Optimization algorithms underlying neural networks

Classification of meditative states by use of recurrent neural networks

by

Oisin Clancy

A thesis submitted in partial fulfillment
of the requirements for the degree of
Bachelor's of Science
Mathematics

at
Lund University
2022

Optimization algorithms underlying neural networks

Classification of meditative states by use of recurrent neural networks

Oisin Clancy

Abstract

Neural networks can be utilized for an ever widening selection of tasks. In this thesis the most common optimization algorithms underlying neural networks are investigated: classical momentum, Nesterov momentum, AdaGrad, AdaDelta, RMSprop, Adam, AdaMax, and Nadam. The underlying mathematics that these algorithms are based on is described. There is a summary of key components of a neural network—activation functions and loss functions—which provides prerequisite knowledge for understanding the place of optimization algorithms within the whole.

Classification of time series data (sequential modelling) can be accomplished through the use of Recurrent Neural Networks (RNNs). These networks allow for the capture of time dependent information. A brief overview of the structure of this type of network is given.

The optimization algorithms, activation functions, loss functions, and RNN that have previously been described theoretically are then implemented explicitly in a real world problem. They are used to classify EEG data of different meditative practices that entail subjective changes in experience. Through this it is shown how the mathematics underlying these tools eventually leads to results in a diverse range of scientific explorations: in this case, the classification of conscious states.

Dedication

To my mum and dad for everything they have given me.

Acknowledgements

I would like to thank my thesis advisor Alexandros Sopasakis for taking on this project and sharing his time with me. Mikael Johansson for introducing me to EEG, sharing his knowledge, and engaging in many interesting dialogues over the years. Philippe Goldin for precious use of his brain in recording and conversing. Alexander Hagelborn for essential discussions regarding neural network design. Lastly, the Math and Psychology departments at Lund University for the education I have received, opportunities offered, and the time given by various professors here.

Contents

List of Figures

Chapter 1

Introduction

1.1 Optimization

The words optimum and optimal stem from the Latin word *optimus* meaning the "best possible" or "most favourable" [14]. Optimisation refers to the process of bringing something into the best possible state. The history of optimisation is long and has had an important role throughout the history of civilisation—before the mathematics of the topic was well established. Optimisation was often done by simulation—imagination, e.g. what is the best route to this location?

"Optimality is a fundamental principle, establishing natural laws, ruling biological behaviours, and conducting social activities." [6]

The beginnings of optimisation within mathematics can be spotted in the geometric studies of Ancient Greece, for instance, when Euclid considered the minimal distance between a point and a line. However, as a whole, the history of optimisation in mathematics can be divided into three periods. [6]

In the first period, there were no known general methods for finding a maximum/minimum point of a function. Only special techniques were found to maximize/minimize some special functions such as the quadratic function of one variable.

The second period began in 1646 by Pierre de Fermat who proposed in a paper [4] a general approach to compute local maxima/minima points of a differentiable function by setting the derivative of the function equal to zero. This approach is likely the most known method for finding such min/max points and is included in most textbooks on calculus as an application of differentiation.

During this period, optimisation existed but was not an important branch of applied mathematics. Hence, little attention was paid to new results and many contributions were not documented, which leaves some mystery in the history of the topic. However, there are works by such well known mathematicians as Joseph-Louis Lagrange, Isaac Newton, and Carl Fredrich Gauss using calculus on continuous optimization [22].

The third period of optimisation started with the discovery of linear programming, with G.B. Dantzig's proposal of the simplex method [3] in 1947 being a significant catalyst point. Mathematical programming relates to the theory, use, and computational solution

of mathematical models to help in making decisions, and now encompasses a wide array of models such as linear programming, integer programming, nonlinear programming, stochastic programming, etc. All of which make use of optimization methods. [22]

Today, optimization has become a very important interdisciplinary field between mathematics, computer science, engineering, management science, and most recently played a prominent role in the rise and resurgence of artificial neural networks and machine learning. This wide applicability of optimization based methods points towards their continued use and expansion within a diverse range of fields.

1.2 Neural Networks

1.2.1 Biological Neural Networks

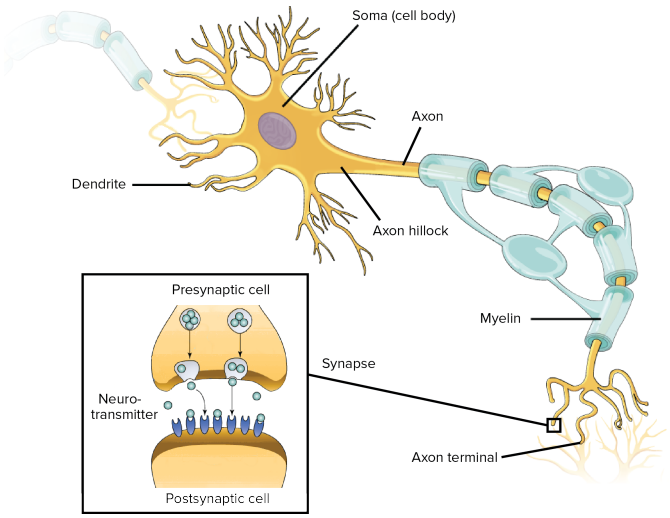


Figure 1.1: Biological neuron diagram.

The neuron is currently considered to be the main computational unit of the brain. While there are many different neuron types to be found in the human brain (and elsewhere in the body) the basic structure remains similar. The neuron is composed of a cell body, known as the **soma**, that contains the nucleus of the cell. Attached to this cell body are spindly arms known as **dendrites**, which act as transmission wires for incoming signals. The membrane is the outer layer of the neuron separating the interior of the neuron from the exterior extracellular fluid in which it is contained. A voltage difference between the interior and exterior of the cell exists and is referred to as the **membrane potential**. Small pores along the membrane, **ion channels**, can open and close to allow different ions to enter and exit the cell.

Electrical changes in membrane potential occur due to the opening and closing of ion channels along the length of the dendrites (and sometimes also on the cell body itself). The changes in membrane potential are carried towards the soma where the individual changes in potential are summed together. If the summation of the electrical potentials cross a specific threshold, approximately -55mV, then an **action potential** is triggered

within the cell body. This action potential then causes an electrical signal to flow down the main axon towards the pre-synaptic terminals. The propagation of this signal is helped along by insulating structures known as **myelin sheaths**, which act similarly to the insulating plastic found on the outside of a metal wire to improve conductivity. When this signal arrives at the pre-synaptic terminal it causes packages of neurotransmitters (vesicles) to be released into the synaptic cleft. These neurotransmitters then attach to the dendrites of other neurons, where they engage in a variety of biochemical signalling, triggering biochemical cascades, opening and closing ion channels, all of which act to change the membrane potential of the next neuron's dendrites at which point the process repeats itself within the next neuron.

The firing of an individual neuron—the triggering of an action potential—is thus dependent on the strength and timing of the incoming signals. Although the process is certainly a continuous one, the firing of the action potential can be modelled discretely as a binary decision: fire or not fire (on or off). The human brain contains approximately 89 billion neurons, all of which are engaged in this constant activity of biochemical and electrical activity—neural signalling. Many individual neurons become tightly interconnected forming groups of neurons that in turn communicate regularly with other groups of neurons, often situated in disparate regions of the brain. These connections between groups of neurons throughout the brain form networks which are often significantly correlated with specific behavioural and psychological functions; the excitation and inhibition of different networks having been shown to effect the cognitive processing of individuals. These neural networks provide the structure from which learning and memories are formed and integrated across the brain. These are the biological neural networks that artificial neural networks are based upon.

1.2.2 Comparison to Biological Neural Network

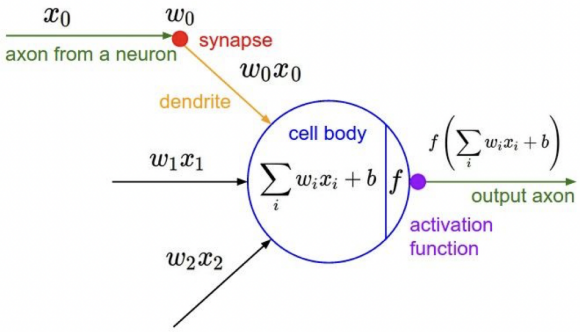


Figure 1.2: Artificial neuron diagram.

Artificial neural networks are coarse approximations of real biological neural networks. They provide the most basic functionality of real neurons: **nodes** acting as cell bodies, **connections** acting as dendrites and axons. Each node takes incoming signals from an incoming connection; the incoming signal being composed of the **output from the previous node**, x, multiplied by a **weight**, w associated with that particular connection.

These individual signals from each incoming connection then arrive at the node where they are summed together,

$$\sum_{i}^{m} w_i x_i.$$

An extra value, known as a **bias**, b, is associated with the node and this is added to the summation of the incoming signals,

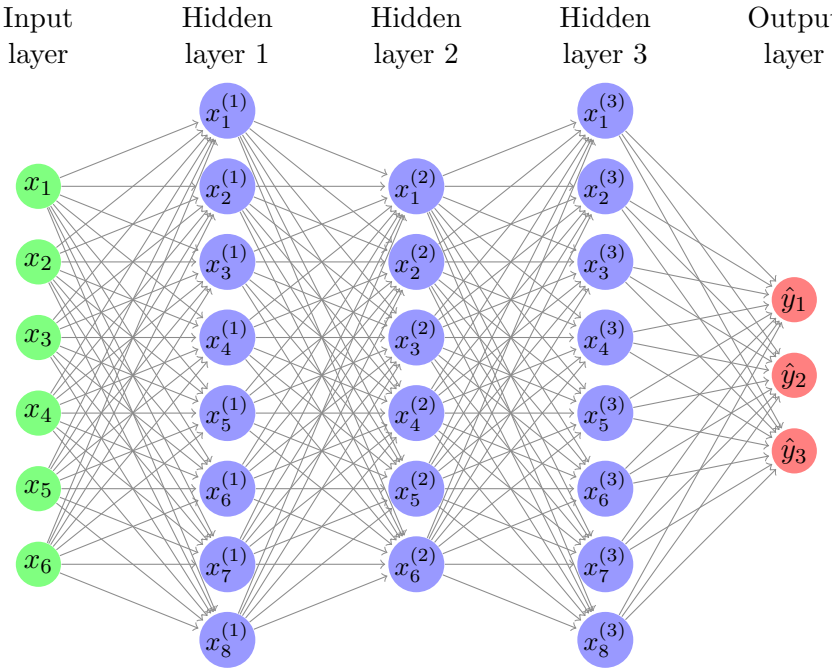
$$\sum_{i}^{m} w_i x_i + b.$$

This value then undergoes a transformation due to a function associated with that node, σ, an **activation function**. The output of this activation function is then the output of this particular node,

$$\sigma\left(\sum_{i}^{m} w_i x_i + b\right).$$

The output may travel along multiple outgoing connections, with each connection having an associated weight, and thus becoming the input to another node. This process is repeated throughout the network with an individual node having multiple incoming connections and multiple outgoing connections. The number of nodes in the network and the number of connections are dependent on the choice of network architecture. In this way an artificial neural network coarsely approximates a biological neural network.

1.2.3 Basic Artificial Neural Network Structure



The ***input layer*** of the network is composed of n input features labelled $x_1, x_2, \ldots, x_n$. Each input feature is normally in the form of a real number and each node in the input

layer represents one of these input features. All of these input features together can be written in vector form, $\mathbf{x} = (x_1, x_2, \ldots, x_n)$. Hence the first node represents x_1, second node represents x_2,..., the n'th node represents x_n. As this is the initial layer in the network we can use the superscript 0 to represent that these values refer to the input layer, $\mathbf{x}^{(0)} = (x_1^{(0)}, x_2^{(0)}, \ldots x_n^{(0)})$.

The ***hidden layers*** of the network can be as numerous as one wishes. The use of networks with many layers is what is often referred to as deep neural networks. Any network with more than 1 hidden layer can technically be referred to as a deep neural network. The number of layers in the network is referred to as the ***depth*** of the network (input layer is not included but output layer is).

Each hidden layer of the network contains a specific number of nodes. The number of nodes in a hidden layer does not need to be equal to the number of nodes in the input layer, nor must the number of nodes per each hidden layer be the same. The number of nodes in a layer is referred to as the ***width*** of a layer.

The values associated with the nodes of a hidden layer can also be represented by the use of $x_1, x_2, ..., x_n$. However, to distinguish these from the input layer we also write a superscript denoting the number of the layer. So in vector form we have, $\mathbf{x}^{(1)} = (x_1^{(1)}, x_2^{(1)}, \ldots, x_m^{(1)})$. Note that the subscript of the last value is given as m rather than n as the number of nodes in the hidden layer does not need to match the number of nodes in the input later.

We can refer to the node values of a hidden layer as $x_i^{(L)}$, where L refers to the hidden layer number and i refers to the node number in that layer. $\mathbf{x}^{(L)}$ is the vector representing the values of the nodes in the L^{th} layer, $\mathbf{x}^{(L)} = (x_1^L, x_2^L, x_3^L, \ldots, x_m^L)$.

If each node in a hidden layer is connected to each node in the previous layer it is referred to as ***fully connected*** (there are occasions when this is not true and a particular node will only be connected to a specific number of nodes in the previous layer but this is not important for an introductory understanding). The connections between these nodes can be referred to as ***edges*** or ***connections***. Therefore if hidden layer 1, $\mathbf{h}^1$, contains 20 nodes then each one of those nodes will be connected to every node in hidden layer 2. So a single node in hidden layer 2 will have 20 incoming connections. And similarly every other node in hidden layer 2 will also have 20 incoming connections.

Each one of these individual edges will have a ***weight***, w, associated with it. The weight is a real value number. The output of the node, $x_i^{(L)}$, is multiplied by the weight value of the edge that connects it to the next node. This weight can be written as , $w_{ij}^{(L)}$, where L refers to the layer of nodes it is being multiplied by, i refers to the number of the node that the weight is applied to, and j denotes the node that the edge is connected to. So the input to a node in the next layer will be given by $w_{ij}^L x^L$.

Each node $x_i^{(L)}$ will connect to m nodes in the next layer. Therefore if there are n nodes in the first layer and m nodes in the second layer there will $n \times m$ edges and so $n \times m$ weights. These weights can be more easily written as a matrix of weights,

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,m} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n,1} & w_{n,2} & \cdots & w_{n,m} \end{bmatrix}.$$

We can then multiply this matrix of weights by the vector representing the node values for the L^{th} layer so we get,

$$\mathbf{Wx} = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,m} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n,1} & w_{n,2} & \cdots & w_{n,m} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} w_{1,1}x_0 + w_{1,2}x_1 + \cdots + w_{1,m}x_n \\ w_{2,1}x_0 + w_{2,2}x_1 + \cdots + w_{2,m}x_n \\ \vdots \\ w_{n,1}x_0 + w_{n,2}x_1 + \cdots + w_{n,m}x_n \end{bmatrix}.$$

This vector thus acts as the input values for the $L + 1^{th}$ layer of the network.

Every individual node will also have a ***bias***, b, associated with it. This is a scalar value which is added to the summation of weights and outputs. This is provided so as to determine a threshold that the summation of weights and outputs must surpass in order for the node to output a particular value.

$$\mathbf{Wx+b} = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,m} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n,1} & w_{n,2} & \cdots & w_{n,m} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} w_{1,1}x_0 + w_{1,2}x_1 + \cdots + w_{1,m}x_n + b_1 \\ w_{2,1}x_0 + w_{2,2}x_1 + \cdots + w_{2,m}x_n + b_2 \\ \vdots \\ w_{n,1}x_0 + w_{n,2}x_1 + \cdots + w_{n,m}x_n + b_n \end{bmatrix}.$$

Associated to each node there will also be an **activation function**, σ, which takes as its input $Wx + b$ and thus produces the output of that particular node.

$$\sigma(\mathbf{Wx} + \mathbf{b}) = \begin{bmatrix} \sigma(w_{1,1}x_0 + w_{1,2}x_1 + \cdots + w_{1,m}x_n + b_1) \\ \sigma(w_{2,1}x_0 + w_{2,2}x_1 + \cdots + w_{2,m}x_n + b_2) \\ \vdots \\ \sigma(w_{n,1}x_0 + w_{n,2}x_1 + \cdots + w_{n,m}x_n + b_n) \end{bmatrix}.$$

The ***output layer*** of the network is a single layer composed of as many nodes as are representative of the type of problem one is dealing with. For instance, in this paper, we deal with a problem that is a multi-class classification problem. This means that we want the neural network to distinguish between multiple separate classes of objects. Specifically, we wish to classify between three different meditative states. Therefore, in this paper, the output layer of the network will contain three nodes. Each output node can be can be labelled as $\hat{y}_1, \hat{y}_2, \hat{y}_3$. These output values can be written in vector form, $\mathbf{y} = (\hat{y}_1, \hat{y}_2, \hat{y}_3)$. The output values are predictions–probabilities–for what class the particular input values belong to. The network is attempting to learn patterns in the data that will allow it to distinguish between each class based off the input values.

