularly difficult to train. The introduction of the backpropagation algorithm by David Rumelhart overcome the training problem of deep neural networks. Thus, allowing networks to tune their internal parameters iteratively over training datasets. The procedure repeatedly adjusts the weights of the connections in the network to minimise a measure of the difference between the actual output vector of the net and the desired output vector (Rumelhart, Hinton & Williams 1986). Using a general-purpose learning procedures, deep learning networks are able to find non-linear relationship within high-dimensional data. Kunihiko Fukushima, Yann LeCun, Geoffrey Hinton and Yoshua Bengio contributed to the creation of richer and more complex multilayered neural networks to tackle different tasks in the processing of images, video, speech, audio and text. With such features surpassing conventional machine-learning techniques and the access to high-end graphic processors and computing power along the rise of Big Data, deep learning networks are being applied to many domains of science, business and government (LeCun, Bengio & Hinton 2015). Hence, allowing tasks such as image recognition, speech recognition, natural language processing, sentiment analysis, fraud detection, etc. to be tackled with promising results.

We start by introducing the perceptron to classify linearly separable datasets and a proof of the perceptron convergence theorem. The solution and limitations of the perceptron for logic functions are explored before introducing feedforward networks. We also explore some common activation function used in feedforward networks. We then introduce and derive the backpropagation algorithm of feedforward networks with multiple hidden layers and introduce different activation function. Finally, we define the RNN, LSTM and BRNN neural network architecture.

2.1 The Perceptron

In 1943, McCulloch and Walter Pitts demonstrated that simple binary threshold units wired up as logical gates could be used to build a digital computer (McCulloch & Pitts 1943). The McCulloch-Pitts (MCP) neuron is a simple mathematical model of a biological neuron. It was the earliest mathematical model of a neural network and had only three types of weights; excitatory (1), inhibitory (-1) and inactive (0). The model had an activation function which had a value of 1 if the weighted sum was greater or equal to a given threshold, else 0. Using the MCP neuron, one of the first digital computers that contained stored programs was built. However, the MCP neuron was very restrictive.

The perceptron algorithm proposed by Rosenblatt (Rosenblatt 1958) is motivated by and overcomes some limitations of the MCP Neuron. For example, the input was not restricted to boolean values but expanded to real numbers. Rosenblatt proved that if the data used to train the perceptron are linearly separable classes, then the perceptron algorithm convergences and separates the two classes by a hyperplane.

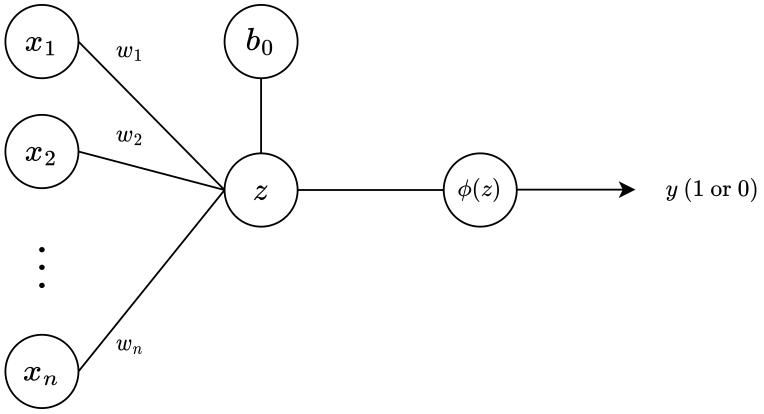

Figure 2.1: Rosenblatt perceptron

Let $\mathbf{x}$ be a vector of inputs where each $x_i \in \mathbb{Z}$ and $\mathbf{w}$ be a vector of weights corresponding to the input signals where each $w_i \in \mathbb{Z}$.

$$\mathbf{x} = \Big[x_0, x_1, \ldots, x_n\Big]^T \quad \text{and} \quad \mathbf{w} = \Big[w_0, w_1, \ldots, w_n\Big]^T$$

Then, the mathematical definition of the perceptron is given by

$$\phi(z) = \begin{cases} 1 & \text{if } \mathbf{w^T x} + \mathbf{b} \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

where $\phi(z)$ is known as the hard delimiter.

Consider sets in $\mathbb{R}^2$ given by $\mathbf{A} = \{a_1\}$ and $\mathbf{B} = \{b_1, b_2\}$ where $a_1 = (1, 2)^T$, $b_1 = (-1, 2)^T$ and $b_2 = (0, -1)^T$. The two sets are linearly separable in $\mathbb{R}^2$ as shown in Figure 2.2. There are an infinite number of separating hyperplanes. One separating hyperplane through the origin is the line $x_1 + x_2 = 0$. The normal vector to this line is given by $w = (1, 1)$ and this line points in the direction of A.

$$\mathbf{A} = \left\{ \begin{bmatrix} 1 \\ 2 \end{bmatrix} \right\} \quad \mathbf{B} = \left\{ \begin{bmatrix} -1 \\ 2 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \end{bmatrix} \right\} \tag{2.2}$$

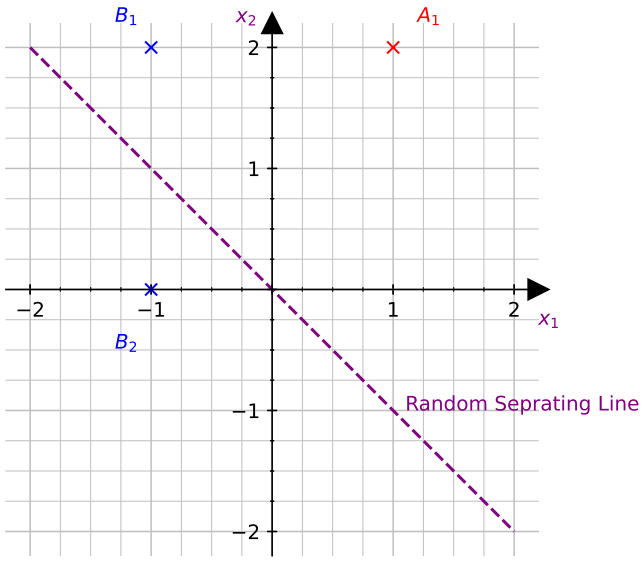

Figure 2.2: Set **A** and **B** with a random separating line

Comparing to the equation (2.1); we can deduce that

$$w_1 = 1 \quad w_2 = 1 \tag{2.3}$$

Visually, we can already see that the separating line does not split the points properly. The computation of the binary classification using the random line is given by

$$\phi(A_1) = \phi\left(\begin{bmatrix}1\\1\end{bmatrix} . \begin{bmatrix}1\\2\end{bmatrix}\right) = \phi(3) = 1$$

$$\phi(B_1) = f\left(\begin{bmatrix}1\\1\end{bmatrix} . \begin{bmatrix}-1\\2\end{bmatrix}\right) = \phi(1) = 1$$

$$\phi(B_2) = f\left(\begin{bmatrix}1\\1\end{bmatrix} . \begin{bmatrix}-1\\0\end{bmatrix}\right) = \phi(-1) = 0$$

The computation confirms the visual representation and groups $\mathbf{A}_1$ and $\mathbf{B}_1$ together and $\mathbf{B}_2$ separated which is incorrect. We can either alter the separating line (by moving the weight vector) with respect to $\mathbf{A}_1$ and/or $\mathbf{B}_1$.

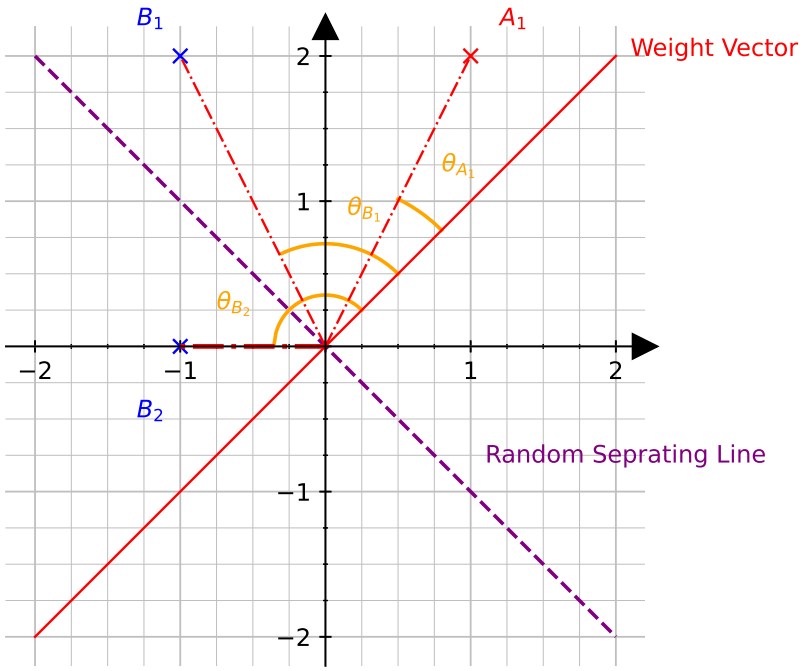

Figure 2.3: Set **A** and **B** with a random separating line and its weight vector

Since $\mathbf{B}_1$ is incorrectly classified, using the property of subtraction of vectors; we move the weight vector ($\mathbf{w}$) away from $\mathbf{B}_1$ and check the classification again.

$$\mathbf{w}^{(1)} = \mathbf{w} - \mathbf{B}_1 \tag{2.4}$$

$$= \begin{bmatrix} 1 \\ 1 \end{bmatrix} - \begin{bmatrix} -1 \\ 2 \end{bmatrix} \tag{2.5}$$

$$= \begin{bmatrix} 2 \\ -1 \end{bmatrix} \tag{2.6}$$

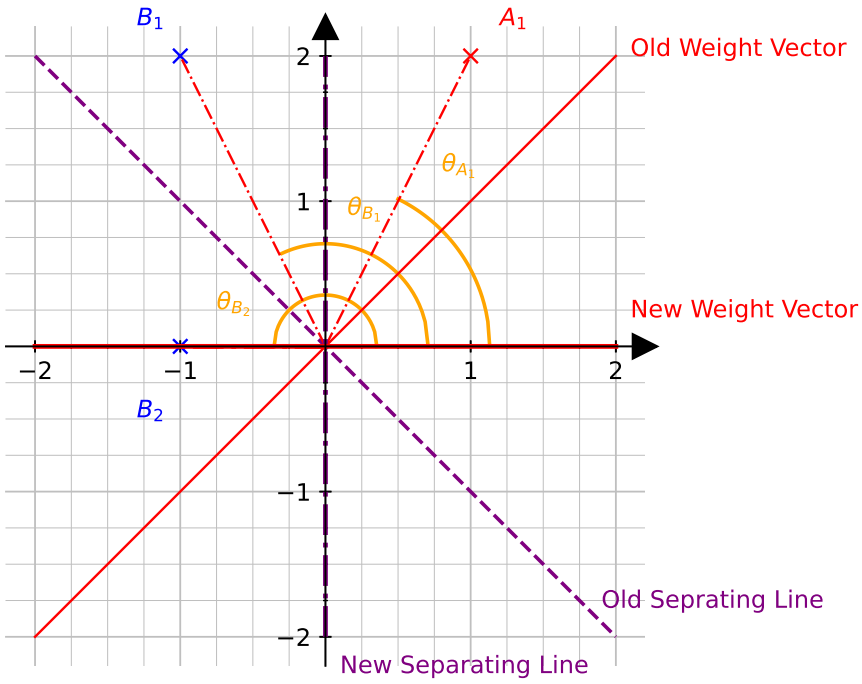

Figure 2.4: Set **A** and **B** after adjusting the separating line and its weight vector

Based on the new separating line, we can observe visually that the points are correctly classified. The computation of the binary classification using the new separating line $\mathbf{w}^{(1)}$ is given by

$$\phi(A_1) = \phi\left(\begin{bmatrix} 2 \\ -1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \end{bmatrix}\right) = \phi(0) = 1$$

$$\phi(B_1) = \phi\left(\begin{bmatrix} 2 \\ -1 \end{bmatrix} \cdot \begin{bmatrix} -1 \\ 2 \end{bmatrix}\right) = \phi(-4) = 0$$

$$\phi(B_2) = \phi\left(\begin{bmatrix} 2 \\ -1 \end{bmatrix} \cdot \begin{bmatrix} -1 \\ 0 \end{bmatrix}\right) = \phi(-2) = 0$$

Thus, the perceptron learned how to classify the two sets of points. The function $f(\mathbf{x})$ could be re-used to classify new points added to the sets.

Perceptron Algorithm

The perceptron algorithm adjusts the weights and as a result, the hard delimiter as well in order to linearly separate a set of binary labelled input.

Algorithm 1 Perceptron Algorithm

Require:

Let $t = 0$ and $\mathbf{w} = \begin{bmatrix} 0 & 0 & \dots & 0 \end{bmatrix}^T$
Consider the training set D $s.t$ $D = C_1 \cup C_2$
$C_1 \leftarrow$ input with label 1
$C_2 \leftarrow$ input with label 0
Start
while !convergence **do**
    Select a random input $\mathbf{x}$
    if $\mathbf{x} \in A$ and $\mathbf{w}.\mathbf{x} < 0$ **then**
        $\mathbf{w} = \mathbf{w} + \mathbf{x}$
    end if
    if $\mathbf{x} \in B$ and $\mathbf{w}.\mathbf{x} \geq 0$ **then**
        $\mathbf{w} = \mathbf{w} - \mathbf{x}$
    end if
end while

If the input data are binary and linearly separable; then algorithm (1) converges. The proof of convergence of the algorithm is known as the **perceptron convergence theorem**.

Theorem

Consider algorithm (1) and let D be a set of training vectors which are linearly separable. Let $\mathbf{w}^*$ be the weight vectors which defines the separating line with $||\mathbf{w}^*|| = 1$. Then the perceptron convergence theorem states that the number of mistakes m made by the perceptron algorithm satisfies

$$m \leq \frac{1}{\gamma^2} \quad \text{where} \quad \gamma = \min_{\mathbf{x} \in D} \frac{|\mathbf{w}^{*T}\mathbf{x}|}{||\mathbf{x}||_2} \tag{2.7}$$

Note

1. Since, $||\mathbf{w}^*|| = 1 \implies \cos(\theta) = \dfrac{\mathbf{w}^{*T}\mathbf{x}}{||\mathbf{x}||_2}$
2. If $||\mathbf{x}||_2 = 1$, that is, we scale all the training examples to have unit norm (which has no effect their orientation), then $\gamma = \min_{\mathbf{x} \in D} |\mathbf{w}^{*T}\mathbf{x}|$ is the minimum distance from any example $\mathbf{x} \in D$ to the separating line.

Proof

We first prove the inequality given by

$$(\mathbf{w}^{(t+1)})^T\mathbf{w}^* \geq (\mathbf{w}^{(t)})^T\mathbf{w}^* + \gamma \tag{2.8}$$

State 1

In state 1, let us consider $\mathbf{x}$ being positive ($\mathbf{x} \in C_1$) and incorrectly classified then

$$(\mathbf{w}^{(t+1)})^T\mathbf{w}^* = (\mathbf{w}^{(t)} + \mathbf{x})^T\mathbf{w}^* = (\mathbf{w}^{(t)})^T\mathbf{w}^* + \mathbf{x}^T\mathbf{w}^*$$

Assuming that all $||\mathbf{x}||_2 = 1$ then $\mathbf{x}^T\mathbf{w}^* \geq \gamma$ since γ is the minimum.

$$(\mathbf{w}^{(t+1)})^T\mathbf{w}^* \geq (\mathbf{w}^{(t)})^T\mathbf{w}^* + \gamma \tag{2.9}$$

Thus, we have proved (2.8) under **State 1**.

State 2

In state 2, let us consider $\mathbf{x}$ being negative ($\mathbf{x} \in C_2$) and incorrectly classified then

$$(\mathbf{w}^{(t+1)})^T\mathbf{w}^* = (\mathbf{w}^{(t)} - \mathbf{x})^T\mathbf{w}^* = (\mathbf{w}^{(t)})^T\mathbf{w}^* - \mathbf{x}^T\mathbf{w}^*$$

Since $\mathbf{x} \in C_2 \implies |\mathbf{w}^{*T}\mathbf{x}| = -\mathbf{x}^T\mathbf{w}^* \geq 0$ and $|\mathbf{x}^T\mathbf{w}^*| \geq \gamma$

$$\begin{aligned}(\mathbf{w}^{(t+1)})^T\mathbf{w}^* &= (\mathbf{w}^{(t)})^T\mathbf{w}^* + |\mathbf{w}^{*T}\mathbf{x}| \\ (\mathbf{w}^{(t+1)})^T\mathbf{w}^* &\geq (\mathbf{w}^{(t)})^T\mathbf{w}^* + \gamma \end{aligned} \tag{2.10}$$

Thus, we have proved (2.8) under **State 2**.

From (2.9) and (2.10), we have shown that $\forall \mathbf{x} \in D$ that inequality (2.8) holds.

Assume that the inequality (2.8) holds for an arbitrary integer value m and $m-1$

$$(\mathbf{w}^{(m)})^T\mathbf{w}^* \geq (\mathbf{w}^{(m-1)})^T\mathbf{w}^* + \gamma \tag{2.11}$$

$$(\mathbf{w}^{(m-1)})^T\mathbf{w}^* \geq (\mathbf{w}^{(m-2)})^T\mathbf{w}^* + \gamma \tag{2.12}$$

Merging inequality (2.11) and (2.12) gives us

$$(\mathbf{w}^{(m)})^T\mathbf{w}^* \geq (\mathbf{w}^{(m-2)})^T\mathbf{w}^* + 2\gamma \tag{2.13}$$

Hence, by induction, after M mistakes, inequality (2.13) becomes

$$(\mathbf{w}^{(m)})^T\mathbf{w}^* \geq (\mathbf{w}^{(0)})^T\mathbf{w}^* + m\gamma \tag{2.14}$$

Since $\mathbf{w}^{(0)} = \mathbf{0}$ (the zero vector), we have

$$(\mathbf{w}^{(m)})^T\mathbf{w}^* \geq m\gamma \tag{2.15}$$

Next, we show that

$$||\mathbf{w}^{(t+1)}||_2^2 \leq ||\mathbf{w}^{(t)}||_2^2 + 1 \tag{2.16}$$

State 1

Consider the same state 1 as before, thus we have

$$(\mathbf{w}^{(t)})^T\mathbf{x} \leq 0 \quad \text{and} \quad \mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \mathbf{x} \tag{2.17}$$

then

$$\begin{aligned} ||\mathbf{w}^{(t+1)}||_2^2 &= (\mathbf{w}^{(t)} + \mathbf{x})^T(\mathbf{w}^{(t)} + \mathbf{x}) \\ &= (\mathbf{w}^{(t)} + \mathbf{x})^T\mathbf{w}^{(t)} + (\mathbf{w}^{(t)} + \mathbf{x})^T\mathbf{x} \\ &= (\mathbf{w}^{(t)})^T\mathbf{w}^{(t)} + \mathbf{x}^T\mathbf{w}^{(t)} + (\mathbf{w}^{(t)})^T\mathbf{x} + \mathbf{x}^T\mathbf{x} \\ &= (\mathbf{w}^{(t)})^T\mathbf{w}^{(t)} + 2(\mathbf{w}^{(t)})^T\mathbf{x} + \mathbf{x}^T\mathbf{x} \end{aligned} \tag{2.18}$$

Since (2.17) and $\mathbf{x}^T\mathbf{x} = 1$, equation (2.18) becomes

$$||\mathbf{w}^{(t+1)}||_2^2 \leq ||\mathbf{w}^{(t)}||_2^2 + 1 \tag{2.19}$$

State 2

Consider the same state 2 as before, thus we have

$$(\mathbf{w}^{(t)})^T\mathbf{x} > 0 \quad \text{and} \quad \mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \mathbf{x} \tag{2.20}$$

then

$$\begin{aligned} ||\mathbf{w}^{(t+1)}||_2^2 &= (\mathbf{w}^{(t)} - \mathbf{x})^T(\mathbf{w}^{(t)} - \mathbf{x}) \\ &= (\mathbf{w}^{(t)} - \mathbf{x})^T\mathbf{w}^{(t)} - (\mathbf{w}^{(t)} - \mathbf{x})^T\mathbf{x} \\ &= (\mathbf{w}^{(t)})^T\mathbf{w}^{(t)} - \mathbf{x}^T\mathbf{w}^{(t)} - (\mathbf{w}^{(t)})^T\mathbf{x} + \mathbf{x}^T\mathbf{x} \\ &= (\mathbf{w}^{(t)})^T\mathbf{w}^{(t)} - 2(\mathbf{w}^{(t)})^T\mathbf{x} + \mathbf{x}^T\mathbf{x} \end{aligned} \tag{2.21}$$

Since (2.20) and $\mathbf{x}^T\mathbf{x} = 1$, equation (2.21) becomes

$$||\mathbf{w}^{(t+1)}||_2^2 \leq ||\mathbf{w}^{(t)}||_2^2 + 1 \tag{2.22}$$

Thus, we have proved (2.16) under **State 2**.
By the same induction defined in (2.14), we have

$$||\mathbf{w}^{(m)}||_2^2 \leq m \tag{2.23}$$

Finally, we merge result (2.15) and (2.23) using the Cauchy-Schwarz (CS) inequality
Recall that the CS inequality is given by

$$|\mathbf{w}^{*T}\mathbf{x}| \leq ||\mathbf{w}^*||_2||\mathbf{x}||_2 \tag{2.24}$$

Hence, using (2.24) we get

$$|(\mathbf{w}^m)^T\mathbf{w}^*| \leq ||(\mathbf{w}^m)^T||_2||\mathbf{w}^*||_2 \tag{2.25}$$

From the result (2.15), (2.25) becomes

$$m\gamma \leq |(\mathbf{w}^m)^T\mathbf{w}^*| \leq ||(\mathbf{w}^m)^T||_2||\mathbf{w}^*||_2 \tag{2.26}$$

Using the result (2.23) and the fact that $||\mathbf{w}^*||_2 = 1$, we get

$$m\gamma \leq ||(\mathbf{w}^m)^T||_2$$

$$m^2\gamma^2 \leq ||(\mathbf{w}^m)^T||_2^2 \tag{2.27}$$

$$m^2\gamma^2 \leq m \tag{2.28}$$

$$m \leq \frac{1}{\gamma^2} \tag{2.29}$$

This proves the Perceptron Convergence Theorem that the number of mistakes is at most $\frac{1}{\gamma^2}$, where γ is the margin.

Even though the perceptron converges, alone it is not always a good model to solve binary classification problem. It has certain limitation by nature of its definition. Consider the logic functions AND, OR and XOR across these 4 points:

$$\mathbf{p}_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \mathbf{p}_2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \mathbf{p}_3 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \mathbf{p}_4 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{2.30}$$

AND Function

The AND function returns a value of 1 if both of the inputs is 1. Let f be the AND function such that

$$f(\mathbf{p}_1) = 0 \quad f(\mathbf{p}_2) = 0 \quad f(\mathbf{p}_3) = 1 \quad f(\mathbf{p}_4) = 0$$

Consider the AND function with the separating line below
The equation of the separating line in figure 2.5 is given by

$$x_1 + x_2 - 1.5 = 0 \tag{2.31}$$

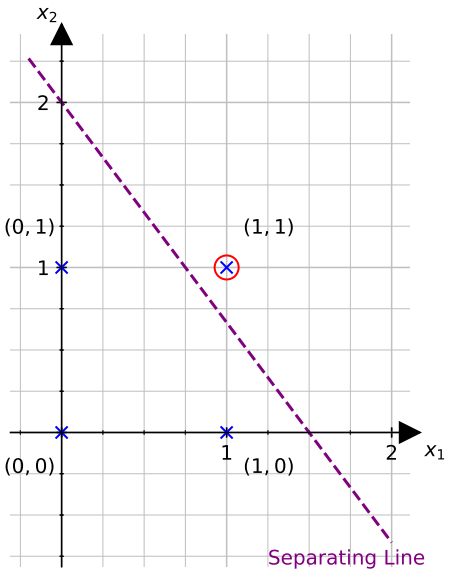

Figure 2.5: AND Function with separating line

We can transform the separating line in the form of $\mathbf{w}^T\mathbf{x} = 0$ to fit the perceptron. Let

$$\mathbf{w} = \begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix} \quad \text{and} \quad \mathbf{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix} \tag{2.32}$$

Applying function (2.1) on each point to check whether they are correctly classified.

$$\phi(p_1) = \phi\left(\begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}\right) = \phi(-1.5) = 0$$

$$\phi(p_2) = \phi\left(\begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}\right) = \phi(-0.5) = 0$$

$$\phi(p_3) = \phi\left(\begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}\right) = \phi(0.5) = 1$$

$$\phi(p_4) = \phi\left(\begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}\right) = \phi(-0.5) = 0$$

Thus, the perceptron is able to solve the AND function.