Chapter 2

Activation Functions & Loss Functions

2.1 Activation Functions[28]

Activation functions determine whether a node should be activated or not by applying a transformation to the weighted sum and bias,

$$\sigma\left(\sum_{i}^{m} w_i x_i + b\right). \tag{2.1}$$

Below is a brief summary of some of the most common activation functions.

2.1.1 Rectified Linear Unit

The Rectified Linear Unit (Relu) provides a simple nonlinear transformation. Given an input x, the function is defined as the maximum of the input and 0,

$$\text{ReLu}(x) = \max(x, 0). \tag{2.2}$$

The ReLU discards all negative inputs by setting the activation output to zero, and thereby retains only the positive inputs. ReLu is piecewise linear as can be seen on the next page.

The derivative of the ReLU is 0 when the input is negative, and 1 when the input is positive. ReLU is not differentiable when the input is equal to 0 so in this case we choose the left-hand side derivative and say that the derivative is equal to 0 (this is suitable in practice because the input is rarely, if ever, zero).

The derivatives of ReLU are very well behaved: they vanish or they allow the argument through. This is the main reason for its usage as it ensures better optimization behaviour and mitigates the problem of vanishing gradients (which will be discussed later).

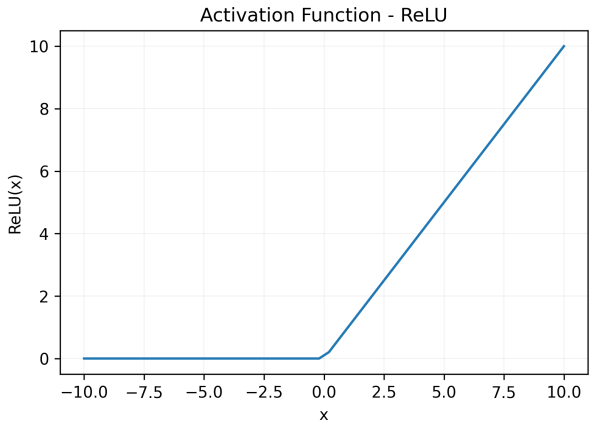


Figure 2.1: ReLU activation function.

2.1.2 Sigmoid (logistic sigmoid)

The sigmoid function transforms its inputs to outputs that lie on the interval $\in (0, 1)$,

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}. \tag{2.3}$$

For this reason, the sigmoid is often referred to as a squashing function: any input $\in (-\infty, +\infty)$ is squashed to some value $\in (0, 1)$. The function approaches a linear transformation when the input is close to zero.

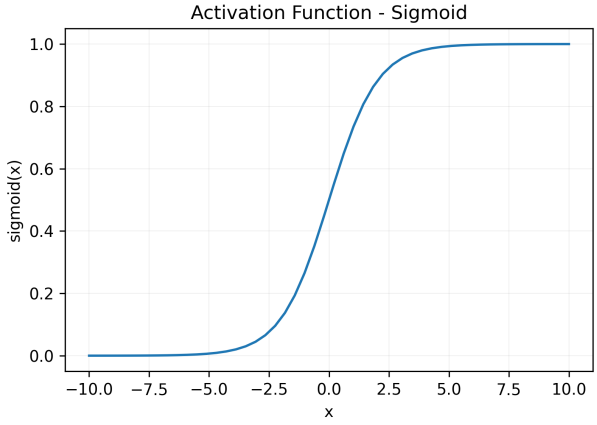


Figure 2.2: Sigmoid activation function.

The sigmoid function is a smooth, differentiable approximation to a thresholding activation function (takes a value of 1 when the input is above a threshold and a value of 0 when below). It is often used as the activation function on the output layer for transforming the output values into probabilites (the sigmoid can be thought of as a special case of the Softmax function discussed below). In hidden layers, they have have mostly been replaced by ReLU which are simpler and easier to train with.

In a later section on recurrent neural networks, we will see how logistic sigmoid functions are used in the network architecture to control the flow of information across time.

The derivative of the sigmoid function is,

$$\frac{d}{dx}\text{sigmoid}(x) = \frac{\exp(-x)}{(1+\exp(-x))^2} = \text{sigmoid}(x)(1-\text{sigmoid}(x)). \tag{2.4}$$

As the input to the sigmoid function approaches 0, the derivative approaches 0.25. As the input moves away from 0, the derivative approaches 0.

2.1.3 Tanh (hyperbolic tangent)

The tanh function transforms its inputs into values on the interval $\in (-1, 1)$,

$$\tanh(x) = \frac{1-\exp(-2x)}{1+\exp(-2x)}. \tag{2.5}$$

Similar to the sigmoid function, it squashes its inputs: any input is squashed to some value $\in (-1, 1)$. The function approaches a linear transformation when the input is close to zero, as can be seen below:

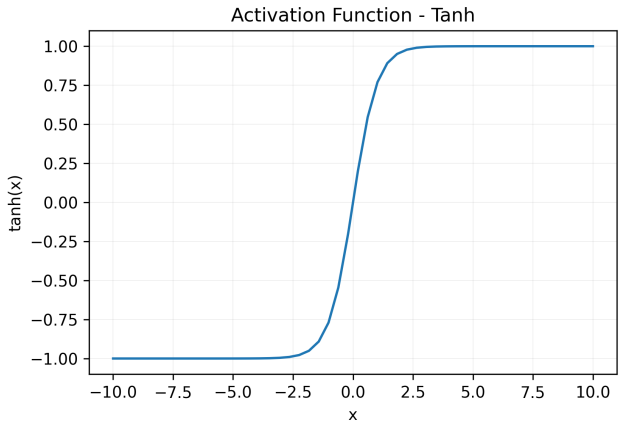


Figure 2.3: Tanh activation function.

The shape of tanh is similar to the sigmoid function, but the tanh function has point symmetry about the origin.

The derivative of the tanh function is,

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x). \tag{2.6}$$

As the input to the tanh function approaches 0, the derivative approaches 1. As the input moves away from 0, the derivative approaches 0.

2.1.4 Softmax

Softmax is a function that transforms its inputs into a probability distribution. It is different from the other activation functions covered previously as it requires a vector of

nodes for input, rather than a single node. It is normally used on the last layer of the network to normalize the output values of the network to a probability distribution. This probability distribution provides information on the prediction of what class has been used as input.

Softmax is defined as,

$$S(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}, \tag{2.7}$$

where $\mathbf{x}$ is an input vector consisting of n elements representing n classes, x_i is the i^{th} element of the input vector and can take any value between $-\infty$ and $+\infty$.

2.2 Loss Functions [13, 28]

Let $\mathbf{y}$ be the true target value and $\hat{\mathbf{y}}$ be a prediction value output by the network then we define a loss function,

$$L(\mathbf{y}, \hat{\mathbf{y}}), \tag{2.8}$$

such that we measure the difference between our prediction value and our true target value.

In our case, the prediction value, $\hat{\mathbf{y}}$, is in the form of a vector representing a probability distribution, as in our previous discussion of the Softmax activation function.

There are a range of different loss functions that can be used depending on the type of network and task that is being implemented. In this paper, we are doing a multi-class classification task and so the most pertinent loss functions are the cross entropy loss functions.

2.2.1 Cross-Entropy Loss

The concept of cross-entropy originates in the field of information theory and the work of Claude Shannon who introduced the term entropy within the context of information in his seminal 1946 paper, The Mathematical Theory of Communication [21], which is why it is sometimes referred to as Shannon entropy.

The entropy of a random variable, X, is the amount of uncertainty inherent in the random variable's possible outcomes. We define **entropy** as follows,

$$H(X) = -\sum_{i=1}^{c} P(x_i) \log P(x_i), \tag{2.9}$$

where X is a discrete random variable, x_i are the possible outcomes, and $P(x_i)$ the probability of the possible outcomes, and c is the number of outcomes. The reason for the negative sign is that $\log(x) < 0$ for all $x \in (0, 1)$, and P is a probability distribution so all values of $P \in [0, 1]$.

An increase in $H(X)$ means an increase in the uncertainty of the probability distribution. A decrease in $H(X)$ means a decrease in the uncertainty of the probability of the distribution.

In our model, the predicted value, $\hat{\mathbf{y}}$, is a probability distribution for the classes that we are attempting to distinguish between. Each predicted probability distribution, $\hat{\mathbf{y}}$, is compared to the true target probability distribution, $\mathbf{y}$, for that particular sample. A loss value is calculated that represents the error between the predicted probability distribution and the expected–target–probability distribution.

Cross-entropy loss is defined as,

$$L_{CE} = -\sum_{i=1}^{c} y_i \log \hat{y}_i, \tag{2.10}$$

where y_i is the target value, $\hat{y}_i$ is the Softmax probability for the i^{th} class, c is the number of classes. The log is calculated to base 2.

As this function is logarithmic it means that large differences are given a high loss value and small differences are given a low loss value.

2.2.2 Binary Cross-Entropy Loss

The binary cross-entropy loss for binary classification (classification between two classes) is defined as,

$$\begin{aligned} L_{BCE} &= -\sum_{i=1}^{2} y_i \log \hat{y}_i \\ &= -[\mathbf{y} \log \hat{\mathbf{y}} + (1 - \mathbf{y}) \log(1 - \hat{\mathbf{y}})], \end{aligned} \tag{2.11}$$

where y_i is the target $\in \{0, 1\}$ and $\hat{y}_i$ is the Softmax probability for the i^{th} class.

In practice, the loss is often calculated over numerous samples in the datatset before being summed and averaged,

$$L_{BCE} = \frac{1}{n} \sum_{i=1}^{n} \Big(- [\mathbf{y} \log(\hat{\mathbf{y}})) + (1 - \mathbf{y}) \log(1 - \hat{\mathbf{y}})] \Big), \tag{2.12}$$

where n is the number of samples chosen to average over.

2.2.3 Multi-Class Cross-Entropy Loss

For multi-class classification problems we make use of the standard cross-entropy loss equation (2.2.3). However, in practice we calculate this over multiple samples in the dataset,

$$L_{MCE} = \frac{1}{n} \sum_{i=1}^{n} \Big(- \sum_{i=1}^{n} y_i \log(\hat{y}_i) \Big), \tag{2.13}$$

where n is the number of samples chosen to average over.

2.2.4 Note on nomenclature

It is common to see the terms objective function, cost function, and loss function used interchangeably in machine learning literature. However, this can be confusing as there are practical differences between those terms. So, for the sake of clarity, in this paper we will make use of the terms loss function and objection function. We will define their meaning below.

The loss function can be seen as the type of function that is used to calculate a value of difference between the prediction value and the actual target value; in our case—cross entropy. The objective function can be seen as that loss function applied to some number of samples in the data set. The objective of the network is to minimize the value output by the objective function by adjusting the parameters of the network, the weights and biases, accordingly. This process is also commonly stated as reducing or decreasing the loss.

Chapter 3

Optimization Algorithms

The main purpose of the optimization algorithms is to allow us to appropriately update the parameters of the neural network–the values of the weights and biases–so that can we decrease the value of the loss function.

3.1 Gradient Descent [19, 28]

There are three versions of gradient descent. The difference in these versions relates to the number of samples that are used to calculate the objective function before we take the gradient of the objective function and update the parameters.

3.1.1 Regular Gradient Descent

In regular gradient descent we calculate the objective function over the entire dataset, i.e. over all our n samples of data, before updating our parameters. In practice, this is very slow as it requires an unreasonable amount of computation, and also contains a lot of redundancy.

The objective function, $h(\theta)$, is the average sum of the loss function, L, over all samples in the data set,

$$h(\theta) = \frac{1}{n}\sum_{i}^{n} L(f(x^{(i),\theta),y^{(i)}}, \tag{3.1}$$

where i counts the samples, n is the total number of samples, f is the prediction based on the sample input, $x^{(i)}$, and the parameters, θ, and $y^{(i)}$ is the expected output.

Let $\theta_t \in \mathbb{R}^n$ be a vector containing the parameters in the network where the sub-index t denotes the iteration step, $\nabla h(\theta_t)$ be the gradient of our objective function with respect to the parameters at iteration step t, and α be the learning rate. Then we have a regular gradient descent iteration given by,

$$\theta_{t+1} = \theta_t - \alpha \nabla h(\theta_t), \tag{3.2}$$

where θ represents the parameters which are composed of the weights, w_j for $j \in \mathbb{Z}^+$, and the biases, b_k for $k \in \mathbb{Z}^+$. So, when taking the gradient we are taking the gradient with respect to every single one of these parameters,

$$\nabla h(\theta_t) = \left(\frac{\partial h}{\partial \mathbf{w}}, \frac{\partial h}{\partial \mathbf{b}}\right)$$
$$= \left(\frac{\partial h}{\partial w_1}, \frac{\partial h}{\partial w_2}, \frac{\partial h}{\partial w_3}, ..., \frac{\partial h}{\partial w_j}, \frac{\partial h}{\partial b_1}, \frac{\partial h}{\partial b_2}, \frac{\partial h}{\partial b_3}, ..., \frac{\partial h}{\partial b_k}\right).$$

3.1.2 Stochastic Gradient Descent

In stochastic gradient descent we calculate the objective function over a single uniformly drawn random sample, $x^{(i)}$ for $i \in \{1, \ldots, n\}$, in our dataset before updating our parameters.

The objective function, $h(\theta)$, is the loss function, L, for a single sample in the data set,

$$h(\theta) = L(f(x^{(i)}, \theta), y^{(i)}), \tag{3.3}$$

where i is the sample number, f is the prediction based on the sample input, $x^{(i)}$, and the parameters, θ, and $y^{(i)}$ is the expected output.

The gradient of the objective function with respect to the parameters at iteration step t is thus,

$$\nabla h(\theta_t) = \nabla_\theta L(f(x^{(i)}; \theta), y^{(i)}). \tag{3.4}$$

Our stochastic gradient descent iteration is then given by,

$$\theta_{t+1} = \theta_t - \alpha \nabla h(\theta_t). \tag{3.5}$$

3.1.3 Mini Batch Gradient Descent

In mini batch gradient descent we calculate the objective function over a mini batch of n random uniformly drawn samples in our dataset, $x^{(i)}$ for $i \in \{1, \ldots, n\}$, before updating our parameters.

The objective function, $h(\theta)$, is the average sum of the loss function, L, over n random samples in the data set,

$$h(\theta) = \frac{1}{n} \sum_i^n L(f(x^{(i)}; \theta), y^{(i)}), \tag{3.6}$$

where i counts the samples, n is the total number of samples, f is the prediction based on the sample input, $x^{(i)}$, and the parameters, θ, and $y^{(i)}$ is the expected output.

The gradient of the objective function with respect to the parameters at iteration step t is thus,

$$\nabla h(\theta_t) = \frac{1}{n} \nabla_\theta \sum_i^n L(f(x^{(i)}; \theta), y^{(i)}). \tag{3.7}$$

Our mini batch gradient descent iteration is then given by,

$$\theta_{t+1} = \theta_t - \alpha \nabla h(\theta_t). \tag{3.8}$$

3.1.4 Version of Gradient Descent Used in Algorithms

As seen above the expanded notation for these algorithms can become quite cumbersome due to the number of indices needed for describing iteration steps, sample number, batch size, etc. For the rest of the paper all our algorithms will make use of mini batch gradient descent. Hence, the notation for the gradient of the objective function will refer to (3.7).

3.2 Background for Optimization Algorithms

We will look at eight of the most common optimisation algorithms currently being used in machine learning:

- Classical momentum
- Nesterov momentum
- AdaGrad
- AdaDelta
- RMSprop
- Adam
- AdaMax
- Nadam

The primary purpose of these algorithms is to increase the convergence rate to the minimum point, i.e. decrease the amount of time needed to arrive at our optimal point, while maintaining a reasonable computational load.

The order they are introduced should provide an increasing level of understanding as later algorithms normally build upon or adapt previous algorithms. Classical momentum is a method that takes into account previous gradients to adapt its current step sizes. Nesterov's momentum is a variant of the classical momentum. AdaGrad is an attempt to adapt the learning rate of the algorithm individually for every parameter, and AdaDelta and RMSprop are variants of AdaGrad. Adam is based upon a combination of Momentum and RMSprop. Adamax is a variant of Adam which makes use of infinity norms. Nadam is a variant of Adam which makes use of concepts from Nesterov's momentum.