OR Function

The OR function returns a value of 1 if either of the inputs is 1. Let f be the OR function such that

$$f(\mathbf{p}_1) = 0 \quad f(\mathbf{p}_2) = 1 \quad f(\mathbf{p}_3) = 1 \quad f(\mathbf{p}_4) = 1$$

Consider the OR function with the separating line below

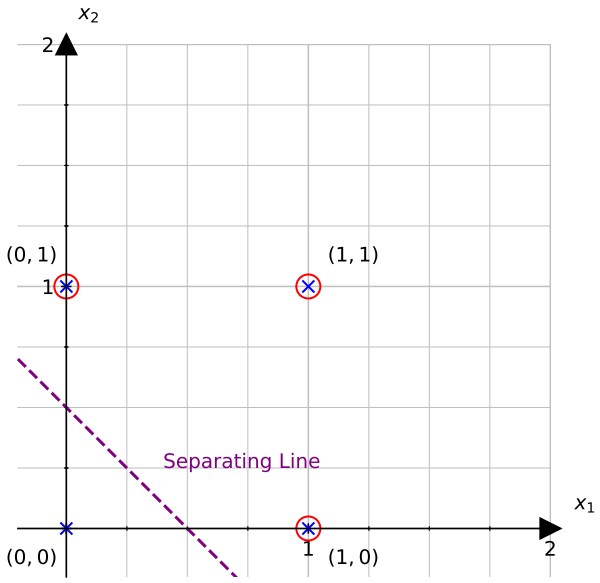

Figure 2.6: OR Function with separating line

The equation of the separating line in figure 2.6 is given by

$$x_1 + x_2 - 0.5 = 0 \tag{2.33}$$

We can transform the separating line in the form of $\mathbf{w}^T\mathbf{x} = 0$ to fit the perceptron. Let

$$\mathbf{w} = \begin{bmatrix} -0.5 \\ 1 \\ 1 \end{bmatrix} \quad \text{and} \quad \mathbf{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix} \tag{2.34}$$

Applying function (2.1) on each point to check whether they are correctly classified.

$$\phi(p_1) = \phi\left(\begin{bmatrix} -0.5 \\ 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}\right) = \phi(-0.5) = 0$$

$$\phi(p_2) = \phi\left(\begin{bmatrix} -0.5 \\ 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}\right) = \phi(0.5) = 1$$

$$\phi(p_3) = \phi\left(\begin{bmatrix} -0.5 \\ 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}\right) = \phi(1.5) = 1$$

$$\phi(p_4) = \phi\left(\begin{bmatrix} -0.5 \\ 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}\right) = \phi(0.5) = 1$$

Thus, the perceptron is able to solve the OR function.

XOR Function

The XOR function returns a value of 1 if either of the inputs is 1 but not both. Let f be the XOR function such that

$$f(\mathbf{p}_1) = 0, \quad f(\mathbf{p}_2) = 1, \quad f(\mathbf{p}_3) = 0, \quad f(\mathbf{p}_4) = 1 \tag{2.35}$$

Consider a plot of the XOR function below

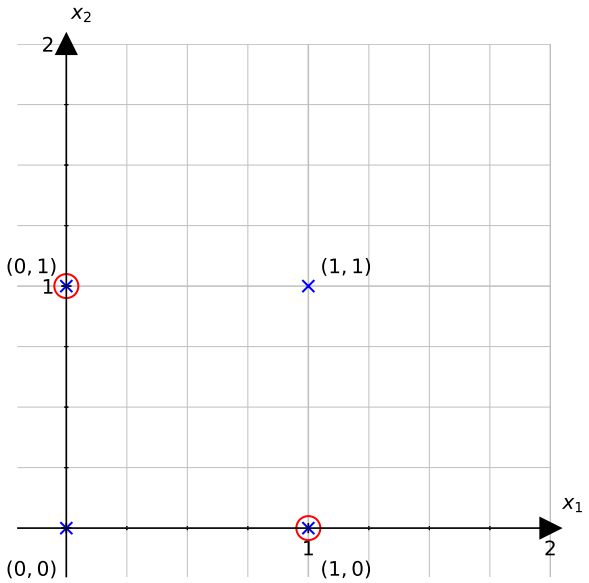

Figure 2.7: XOR function

From figure 2.7; we can observe that no separating line alone could group the two different classes together. We attempt to find a combination of weight for the perceptron which reduces the error (margin) as much as possible. Let f^* be the exact function and $f(\mathbf{x}, \mathbf{w}) = \mathbf{w}^T\mathbf{x} + b$, where $\mathbf{w} = [w_1 \ w_2]^T$, be the approximate function. Consider the error function (the difference between the exact and approximate value)

$$\mathbb{E}(\mathbf{w}, b) = \frac{1}{4}\sum_{x \in D}(f^*(\mathbf{x}) - \mathbf{w}^T\mathbf{x} - b)^2 \tag{2.36}$$

we minimise the error function $\mathbb{E}$ by setting the partial derivatives to 0. The partial derivative with respect to the bias is given by

$$\frac{\partial \mathbb{E}}{\partial b} = -\frac{1}{2}\sum_{x \in D}(f^*(\mathbf{x}) - \mathbf{w}^T\mathbf{x} - b)$$

$$\frac{\partial \mathbb{E}}{\partial b} = 0 \rightarrow \sum_{x \in D} f^*(\mathbf{x}) = \sum_{x \in D} \mathbf{w}^T\mathbf{x} - \sum_{x \in D} b \tag{2.37}$$

The partial derivative with respect to the weight vector is given by

$$\begin{aligned}\frac{\partial \mathbb{E}}{\partial \mathbf{x}} &= \frac{1}{2}\sum_{x \in D}(f^*(\mathbf{x}) - \mathbf{w}^T\mathbf{x} - b)\frac{\partial(-\mathbf{w}^T\mathbf{x})}{\partial \mathbf{w}} \\ &= -\frac{1}{2}\sum_{x \in D}(f^*(\mathbf{x}) - \mathbf{w}^T\mathbf{x} - b)\mathbf{x}\end{aligned}$$

$$\frac{\partial \mathbb{E}}{\partial \mathbf{w}} = 0 \rightarrow \sum_{x \in D} \mathbf{x} f^*(\mathbf{x}) = \sum_{x \in D}(\mathbf{w}^T\mathbf{x})\mathbf{x} + b\sum_{x \in D}\mathbf{x} \tag{2.38}$$

Substituting for values in equation (2.37)

$$\begin{aligned}\sum_{x \in D} f^*(\mathbf{x}) &= f^*(\mathbf{p}_1) + f^*(\mathbf{p}_2) + f^*(\mathbf{p}_3) + f^*(\mathbf{p}_4) \\ &= 0 + 1 + 1 + 0 \\ &= 2 \qquad (2.39)\end{aligned}$$

$$\begin{aligned}\sum_{x \in D} \mathbf{w}^T\mathbf{x} &= \mathbf{w}^T\begin{bmatrix}0\\0\end{bmatrix} + \mathbf{w}^T\begin{bmatrix}1\\0\end{bmatrix} + \mathbf{w}^T\begin{bmatrix}1\\1\end{bmatrix} + \mathbf{w}^T\begin{bmatrix}0\\1\end{bmatrix} \\ &= 2w_1 + 2w_2 \qquad (2.40)\end{aligned}$$

$$\begin{aligned}\sum_{x \in D} b &= b\sum_{x \in D} 1 \\ &= 4b \qquad (2.41)\end{aligned}$$

Thus, we get

$$2 = 2w_1 + 2w_2 + 4b$$

$$w_1 + w_2 + 2b = 1 \tag{2.42}$$

Substituting for values in equation (2.38)

$$\sum_{x \in D} \mathbf{x} f^*(\mathbf{x}) = \begin{bmatrix} 0 \\ 0 \end{bmatrix} (0) + \begin{bmatrix} 1 \\ 0 \end{bmatrix} (1) + \begin{bmatrix} 1 \\ 1 \end{bmatrix} (0) + \begin{bmatrix} 0 \\ 1 \end{bmatrix} (1)$$

$$= \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{2.43}$$

$$\sum_{x \in D} (\mathbf{w}^T \mathbf{x}) \mathbf{x} = \begin{bmatrix} w_1 & w_2 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} w_1 & w_2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$+ \begin{bmatrix} w_1 & w_2 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} w_1 & w_2 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} 2w_1 + w_2 \\ w_1 + 2w_2 \end{bmatrix} \tag{2.44}$$

$$b \sum_{x \in D} \mathbf{x} = b \begin{bmatrix} 0 \\ 0 \end{bmatrix} + b \begin{bmatrix} 1 \\ 0 \end{bmatrix} + b \begin{bmatrix} 1 \\ 1 \end{bmatrix} + b \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{2.45}$$

$$= b \begin{bmatrix} 2 \\ 2 \end{bmatrix} \tag{2.46}$$

Thus, we get

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2w_1 + w_2 \\ w_1 + 2w_2 \end{bmatrix} + b \begin{bmatrix} 2 \\ 2 \end{bmatrix} \tag{2.47}$$

From equations (2.42) and (2.47), we get the set of equations that forms the approximate function with the minimum error

$$w_1 + w_2 + 2b = 1$$

$$2w_1 + w_2 + 2b = 1$$

$$w_1 + 2w_2 + 2b = 1$$

Solving the system of linear equations we get

$$w_1 = 0 \quad w_2 = 0 \quad b = \frac{1}{2} \tag{2.48}$$

Hence, the approximate function $f(\mathbf{x}, \mathbf{w})$ is given by

$$f(\mathbf{x}, \mathbf{w}) = \frac{1}{2} \tag{2.49}$$

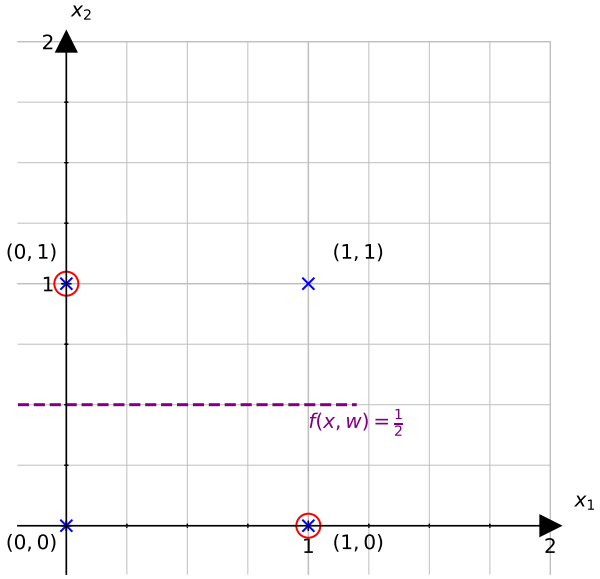

Figure 2.8: XOR approximate solution

Clearly from figure 2.8, the perceptron model cannot solve the XOR logic function. The XOR problem is actually not linearly separable. Thus, we introduce **feedforward networks** to overcome this problem.

2.2 Feedforward Network

We have seen that the single perceptron cannot solve the XOR problem. In order to solve the problem, we use a set of perceptrons that learns two different functions within a connected network.

First, let's consider two NOT boolean function $f_1^* = x_1$ AND (NOT x_2) and $f_2^* =$ (NOT x_1) AND x_2.

Consider f_1^* which returns 1 if x_1 and (not x_2) are equal to 1.

x_1	x_2	NOT (x_2)	x_1 AND (NOT x_2)
0	0	1	0
1	0	1	1
1	1	0	0
0	1	0	0

Table 2.1: Truth Table for f_1^*

The function f_1^* can be separated using the perceptron.

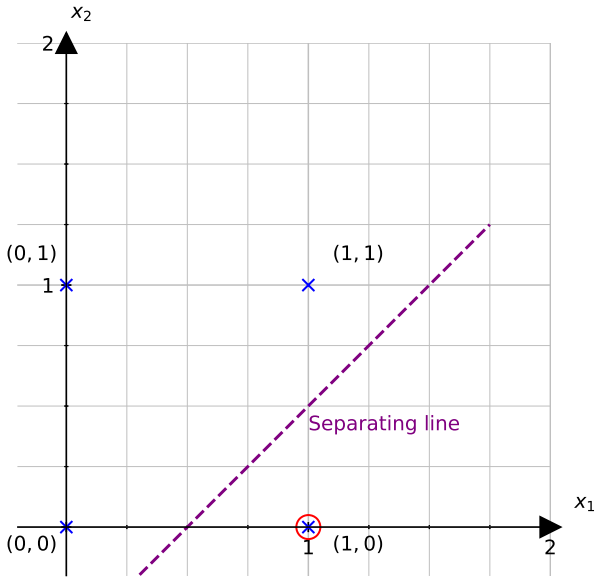

Figure 2.9: Representation of Truth Table for function f_1^*

The equation of the separating line in figure 2.9 is given by

$$x_1 - x_2 - \frac{1}{2} = 0 \tag{2.50}$$

We can transform the separating line in the form of $\mathbf{w}^T\mathbf{x} = 0$ to fit the perceptron.

$$\mathbf{w} = \begin{bmatrix} -0.5 \\ 1 \\ -1 \end{bmatrix} \quad \text{and} \quad \mathbf{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix} \tag{2.51}$$

Similarly, consider f_2^* which returns 1 if (not x_1) and x_2 are equal to 1.

x_1	x_2	NOT (x_1)	(NOT x_1) AND x_2
0	0	1	0
1	0	0	0
1	1	0	0
0	1	1	1

Table 2.2: Truth Table for f_2^*

The function f_2^* can be separated using the perceptron.

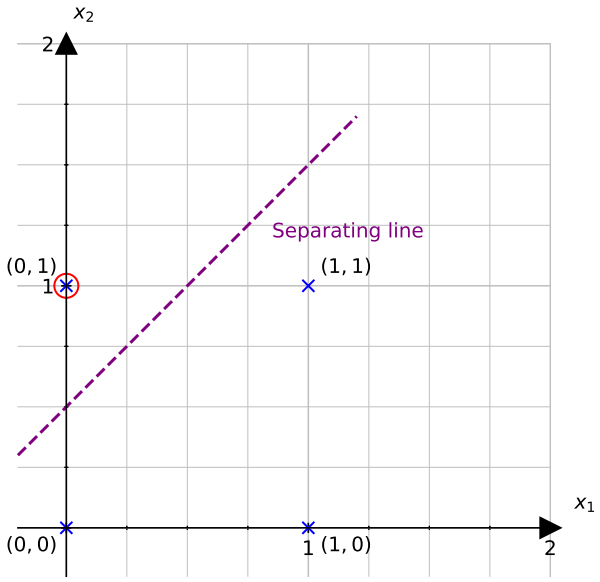

Figure 2.10: Representation of Truth Table for function f_2^*

The equation of the separating line in figure 2.10 is given by

$$-x_1 + x_2 - \frac{1}{2} = 0 \tag{2.52}$$

We can transform the separating line in the form of $\mathbf{w}^T\mathbf{x} = 0$ to fit the perceptron.

$$\mathbf{w} = \begin{bmatrix} -0.5 \\ -1 \\ 1 \end{bmatrix} \quad \text{and} \quad \mathbf{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix} \tag{2.53}$$

Consequently, the XOR function defined in (2.35) can be expressed as f^* such that

$$f^*(x_1, x_2) = f_1^*(x_1, x_2) \text{ OR } f_2^*(x_1, x_2)$$

x_1	x_2	$f_1^*(x_1, x_2)$	$f_2^*(x_1, x_2)$	$f_2^*(x_1, x_2)$
0	0	0	0	0
1	0	1	0	1
1	1	0	0	0
0	1	0	1	1

Table 2.3: Truth Table for f^*

Consider a feedforward network with a hidden layer with two units h_1 and h_2 such that h_1 learns $f_1^*(x_1, x_2)$ and h_2 learns $f_2^*(x_1, x_2)$.

$$\begin{aligned} h_1 &= f_1^*(x_1, x_2) \\ &= \phi(z_1) \end{aligned}$$

where $z_1 = w_{11}x_1 + w_{12}x_2 + b_1$ and $w_{11} = 1$, $w_{12} = -1$ and $b_1 = -0.5$ from separating line in (2.51).

$$\begin{aligned} h_2 &= f_2^*(x_1, x_2) \\ &= \phi(z_2) \end{aligned}$$

where $z_2 = w_{21}x_1 + w_{22}x_2 + b_1$ and $w_{21} = -1$, $w_{22} = 1$ and $b_2 = -0.5$ from separating line in (2.53).

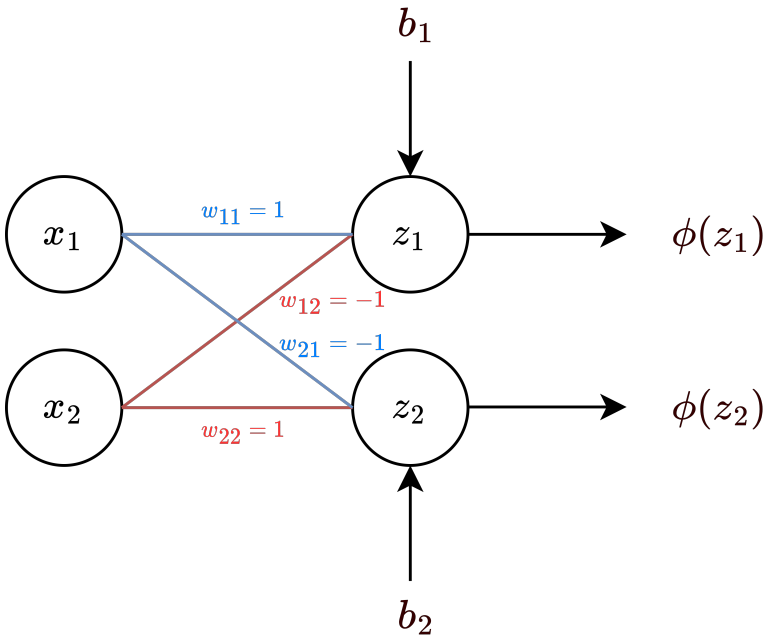

Figure 2.11: FFN hidden layer

Thus we obtain the system below for the hidden layer.

$$\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} b_1 & w_{11} & w_{12} \\ b_1 & w_{21} & w_{22} \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix} \tag{2.54}$$

We then feed forward the information from the hidden layer to another layer.

$$\begin{bmatrix} h_1 \\ h_2 \end{bmatrix} = \phi \left(\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \right) \tag{2.55}$$

At the hidden layer, we are left with the following truth table.

x_1	x_2	XOR	
0	0	0	
1	0	1	
0	0	0	redundant
0	1	1	

Table 2.4: Truth Table from hidden layer

The XOR problem can not be separated from using another perceptron after the hidden layer.

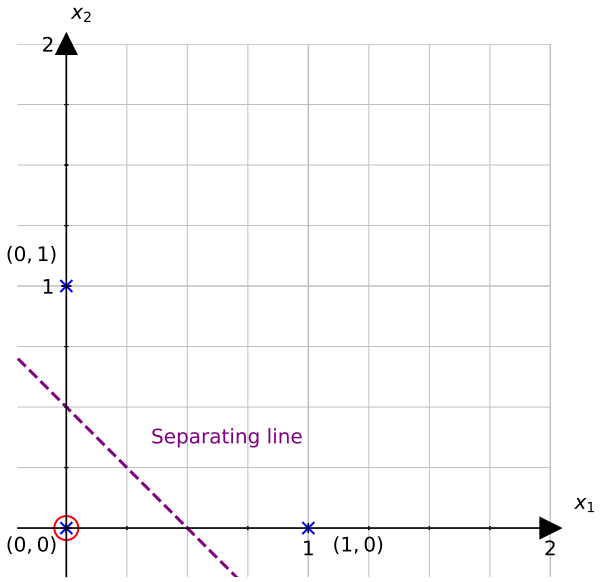

Figure 2.12: Representation of the hidden layer

The equation of the separation line is in figure (2.12) is given binary

$$h_1 + h_2 - \frac{1}{2} = 0 \tag{2.56}$$

We can now transform the separating line in the form of $\mathbf{w}^T\mathbf{x} = 0$ to fit the perceptron.

$$\mathbf{w} = \begin{bmatrix} -0.5 \\ 1 \\ 1 \end{bmatrix} \quad \mathbf{h} = \begin{bmatrix} 1 \\ h_1 \\ h_2 \end{bmatrix} \tag{2.57}$$

Finally, the XOR function can be represented using the set of transform defined below.

$$f(x_1, x_2) = \phi' \left(\mathbf{w}_2^T \left(\phi \left(\mathbf{w}_1^T \mathbf{x} \right) \right) \right) \tag{2.58}$$

where

$$\mathbf{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix} \quad \mathbf{w}_1 = \begin{bmatrix} -0.5 & -0.5 \\ 1 & -1 \\ -1 & 1 \end{bmatrix} \quad \mathbf{w}_2 = \begin{bmatrix} -0.5 \\ 1 \\ 1 \end{bmatrix} \tag{2.59}$$

ϕ^* takes an augmented input from the calculation of $\phi(\mathbf{w}_1^T\mathbf{x})$ to accommodate for the bias.

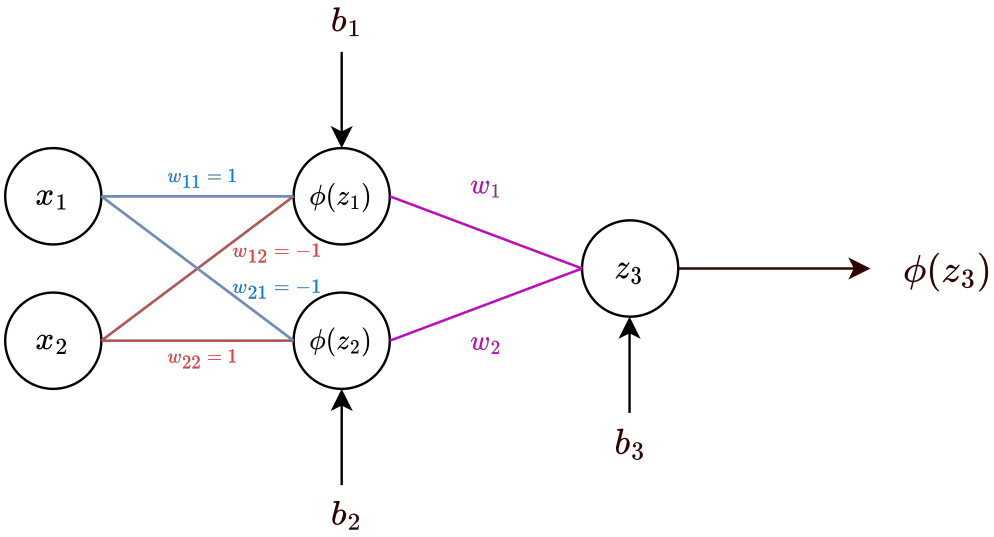

Figure 2.13: Feedforward network for XOR problem

Checking for $\mathbf{x} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^T$

$$\mathbf{w}_1^T\mathbf{x} = \begin{bmatrix} -0.5 & 1 & -1 \\ -0.5 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} -0.5 \\ -0.5 \end{bmatrix}$$

$$\phi(\mathbf{w}_1^T\mathbf{x}) = \phi\left(\begin{bmatrix} -0.5 \\ -0.5 \end{bmatrix}\right) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\phi^*\left(\mathbf{w}_2^T\left(\phi\left(\mathbf{w}_1^T\mathbf{x}\right)\right)\right) = \phi^*\left(\begin{bmatrix} -0.5 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}\right) = \phi^*(-0.5)$$

Thus, for $\mathbf{p}_1 = \begin{bmatrix}0, 0\end{bmatrix}^T$

$$f(x_1, x_2) = \phi^*(-0.5) = 0$$

Checking for $\mathbf{x} = \begin{bmatrix}1 & 0 & 1\end{bmatrix}^T$

$$\mathbf{w}_1^T\mathbf{x} = \begin{bmatrix}-0.5 & 1 & -1\\ -0.5 & -1 & 1\end{bmatrix}\begin{bmatrix}1\\0\\1\end{bmatrix} = \begin{bmatrix}-1.5\\0.5\end{bmatrix}$$

$$\phi(\mathbf{w}_1^T\mathbf{x}) = \phi\left(\begin{bmatrix}-1.5\\0.5\end{bmatrix}\right) = \begin{bmatrix}0\\1\end{bmatrix}$$

$$\phi^*\left(\mathbf{w}_2^T\left(\phi\left(\mathbf{w}_1^T\mathbf{x}\right)\right)\right) = \phi^*\left(\begin{bmatrix}-0.5 & 1 & 1\end{bmatrix}\begin{bmatrix}1\\0\\1\end{bmatrix}\right) = \phi^*(0.5)$$

Thus, for $\mathbf{p}_4 = \begin{bmatrix}0, 0\end{bmatrix}^T$

$$f(x_1, x_2) = \phi^*(0.5) = 1$$

2.3 Activation Function

The hard-delimiters used in our previous feedforward network are linear. However, different activation functions can use with a perceptron to introduce non-linearity in our network. The non-linearity enables a wider range of problems to be tackled. The function to be used is subjective to the problem being solved and the form of the desired result we want. Some common activation functions are defined below.

Linear

$$\phi(z) = z \tag{2.60}$$

Unit Step (Heaviside Function)

$$\phi(z) = \begin{cases} 0 & z < 0 \\ 0.5 & z = 0 \\ 1 & z > 0 \end{cases} \tag{2.61}$$

Signum

$$\phi(z) = \begin{cases} -1 & z < 0 \\ 0 & z = 0 \\ 1 & z > 0 \end{cases} \tag{2.62}$$

Sigmoid

$$\phi(z) = \frac{1}{1 + e^{-z}} \tag{2.63}$$

Hyperbolic Tangent(tanh)

$$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \tag{2.64}$$

ReLU

$$\phi(z) = \begin{cases} 0 & z < 0 \\ z & z > 0 \end{cases} \tag{2.65}$$

2.4 Backpropagation

We have seen how information flows in a feedforward neural network. Backpropagation algorithm is the process of spreading the error at the output layer throughout the other layers in the network. The goal is to understand how does the error changes with respect to the weights and biases. Consider the neural network below with N number of layers (including the input and output layer).