Before examining any of the optimisation algorithms in detail it is instructive to first understand the concepts of Exponentially Weighted Moving Averages (EWMA), bias-correction, and learning rate (step size) as these concepts provide much of the mathematical basis and motivation behind these algorithms.

3.2.1 Exponentially Weighted Moving Averages

Exponentially weighted moving averages (EWMA) can simply be thought of as averages of a sequence of values in which weights are assigned to each value before taking the average. The most recent values receive larger weights while the earlier values receive smaller

weights. The purpose of an EWMA is to take an average over all previous measurements yet only a specific amount of the most recent values—a moving window—have a large impact on the average.

Let θ_t be a vector containing the parameters of the network at iteration step t and $\beta \in (0, 1)$ be a hyperparameter that controls the size of the moving window. At every time step, t, we will have a new parameter value, θ_t. We then create a weighted sum of these values,

$$v_t = \beta v_{t-1} + (1 - \beta)\theta_t. \tag{3.9}$$

By looking at multiple iterations of this equation we can clearly see how it produces an exponentially weighted average. We will change the order of the RHS to make this more clear,

$$\begin{aligned} v_1 &= (1 - \beta)\theta_1 + \beta v_0 \\ v_2 &= (1 - \beta)\theta_2 + \beta v_1 \\ &\vdots \\ v_{t-2} &= (1 - \beta)\theta_{t-2} + \beta v_{t-3} \\ v_{t-1} &= (1 - \beta)\theta_{t-1} + \beta v_{t-2} \\ v_t &= (1 - \beta)\theta_t + \beta v_{t-1} \end{aligned}$$

If we now expand v_t,

$$\begin{aligned} v_t &= (1 - \beta)\theta_t + \beta v_{t-1} \\ &= (1 - \beta)\theta_t + \beta\big((1 - \beta)\theta_{t-1} + \beta v_{t-2}\big) \\ &= (1 - \beta)\theta_t + \beta\big((1 - \beta)\theta_{t-1} + \beta\big((1 - \beta)\theta_{t-2} + \beta v_{t-3}\big)\big) \\ &= (1 - \beta)\theta_t + \beta\big((1 - \beta)\theta_{t-1} + \beta\left((1 - \beta)\theta_{t-2} + \beta\big((1 - \beta)\theta_{t-3} + \beta v_{t-4}\big)\right)\big) \end{aligned}$$

If we continue to expand and then multiply out we get,

$$\begin{aligned} v_t &= (1 - \beta)\theta_t + \beta(1 - \beta)\theta_{t-1} + \beta^2(1 - \beta)\theta_{t-2} + \cdots \beta^{t-2}(1 - \beta)\theta_2 + \beta^{t-1}(1 - \beta)\theta_1 \\ &= (1 - \beta)\sum_{i=1}^{t} \beta^{t-i}\theta_i. \end{aligned}$$

So this is a weighted sum of $\theta_t, \theta_{t-1}, ..., \theta_1$ and since all of the coefficients $\beta^{t-i}(1-\beta)$ add up to 1 it it actually an exponentially weighted average, where the value of β determines the size of the moving window. A heuristic for choosing the value of β is that v_t is an approximate average over $\frac{1}{1-\beta}$ time steps. Note that an EWMA may also be referred to as a 'decaying mean' and the β term as the term which controls the 'decay rate'.

One of the advantages of this formula, (3.9), is that it takes up a small amount of memory due to continuously overwriting previous averages. Although it is not the most accurate way to compute an average, i.e. for a previous k iterations we could have a moving window average where we explicitly sum the last k values and then divide by k, $\frac{\theta_{t-k+1}+\cdots+\theta_{t-1}+\theta_t}{k}$, but the disadvantage of this method is that it requires us to keep the last k parameter values in memory and so is computationally expensive. Therefore when we need to compute averages of many variables (3.9) is a very useful way to do so due to computational efficiency.

Decaying Sum Over Previous Updates

In the cases discussed below of classical momentum, section 3.3.1, and Nesterov momentum, section 3.3.2, the original authors write the momentum term using a decaying sum over previous updates rather than a decaying mean over previous gradients (EWMA).

For the sake of clarity, classical momentum and Nesterov momentum as they are described in this paper have been written as versions using EWMA. This has precedence as discussed by Andrew Ng [15] who suggests that writing classical momentum in this manner is easier when implementing the algorithm due to the separation of the decay rate, β, from the learning rate, α, when defining the momentum term. This allows for easier manual tuning of these hyperparameters.

3.2.2 Bias Correction

Let us look at bias correction specifically in relation to EWMA. If we look at the initial iterations of EWMA we will see that at the beginning we are largely dependent on the $(1-\beta)$ coefficient for the average value that is produced,

$$
\begin{aligned}
v_0 &= 0 \\
v_1 &= \underbrace{\beta v_0}_{=0} + (1-\beta)\theta_1 \\
v_2 &= \beta \underbrace{v_1}_{<<\theta_1} + (1-\beta)\theta_2 \\
v_3 &= \beta \underbrace{v_2}_{<<\frac{\theta_1+\theta 2}{2}} + (1-\beta)\theta_3 \\
&\vdots
\end{aligned}
$$

This means that our initial iterations will produce averages that are significantly less than the actual average. To rectify this bias for out initial iterations we introduce the bias correction formula:

$$
\frac{v_t}{1-\beta^t}. \tag{3.10}
$$

As t becomes large β_t goes to zero so the bias correction only has a noticeable impact at the beginning of the iterations where the bias is an issue.

3.2.3 Learning Rate

The learning rate (step size), α, is typically manually tuned for the highest possible value. Choosing greater than this value this may cause the algorithms to diverge from the minimum, and choosing less than this value may cause slow learning. The choice of this learning rate is often down to practical experience and a compromise must be made between speed of convergence and accuracy of convergence.

Adapted Learning Rate

We shall see that many of the algorithms are aimed towards adapting the learning rate of the parameter update rule. In our basic MBGD the learning rate, α, remains constant. It is constant over time, i.e. the learning rate stays the same as the number of iterations increase and we repeatedly apply our parameter update rule. It is also constant over the parameter space, i.e. every parameter is updated using the same learning rate.

Further improvements in our parameter update rule make it so that the learning rate is not constant over time, i.e. the learning rate changes as the number of iterations increase. And that it is not constant over the parameter space, i.e. every parameter is updated using an individually adapted learning rate.

3.3 Optimization Algorithms

3.3.1 Classical Momentum [17, 18, 25]

The classical momentum algorithm computes an exponentially weighted moving average (EWMA) of gradients, m_t, and then uses this average to decide our direction of descent. The primary benefit of momentum is that it takes into account previous gradients and so prevents large oscillations in valley-like areas of the loss surface and subsequently increases the speed in direction of the decent.

Let m_t be an EWMA of our gradients,

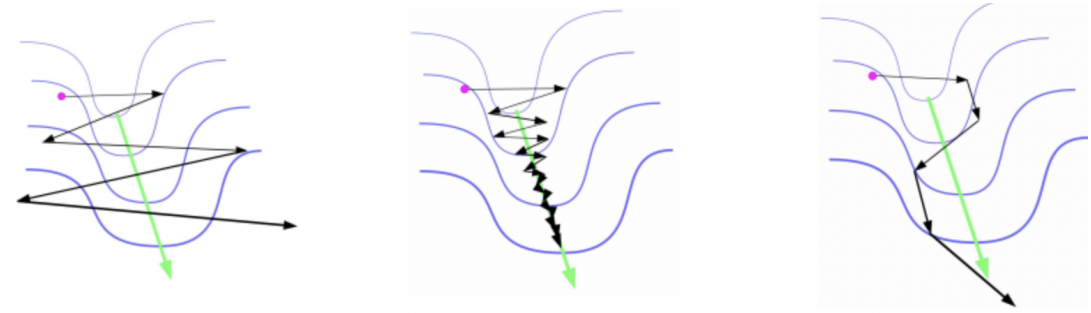
$$m_t = \beta m_{t-1} + (1 - \beta)\nabla h(\theta_t). \tag{3.11}$$

We then have our parameter update rule for classical momentum as,

$$\theta_{t+1} = \theta_t - \alpha m_t. \tag{3.12}$$

For instance, assume that we are the top of a valley as in Fig.3.1a. When we move in the direction of steepest descent we may take an overly large step that brings us up to the other side of the valley. Our next step will then move in the new direction of steepest descent and take us back to the previous side. This process will repeat and rather than getting to the bottom of the valley and then moving along the base we may oscillate up and down the sides. If we were to take a smaller step size (smaller α) as in Fig.3.1b we may be able to prevent these oscillations from occurring but the smaller step sizes will then cause us to move very slowly.

To account for both of these issues—preventing oscillations and preventing too small of a step size—we use the momentum algorithm. The momentum algorithm takes into account previous gradients m_t and prevents us from oscillating while maintaining our learning rate parameter, α. For instance, assume that we are at the top of a valley as in Fig.3.1c. Although the first step we take may bring us up the other side, the next step we take does not take us as far back up the previous side of the valley. Over time the oscillations are dampened out. This is because while previously we were only moving in the direction of our current gradient, $\nabla h(\theta_t)$, we are now moving in the direction of our current gradient plus a weighted sum of our previous gradients, m_t. The knowledge of these previous gradients ensures that we do not oscillate as much and hence we converge



(a) Gradient descent with large learning rate

(b) Gradient descent with small learning rate

(c) Gradient descent with momentum

Figure 3.1

more quickly to the base of the valley. If there is shallow decline at the base then we will we will also begin to move more quickly along it due to the accumulated gradients in the same direction.

The concept of momentum is commonly used throughout the remaining algorithms but is rarely implemented by itself.

3.3.2 Nesterov Momentum [25]

Nesterov momentum, is a variant of the classical momentum method. The difference between the two is where the current gradient, ∇h, is evaluated in the momentum step (3.11). In Nesterov momentum the gradient is evaluated with the previous momentum term, m_{t-1}, already applied.

Let us look at the momentum equation and parameter update rule from classical momentum and let us expand the parameter update rule,

$$\begin{aligned} m_t &= \beta m_{t-1} + (1-\beta)\nabla h(\theta_t). \\ \theta_{t+1} &= \theta_t - \alpha m_t \\ &= \theta_t - \alpha(\beta m_{t-1} + (1-\beta)\nabla h(\theta_t)) \\ &= \theta_t - \underbrace{\alpha\beta m_{t-1}}_{(*)} - \underbrace{\alpha(1-\beta)\nabla h(\theta_t)}_{(**)}. \end{aligned}$$

From this expansion we can see that we step in the direction of the previous momentum vector, (*), and a step in the direction of our current gradient vector, (**). Recall that both $\alpha \in (0,1)$ and $\beta \in (0,1)$ are real valued scalars with beta normally being assigned a larger value than alpha.

This expansion shows that we update our parameter value, θ_{t+1}, using $(*)$ and $(**)$. Let us focus only on $(*)$. Notice that m_{t-1} has already been computed in a previous iteration step. In Nesterov Momentum we use this $(*)$ to update where we take our gradient from during our m_t calculation,

$$m_t = \beta m_{t-1} + (1-\beta)\nabla h(\theta_t - \beta m_{t-1}). \tag{3.13}$$

This allows us a small glimpse of the future direction of θ_{t+1}. By doing so earlier in the iteration procedure it attempts to add a correction factor to the standard method of momentum.

We then have our parameter update rule for Nesterov momentum,

$$\theta_{t+1} = \theta_t - \alpha m_t. \tag{3.14}$$

3.3.3 AdaGrad [7]

The AdaGrad algorithm is the first of the so called 'adaptive learning' algorithms. While previously the same learning rate, α, was used for all parameters it is now adapted individually for all parameters. This ensures that we are not taking the same step size in every parameter direction. It does this by dividing the learning rate by the square root of past squared gradients; gradients that relate to only the specific parameter being updated.

First we create a sum of squared gradients,

$$a_t = a_{t-1} + \nabla h(\theta_t)^2. \tag{3.15}$$

We then divide our learning parameter by the square root of this to get our parameter update rule for AdaGrad,

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{a_t + \epsilon}} \odot \nabla h(\theta_t). \tag{3.16}$$

We add an epsilon, ϵ, in the denominator to ensure that we do not divide by zero.

Here the operations are applied element wise (Hadamard product). That is, for arbitrary vectors of the same size, u and v, v^2 has entries v_i^2, $\frac{1}{\sqrt{v}}$ has entries $\frac{1}{\sqrt{v_i}}$, and $u \odot v$ has entries $u_i v_i$.

The spaces we are optimizing over are high dimensional spaces. We may successfully optimize in one direction of the space while failing to optimize in another direction of the space. If the learning rate was only to decrease over time, for example $\alpha_t = \frac{\alpha_0}{\sqrt{t+\epsilon}}$, then we may successfully optimise a specific parameter, w_1, but fail to optimise a different parameter, w_2; the step size may have become too small to continue successful minimisation in the direction of w_2.

For this reason it is not enough to simply decrease the learning rate, α, over time. We must also adapt it to each parameter individually. We do this by dividing the learning rate by the square root of the past gradients—the gradients associated with that parameter. If the gradients for a particular parameter, w_1, are repeatedly large then we may wish to have a slightly smaller learning parameter (to ensure that we do not oscillate in valleys, see section 3.3.1). While if the gradients for a different parameter, w_2, are repeatedly small then when we divide by them it will produce a larger learning rate, thus ensuring that we can move more quickly along a shallow decline.

An issue with AdaGrad is that our accumulated squared gradients, a_t, constantly grow in size. Eventually this will cause our learning rate, α, to become extremely small after a certain amount of iterations. Hence, AdaGrad is rarely used as there are now better algorithms, which we will discuss below, that have improved upon it.

3.3.4 AdaDelta [27, 28]

AdaDelta is a variant of AdaGrad that seeks to rectify the issue of a monotonically decreasing learning rate. It does this by using an EWMA of squared gradients rather then just a sum of squared gradients (3.15). It also adds a 'correcting factor' on the numerator to the make the units of the LHS of our algorithm match the units of the RHS of our algorithm. In doing so it replaces our α value and so AdaGrad is sometimes said to have no learning rate.

Let d_t be the EWMA of squared gradients,

$$d_t = \beta d_{t-1} + (1-\beta)\nabla h(\theta_t)^2.$$

This gives us an intermediate parameter update rule,

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{d_t + \epsilon}} \odot \nabla h(\theta_t). \tag{3.17}$$

In the original paper, [27], the author notes that the square root of an EWMA "effectively becomes the RMS of previous squared gradients up to time t", and so begins to refer to the EWMA as a root mean square. This is slightly confusing due to the fact that the actual calculation is not a root mean square so we will instead make use of the explicit formulas, as in [28].

At this point the authors consider that as the parameter updates, $\Delta\theta_t$, are being applied to θ, they should have matching units. To rectify this they decide to add a new term to the numerator, choosing to use the square root of an EWMA of our $\Delta\theta_t$,

$$\sqrt{c_t} = \sqrt{\beta c_{t-1} + (1-\beta)\Delta\theta_t^2}.$$

They then drop our manually chosen learning rate α to give the delta that the algorithm derives its name from,

$$\Delta\theta_t = -\frac{\sqrt{c_{t-1} + \epsilon}}{\sqrt{d_t + \epsilon}} \odot \nabla h(\theta_t).$$

This leaves us with our parameter update rule for AdaDelta,

$$\begin{aligned} \theta_{t+1} &= \theta_t - \frac{\sqrt{c_{t-1} + \epsilon}}{\sqrt{d_t + \epsilon}} \odot \nabla h(\theta_t) \\ &= \theta_t + \Delta\theta_t. \end{aligned} \tag{3.18}$$

Here the operations are applied element wise (Hadamard product). That is, for arbitrary vectors of the same size, u and v, v^2 has entries v_i^2, $\sqrt{u+\epsilon}$ has entries $\sqrt{u_i+\epsilon}$, $\frac{1}{\sqrt{v}}$ has entries $\frac{1}{\sqrt{v_i}}$, the operation $\frac{\sqrt{u}}{\sqrt{v}}$ has entires $\frac{\sqrt{u_i}}{\sqrt{v_i}}$, and $u \odot v$ has entries $u_i v_i$.