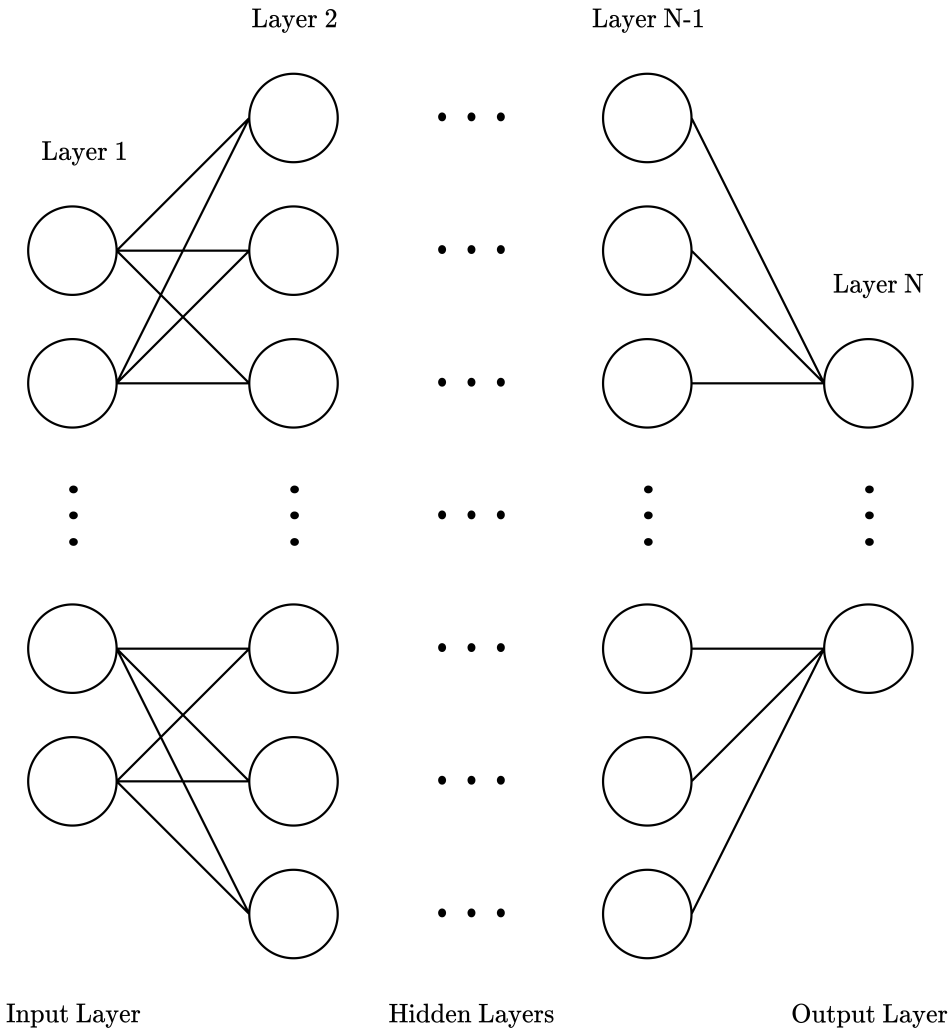

Figure 2.14: Neural Network with N number of layers

The annotation between two consecutive layers, l and $l+1$, in the network is given by the following set of indexed values.

$b_j^{l+1} \rightarrow$ Bias input at the j^{th} neuron in the $l+1^{th}$ layer

$w_{jk}^{l+1} \rightarrow$ Weight between the k^{th} and j^{th} neuron in the l^{th} and $(l+1)^{th}$ layer respectively

$z_j^{l+1} \rightarrow$ Input of the activation function at the j^{th} neuron in the $(l+1)^{th}$ layer

$a_j^l \rightarrow$ Output from the activation function at the j^{th} neuron in the l^{th} layer

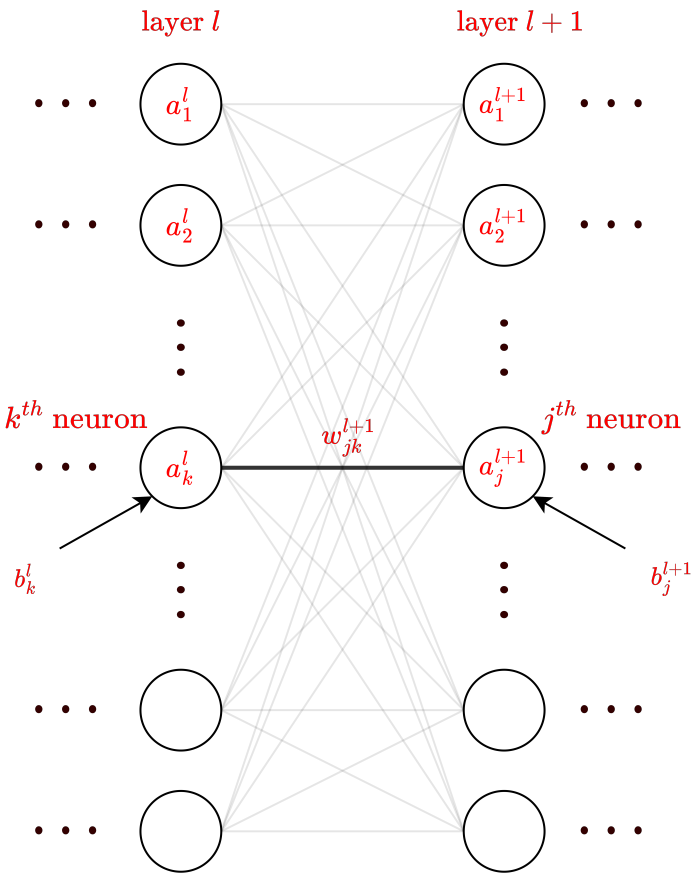

Figure 2.15: Consecutive layers, l and $l+1$, in a neural network

Let the activation function be denoted as σ such that

$$a_j^{(l+1)} = \sigma\left(z_j^{(l+1)}\right) \tag{2.66}$$

$$z_j^{(l+1)} = \sum_k w_{jk}^{(l+1)} a_k^{(l)} + b_j^{(l+1)} \tag{2.67}$$

We start at the end of the network by calculating the value of the cost/error function. Let C be the cost function which takes as input the output layer

$$\begin{aligned} C &= C\left(a_j^L\right) \\ &= \frac{1}{2}\sum_j (y_j - a_j^L)^2 \end{aligned} \tag{2.68}$$

where y_j is the desired exact value that we want the network to predict. Also, let the change in the cost function with respect to the input of the activation function be given by δ_j^L where

$$\delta_j^L = \frac{\partial C}{\partial z_j^L}$$

Using result (2.66) and the definition (2.68)

$$\begin{aligned} \delta_j^L &= \frac{\partial C(a_j^L)}{\partial z_j^L} \\ &= \frac{\partial C(\sigma(z_j^L))}{\partial z_j^L} \end{aligned}$$

Applying the chain rule

$$\begin{aligned} \delta_j^L &= \frac{\partial C(\sigma(z_j^L))}{\partial z_j^L} \\ &= \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} \end{aligned}$$

From equation (2.66)

$$\begin{aligned} a_j^{(L)} &= \sigma(z_j^{(L)}) \\ \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} &= \sigma'(z_j^{(L)}) \end{aligned}$$

Thus, the change in the cost function with respect to the input of the activation

function in the last layer is given by

$$\delta_j^{(L)} = \frac{\partial C}{\partial a_j^{(L)}} \sigma'(z_j^{(L)}) \tag{2.69}$$

Next, we find how to back-propagate the changes from an upper layer $l+1$ to the previous layer l. Consider the change in the cost function based at the l^{th} layer

$$\delta_j^{(l)} = \frac{\partial C}{\partial z_j^{(l)}} \tag{2.70}$$

From figure (2.15); an element in the l^{th} layer influences all the neurons in the $(l+1)^{th}$ layer. Thus, when applying the chain rule we sum over all the neurons which gives

$$\delta_j^l = \sum_k \frac{\partial C}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} \tag{2.71}$$

From equations (2.66) and (2.67)

$$\begin{aligned} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} &= \frac{\partial}{\partial z_j^{(l)}} \Big(\sum_s w_{ks}^{(l+1)} \sigma(z_s^{(l)}) + b_k^{(l+1)} \Big) \\ &= \sum_s w_{ks}^{(l+1)} \frac{\partial \sigma(z_s^{(l)})}{\partial z_j^{(l)}} \\ &= \sum_s w_{ks}^{(l+1)} \sigma'(z_s^{(l)}) \frac{\partial z_s^{(l)}}{\partial z_j^{(l)}} \end{aligned}$$

The partial derivative only exist and is 1 when $s = j$; else the value is 0.

$$\frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = w_{kj}^{(l+1)} \sigma'(z_j^{(l)}) \tag{2.72}$$

Substituting (2.72) and (2.70) into (2.71)

$$\delta_j^l = \sum_k \delta_k^{(l+1)} w_{kj}^{(l+1)} \sigma'(z_j^{(l)}) \tag{2.73}$$

Equation (2.73) are the changes in l^{th} layer from the changes that happened in

the $(l+1)^{th}$ layer. Since we do not explicitly adjust the output and input of the activation functions, we derive the changes of the cost function with respect to the bias and weight. Consider the effect of the bias at each layer given by

$$\frac{\partial C}{\partial b_j^{(l)}} = \frac{\partial C}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}}$$

Using equation (2.67)

$$\begin{aligned} \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} &= \frac{\partial}{\partial b_j^{(l)}} \Big(\sum_k w_{jk}^{(l+1)} a_k^{(l)} + b_j^{(l+1)} \Big) \\ &= 1 \end{aligned}$$

Thus, the change of in the cost function with respect to the bias at any layer is given by

$$\frac{\partial C}{\partial b_j^{(l)}} = \delta_i^{(j)} \tag{2.74}$$

Finally, consider the rate of change of the cost function with respect to any weight in the network given by

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial z_m^{(l)}} \frac{\partial z_m^{(l)}}{\partial w_{jk}^{(l)}} \tag{2.75}$$

Using equation (2.67)

$$\begin{aligned} \frac{\partial z_m^{(l)}}{\partial w_{jk}^{(l)}} &= \frac{\partial}{\partial w_{jk}^{(l)}} \Big(\sum_s w_{ms}^{(l)} a_s^{(l-1)} + b_m^{(l)} \Big) \\ &= \sum_s a_s^{(l-1)} \frac{\partial w_{ms}^{(l)}}{\partial w_{jk}^{(l)}} \end{aligned}$$

The partial derivative only exist and is 1 when $s = k$ and $m = j$; else the value is 0.

$$\frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} = a_k^{(l-1)} \tag{2.76}$$

Substituting (2.72) and (2.76) into (2.75)

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} a_k^{(l-1)} \tag{2.77}$$

The four main back-propagation equations are given by

$$\delta_j^{(L)} = \frac{\partial C}{\partial a_j^{(L)}} \sigma^{'}(z_j^{(L)}) \tag{2.78}$$

$$\delta_j^{l} = \sum_k \delta_k^{(l+1)} w_{kj}^{(l+1)} \sigma^{'}(z_j^{(l)}) \tag{2.79}$$

$$\frac{\partial C}{\partial b_j^{(l)}} = \delta_i^{(j)} \tag{2.80}$$

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} a_k^{(l-1)} \tag{2.81}$$

We vectorize the above set of equations by using the Hadamard product. Given two matrices of the A and B of the same dimension, $m \times n$, the Hadamard product is given as

$$(A \odot B)_{mn} = (A)_{mn}(B)_{mn}$$

The vectorized form of the back-propagation equations are given by

$$\delta^{(L)} = \nabla_{a^{(L)}} C \odot \sigma^{'}(z^{(L)}) \tag{2.82}$$

$$\delta^{(l)} = (w^{(l+1)})^T \sigma^{(l+1)} \odot \sigma^{'}(z^{(l)}) \tag{2.83}$$

$$\nabla_{b^{(l)}} C = \delta^{(l)} \tag{2.84}$$

$$\nabla_{w^{(l)}} C = \delta^{(l)} a^{(l-1)} \tag{2.85}$$

Algorithm 2 Back-Propagation Algorithm

Start
Step 1 (Input) Compute the first activation layer $a^{(1)}$ using input values x
    $a^{(1)} = \sigma(w^{(1)}x + b^{(1)})$
Step 2 (Feed-Forward)
    Feed forward the input to the next layers
for $l = 2, 3, \ldots, L$ **do**
    $z^{(l)} = w^{(l)}a^{(l-1)} + b^{(l)}$
    $a^{(l)} = \sigma(z^{(l)})$
end for
Step 3 (Cost/Error Calculation)
    Compute the cost function, C, at the output layer L
Step 4 (Backpropagate)
    Compute the rate of change of the cost function $w.r.t$ to $z^{(L)}$
    $\delta^{(L)} = \nabla_{a^{(L)}}C \odot \sigma'(z^{(L)})$
for $l = L-1, L-2, \ldots, 2$ **do**
    $\delta^{(l)} = (w^{(l+1)})^T\sigma^{(l+1)} \odot \sigma'(z^{(l)})$
    $\nabla_{b^{(l)}}C = \delta^{(l)}$
    $\nabla_{w^{(l)}}C = \delta^{(l)}a^{(l-1)}$
end for

2.5 Sequential Neural Networks

Recurrent Neural Network(RNN)

Developed through the work of (Rumelhart et al. 1986), recurrent neural network is a type of neural network which is capable of modelling sequential data for predictions. Similar to the feedforward network; the RNN consist of an input layer and an output layer with a hidden state instead of a hidden layer. The hidden state act as a feedback loop. That is, the output at a particular step in the sequence is fed to the input of the next step to compute the output of the next step. Hence, recurrent neural networks are neural networks with hidden states. Consider the sequential input data $\mathbf{X}_t \in \mathbb{R}^{n\times d}$, with batch size n and d inputs, at a particular time step t, $\mathbf{H}_t \in \mathbb{R}^{n\times h}$ be the hidden layer, $\mathbf{W}_{xh} \in \mathbb{R}^{d\times h}$ be the weight applied on the input data, $\mathbf{W}_{hh} \in \mathbb{R}^{h\times h}$ be the weight applied to the output hidden state, $\mathbf{b}_h \in \mathbb{R}^{1\times h}$ and ϕ be the activation function; thus the calculation at the hidden layer is given by:

$$\mathbf{H}_t = \phi(\mathbf{X}_t\mathbf{W}_{xh} + \mathbf{H}_{t-1}\mathbf{W}_{hh} + \mathbf{b}_h) \tag{2.86}$$

Consider the weights $\mathbf{W}_{hq} \in \mathbb{R}^{d \times h}$ and the bias $\mathbf{b}_h \in \mathbb{R}^{1 \times q}$, thus the computation at the output layer is given by:

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q \tag{2.87}$$

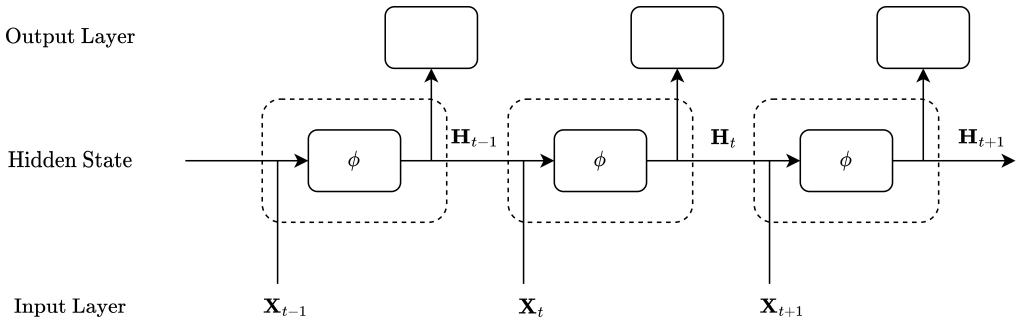

Figure 2.16: An RNN with a hidden state

As we unfold a recurrent neural network for more time steps; the number of parameters does not increase. The input of the activation function gets recursively very large or very small. Consequently, we are limited by the problem of vanishing or exploding gradients when optimizing recurrent neural network (Bengio, Simard & Frasconi 1994) (Kolen & Kremer 2001). A preventive measure would be to truncate the recursive gradient term during backpropagation. RNN also suffer from short-term memory, important information in very long sequence from the beginning are left out.

Long Short Term Memory(LSTM)

Introduced in 1997 by (Hochreiter & Schmidhuber 1997), the long short-term memory model was introduced to solve the vanishing and exploding problem in the RNN. While the architecture is similar to the RNN, the LSTM replaces the hidden state by a memory cell. The memory cell comprises three main "gates" namely and an input node: the forget gate, the input gate, the output gate and the input node respectively. The forget gate controls whether we keep the input from the previous hidden state or not. The input gate determines how much of the input should contribute to the internal cell memory and passed to the input node. At the input

node, the memory cell is calculated using the input value and the value from the input gate. Finally, the output gate determines how much of the memory cell of the current time step impacts the next time step in the next memory cell. Consider the input $\mathbf{X}_t \in \mathbb{R}^{n\times d}$, the hidden state of the previous time step be $\mathbf{H}_t \in \mathbb{R}^{n\times h}$, the input gate be given by $\mathbf{I}_t \in \mathbb{R}^{n\times h}$, the forget gate given by $\mathbf{F}_t \in \mathbb{R}^{n\times h}$, the input node given by $\tilde{\mathbf{C}}_t \in \mathbb{R}^{n\times h}$, the output gate be $\mathbf{O}_t \in \mathbb{R}^{n\times h}$ and the memory cell state be given by $\mathbf{C}_t \in \mathbb{R}^{n\times h}$ then the computations of the LSTM are given by:

$$\mathbf{I}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xi} + \mathbf{H}_{t-1}\mathbf{W}_{hi} + \mathbf{b}_i) \tag{2.88}$$

$$\mathbf{F}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xf} + \mathbf{H}_{t-1}\mathbf{W}_{hf} + \mathbf{b}_f) \tag{2.89}$$

$$\mathbf{O}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xo} + \mathbf{H}_{t-1}\mathbf{W}_{ho} + \mathbf{b}_o) \tag{2.90}$$

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xc} + \mathbf{H}_{t-1}\mathbf{W}_{hc} + \mathbf{b}_c) \tag{2.91}$$

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t \tag{2.92}$$

where $\mathbf{W}_{xi}$, $\mathbf{W}_{xf}$, $\mathbf{W}_{xo}$, $\mathbf{W}_{xc} \in \mathbb{R}^{d\times h}$ and $\mathbf{W}_{hi}$, $\mathbf{W}_{hf}$, $\mathbf{W}_{ho}$, $\mathbf{W}_{hc} \in \mathbb{R}^{h\times h}$ are weight parameters and $\mathbf{b}_i$, $\mathbf{b}_f$, $\mathbf{b}_o$, $\mathbf{b}_c \in \mathbb{R}^{1\times h}$ are the bias parameters.

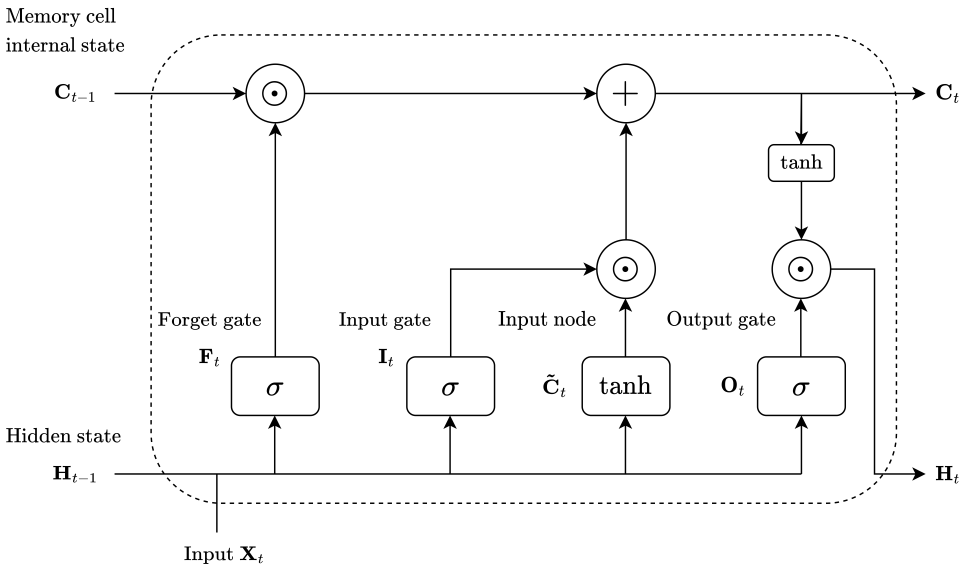

Figure 2.17: Computing the hidden state in an LSTM model

LSTM addresses the vanishing and exploding gradient problem of the RNN by controlling the amount of information being carried to the memory and by providing

a continuous gradient flow to facilitate backpropagation. Many variants of LSTM have been developed over the years due to their dominance for sequence learning. Some LTSM variants are namely: The Peephole Variant, the Coupled Gate Variation, Gated Recurrent Units (GRU). However, they are quite costly to compute due to their long range dependencies of the sequence.

Bidirectional RNN(BRNN)

In order to overcome the uni-direction learning of RNN and LSTM,(Schuster & Paliwal 1997) introduced the bidirectional recurrent neural network. The implementation consists of two unidirectional layers chained together in opposite directions but taking the same inputs. The output is obtained by concatenating the two directional outputs from the hidden layers. The obtain layers is then computed to get the prediction. Consider the input $X_t \in \mathbf{R}^n$ for time step t, the forward and backward hidden unidirectional layers for time step t be given by $\overrightarrow{H_t} \in \mathbb{R}^{n\times h}$ and $\overleftarrow{H_t} \in \mathbb{R}^{n\times h}$ respectively and the output layer be given by $\mathbf{O}_t$, then the computations of the BRNN is given by:

$$\overrightarrow{H_t} = \phi(\mathbf{X}_t\mathbf{W}_{xh}^{(f)} + \overrightarrow{\mathbf{H}}_{t-1}\mathbf{W}_{hh}^{(f)} + \mathbf{b}_h^{(f)}) \tag{2.93}$$

$$\overleftarrow{H_t} = \phi(\mathbf{X}_t\mathbf{W}_{xh}^{(b)} + \overleftarrow{\mathbf{H}}_{t+1}\mathbf{W}_{hh}^{(b)} + \mathbf{b}_h^{(b)}) \tag{2.94}$$

$$\mathbf{O}_t = \mathbf{H}_t\mathbf{W}_{tq} + \mathbf{b}_q$$

where $\mathbf{W}_{xh}^{(f)}$ and $\mathbf{W}_{xh}^{(b)} \in \mathbb{R}^{d\times h}$, $\mathbf{W}_{hh}^{(f)}$ and $\mathbf{W}_{hh}^{(b)} \in \mathbb{R}^{h\times h}$, $\mathbf{W}_{hq} \in \mathbb{R}^{2h\times q}$ are the weight parameters and $\mathbf{b}_h^{(f)}$ and $\mathbf{b}_h^{(b)} \in \mathbb{R}^{1\times h}$ and $\mathbf{b_q} \in \mathbb{R}^{1\times q}$ are the bias parameters.

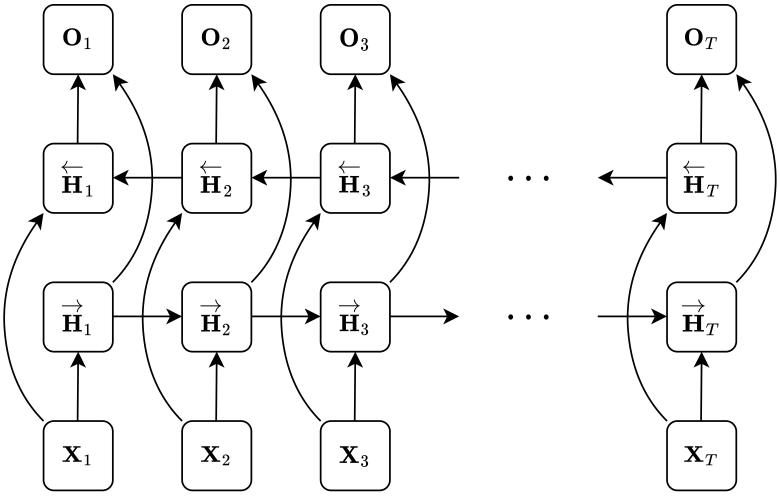

Figure 2.18: Architecture of a bidirectional RNN

Bidirectional recurrent neural network are very efficient in sequential modelling in cases where context is a priority. Natural language processing, speech recognition, handwriting recognition, part-of-speech tagging; leverages BRNN as context input is required.

Chapter 3

Optimization

The optimization problem is a computational problem in which the objective is to find the best of all possible solutions. Deep neural networks is a form of an optimization problem to find the best possible set of weights in order to reduce the error in a network.

A generic form of an optimization problem is given by

$$\begin{aligned} \underset{x}{\text{minimize/maximize}} \quad & f(x) \\ \text{subject to} \quad & g_i(x) \leq 0, \quad i = 1, \ldots, m \\ & h_j(x) = 0, \quad j = 1, \ldots, p \end{aligned} \tag{3.1}$$

where

$f : \mathbb{R}^n \to \mathbb{R}$ is the objective/loss function to be minimised,

$g_i(x) \leq 0$ are called inequality constraints,

$h_j(x) = 0$ are called equality constraints, and

$m \geq 0$ and $p \geq 0$ If $m = p = 0$, the problem is an unconstrained optimization problem.