3.3.5 RMSprop [9, 28]

The RMSprop algorithm is another variant of AdaGrad that adapts the learning rate individually for all parameters. It does this by dividing the learning rate by the square root

of an EWMA of past squared gradients rather than an ever increasing sum of gradients. Thereby preventing the rapidly decreasing learning rate that occurs in AdaGrad. In this way it is actually the exact same as AdaDelta at (3.17). It simply forgoes the process of matching units that takes place in AdaDelta. Hence, we have an EWMA of Squared Gradients,

$$r_t = \beta r_{t-1} + (1-\beta)\nabla h(\theta_t)^2. \tag{3.19}$$

Which then gives us our parameter update rule for RMSprop,

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{r_t + \epsilon}} \odot \nabla h(\theta_t). \tag{3.20}$$

Here the operations are applied element wise (Hadamard product). That is, for arbitrary vectors of the same size, u and v, v^2 has entries v_i^2, $\sqrt{u+\epsilon}$ has entries $\sqrt{u_i+\epsilon}$, $\frac{1}{\sqrt{v}}$ has entries $\frac{1}{\sqrt{v_i}}$, the operation $\frac{\sqrt{u}}{\sqrt{v}}$ has entires $\frac{\sqrt{u_i}}{\sqrt{v_i}}$, and $u \odot v$ has entries $u_i v_i$.

3.3.6 Adam [12]

Adam combines the improvements discussed in previous algorithms into a single algorithm. It replaces the gradient term, $\nabla h(\theta_t)$, with an EWMA of the gradient (3.11),

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla h(\theta_t). \tag{3.21}$$

It also divides the learning rate by the square root of an EWMA of squared gradients (3.19),

$$r_t = \beta_2 r_{t-1} + (1-\beta_2)\nabla h(\theta_t)^2. \tag{3.22}$$

So, it is most easily understood as a combination of classical momentum and RMSprop. It uses bias corrections to the estimates of both the momentum term and the RMSprop term to account for their initialisation at the origin, i.e. $m_0 = 0$ and $r_0 = 0$. So we get,

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \tag{3.23}$$

$$\hat{r}_t = \frac{r_t}{1-\beta_2^t}. \tag{3.24}$$

This is different to the previous discussions of classical momentum and RMSprop where no bias corrections were implemented. Note that we make use of a β_1 and a β_2 for (3.23) and (3.24) respectively, as we may choose different decay rates for each. This then gives us our parameter update rule,

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{r}_t} + \epsilon}\hat{m}_t. \tag{3.25}$$

Note that the ϵ term in the denominator is in this is case placed outside the square root. While placing it inside the square root would also work it has been found in practice that it works slighter better when placed outside.

The name Adam derives from 'adapted moment estimation'. The (3.21) term is commonly referred to as the first moment and the (3.22) term is commonly referred to as the second moment. The 'adapted' may refer to the adapted learning rate or the fact that (3.21) and (3.22) are adapted by the bias-correction.

Adam has been show to work well across a wide range of deep learning architectures and is currently the most popularly used optimization algorithm. In general, Adam is deemed to be robust to the choice of hyperparameters, though the learning rate may need to be changed from the default.

3.3.7 AdaMax [12]

The AdaMax algorithm is a variant of Adam that makes use of infinity norms. Instead of dividing by the square root of an EWMA of squared gradients, which can be understood as a scaled L^2 norm of gradients, we will now extend this to an infinity norm, L^∞, of gradients.

An L^p norm of a vector $x = (x_1, x_2, \ldots, x_n)$ is defined as,

$$||x||_{L^p} := \left(\sum_{i=1}^{n} |x_i|^p \right)^{\frac{1}{p}}, \tag{3.26}$$

where $p \geq 1$ is a real number.

The square root of the EWMA of squared gradients can be considered a scaled L^2 norm,

$$\sqrt{r_t} = \sqrt{\beta_2 r_{t-1} + (1-\beta_2)\nabla h(\theta_t)^2} = \left((1-\beta_2) \sum_{i=1}^{t} \beta_2^{t-i} \cdot |\nabla h(\theta_i)|^2 \right)^{1/2}.$$

We can generalize this to an L^p norm,

$$\sqrt{r_t} = \sqrt{\beta_2 r_{t-1} + (1-\beta_2)\nabla h(\theta_t)^p} = \left((1-\beta_2) \sum_{i=1}^{t} \beta_2^{t-i} \cdot |\nabla h(\theta_i)|^p \right)^{1/p}.$$

For large values of p this tends to make the parameter update rule unusable, however, if we let $p \to \infty$ then a usable algorithm occurs. Let $p \to \infty$ and define $u_t = \lim_{p\to\infty}(r_t)^{\frac{1}{p}}$, then,

$$
\begin{aligned}
u_t &= \lim_{p\to\infty} (r_t)^{\frac{1}{p}}, \\
&= \lim_{p\to\infty} \left((1-\beta_2) \sum_{i=1}^{t} \beta_2^{t-i} \cdot |\nabla h(\theta_i)|^p \right)^{\frac{1}{p}}, \\
&= \lim_{p\to\infty} (1-\beta_2)^{\frac{1}{p}} \left(\sum_{i=1}^{t} \beta_2^{t-i} \cdot |\nabla h(\theta_i)|^p \right)^{\frac{1}{p}}, \\
&= \lim_{p\to\infty} \left(\sum_{i=1}^{t} \beta_2^{t-i} \cdot |\nabla h(\theta_i)|^p \right)^{\frac{1}{p}}, \\
&= \max(\beta_2^{t-1}|\nabla h(\theta_1)|, \beta_2^{t-2}|\nabla h(\theta_2)|, ..., \beta_2^1|\nabla h(\theta_{t-1})|, |\nabla h(\theta_t)|).
\end{aligned}
$$

This then corresponds to the recursive formula,

$$u_t = \max(\beta_2 \cdot u_{t-1}, |\nabla h(\theta_t)|). \tag{3.27}$$

with the initial value $u_0 = 0$. In this case, there is no need to correct for initialization bias.

The parameter update rule for AdaMax is then given by,

$$\theta_{t+1} = \theta_t - \frac{\alpha}{u_t}\hat{m}_t. \tag{3.28}$$

3.3.8 Nadam [5]

The Nadam algorithm modifies Adam with a similar concept as was applied in Nesterov momentum. It replaces the bias corrected momentum term, $\hat{m}_t$, (3.23), that occurs in Adam's update rule with an expanded version which then shows that we can replace a previous time-step component, $\hat{m}_{t-1}$ for a current time-step component, $\hat{m}_t$.

Looking at the parameter update rule for Adam,

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{r}_t} + \epsilon}\hat{m}_t.$$

We first expand the $\hat{m}_t$ term,

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{r}_t} + \epsilon}\left(\frac{m_t}{1-\beta_1^t}\right).$$

Then expanding m_t,

$$
\begin{aligned}
\theta_{t+1} &= \theta_t - \frac{\alpha}{\sqrt{\hat{r}_t} + \epsilon}\left(\frac{\beta_1 m_{t-1} + (1-\beta_1)\nabla h(\theta_t)}{1-\beta_1^t}\right) \\
&= \theta_t - \frac{\alpha}{\sqrt{\hat{r}_t} + \epsilon}\left(\frac{\beta_1 m_{t-1}}{1-\beta_1^t} + \frac{(1-\beta_1)\nabla h(\theta_t)}{1-\beta_1^t}\right).
\end{aligned}
$$

Notice that m_{t-1} is the momentum vector for the previous time step and since we have already calculated the momentum vector for the current time step, m_t, we can instead replace it with this, which gives us our parameter update rule for Nadam,

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{r}_t} + \epsilon} \left(\frac{\beta_1 m_t}{1 - \beta_1^t} + \frac{(1 - \beta_1) \nabla h(\theta_t)}{1 - \beta_1^t} \right). \tag{3.29}$$

Chapter 4

Recurrent Neural Network

4.1 Recurrent Neural Network (RNN) [20, 23, 28]

Recurrent Neural Networks (RNNs) are a type of neural network architecture that are most often used for detecting patterns in sequential data.

RNNs and multi-layer perceptrons (MLPs), a type of feedforward neural network (FNNs), differ in how information passes through the network. In MLPs, information is passed through the network without any cycles; an input is passed through the network and produces an output. In RNNs, information is passed through a network that also has cycles which transmit information back to itself. This allow RNNs to take account of both the current input, $\mathbf{X}_t$, the previous inputs, $\mathbf{X}_{0:t-1}$.

Let $\mathbf{H}_t \in \mathbb{R}^{n\times h}$ be the hidden state, $\mathbf{X}_t \in \mathbb{R}^{n\times d}$ the input at time step t, $\mathbf{W}_{xh} \in \mathbb{R}^{d\times h}$ be a weight matrix, $\mathbf{W}_{hh} \in \mathbb{R}^{h\times h}$ be a hidden-state-to-hidden-state matrix, and $\mathbf{b}_h \in \mathbb{R}^{1\times h}$ a bias parameter, where n is number of samples, d is the number of inputs of each sample, and h is the number of hidden units. Lastly, let ϕ be an activation function which the above information is passed through.

We then have our hidden variable,

$$\mathbf{H}_t = \phi(\mathbf{X}_t\mathbf{W}_{xh} + \mathbf{H}_{t-1}\mathbf{W}_{hh} + \mathbf{b}_h), \tag{4.1}$$

and our output variable,

$$\mathbf{O}_t = \phi(\mathbf{H}_t\mathbf{W}_{ho} + \mathbf{b}_o), \tag{4.2}$$

where $\mathbf{W}_{ho} \in \mathbb{R}^{h\times o}$ is the weight and $\mathbf{b}_o \in \mathbb{R}^{1\times o}$ is the bias for the output layer.

As we can see, our hidden variable, $\mathbf{H}_t$, is a recurrence relation as it recursively includes $\mathbf{H}_{t-1}$, and hence contains information about previous hidden states.

The equivalent equations for MLPs clearly show the difference between them and RNNs. The hidden variable equation is not a recurrence relation, does not include $\mathbf{H}_{t-1}$, and contains no information about previous hidden states,

$$\mathbf{H} = \phi(\mathbf{X}\mathbf{W}_{xh} + \mathbf{b}_h). \tag{4.3}$$

Hence, our output variable does not include any information about previous hidden states either,

$$\mathbf{O} = \phi(\mathbf{H}\mathbf{W}_{ho} + \mathbf{b}_o). \tag{4.4}$$

4.2 Deep Recurrent Neural Networks (DRNNs)

The RNNs discussed so far have a single unidirectional hidden layer. We can stack multiple RNNs on top of each other to create a Deep Recurrent Neural Network (DRNN). In a DRNN, each hidden state passes to the next time step of the current layer (as in our regular RNN) and also passes to the next layer of the current time step. This creates a flexible mechanism in which different forms of information may be held at different levels in the stack. High-level data at the top and low-level data at the bottom.

Let $\mathbf{X}_t \in \mathbb{R}^{n\times d}$ be a minibatch input at time step t, $\mathbf{H}_t^{(l)} \in \mathbb{R}^{n\times h}$ be the hidden state of the l^{th} hidden layer ($l = 1, ..., L$), $\mathbf{O}_t \in \mathbb{R}^{n\times o}$ be the output layer variable, and ϕ_l be the activation function at the l^{th} hidden layer, where n is the number of examples, d is the number of inputs for each example, h be the number of hidden units, o the number of outputs. Also, let $\mathbf{W}_{xh}^{(l)} \in \mathbb{R}^{h\times h}$ and $\mathbf{W}_{hh}^{(l)} \in \mathbb{R}^{h\times h}$ be the weights, and $\mathbf{b}_h^{(l)} \in \mathbb{R}^{1\times h}$ the bias for the l^{th} hidden layer. Then the hidden state of the 0^{th} and l^{th} hidden layer is given by,

$$\begin{aligned} \mathbf{H}_0^{(l)} &= \mathbf{X}_t, \\ \mathbf{H}_t^{(l)} &= \phi_l(\mathbf{H}_t^{(l-1)}\mathbf{W}_{xh}^{(l)} + \mathbf{H}_{t-1}^{(l)}\mathbf{W}_{hh}^{(l)} + \mathbf{b}_h^{(l)}). \end{aligned} \tag{4.5}$$

The output layer is based only on the hidden state of the final L^{th} hidden layer,

$$\mathbf{O}_t = \phi(\mathbf{H}_t^{(L)}\mathbf{W}_{ho} + \mathbf{b}_o), \tag{4.6}$$

where $\mathbf{W}_{hq} \in \mathbb{R}^{n\times o}$ is weight and $\mathbf{b}_o \in \mathbb{R}^{1\times o}$ is the bias of the output layer.

The number of hidden layers and the number of hidden units are hyperparameters that can be tuned or specified just as in a standard MLP.

Chapter 5

Experiment

5.1 Electroencephalography (EEG)

Electroencephalography (EEG) is a method for recording electrical activity between neurons in the cortex of the brain. Electrodes are placed on the scalp where they record changes in voltage that are representative of electrical activity occurring in the cortex that has passed through the skull.

The particular EEG device used in this study is an Emotiv Epoc Flex with 34 Ag/AgCl (silver-silver chloride) electrodes. This devices has a sample rate of 128 SPS (samples per second) at a frequency range of 0.16–43 Hz.

5.2 Meditation

Meditation refers to a wide and diverse collection of practices that can be succinctly summarised as methods that are used to navigate the subjective experience of consciousness.

In this study, three different meditation practices were engaged with: Breath, Body, Gratitude. The subjective experience of the meditator differs during each of these meditation types. Although the inner subjective experience may differ, from the perspective of an external observer the outward behaviour of the meditator may seem relatively similar. The fact that the subjective expererience of the meditator differs points to the liklihood that there are distinct patterns of neural activity occurring in conjunction with these different states of consciousness. By recording the neural activity that occurs during these practices–in this case EEG signals–we can then input these signals into an artifical neural network to see if it can find patterns in the data that can distinguish the meditation practices from one another. Thereby providing evidence that there are distinct processes occurring in the brain in relation to these meditative states.

These three meditative states are the three distinct classes in this multi-class classification problem.

5.3 Recording

An electroconductive gel, EasyCap Abralyt HiCl, is spread underneath each electrode to improve conductivity. The area of the scalp beneath each electrode is cleaned with

disinfectant alcohol before application of the gel. This is done to remove any dead skin cells or grease, which may disrupt the recording signal. Electrodes are placed according to the International 10-20 system.

A recording session lasts a total of 36mins. Each meditation practice lasts for 12mins upon which a bell sounds that notifies the participant to move into the next meditation practice. In this way, 3 meditation types are practiced in a single recording session ($3 \times$ 12mins = 36mins).

5.4 Data

5.4.1 Recorded Data

The data was recorded on the Emotiv Pro software.

5.4.2 Exported Data

The recorded session is exported from the Emotiv Pro software program as a Comma Separate Values (CSV) file. These files contain a column for each electrode specifying the change in voltage over time, along with a number of other extraneous columns that are removed during preprocessing. There are 128 SPS which corresponds to 128 rows in the CSV file. Hence the CSV file contain a large number of rows, approximately 276,480 (36mins $\times$ 60sec $\times$ 128samples).

5.5 Preprocessing

The exported CSV file contains the raw data of the recording. This raw data must be transformed into the appropriate format for use in the analysis. The extraneous columns are removed from the file, leaving only the 31 electrodes that have the voltage information (one electrode was dropped pre-recording). The data is searched for any errors such as NaN values (cells which contain no value) and these are then removed. Each row of data belongs to one of the meditation types so an extra column is added that contains the different meditation type names. This is referred to as the the **label** column and the values as *labels*. The first 60 seconds and the last 60 seconds of each meditation type is deleted so that the transition between meditation types is not included. This leaves approximately 10mins of data per meditation type. The feature values are normalized before being input into the network, to be in-between -1 and 1, as is common practice.

5.6 Train, Validation, Test Sets

The data must then be split into training, validation, and testing sets. Each set contains a balanced number of each meditation type.

The training set is used to train the model; it is the set from which the model learns to distinguish patterns that may be useful in differentiating–classifying–the different meditation types. The validation set is used during the training process to provide validation that the model is learning patterns that are general instead of just memorizing the training

set. The model attempts to predict the correct labels for the validation set based off of what it has learnt from the training set.

This provides real time feedback on how well the model is learning and allows for the output of metrics that can be used to stop the model during the learning process, i.e. validation loss and validation accuracy can be used to determine whether to stop training the model early because it is already of sufficient quality or because it it does not seem to be improving.

The test set is then used to compare the trained model against an unknown dataset. The model attempts to predict the correct labels for the test set based off of the patterns that it has learnt. The data that the test set contains has not been used in training the model so a high prediction accuracy for the test set provides evidence that model has learnt general patterns related to each label rather than patterns specific to the training set.