Minimum/Maximum Point

Let x^* be points of the function $f(x)$ defined in (3.1) and $u \in \mathbb{N}(x^*, \delta)$ where $\mathbb{N}$ is a δ-neighbourhood of x^*. Then, x^* is said to be a

(i) local minimum if $f(x^*) < f(u)\ \forall u \in \mathbb{N}$

(ii) local maximum if $f(x^*) > f(u)\ \forall u \in \mathbb{N}$

(iii) global minimum if $f(x^*) < f(u)\ \forall u \in \mathbb{R}$

(iv) global maximum if $f(x^*) > f(u)\ \forall u \in \mathbb{R}$

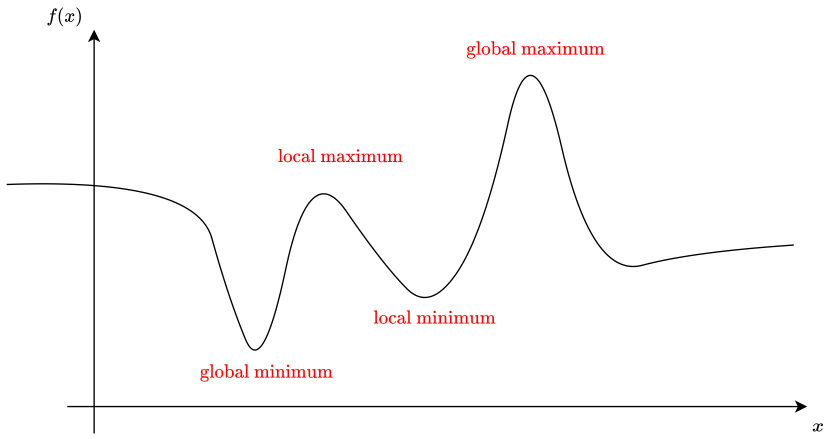

Figure 3.1: Minimum and Maximum points

Determining Minimum/Maximum Point

For a function $f(\mathbf{x})$ where $\mathbf{x} = (x_1, x_2 \ldots, x_n) \in \mathbb{R}^n$, the condition for the presence of a stationary point (minimum or maximum point) is given by

$$\mathbf{G} = \nabla f \left(\frac{\partial f}{\partial x_1} \quad \frac{\partial f}{\partial x_2} \quad \ldots, \frac{\partial f}{\partial x_n} \right) = \mathbf{0} \tag{3.2}$$

The second derivative of $f(x)$ is given by the Hessian matrix, $\mathbf{H}$

$$\mathbf{H} = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \dfrac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_1 \partial x_n} \\ \dfrac{\partial^2 f}{\partial x_2 \partial x_1} & \dfrac{\partial^2 f}{\partial x_2^2} & \cdots & \dfrac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial x_n \partial x_1} & \dfrac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_n^n} \end{bmatrix} \tag{3.3}$$

The nature of the stationary points of $f(x)$ can be determined by studying the positive definiteness of (3.3) using its eigenvalues. If all the eigenvalues of $\mathbf{H}$ are

positive, then **H** is symmetric positive definite, indicating the presence of a **local minimum**.

3.1 Optimization algorithms

The solutions to the optimization problem are vital in modern machine learning and artificial intelligence algorithms, which includes weight optimization in deep learning. There are a number of popular optimization algorithm currently developed to solve the problem. Hence, choosing the right algorithm can be challenging as well. Let the network cost function, c, be given by the squared difference between the predicted value and the true value:

$$c = \frac{1}{2n} \sum_{i}^{n} (\widehat{y_i} - y_i)^2 \tag{3.4}$$

where n is the total number of input.

The difference is squared to avoid the sum of errors of multiple input vector to be zero which can mislead the network to have perfect predictive power. The goal of the network is to adjust the weight to reduce the network error as much as possible. The idea of reducing a function to a value is synonymous to (3.1) an optimization problem. The network error in (3.4) is a quadratic equation in this case.

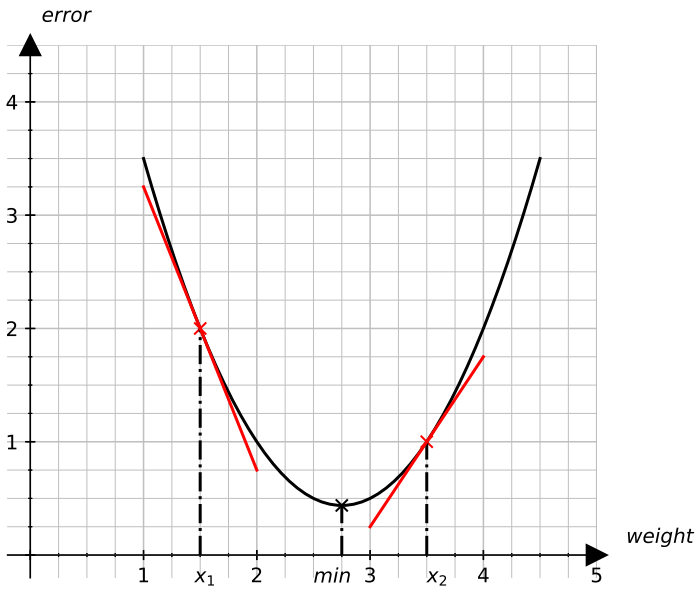

Figure 3.2: Network Error over different weight

Figure (3.2) is a graphical representation of the network error function. Our goal is to reach the minimum of the function by adjusting the weight value.

Gradient Descent

In order to reach the bottom of the function; we need to adjust the weight such that the derivative of the error function is 0. The approach of updating the weight based on the gradient of the error function is known as **gradient descent**. The derivative with respect to the weights of the network error (3.4) for a single input is given by

$$\frac{\partial e}{\partial w_{ij}} = (\widehat{y}_i - y_i)\frac{\partial \widehat{y}_i}{\partial w_{ij}} \tag{3.5}$$

From figure (3.2); we can observe that if the gradient is negative then we have underestimated the predicted value and need to increase the weight to reach the optimal value. On the other hand, if the gradient is positive then we have overestimated the predicted value and need to decrease the weight to reach the optimal value. The equation (3.5) provides us with the opposite direction and amount to adjust the weight. Hence, the update rule of the weights for gradient descent method is given

by

$$w_{ij}^{t+1} = w_{ij}^{t} - \frac{de}{dw_{ij}} \tag{3.6}$$

In the gradient descent method, the network learns from the gradient of the error function and adjust the weights accordingly to reduce the error. However, the gradient alone can be quite large, causing oscillations as we go down the error function. This problem by introducing a **learning rate** α prior to adjusting the weight.

Learning Rate

The learning rate is typically denoted by the Greek letter alpha α. It helps the network to control the rate at which the weights are changing. Having a system with a high learning rate may lead to an oscillating network when trying to find the optimal weight and having a slow learning rate increases the number of iterations required when optimizing the network. The adjusted update rule of the weight is given by

$$w_{ij}^{t+1} = w_{ij}^{t} - \alpha \frac{\partial e}{\partial w_{ij}} \tag{3.7}$$

where $\alpha \in (0, 1)$

Stochastic Gradient Descent

The Stochastic Gradient Descent dates back to the Robbins-Monro(ADD REF) algorithm from the 1950s and is still an important optimization algorithm. Instead of adjusting the weight to the average loss function, we can approximate the gradient by only one random directional derivative of the error. Thus, the number of computation in the gradient descent method can be significantly reduced by a factor of n (where n is the number of direction). The stochastic gradient method (SGM) is

given by

$$w^{t+1} = w^t - \alpha_t(\nabla e)_i \tag{3.8}$$

where α_t is the learning rate at the t^{th} iteration which may vary with iterations. As we move closer to the bottom of the minimum of the function; the solution starts to oscillate since we are moving the weight in random directions. Thus, the learning rate should be reduced gradually as we iterate. Some commonly used reduction of learning rate is given by

$$\eta(t) = \eta_i \quad \text{if } t_i \leq t \leq t_{i+1} \qquad \text{piecewise constant} \tag{3.9}$$

$$\eta(t) = \eta_0 e^{-\lambda t} \qquad \text{exponential decay} \tag{3.10}$$

$$\eta(t) = \eta_0(\beta t + 1)^{-\alpha} \qquad \text{polynomial decay} \tag{3.11}$$

where λ, β, and α are known as hyperparameter.
Training dataset can be very large in certain cases which leads to a greater cost of compute at each iteration for the gradient descent, so stochastic gradient descent is preferred in these cases.

Mini-Batch Stochastic Gradient Descent

In order to make use of the features of both gradient descent and stochastic gradient descent; we introduce the mini-batch stochastic gradient descent. In this approach, instead of iterating in the direction of the full gradient or only in one direction of the full gradient, we split the dataset into small batches and compute the gradient of each batch. The formula of the stochastic gradient descent is given by

$$\begin{aligned} \mathbf{w}^{t+1} &= \mathbf{w}^t - \eta_t \mathbf{g}_t \\ \mathbf{g}_t &= \nabla_{\mathcal{B}_t} e(\mathbf{x}, \mathbf{w}) \end{aligned} \tag{3.12}$$

where $\mathcal{B}_t$ is a mini-batch of elements drawn uniformly at random from the training set (the expectation of the gradient remains unchanged).

Since we are using a mini-batch, the updates are closer to the full gradient but at a reduced cost.

Newton's Method

For minimizing $f(x)$, $x \in \mathbb{R}$, we need to solve $g(x) = e^{'}(x) = 0$. Newton's iteration is given by

$$x_{n+1} = x_n - \frac{g(xn)}{g^{'}(x_n)} \tag{3.13}$$

$$= x_n - \frac{f^{'}(xn)}{f^{''}(x_n)} \tag{3.14}$$

For multivariate functions we need to minimise $f(\mathbf{x})$ over $\mathbf{x} \in \mathbb{R}^n$, that it

$$\underset{\mathbf{x}\in\mathbb{R}^n}{min} f(\mathbf{x}), \quad \mathbf{x} = (x_1, x_2, \ldots, x_n)^T \in \mathbb{R}^n \tag{3.15}$$

The Newton's iteration for multivariate function is given by

$$\mathbf{x_{n+1}} = \mathbf{x_n} - H(\mathbf{x_n})^{-1} \nabla f(\mathbf{x_n}) \tag{3.16}$$

where $H(\mathbf{x_n})$ is the Hessian matrix of $f(\mathbf{x})$.
We can observe from (3.3) that calculating the inverse of the Hessian matrix can be computationally very expensive for higher dimensions. Replacing $H(\mathbf{x_n})^{-1}$ by $\alpha\mathcal{I}$ where $\mathcal{I}$ is the identity matrix; we get the **method of steepest descent** given by

$$\mathbf{x_{n+1}} = \mathbf{x_n} - \alpha \mathbf{I} \nabla f(\mathbf{x_n}) \tag{3.17}$$

where $\alpha \in (0, 1)$

Momentum

Exponentially Weighted Moving Average

The exponentially weighed moving average (EWMA), also known as exponential moving average (EMA), of a series of data points S_t is given by

$$\begin{aligned} \mu_t &= \beta\mu_{t-1} + (1-\beta)S_t \\ \mu_0 &= c \end{aligned} \tag{3.18}$$

where $c \in \mathbb{R}$ and $\beta \in (0, 1)$ is known as the smoothing constant. β represents the weightage that is going to be assigned to the past values. The average number of previous reading is approximately given by $n = (1-\beta)^{-1}$. The higher the value of β the greater the number of points we average over.

SGD with Momentum

The momentum method was mentioned in Rumelhart, Hinton and Williams' paper on back-propagation learning in 1986 (ADD REF). From the SGD, it was observed that the weight update can be very noisy. In order to smoothen the search direction in the SGD, an exponentially moving average is implemented on the gradient. The SGC with momentum can be written in the form

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta\boldsymbol{\mu}^t \tag{3.19}$$

where the exponential moving average of the gradient is given by

$$\begin{aligned} \boldsymbol{\mu}^t &= \beta\boldsymbol{\mu}^{t-1} + (1-\beta)\nabla(e)^t \\ \boldsymbol{\mu}_0 &= \mathbf{c} \end{aligned}$$

RMSProp

Proposed by Geoff Hilton (ADD REF) in lecture 6 of the online course "Neural Network for Machine Learning", he Root Mean Square Propagation (RMSProp)

algorithm alleviates undesirable fluctuations when adjusting the weight during optimization. RMSProp is an adaptive learning rate. The idea is to reduce the step size at very large gradient to avoid fluctuations and increase the step size at smaller gradient to move steadily in the correct direction of the optimal solution; hence an adaptive learning rate at each iteration. The equations for the RMSProp is given by

$$\begin{aligned} S_i^t &= \gamma S_i^{t-1} + (1-\gamma)(\nabla e^t)_i^2 \\ w_i^{t+1} &= w_i^t - \frac{\eta}{\sqrt{S_i^t + \epsilon}}(\nabla e^t)_i \end{aligned} \tag{3.20}$$

where $\epsilon \approx 10^{-6}$ to ensure that we avoid dividing by zero during iterations.

Adam

The Adam algorithm (Adaptive Moment Estimation algorithm) is a robust update rule for the weight optimization, it was proposed by Kingma and BA in 2014 (ADD REF). The algorithm combines the benefits of momentum (3.19) and RMSProp (3.20). Adam is the most popular generalized algorithm performs very well in many cases. It is considered as a state-of-the-art algorithm for deep neural network optimization. The set of equations for the Adam algorithm is given by

$$\begin{aligned} \mu_i^{(t)} &= \beta_1 \mu_i^{(t-1)} + (1-\beta_1)\nabla(e^{(t)})_i && \text{Momentum} \\ S_i^{(t)} &= \beta_2 S_i^{(t-1)} + (1-\beta_2)(\nabla e^{(t)})_i^2 && \text{RMSProp} \end{aligned}$$

To prevent S^t and μ^t from becoming zero during the initial steps, a bias correction is introduced, such that

$$\hat{\mu^{(t)}} = \frac{\mu^t}{1-\beta_1^t} \qquad\qquad \hat{S^{(t)}} = \frac{S^t}{1-\beta_2^t}$$

Finally, the update rule is given by

$$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta}{\sqrt{S_i^t + \epsilon}}\mu_i \tag{3.21}$$

The parameters β_1, β_2 and ϵ are knows are the hyperparameters of the Adam algorithm. A common set of values that works well in literature are $\beta_1 = 0.9$, $\beta_2 = 0.9999$ and $\epsilon = 10^{-8}$ (ADD REF).