5.7 Model Architecture

The model is trained on a specific configuration—model architecture—of a neural network. There are a wide variety of model architectures that can be chosen with many hyperparameters that can be manually tuned. In theory, there is no perfect configuration of a neural network, so in practice, different configurations must be implemented in order to discover what works best.

Due to time and computational constraints a specific choice of model architecture and hyperparameter configuration was needed to adequately compare the different optimization algorithms. The model architecture chosen for comparison of the optimization algorithms is shown below:

Hyperparameters	
Data Structure	
Window Size	256
Step Size	32
Batch Size	32
Network	
Neural Nework	LSTM
Nbr. of Layers	1
Nbr. of Hidden Units	16
Reccurent Dropout	0.4
Compile	
Loss Function	Categorical Cross Entropy
Learning Rate	0.005
Nbr. of Epochs	50

Figure 5.1: Model architecture of recurrent neural network used for comparing the optimzation algorithms with the values chosen for various hyperparamters

Chapter 6

Results

In this section, we display the results for the analysis described in the previous section. An LSTM recurrent neural network was run using each optimizer algorithm with the model architecture described in section 5.7. The results of these analyses are shown using a selection of plots and tables: a prediction accuracy plot for the training dataset and the testing dataset–displaying the model's ability to accurately predict the correct meditation class during learning; a loss plot for the training dataset and the testing dataset–displaying the value output by the loss function during learning; a best performance table providing the accuracy, loss, and epoch values related to the top prediction accuracy result; and a confusion matrix giving the normalized prediction accuracy per meditation class.

There is also a comparison of the optimization algorithms and an example of how changing even a single hyperparameter of the model architecture can produce significantly different results.

6.1 Analysis 1 - Classical Momentum

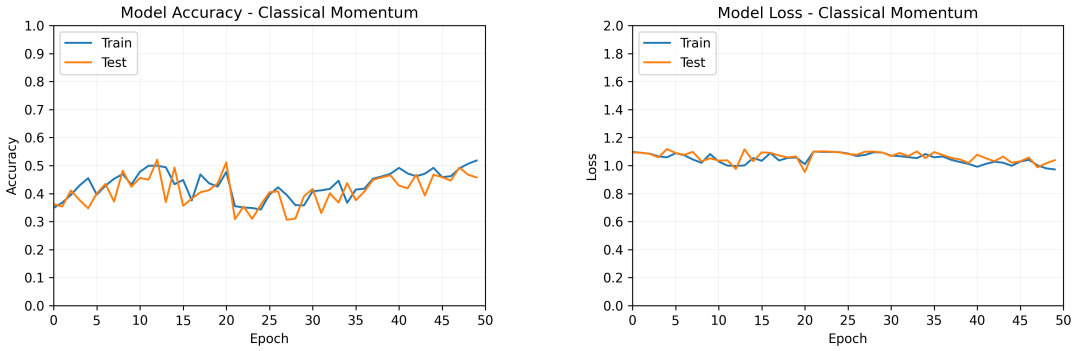


Figure 6.1: Accuracy plot (left figure) and loss plot (right figure) for classical momentum displaying learning trajectories for the training dataset (labelled Train) and the testing dataset (labelled Test) during model learning over a duration of 50 epochs. The train and test curves converge with one another in both the accuracy and loss plots, hence, there is no overfitting of the train data. However, there is barely any improvement in performance between the start and end as can be seen from the relatively static behaviour of the loss and accuracy curves.

Classical Momentum	Value
Accuracy	52%
Loss	0.9756
Epoch	13

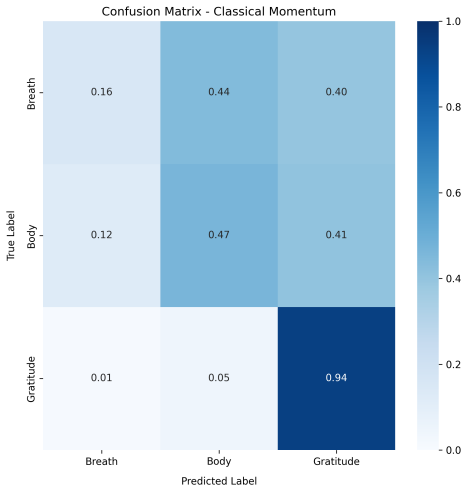


Figure 6.2: Best performance values (left figure) and confusion matrix (right figure) for classical momentum—based on the testing dataset—displaying the accuracy/loss/epoch values related to the top prediction accuracy result and the normalized prediction accuracy per meditation class, respectively. The classical momentum optimizer achieved a top overall accuracy of 52% with a loss of 0.98 at epoch 13. The confusion matrix shows that the model classified gratitude correctly 94% of the time. But it was poor at classifying breath and body, correctly distinguishing them only 16% and 12% of the time, respectively.

6.2 Analysis 2 - Nesterov Momentum

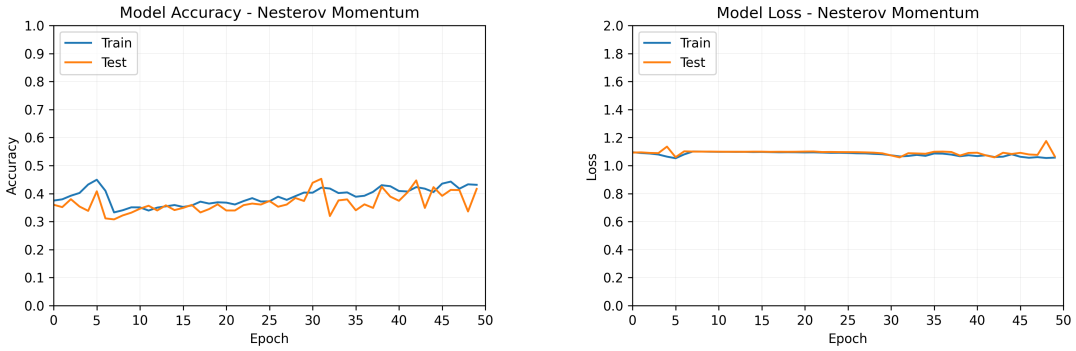


Figure 6.3: Accuracy plot (left figure) and loss plot (right figure) for Nesterov momentum displaying learning trajectories for the training dataset (labelled Train) and the testing dataset (labelled Test) during model learning over a duration of 50 epochs. The train and test curves, once again, converge with one another in both the accuracy and loss plots. There is little improvement in performance as loss and accuracy curves exhibit relatively static behaviour.

Nesterov Momentum	Value
Accuracy	45%
Loss	1.06
Epoch	32

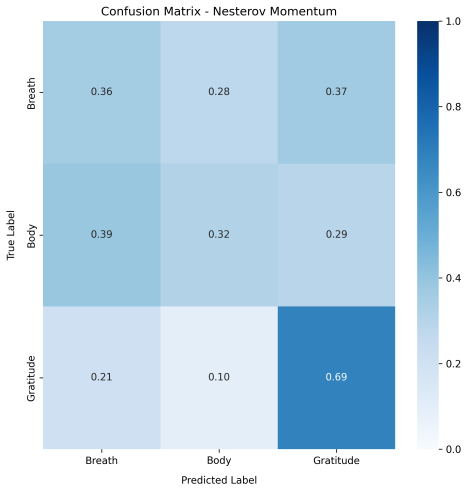


Figure 6.4: Best performance values (left figure) and confusion matrix (right figure) for Nesterov momentum—based on the testing dataset—displaying the accuracy/loss/epoch values related to the top prediction accuracy result and the normalized prediction accuracy per meditation class, respectively. The Nesterov momentum optimizer achieved a top overall accuracy of 45% with a loss of 1.06 at epoch 32. The confusion matrix shows that the model classified gratitude correctly 69% of the time. It was close to random at classifying breath and body, achieving 36% and 39% respectively.

6.3 Analysis 3 - AdaGrad

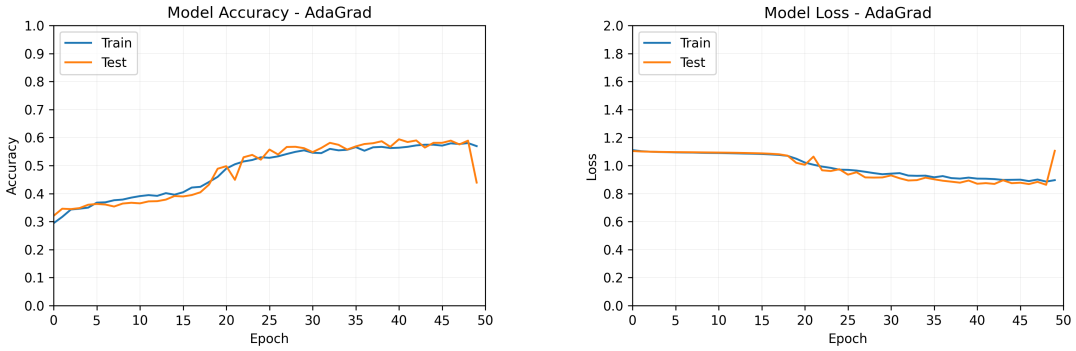


Figure 6.5: Accuracy plot (left figure) and loss plot (right figure) for AdaGrad displaying learning trajectories for the training dataset (labelled Train) and the testing dataset (labelled Test) during model learning over a duration of 50 epochs. The train and test curves converge with another implying that overfitting of the train set was not occurring. There is a steady, if slow, increase in the accuracy for most of the learning and a similar decrease in the loss; this behaviour provides some evidence that AdaGrad is an improvement upon classical momentum and Nesterov momentum.

AdaGrad	Value
Accuracy	59%
Loss	0.87
Epoch	41

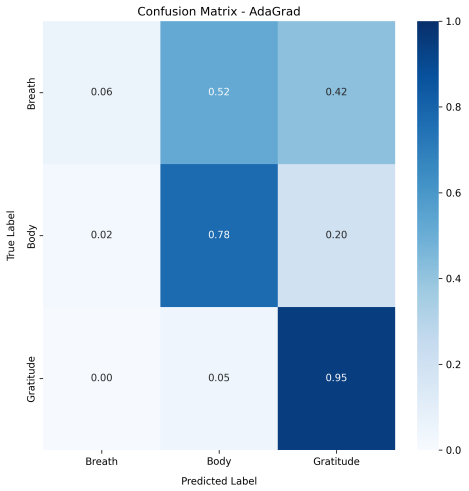


Figure 6.6: Best performance values (left figure) and confusion matrix (right figure) for AdaGrad—based on the testing dataset—displaying the accuracy/loss/epoch values related to the top prediction accuracy result and the normalized prediction accuracy per meditation class, respectively. The AdaGrad algorithm achieved a top overall accuracy of 59% with a loss of 0.87 at epoch 41. The confusion matrix provides some further evidence of this, as gratitude is classified correctly 95% of the time, and body is classified correctly 78% of the time. Breath is classified correctly only 6% of the time, however, overall the model appears to be improved by being able to distinguish between two out of three meditations.

6.4 Analysis 4 - AdaDelta

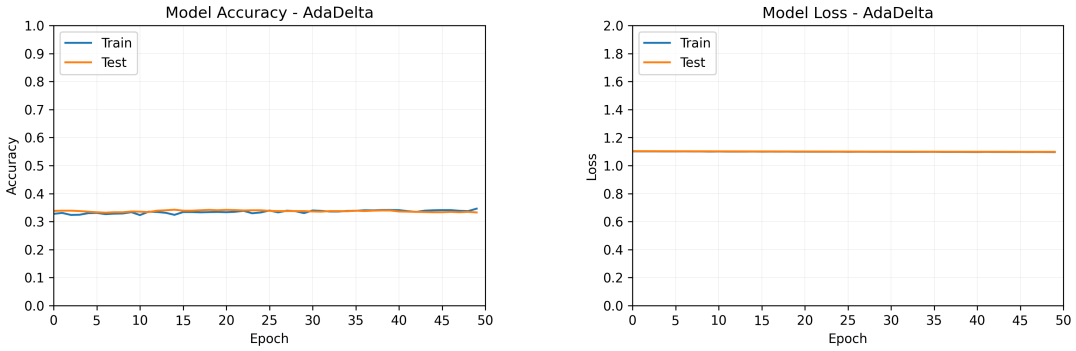


Figure 6.7: Accuracy plot (left figure) and loss plot (right figure) for AdaDelta displaying learning trajectories for the training dataset (labelled Train) and the testing dataset (labelled Test) during model learning over a duration of 50 epochs. The accuracy and loss appears to remain static throughout the learning.

AdaDelta	Value
Accuracy	34%
Loss	1.10
Epoch	15

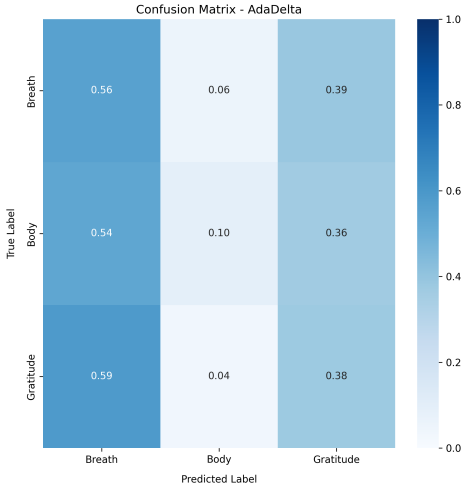


Figure 6.8: Best performance values (left figure) and confusion matrix (right figure) for AdaDelta—based on the testing dataset—displaying the accuracy/loss/epoch values related to the top prediction accuracy result and the normalized prediction accuracy per meditation class, respectively. The AdaDelta algorithm achieved a top overall accuracy of 34% with a loss value of 1.10 at epoch 15. However, these particular values are representative of the entirety of the model, as the accuracy and loss appears to remain static throughout the learning. The confusion matrix shows that the model classified breath, body, gratitude with 56%, 10%, 38% accuracy, respectively. So, it may be that the model learnt patterns for breath more easily and hence recognised it throughout the model, as body is classified as breath 54% of the time and gratitude classified as breath 59% of the time.

6.5 Analysis 5 - RMSprop

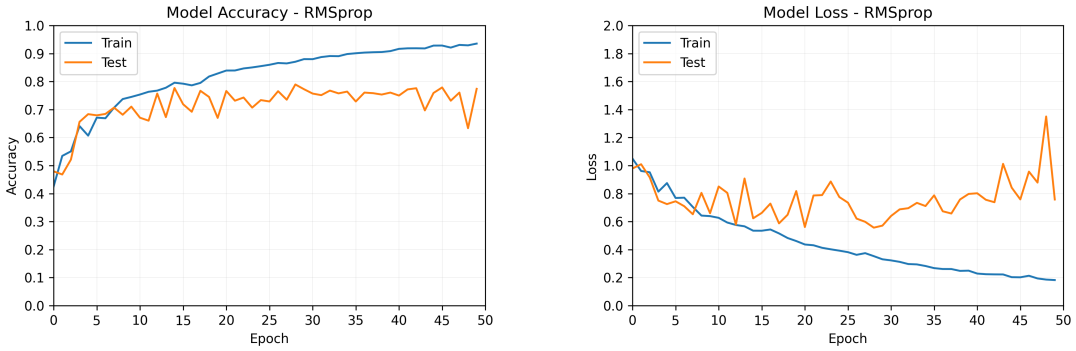


Figure 6.9: Accuracy plot (left figure) and loss plot (right figure) for RMSprop displaying learning trajectories for the training dataset (labelled Train) and the testing dataset (labelled Test) during model learning over a duration of 50 epochs. The train and test curves in both the accuracy and loss plots begin to diverge after approximately 10-15 epochs. The train set steadily improves in both accuracy and loss, while the test set stops improving. Overfitting of the train set is likely occurring. Yet, even before this, RMSprop is already more successful than previous optimizers.

RMSprop	Value
Accuracy	79%
Loss	0.56
Epoch	29

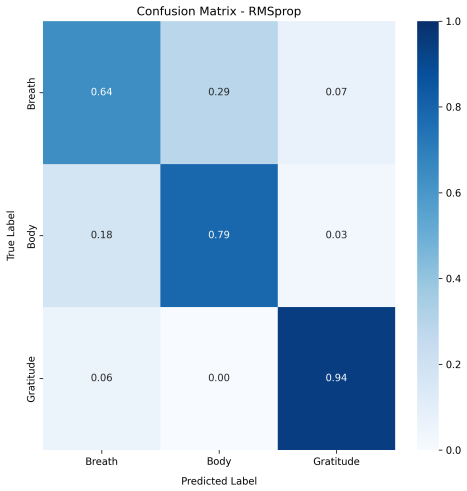


Figure 6.10: Best performance values (left figure) and confusion matrix (right figure) for RMSprop—based on the testing dataset—displaying the accuracy/loss/epoch values related to the top prediction accuracy result and the normalized prediction accuracy per meditation class, respectively. The RMSprop algorithm achieved a top overall accuracy of 79% with a loss of 0.56 at epoch 29. The confusion matrix shows that the model classified breath, body, gratitude with 64%, 79%, 94% accuracy, respectively. This is a marked improvement upon previous algorithms and begins to provide evidence that the network may be able to successfully distinguish between meditations.