3.2 Backpropagation using Adam

The Adam algorithm for optimization can be introduced in the weight update when training deep neural network. The problem of finding the right set of weights and biases for a deep neural network can be reduced to an optimization problem same as we defined in (3.4). After finding the gradients through the backpropagation algorithm, the set of updates proposed in the Adam algorithm are used to update the weights and biases in the neural network.

Algorithm 3 Backpropagation using Adam optimization with n total number of inputs for N epochs

Start
for $t = 1, 2, \ldots, N$ **do**
  for $s = 1, 2, \ldots, n$ **do**
    Compute the activation layer $a^{(1)}$ using n input values in a batch
    $a^{(1)} = \sigma(w^{(1)}x + b^{(1)})$
    Feed forward the n input to the next layers
    for $l = 2, 3, \ldots, L$ **do**
      $z^{(l)} = w^{(l)}a^{(l-1)} + b^{(l)}$
      $a^{(l)} = \sigma(z^{(l)})$
    end for
  end for
  Compute the average cost function, C, at the output layer L
  $c^{(t)} = \dfrac{1}{2n}\sum_s^n(\widehat{y}_s - y_s)^2$
  Compute the rate of change of the cost function $w.r.t$ to $z^{(L)}$
  $\delta^{(L)} = \nabla_{a^{(L)}}C \odot \sigma'(z^{(L)})$
  for $l = L-1, L-2, \ldots, 2$ **do**
    Computing the gradients
    $\delta^{(l)} = (w^{(l+1)})^T\sigma^{(l+1)} \odot \sigma'(z^{(l)})$
    $\nabla_{b^{(l)}}C = \delta^{(l)}$
    $\nabla_{w^{(l)}}C = \delta^{(l)}a^{(l-1)}$

    Bias update
    $\mu_i^{(t)} = \beta_1\mu_i^{(t-1)} + (1-\beta_1)\delta_i^{(l)}$
    $S_i^{(t)} = \beta_2 S_i^{(t-1)} + (1-\beta_2)(\delta_i^{(l)})^2$
    $\hat{\mu^{(t)}} = \dfrac{\mu^t}{1-\beta_1^t}$
    $\hat{S^{(t)}} = \dfrac{S^t}{1-\beta_2^t}$
    $b_i^{(t+1)} = b_i^{(t)} - \dfrac{\eta}{\sqrt{S_i^t + \epsilon}}\mu_i$

    Weight update
    $\mu_i^{(t)} = \beta_1\mu_i^{(t-1)} + (1-\beta_1)\delta^{(l)}a_i^{(l-1)}$
    $S_i^{(t)} = \beta_2 S_i^{(t-1)} + (1-\beta_2)(\delta^{(l)}a_i^{(l-1)})^2$
    $\hat{\mu^{(t)}} = \dfrac{\mu^t}{1-\beta_1^t}$
    $\hat{S^{(t)}} = \dfrac{S^t}{1-\beta_2^t}$
    $w_{ij}^{(t+1)} = w_{ij}^{(t)} - \dfrac{\eta}{\sqrt{S_i^t + \epsilon}}\mu_i$
  end for
end for

Chapter 4

Sentiment Analysis

People's opinions, feelings and sentiments towards entities such products, services, other people, events, news, issues, topics, etc. can be in very large volume, complex and difficult to be understood and processed by machines and computers. Thus, sentiment analysis, also known as opinion mining, started to popularised along the rise of social media when large amount of digital text data were suddenly available for mining. Natural language processing (NLP) helps computers process and understand human based language to perform repetitive task. Sentiment analysis is a niche of NLP. It aims at quantifying the positivity, negativity and/or neutrality of implied or expressed in a given text.

Social media have been providing large platforms for people to share their opinion freely and expressed their views on any subject across various geographical and spatial boundaries. They have also allowed people to connect, influence and be influenced by such opinions and views. These interactions have been studied in the 1940s and 1950s among people in organizations by management science researchers. Since 2002, with social media, those studies have been performed at grand scales with the abundance of data. Thus, advanced sentiment analysis research have been performed in field political science, economics, finance and management science as they are heavily dependent on public opinions.

Asur and Huberman (2010) (ADD REF) attempted to solve the revenue prediction problem using both the tweet volume and the tweet sentiment. Same kind of data along with polling results were also used, in Bermingham and Smeaton (2011)

(ADD REF), to train a linear regression model to predict election results. Zhang et al. (2010) (ADD REF) leveraged sentiments on Twitter to help predict the movement of stock market indices such as Dow Jones, S&P500 and NASDAQ. It was shown that large negative emotions and opinions caused Dow to go down the next day and lower negative emotions cause Dow to go up on the next day. Bar-Haim et al. (2011) (ADD REF) on one hand leverage sentiments on Twitter but on the other did not treat all Twitter authors equally. Only expert investors were used as features in training stock price movement predictors. Undeniably, modern social media (started early 2000s) have grown into a major influencer of human opinion and sentiments.

4.1 Rule-Based Methods

Conventional NLP techniques relied on a set of rules to process textual data to extract opinion, polarity, topic, and other information within the data. Tokenization, part-of-speech tagging, lemmatization and removal of stop words are some rules and technique used to processed data prior to analysis. Tokenization is the separation/breaking down of text data into words. The space between words in a document is commonly used as the delimiter of separation. Consider the example below:

"The price of Cryptocurrency wasn't great today! See y'all tomorrow."

Tokenizing the result above results in the list:

"["The", "price", "of", "Bitcoin", "was", "n't", "great", "today", "!", "See", "y'", "all', "tomorrow", "."]"

In sentiment analysis, the tokenized sentence would be compared to predefined list polarized words (positive and negative). A naive way to get the polarity of the sentence would be to count the number of positive and negative words in the predefined polarized lists. The limitations are quite obvious, we are not taking into account the context in which the words are used, nor the preceding words. Part-of-speech tagging refers to the classification of the token in a document based on predefined assignment. Tokens are classified as adjective, adverb, noun, verb, etc. (the full list can be obtained through the universal POS tags - (ADD REF)). POS tagging is used to describe the syntactic structuring of a sentence. The part-of-speech tagging of our example in () is given by:

Placeholder

Lemmatization refers to the process of reducing words to their base form, also known as lemma. For example, the words below can be reduced as follows:

great → good

happiness → happy

stemming → stem

It allows to map the meaning of multiple words at one time by reducing the former to its base. The assumption is that both form retain the same meaning syntactically. Moreover, the removal of stop words is also often perform to eliminate words that are neutral or bring no additional value within a sentence. Examples of such word would be “the”, “a”, “is”, etc. Since rule-based models is performed individually

over each word; the simple removal stop words reduced computation time of such models. However, such models are highly limited, computationally demanding and often inaccurate. Thus, with the rise of computation power available and machine learning, statistical and embedding based models were developed to tackle those limitations.

4.2 Word Embedding

Processing raw data from ruled based method is not ideal. A numerical representation of text data would be more appropriate for mathematical models to perform NLP. Word embedding is the representation of text data as vectors in a vector space. Words with similar meaning would be considered as very close to each other in such a space. Moreover, from linear algebra, we would expect that normalized vector of words with similar meaning to be close. Word embedding techniques are categorized among two types: frequency based embedding and prediction based embedding.

Frequency based embedding refers to the vectorization of words based on the frequency of occurrence of words in a document. The three types of frequency based embedding are: Count Vector, TF-IDF (Term Frequency-Inverse Term Frequency) Vector and Co-Occurence Matrix with a fixed context window. Such type of embedding results in very sparse vector of high dimensionality which are computationally challenging to manipulate.

Prediction based embedding refers to the prediction of a target word based on the context (neighboring words). Developed by Mikolov et al in 2013 (ADD 2 REF), the word2vec was introduced. Word2vec is a word embedding method derived from two techniques: CBOW (Continuous Bag of Words) and Skip-Gram Model. The CBOW attempts to predict the probability of a word in the center from a given context while the Skip-Gram model is the opposite of the CBOW model; it tries to predict the context of words given a center word.

4.3 Transformer Model (Attention Mechanism)

Attention in neural network attempts to emulate the cognitive attention in the human brain. It focuses on only relevant part of the data while neglecting the rest. Developed by at team at Google Brain in 2017 (Vaswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser & Polosukhin 2017), transformers are the current state-of-the-art attention techniques (instead of recurrence) for processing contextual data; text, audio, video, speech, etc. Similar to the RNN, LSTM and GRU, transformers process sequential data as well with the exception that transforms process all input at once thus allowing parallelization. Transformers consist of two main parts: the encoder and the decoder. The encoder processes the input document, identifies the important text and creates an embedding for the text based on the importance of the latter to other words in the document. The decoder on the other hand tries to get the text back from the embedding created by the encoder.

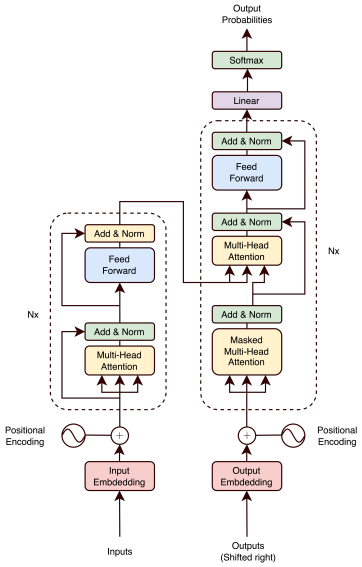

Figure 4.1: !!!TO BE CHANGED!!!-Transformer Layer

The left part of the architecture is the encoder while the right side is the decoder. This structure allows the transformer model for parallelization computations instead of sequential (RNN, LSTM and GRU). Transformer models are now able to solve natural language processing problem with higher accuracy. We introduce two commonly used transformation models: BERT and RoBERTa.

Bidirectional Encoder Representations from Transformers (BERT) which was developed by Google (Devlin, Chang, Lee & Toutanova 2018). BERT uses only the encoder part of the transformer similar to the original transformer in (Vaswani et al. 2017).

BERT is able to create word representations that are dynamically influenced by the surrounding words, whereas word2vec has a fixed representation for each word independent of the context in which it occurs. Leading BERT to achieve multiple state-of-the-art NLP tasks when it was published, including at sentiment analysis (Chiorrini, Diamantini, Mircoli & Potena 2021) A number of pre-trained BERT models from unlabeled text are currently available (due to the large number of data and parameters required for training) and can be re-adapted using transfer learning.

RoBERTa stands for Robustly Optimized BERT Pretaining Approach was also developed by Google (Liu, Ott, Goyal, Du, Joshi, Chen, Levy, Lewis, Zettlemoyer & Stoyanov 2019). The approach found that BERT was currently under trained and could potentially do much better. RoBERTa uses an extensively large amount of hyperparameter, tuning and optimization. Additionally, much more data was also used during training. BERT used 16gb of data from CC-news while RoBERTa was trained on 160bg of data from CC-news (92gb), OpenWebText (38bg) and addition data from Stories (31gb) (Liu et al. 2019).

Bibliography

Bengio, Y., Simard, P. & Frasconi, P. (1994), 'Learning long-term dependencies with gradient descent is difficult', *IEEE Transactions on Neural Networks* **5**(2), 157–166.

Chiorrini, A., Diamantini, C., Mircoli, A. & Potena, D. (2021), Emotion and sentiment analysis of tweets using bert.

Devlin, J., Chang, M., Lee, K. & Toutanova, K. (2018), 'BERT: pre-training of deep bidirectional transformers for language understanding', *CoRR* **abs/1810.04805**.
URL: *http://arxiv.org/abs/1810.04805*

Hochreiter, S. & Schmidhuber, J. (1997), 'Long Short-Term Memory', *Neural Computation* **9**(8), 1735–1780.
URL: *https://doi.org/10.1162/neco.1997.9.8.1735*

Kolen, J. F. & Kremer, S. C. (2001), *Gradient Flow in Recurrent Nets: The Difficulty of Learning LongTerm Dependencies*, pp. 237–243.

LeCun, Y., Bengio, Y. & Hinton, G. (2015), 'Deep learning', *Nature* **521**(7553), 436–444.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L. & Stoyanov, V. (2019), 'Roberta: A robustly optimized BERT pretraining approach', *CoRR* **abs/1907.11692**.
URL: *http://arxiv.org/abs/1907.11692*

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.

Rosenblatt, F. (1958), 'The perceptron: a probabilistic model for information storage and organization in the brain.', *Psychol Rev* **65**(6), 386–408.

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

Schuster, M. & Paliwal, K. (1997), 'Bidirectional recurrent neural networks', *IEEE Transactions on Signal Processing* **45**(11), 2673–2681.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. & Polosukhin, I. (2017), 'Attention is all you need'.
URL: *https://arxiv.org/abs/1706.03762*