6.6 Analysis 6 - Adam

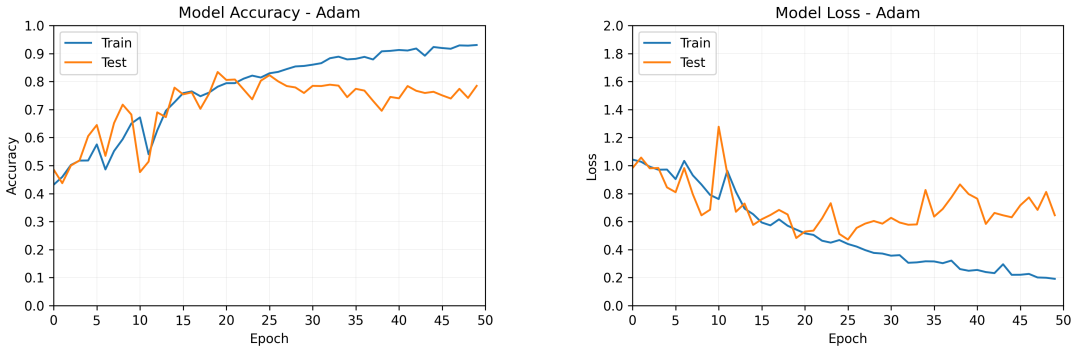


Figure 6.11: Accuracy plot (left figure) and loss plot (right figure) for Adam displaying learning trajectories for the training dataset (labelled Train) and the testing dataset (labelled Test) during model learning over a duration of 50 epochs. The train and test curves in both the accuracy and loss plots begin to diverge after approximately 25 epochs–displaying overfitting of the train set. However, before they diverge Adam has already achieved the best results thus far.

Adam	Value
Accuracy	83%
Loss	0.48
Epoch	20

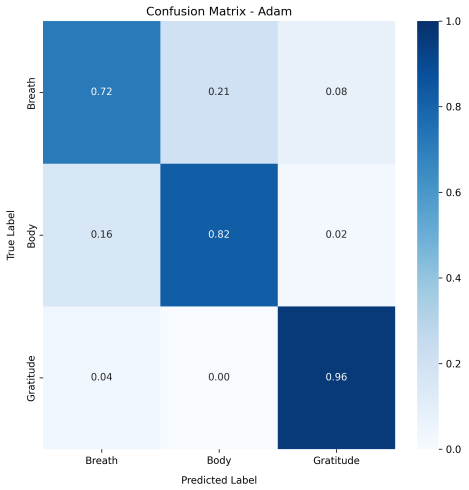


Figure 6.12: Best performance values (left figure) and confusion matrix (right figure) for Adam—based on the testing dataset—displaying the accuracy/loss/epoch values related to the top prediction accuracy result and the normalized prediction accuracy per meditation class, respectively. The Adam algorithm achieved a top overall accuracy of 83% with a loss of 0.48 at epoch 20. The confusion matrix shows that the model classified breath, body, gratitude with 72%, 82%, 96% accuracy, respectively. These are the best overall accuracy scores per meditation and provide further evidence that the network may be able to successfully distinguish between meditations with high accuracy.

6.7 Analysis 7 - AdaMax

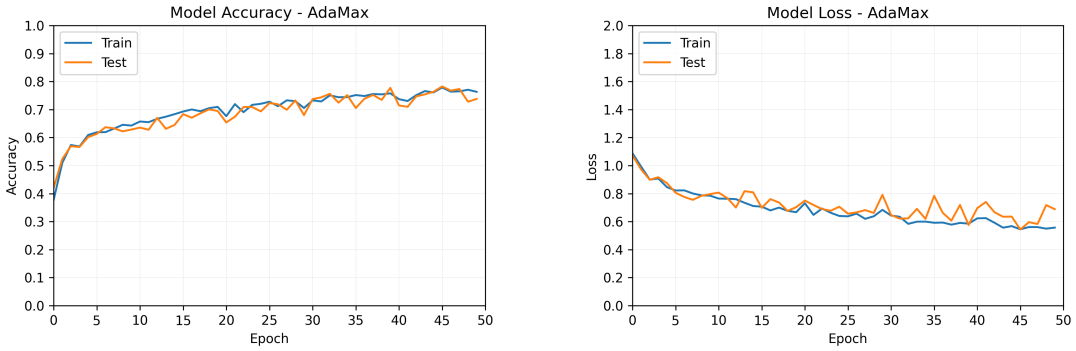


Figure 6.13: Accuracy plot (left figure) and loss plot (right figure) for AdaMax displaying learning trajectories for the training dataset (labelled Train) and the testing dataset (labelled Test) during model learning over a duration of 50 epochs. The train and test curves converge with one another in both the accuracy and loss plots. There is a steady improvement in accuracy and loss over the epochs, although the rate of improvement appears to be slowing near the end. It may be that increasing the number of epochs would improve the model.

AdaMax	Value
Accuracy	78%
Loss	0.54
Epoch	46

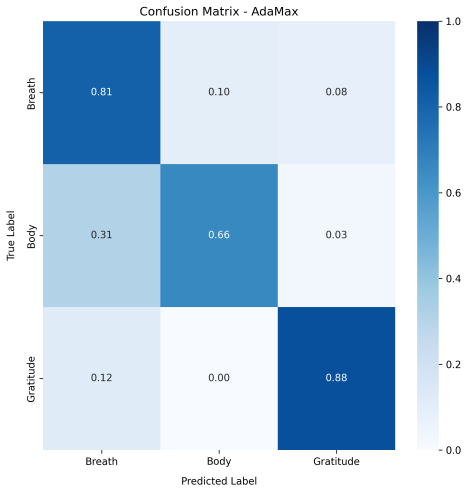


Figure 6.14: Best performance values (left figure) and confusion matrix (right figure) for AdaMax—based on the testing dataset—displaying the accuracy/loss/epoch values related to the top prediction accuracy result and the normalized prediction accuracy per meditation class, respectively. The AdaMax algorithm achieved a top overall accuracy of 78% with a loss of 0.54 at epoch 46. The confusion matrix shows that the model classified breath, body, gratitude with 81%, 66%, 88% accuracy, respectively. This is the best result for breath, but body and gratitude do not achieve as high an accuracy as in RMSprop and Adam.

6.8 Analysis 8 - Nadam

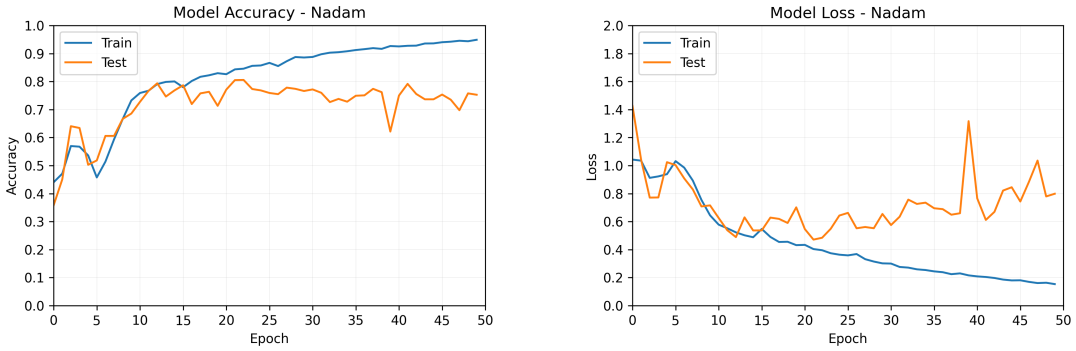


Figure 6.15: Accuracy plot (left figure) and loss plot (right figure) for Nadam displaying learning trajectories for the training dataset (labelled Train) and the testing dataset (labelled Test) during model learning over a duration of 50 epochs. The train and test curves for both accuracy and loss plots begin to diverge after approximately 15epochs–displaying overfitting of the train set. These plots are very similar to the plots for RMSprop and Adam and the top overall accuracy and loss values are comparable.

Nadam	Value
Accuracy	80%
Loss	0.48
Epoch	23

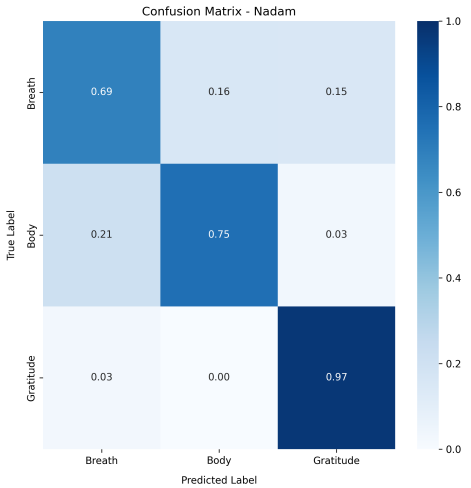


Figure 6.16: Best performance values (left figure) and confusion matrix (right figure) for Nadam—based on the testing dataset—displaying the accuracy/loss/epoch values related to the top prediction accuracy result and the normalized prediction accuracy per meditation class, respectively. The Nadam optimizer achieved a top overall accuracy of 80% with a loss of 0.48 at epoch 23. The confusion matrix shows that the model classified breath, body, gratitude with 69%, 75%, 97% accuracy, respectively. Once again, these are comparable to the values of RMSprop and Adam.

6.9 Comparison of Optimizers

Analysis	Optimizer	Accuracy	Loss	Epoch
1	Classical Momentum	52%	0.98	13
2	Nesterov Momentum	45%	1.06	32
3	AdaGrad	59%	0.87	41
4	AdaDelta	34%	1.10	15
5	RMSprop	79%	0.56	29
6	Adam	83%	0.48	20
7	AdaMax	78%	0.54	46
8	Nadam	80%	0.48	23

Figure 6.17: Table depicting best performance values—based on the testing dataset—per optimizer. Shows the accuracy/loss/epoch values related to the top prediction accuracy result.

The best performing algorithm was Adam with a top accuracy of 83%, a loss of 0.48, at epoch 20. This is the most widely used and common choice of optimizer throughout neural network models and so it is not unexpected that it provides the best performance. As discussed in earlier section 3.3.6, Adam makes use of an adaptive gradient, an adaptive learning rate, and combines benefits from previous algorithms into a single optimizer.

RMSprop, AdaMax, and Nadam follow close behind in model performance. The first four algorithms perform surprisingly poorly. However, by re-running the analyses with recommended and default learning rates for the various optimizers we get significantly different results.

Analysis	Optimizer	Accuracy	Loss	Epoch	Learning Rate
1	Classical Momentum	81%	0.46	49	0.01
2	Nesterov Momentum	69%	0.67	46	0.01
3	AdaGrad	82%	0.56	26	1.00
4	AdaDelta	82%	0.52	45	1.00
5	RMSprop	83%	0.52	50	0.001
6	Adam	67%	0.72	26	0.001
7	AdaMax	63%	0.75	46	0.001
8	Nadam	79%	0.57	42	0.001

Figure 6.18: Table depicting best performance values—based on the testing dataset—per optimizer with specific learning rates. Shows the accuracy/loss/epoch values related to the top prediction accuracy result.

Results improve across all algorithms except for Adam, AdaMax, and Nadam. The first four algorithms are improved greatly–with classical momentum, AdaGrad, and AdaDelta now rivalling the top performance of Adam from the previous analyses. Classical momentum improves by 31%, Nesterov Momentum by 24% and AdaGrad by 23%, and AdaDelta by 48%. Notably, AdaDelta has over doubled in performance. RMSprop is now equivalent to the top accuracy of Adam at 83% with a small improvement of 3% (thought slightly higher in loss value). Adam and AdaMax have actually decreased in performance, by 16% and 15% to become the worst choice of algorithms while Nadam has remained relatively similar with a decrease of only 1%.

These re-analyses provide an example of the importance of the choice of values for hyperparameters and the necessity for tuning of these hyperparameters during training. The model architecture chosen has a decisive impact on model performance and changing even a single hyperparameter within this architecture can have consequential effects for

the results. Therefore, it is necessary to run multiple model configurations on a single problem in order to find out what the optimal configuration is.

Depending on the number of epochs chosen we may see divergent behaviour between training and testing datasets—overfitting—or we may see continual convergent behaviour between them. This is of particular interest in cases where the model is still improving in performance when the it finishes. Classical momentum, AdaDelta, RMSprop, and AdaMax (see Fig.6.18) all look promising in this regard as we can see that their best accuracy occurs near the end of the model (and they have a high accuracy result) so it is possible that by simply extending the length of training—by increasing the number of epochs—that better results may be attained.

In this thesis, a standard configuration was chosen to compare and contrast optimization algorithms with as many of the hyperparameters as possible held constant. The consequence of this is that the specific implementation of the optmizers may not be optimally adapted to the problem. In practice, it is necessary to adapt the optimizer individually to the task at hand, along with the rest of the hyperparameters in the model architecture.

Overall, with further time and computational resources–and an adequate search of the hyperparameter space–one can hope to further improve model performance.

Chapter 7

Conclusion

This thesis has documented the mathematical underpinnings of eight of the most commonly used optimisation algorithms in neural networks. These algorithms use a variety of techniques to improve upon the basic method of gradient descent. As most machine learning problems have very many parameters with many dimensions over which to optimise, these algorithms are highly important in the functioning of these networks.

We see how incremental improvements in algorithmic architecture have progressed to make general optimizers that are successful across a wide range of applications. Beginning with the decision to implement an exponentially weighted moving average of gradients to act as a mechanism to prevent excessive oscillation in the loss landscape and the ability to move more quickly along areas of shallow decline. Methods to adapt the learning rate to each parameter individually were then developed that allow individualised step sizes for each dimension in the loss landscape—dividing the learning rate by the square root of past gradients associated with each individual parameter. This ensures that the different gradients in different directions are taken into account; movement through the landscape as a whole is better optimised rather than solely movement along any specific dimension.

This per-dimension learning rate will monotonically decrease over time due to the division of past gradients and so the concept of exponentially weighted moving averages is once again implemented—dividing the learning rate by the square root of an EWMA of past gradients—to ensure that our learning rate does not become too small. We then have a per-dimensional learning rate that is also better adapted to time.

These improvements along spatial and temporal dimensions of the optimisation process provide a basis from which further algorithms are and will continue to be developed.

After dealing with the mathematical structure of these optimisation algorithms we then see how they are practically implemented within a neural network—specifically a recurrent neural network—in the context of an experimental paradigm.

Recurrent neural networks are networks that can make use of time dependent information to better learn patterns in sequential data. We see that those networks have a cyclical structure that allows information from previous time steps to be learnt in relation to information from future time steps, thereby allowing the successful learning of patterns in time series data.

By comparing these algorithms we see that they behave differently from one another and produce different results, as expected, and that their successful implementation de-

pends on an appropriate choosing of hyperparameters for the algorithms, such as their base learning rate. The place of these algorithms within the larger structure of the neural network is made evident by the significant difference in results that one algorithm may have from another. It is also made obvious that these algorithms are only one part of the whole, and that many other properties of the neural network must be appropriately chosen to achieve the best results such as the type of network, the number of hidden units, the number of layers, the number of batches, and more.

Overall, the algorithms are core components of neural networks and are essential for succeeding in the broad range of tasks that these networks can be applied to. In this case, allowing for successful classification between different classes of information, specifically, distinguishing between different meditative states recorded by EEG. This shows how the mathematical formalism underlying the design of algorithms eventually leads to their practical implementation in a diverse range of applications and scientific experiments, shown here through the the classification of conscious states.

Appendix A

Data Collection

In this Appendix A you will find information relating to data collection. There are two sections covering:

1) Electroencephalogram (EEG) apparatus and the process of physically setting up the apparatus for recording.

2) Emotiv Pro software used to record the data.

A.1 Electroencephalogram (EEG)

An electroencephalogram (EEG) is a brain-machine-interface device that is used to record electrical signalling between neurons in the neocortex. Electrodes are placed on the scalp where they record changes in voltage that are representative of electrical activity that has passed through the skull. EEGs tend to have very good temporal resolution (on the order of milliseconds), although the spatial resolution is not as precise.

The particular EEG device used in this study is an Emotiv Epoc Flex Headset with 34 Ag/AgCl (silver-silver chloride) electrodes (32 recording electrodes and 2 reference electrodes). This device has a sample rate of 128 SPS (samples per second) at a frequency range of 0.16–43 Hz. Further detailed information about this device can be found on the company's website - https://www.emotiv.com.

To set up the EEG it is necessary to connect the headset to the Emotiv Pro recording software. There is a control box that receives signals from the electrodes and transmits them wirelessly to a USB receiver connected to a computer. The software allows you to see whether the electrodes are recording the appropriate voltage information. Electrodes are placed according to the International 10-20 system.

An electroconductive gel, EasyCap Abralyt HiCl, is used to improve conductivity. The area of the scalp beneath each electrode is cleaned with disinfectant alcohol before application of the gel. This is done to remove any dead skin cells or grease, which may disrupt the recording signal. The gel is then placed on the scalp beneath each electrode by using a small syringe and spread evenly with a cotton bud. Once the electrodes are conducting the voltage appropriately then the recording session is ready to begin.

Figure A.1: Emotiv Epoc Flex gel sensor kit with 32 Ag/AgCl (silver-silver chloride) electrodes. This device has a sample rate of 128 SPS (samples per second) at a frequency range of 0.16–43 Hz. The electrodes and wires are individually thread through the cap and positioned according to the International 10-20 system. There is a control box that receives signals from the electrodes and transmits them wirelessly to a USB receiver connected to a computer.

Figure A.2: The Emotiv Epoc Flex EEG assembled and placed on the head in preparation for recording.

A.2 Emotiv Software

The Emotiv Pro software is a custom built application by the Emotiv company for interaction with their devices. It allows the data to be transmitted wirelessly from the EEG to the USB receiver to the Emotiv Pro platform. Wave forms from each electrode can be viewed in real time and activity can be recorded and stored on the platform for later exporting. Data is then exported as CSV files for analysis with other tools. Further information about the software can be found on the company's website - https://www.emotiv.com.

Appendix B

Meditation

In this Appendix B the reader will find information relating to meditation practices. It will provide brief summaries of each mediation practice that was engaged in: the instructions for each meditation and an account of why these meditative practices are experientially different. There are four sections covering four different meditation practices:

1) Breath Focus
2) Body Scan
3) Gratitude
4) Loving-kindness

These practices are, for the most part, based on Buddhist meditation practices but may also be found in other religious and contemplative disciplines.

It is common for different lineages and schools of Buddhism to use different terminology to refer to a similar practice, and the same terminology to refer to different practices. For this reason, it could be misleading to use the Pali, Sanskrit, Tibetan etc. terminology to define the practices engaged with during the study. There are Buddhist teachings that are aligned with the described practices below, such as Anapanasati (mindfulness of breathing), Kayagatsati (mindfulness of the body), Metta (benevolence, loving-kindness, friendliness, good will) but for the sake of clarity we will not say that the summaries below are a definition of such terminology; to define such terminology precisely may require much more detailed accounts of Buddhist philosophy.

For this reason, the meditations used during the experiment are described explicitly and act as the definitions for what was recorded. It is all too common that the words 'meditation' and 'mindfulness' are used in studies without appropriate definition. This can lead to confusion as to what the participant was actually doing during the recording sessions.

B.1 Breath Focus

The process of this meditation is very simple; become aware of the movement of breath. In some traditions people may focus on the abdomen or the chest, in this case the focus is on the nostrils. Pay attention to the movement of the breath inwards through the nostrils and the movement of the breath outwards through the nostrils. Attempt to pay attention for the full duration of the breath, noticing the transition between breathing in and breathing

out and vice versa. Do not attempt to control the breath, rather allow oneself to breath naturally while paying attention to it.

During this process one's mind will continuously wander, thoughts arising and falling. Remain with the breath and do not be carried away by any thoughts. If one notices that they have been distracted by some train of thought then calmly bring attention back to the breath and begin focusing on its movement again. It is completely normal to be consistently distracted by a thought and then return to the breath; in many ways this is the point of the meditation—to be become aware when one is lost in non purposeful thought and to then bring one's attention back to the present.

Over time it becomes easier to remain with the breath and become less distracted by thoughts appearing in one's mind—they simply arise and fall like the beating of the heart with a decreased capacity to grab one's attention. Eventually one may experience moments where it feels like there are barely any thoughts at all and it is very quiet and still. A primary purpose of this meditation is to build and increase one's concentration.

B.2 Body Scan

The process of this meditation is also quite simple; become aware of the sensations occurring on and in one's body. One can normally start at the top of the head, paying attention to any sensations on one's scalp whether that be a light tingling or heat or anything that one becomes aware of—there is no need for any strong preconceptions of what one will feel, just an attempt to become aware of any sensations that are occurring. One then moves down through the entirety of the body body, going to forehead, brows, eyes, cheeks, lips, chin, ears, neck, throat, shoulders, arms, elbows, wrists, hands, fingers, chest, nipples, abdomen, belly, upper back, lower back, bum, crouch, thighs, knees, calves, ankles, feet, toes. Turning attention to all parts of the body and becoming aware of any sensations occurring. One repeats this process moving from toes to scalp and continuously repeats, up and down through the body.

Occasionally one may feel strong sensations in specific regions of the body at which point they may turn attention specifically to this region. Once they have paid attention and stayed with these sensations for a while they may return to moving through the body in sequence. The purpose of paying attention to all parts of the body in sequence is to ensure that one does not end up being continuously captured by a specific region and ignoring others. Of course, it is normal that one may begin to feel strong sensations in certain areas, for instance pain in one's bum from sitting, so it is okay to pay specific attention to these sensations when they occurs but one should then try and return to other regions of the body when they can.

Over time one may begin to feel more sensations on the body, or settle into feeling sensations more readily—forming a tactile granularity that gives discrepancy between different sensations. Also, at the beginning one is mainly focused on sensations on the skin, eventually one may begin to pay attention and become more aware of internal sensations such as the beating of the heart and other physiological processes.

A primary purpose of this meditation is to become aware that one is constantly experiencing sensations all over and through one's body that they are often reacting to unaware—the sensations are shaping thought and the thoughts are shaping sensations—by

becoming aware of this constant process the labels we associate with specific sensations can change and we can relate differently to them. For instance, what is the sensation of physical pain? Is it hot, cold, sharp, blunt, throbbing, tingling; does it stay in the same place or is it constantly moving? Over time one becomes aware that what we broadly refer to as pain is composed of a variety of different sensations, some of which individually do not directly cause us pain, and which are actually arising and falling continuously, much like our thoughts, and that often what is causing us to suffer is the thoughts that are related to these sensations rather then the sensations themselves.

In this way we become more adept at noticing the boundaries, or lack thereof, between mind and body, and learn how to be more aware of what it is we are physically feeling and how this effects our thoughts and vice versa.

B.3 Gratitude

The process of this meditation involves more directly cultivating a specific state of mind, in comparison to the process of paying attention to one's present condition from which specific states can then arise. One uses phrases that they repeat, changing the focus or object of the phrase, and giving reasons for that object's particular importance—a reminder of why feelings of gratitude are an appropriate response. These phrases are said silently in one's mind and have this general structure: *"May I be grateful for ——."* Followed by reasons or reminders if deemed necessary. To give some examples:

1) *May I be grateful for my physical health. That I am fit and healthy and free of disease or illness. That I can move easily and without pain.*

2) *May I be grateful for my eyes. That they give vision and allow me to see.*

3) *May I grateful for safety. That I do not need to constantly worry about being hurt on the street or for bombs to fall for the sky or for war to break out in the city.*

4) *May I grateful for food. That I have food in my home, that I do not have to go hungry, that I have the fortune to eat when needed.*

5) *May I be grateful for farmers around the world. That they labour to grow crops and foods that I can then purchase and eat.*

As seen from the above examples, the focus of one's gratitude is broad, it can be: something about oneself, something one owns, something that one has access to, something about the society one finds themselves in, someone or some people that one may not even know, objects or skills that one is in possession of, the appreciation that someone else has found happiness, and so on. Regardless of the focus or object, the purpose is to cultivate and feel this sense of gratitude and to sit in this state and become more comfortable and practiced at it.

Sometimes one may repeat the same mantra at the beginning and end:

1)* *May I be grateful for for my physical health. That I am fit and healthy and free of disease or illness. That I can move easily and without pain. May I be grateful for my physical health.*

One may remind themselves of the suffering of others and the great fortune that they themselves are free of such suffering:

1)** *May I grateful for my physical health. Many others in this world are burdened with difficult physical disabilities, hardships, and pain, yet I am fortunate to be free of these.*

It is important in instances like this that one does not become overwhelmed or overly focused on feelings of empathy towards others to an extent that they themselves become sad or upset; this is something that one can meditate on in different practices but the purpose here is to cultivate feelings of gratitude, so while one can become aware of the unfortunate position others find themselves in, one uses this as a catalyst for gratitude rather than emotional contagion.

It is not always necessary to directly state reasons after choosing what one is grateful for, sometimes simply stating the focus of one's gratitude is enough to give rise to all the reasons and feelings without stating them explicitly. What is important is the experience of this sense of gratitude; one becomes cognitively aware that these different things are worthy of gratitude and this cognitive awareness can transform to a felt experience of gratitude. One can then also be aware that sense of gratitude is generally positive and gives rise to a more peaceful mind—something else to be grateful for.

The overall purpose of this meditation is to get more in touch with the actual felt experience of gratitude and to improve one's ability to feel it.

B.4 Loving-Kindess

The process of this meditation also involves more directly cultivating a specific state of mind. In this case the state of mind is one of love towards other beings. It could also be stated as goodwill, kindness, well wishing, etc. as these are often easier states to first engage with but the end goal is to feel genuine love towards the person or persons of your focus.

The general phrasing used can remain the same, but the person the phrase is focused on changes. The changing of person is where a major difficulty lies; while it may be easy to cultivate feelings of love towards a specific individual it may be much more difficult to hold similar feelings for a different individual. Similarly we may find ourselves more readily able to cultivate feelings of love towards certain groups over others.

By engaging in the practice one becomes aware of how it feels to be truly loving towards someone else and how this feeling of love is a positive experience. The cognitive awareness of this is once again an important recognition. Throughout the process one may run into the difficulty of feeling love towards certain individuals—this is perfectly normal and a purpose of the meditation is to become more aware of these difficulties and their character; by gaining more awareness of what it feels like to truly wish another person well we can then also notice when we do not feel this way. One may wish to return to someone for whom the feeling of love comes easy if one gets stuck on a specific individual before returning and trying again.

The sequence of this practice often moves from individuals for whom it is easy to feel love towards, to strangers, to difficult people, to people whom you really dislike, then

outwards to larger groups, and eventually everyone and all sentient beings. One can also focus on oneself; this may be even more difficult and so is important to include.

There is generally a specific phrase or multiple sentences that are repeated. This phrase can be altered to best suit the practitioner, however it is often something akin to:

May —— be free from danger. May —— have mental happiness. May —— have physical happiness. May —— have ease of well being.

The phrase can act as an anchor from which the feelings can stem so its repetition allows one to focus on the feelings that it evokes. The forms given below are what are used most commonly in this study.

Loving-Kindness 1 (directed towards oneself)

The below phrase is repeated.

May I feel love, happiness, kindness, and peace. May my suffering, concerns, and ego disappear. May I feel excitement, adventure, dreams, and boundless love.

One puts the focus of the practice on oneself. This can often feel strange, uncomfortable, silly, selfish, etc. For many people this is the most difficult focus for the practice. One will likely become aware of many different types of resistance and retorts in one's own mind—reasons and insults for why one should not be feeling such love for oneself. This can give a greater understanding of the difficulties we have regarding our own self image. Once again, a cognitive understanding that having love for oneself is normally more beneficial both for oneself and others is an important recognition. One can become aware that wishing oneself well is deeply intertwined with wishing others well.

Loving-Kindness 2 (directed towards other individuals)

The below phrase is repeated with new individuals being inserted after a number of repetitions on a specific person.

May —— feel love, happiness, kindness, and peace. May —— suffering, concerns, and ego disappear. May —— feel excitement, adventure, dreams, and boundless love.

One normally beings with someone who it is easy to have feelings of love towards; a person, perhaps a family member or friend, for whom you have an uncomplicated relationship and with whom feelings of love and goodwill come readily. By beginning with this person we come in contact with the feeling we are attempting to cultivate and gain more understanding and awareness of this state.

Then one moves onto a stranger or an acquaintance, someone they have only seen or someone they know very little and so do not have any strong feelings towards them either way. It may be a cashier at the supermarket, someone you saw on the street earlier, some member of a shared community, etc.

Then one can move onto someone whom they are having a difficult relationship with; this may be a friend, a partner, a workmate, a boss, etc. It is normally good to start with someone with whom the difficulty is not too great in order for progress to be made more readily.

One can then move onto a person for whom they have feelings of more extreme dislike. This may be someone in your day to day life or even a public figure such as a politician who causes strong feelings feelings of dislike to arise.

Throughout the process the purpose is to cultivate feelings of love. Therefore if one is struggling on a specific individual or type of individual they can feel free to return to someone for whom it is easier before trying again.

Loving-Kindness 3 (directed towards groups)

The below phrase is repeated with new groups being inserted after a number of repetitions on a specific group.

May —— feel love, happiness, kindness, and peace. May —— suffering, concerns, and ego disappear. May —— feel excitement, adventure, dreams, and boundless love.

One can do this practice geographically; beginning with their local town/city, then the country they live in, and then onwards throughout the continents of the world. Moving from one continent to the next, or even from one country to the next. Repeating the phrase and cultivating the sense towards people of different nationality and cultures. One moves through all the areas of the world, eventually encompassing the entire planet and wishing love towards all people.

Or instead of using geographical groups may one wish to mention groups by some other categorisation that is often present in one's mind (or in the media) for instance towards women, men, rightwing, leftwing, protestors, bankers, artists, politicians, etc. This can be an interesting way of discovering differences in feelings towards groups that one was unaware of before. Eventually one wishes to encompass all groups within this same frame of loving kindness and wish all of them well.

Appendix C

Data Analysis

In this Appendix C you will find information relating to the analysis of data. A python script was used to clean the data, transform the data into the correct structure for input into the neural network, define the neural network, run the training process to create a model, and provide readable output of the results. Tensorflow, an open source machine learning library developed at Google, allows for easy access to neural network network functionality and was the library used in this study.

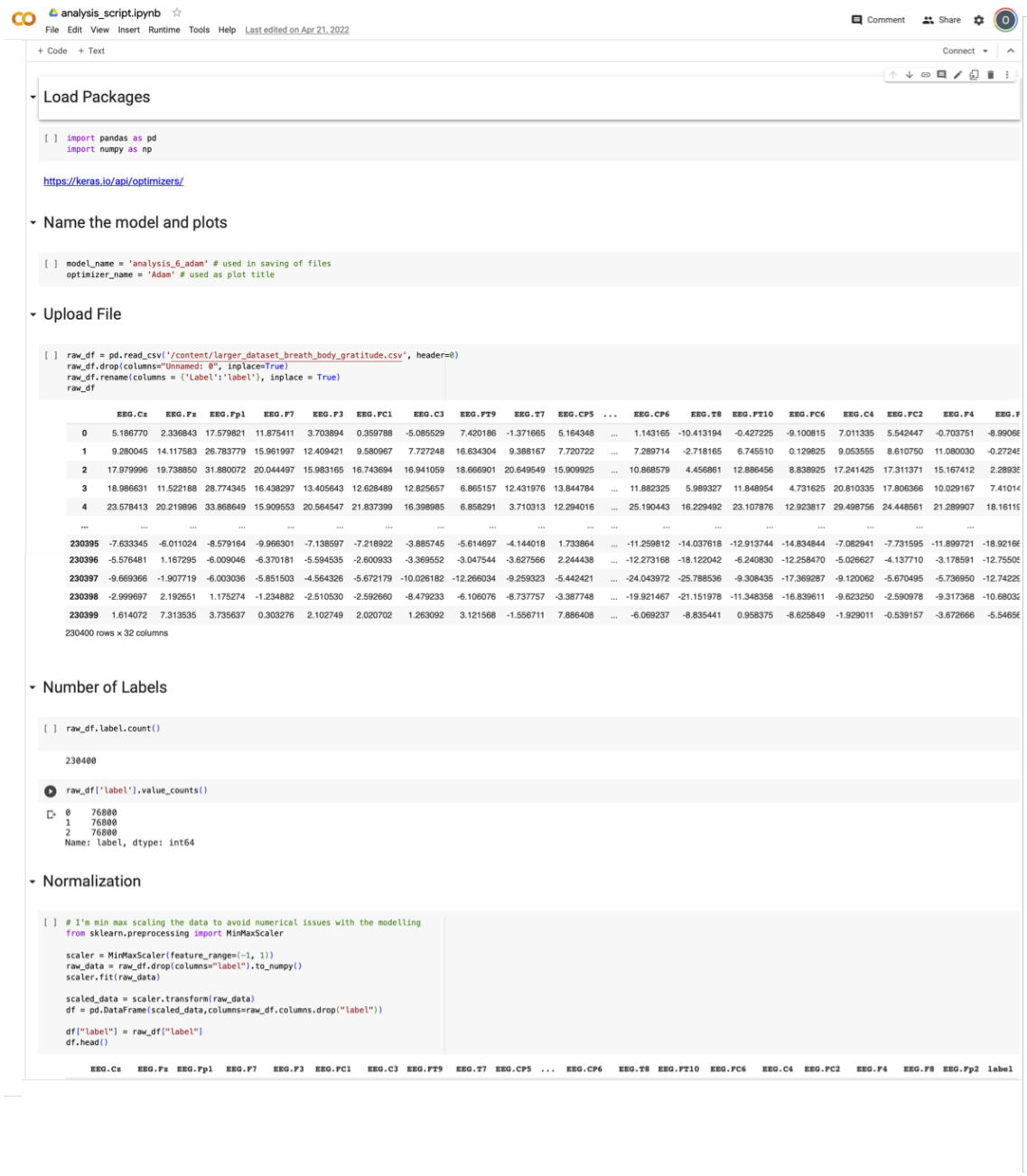
A summary of the steps taken for data analysis is as follows:

1) Export raw data from Emotiv Pro software as a CSV file.

2) Clean CSV file by removing rows and columns that are not related to analysis or that for some reason (e.g. momentary error in recording) have empty cells. Only time-series information related to the voltage change over the electrodes is necessary (file initially includes extraneous columns with information related to motion, contact quality, eeg quality, etc.)

3) Label the rows of data the appropriate class, normalize the values, structure the data for use in the neural network, decide on hyperparameters for the network, define the neural network (in this case a recurrent neural network), run the training process to create a model, test the model, provide readable output of the results.

The following pages provide images of the main script used in the final step.



analysis_script.ipynb

File Edit View Insert Runtime Tools Help Last edited on Apr 21, 2022

Comment Share

+ Code + Text

Connect

Load Packages

```
[ ] import pandas as pd
    import numpy as np
```

https://keras.io/api/optimizers/

Name the model and plots

```
[ ] model_name = 'analysis_6_adam' # used in saving of files
    optimizer_name = 'Adam' # used as plot title
```

Upload File

```
[ ] raw_df = pd.read_csv('/content/larger_dataset_breath_body_gratitude.csv', header=0)
    raw_df.drop(columns="Unnamed: 0", inplace=True)
    raw_df.rename(columns = {'Label':'label'}, inplace = True)
    raw_df
```

	EEG.Cz	EEG.Fz	EEG.Fp1	EEG.F7	EEG.F3	EEG.FC1	EEG.C3	EEG.FT9	EEG.T7	EEG.CP5	...	EEG.CP6	EEG.T8	EEG.FT10	EEG.FC6	EEG.C4	EEG.FC2	EEG.F4	EEG.F
0	5.186770	2.336843	17.579821	11.875411	3.703894	0.359788	-5.085529	7.420186	-1.371665	5.164348	...	1.143165	-10.413194	-0.427225	-9.100815	7.011335	5.542447	-0.703751	-8.9906
1	9.280045	14.117583	26.783779	15.961997	12.409421	9.580967	7.727248	16.634304	9.388167	7.720722	...	7.289714	-2.718165	6.745510	0.129825	9.053555	8.610750	11.080030	-0.2724
2	17.979996	19.738850	31.880072	20.044497	15.983165	16.743694	16.941059	18.666901	20.649549	15.909925	...	10.868579	4.456861	12.886456	8.838925	17.241425	17.311371	15.167412	2.2893
3	18.986631	11.522188	28.774345	16.438297	13.405643	12.628489	12.825657	6.865157	12.431976	13.844784	...	11.882325	5.989327	11.848954	4.731625	20.810335	17.806366	10.029167	7.4101
4	23.578413	20.219896	33.868649	15.909553	20.564547	21.837399	16.398985	6.858291	3.710313	12.294016	...	25.190443	16.229492	23.107876	12.923817	29.498756	24.448561	21.289907	18.1611
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
230395	-7.633345	-6.011024	-8.579164	-9.966301	-7.138597	-7.218922	-3.885745	-5.614697	-4.144018	1.733864	...	-11.259812	-14.037618	-12.913744	-14.834844	-7.082941	-7.731595	-11.899721	-18.9216
230396	-5.576481	1.167295	-6.009046	-6.370181	-5.594535	-2.600933	-3.369552	-3.047544	-3.627566	2.244438	...	-12.273168	-18.122042	-6.240830	-12.258470	-5.026627	-4.137710	-3.178591	-12.7550
230397	-9.669366	-1.907719	-6.003036	-5.851503	-4.564326	-5.672179	-10.026182	-12.266034	-9.259323	-5.442421	...	-24.043972	-25.788536	-9.308435	-17.369287	-9.120062	-5.670495	-5.736950	-12.7422
230398	-2.999697	2.192651	1.175274	-1.234882	-2.510530	-2.592660	-8.479233	-6.106076	-8.737757	-3.387748	...	-19.921467	-21.151978	-11.348358	-16.839611	-9.623250	-2.590978	-9.317368	-10.6803
230399	1.614072	7.313535	3.735637	0.303276	2.102749	2.020702	1.263092	3.121568	-1.556711	7.886408	...	-6.069237	-8.835441	0.958375	-8.625849	-1.929011	-0.539157	-3.672666	-5.5465

230400 rows × 32 columns

Number of Labels

```
[ ] raw_df.label.count()
```

```
230400
```

```
raw_df['label'].value_counts()
```

```
0    76800
1    76800
2    76800
Name: label, dtype: int64
```

Normalization

```
[ ] # I'm min max scaling the data to avoid numerical issues with the modelling
    from sklearn.preprocessing import MinMaxScaler

    scaler = MinMaxScaler(feature_range=(-1, 1))
    raw_data = raw_df.drop(columns="label").to_numpy()
    scaler.fit(raw_data)

    scaled_data = scaler.transform(raw_data)
    df = pd.DataFrame(scaled_data,columns=raw_df.columns.drop("label"))

    df["label"] = raw_df["label"]
    df.head()
```

	EEG.Cz	EEG.Fz	EEG.Fp1	EEG.F7	EEG.F3	EEG.FC1	EEG.C3	EEG.FT9	EEG.T7	EEG.CP5	...	EEG.CP6	EEG.T8	EEG.FT10	EEG.FC6	EEG.C4	EEG.FC2	EEG.F4	EEG.F8	EEG.Fp2	label

0	-0.115762	-0.160547	0.311433	0.077191	-0.081477	-0.162250	-0.185204	0.222705	0.007566	0.016682	...	-0.161106	-0.164162	-0.018791	-0.187865	-0.128718	-0.103011	-0.128534	-0.145134	0.138007	0
1	-0.072397	-0.017753	0.407473	0.132925	0.024506	-0.051045	-0.004783	0.370760	0.202474	0.060264	...	-0.082831	-0.060822	0.069327	-0.064224	-0.109749	-0.066523	0.012124	-0.047072	0.210471	0
2	0.019771	0.050382	0.460652	0.188602	0.068013	0.035335	0.124960	0.403420	0.406469	0.199874	...	-0.037254	0.035534	0.144769	0.052432	-0.033695	0.036942	0.060913	-0.018257	0.268359	0
3	0.030435	-0.049212	0.428244	0.139420	0.036634	-0.014293	0.067009	0.213787	0.257611	0.164667	...	-0.024345	0.056115	0.132023	-0.002584	-0.000545	0.042828	-0.000420	0.039341	0.268183	0
4	0.079081	0.056213	0.481402	0.132209	0.123788	0.096764	0.117327	0.213676	0.099623	0.138230	...	0.145132	0.193634	0.270340	0.107148	0.080158	0.121816	0.133995	0.160268	0.335682	0

5 rows × 32 columns

Functions for Structuring Data

```
# This takes the dataframe, looks at the 'label' column and determines how many classes are present.
# It then creates an empty list, loops through the classes, for every class it slices off the
# features of the class by dropping the label column.
# it then stores this dataframe as numpy matrix, and then creates a dictionary,
# linking this matrix with the class
# We not have a list containing a dictionary

def split_df_into_dict(df):
  nbr_classes = df["label"].nunique()

  class_recordings = []
  for i in range(nbr_classes):
    tmp_df = df.loc[df.label == i].drop(columns=["label"])
    matrix = tmp_df.to_numpy() # Change .label to name of your label column
    sample = {"label": i, "matrix": matrix}
    class_recordings.append(sample)

  return class_recordings
```

```
[ ] df_as_dict = split_df_into_dict(df) #
    df_as_dict
```

```
[{'label': 0,
  'matrix': array([[-1.15762006e-01, -1.60546763e-01,  3.11432653e-01, ...,
          -1.28534057e-01, -1.45133548e-01,  1.38007371e-01],
         [-7.23973657e-02, -1.77528799e-02,  4.07473217e-01, ...,
           1.21239709e-02, -4.70717607e-02,  2.10470623e-01],
         [ 1.97709450e-02,  5.03822751e-02,  4.60651519e-01, ...,
           6.09133273e-02, -1.82567379e-02,  2.68359436e-01],
         ...,
         [-1.11493042e-01, -2.85632372e-02,  3.02207530e-01, ...,
          -5.43433390e-02,  1.40121659e-01,  2.53609087e-01],
         [-8.44150158e-02,  5.20020475e-02,  3.60836869e-01, ...,
           6.17796842e-02,  2.20611003e-01,  3.30790733e-01],
         [-1.44203241e-01, -4.13837468e-02,  3.12492025e-01, ...,
           4.45759590e-04,  1.37551269e-03,  2.82211486e-01]])},
 {'label': 1,
  'matrix': array([[ 0.11549141, -0.16308867,  0.03598773, ...,  0.00926422,
           0.07093923,  0.11336311],
         [-0.06932803, -0.299727  , -0.04945273, ..., -0.1009388 ,
          -0.04442355,  0.02632939],
         [ 0.00655487, -0.13816495,  0.07902341, ..., -0.03369078,
           0.01320078,  0.12790824],
         ...,
         [ 0.21790073,  0.04042928,  0.20825782, ...,  0.07586898,
           0.11103738,  0.16115743],
         [-0.04843284, -0.25786375,  0.03176687, ..., -0.18116548,
          -0.10808854, -0.00810188],
         [-0.08654726, -0.24537543,  0.0799751 , ..., -0.16275884,
          -0.25208424,  0.02583558]])},
 {'label': 2,
  'matrix': array([[ 0.00725305, -0.11580494,  0.19234586, ..., -0.04323659,
           0.03128397,  0.12099626],
         [-0.05805431, -0.1655553 ,  0.13347794, ..., -0.1105807 ,
          -0.06675198,  0.07746081],
         [-0.107014  , -0.17799794,  0.15485558, ..., -0.10447505,
          -0.05520446,  0.10164472],
         ...,
         [-0.27314967, -0.21199493,  0.06535255, ..., -0.18861323,
          -0.18733141,  0.05706184],
         [-0.20249041, -0.16229451,  0.14025608, ..., -0.23135117,
          -0.16413853,  0.05226222],
         [-0.15361159, -0.10022448,  0.16697271, ..., -0.16397274,
          -0.10639447,  0.07647133]])}]
```

```
[ ] # Since I don't have any test data I split the data so that the last 20% of the sessions are testing data
    from sklearn.model_selection import train_test_split

    nbr_labels = len(df_as_dict)
    train_data = []
    test_data = []
    for i in range(nbr_labels):
        label = df_as_dict[i]["label"]
        train, test = train_test_split(df_as_dict[i]["matrix"], shuffle=False, test_size=0.2)
        train_data.append({"label":label, "matrix": train})
        test_data.append({"label":label, "matrix": test})
```

```
[ ] test_data[2]["matrix"].shape
```

```
(15360, 31)
```

```
(15360, 31)
```

```
# Now we have a list of numpy arrays with the measurements,
# given by df_as_dict[0]["matrix"], df_as_dict[1]["matrix"], df_as_dict[2]["matrix"]

# Each numpy array should correspond to a specific meditation class.
# We now want to window these to construct training samples


# We create a function that makes an empty list
# We loop for the number of windows that will be created
# we sample a number of rows (corresponding to the window_size) from the dataframe matrix
# and if the length if the sample is appropriate then we append window of samples to our list
# once we have windowed our entire dataset we then stack this windows to form a windowed dataset

def make_windows(mat, window_size, step_size):
  windowed_samples = []
  nbr_windows = int(np.floor((mat.shape[0] - window_size) / step_size))
  start = 0
  for _ in range(nbr_windows):
    sample = mat[start:start+window_size]
    start += step_size
    if len(sample) == window_size:
      windowed_samples.append(sample)

  return np.stack(windowed_samples, axis=0)
```

```
# Loop through df_as_dict and apply make_windows to each
# concatenate this windows into a single windowed feature dataset, X
# concatenate the appropriate label of these windows into a single label dataset, y

def window_data(data, window_size=128, step_size=32):
  x_list = []
  y_list = []

  for class_data in data:
    matrix = class_data["matrix"]
    x = make_windows(matrix, window_size, step_size)
    x_list.append(x)
    label = class_data["label"]
    y = np.ones((x.shape[0],1)) * label
    y_list.append(y)

  X = np.concatenate(x_list, axis=0)
  y = np.concatenate(y_list, axis=0)
  return X, y
```

Decide on hyperparameters: Window Size, Step Size, Number of Features

```
window_size = 512
step_size = 32
n_features = 31
```

Create Train and Test set.

```
X_train, y_train = window_data(train_data, window_size=window_size, step_size=step_size)
X_test, y_test = window_data(test_data, window_size=window_size, step_size=step_size)
```

```
X_train.shape
```

```
(5736, 256, 31)
```

One_Hot_Encoding

```
import tensorflow.keras as keras
#from keras.utils import to_categorical

y_train_onehot = keras.utils.to_categorical(y_train, num_classes=3)
y_test_onehot = keras.utils.to_categorical(y_test, num_classes=3)
```

Model Function

```
import tensorflow.keras as keras
from tensorflow.keras import Sequential
import matplotlib.pyplot as plt
from google.colab import files
from sklearn.metrics import confusion_matrix, classification_report, ConfusionMatrixDisplay
import seaborn as sns #confusion matrix heatmap

#from plot_keras_history import show_history, plot_history

def train_model(n_hidden_nodes=32, dropout=0.3, lr=0.001, epochs=50, patience=5):
```

```
# Create lstm layer/s
lstm_layer = [keras.layers.LSTM(n_hidden_nodes, input_shape=(window_size, n_features), recurrent_dropout=dropout, activation="tanh", recurrent_activation="sigmoid", return_sequences=False)]
dense_layer = [keras.layers.Dense(nbr_labels, activation="softmax")]

# Model architecture
model = Sequential(lstm_layer + dense_layer)
print(model.summary())

# Model Opimizer
opt = keras.optimizers.Adam(learning_rate=lr, beta_1=0.9, beta_2=0.999,epsilon=1e-07)

# Model Compile
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])

# Model early stopping
early_stopping = keras.callbacks.EarlyStopping(monitor="val_loss", restore_best_weights=True, mode="min", patience=patience)

# Model Checkpoint
checkpoint_filepath = '/best_accuracy/best_weights'
best_weights_accuracy= keras.callbacks.ModelCheckpoint(filepath=checkpoint_filepath, save_weights_only=True, monitor='val_accuracy', mode='max', save_best_only=True)

# Model fit
history = model.fit(X_train, y_train_onehot, epochs=epochs, callbacks=[early_stopping, best_weights_accuracy], validation_data=(X_test, y_test_onehot), shuffle=True)

# Model Accuracy
model.load_weights('/best_accuracy/best_weights')
model_acc = model.evaluate(X_test, y_test_onehot, verbose=1)
print('Test Loss:', model_acc[0])
print('Test Accuracy:', model_acc[1])

# accuracy, loss, and epochs
# accuracy values
best_train_acc = np.max(history.history['accuracy'])
best_train_acc_epoch = (np.argmax(history.history['accuracy']) + 1)
best_test_acc = np.max(history.history['val_accuracy'])
best_test_acc_epoch = (np.argmax(history.history['val_accuracy']) + 1)

# loss values
best_train_loss =  np.max(history.history['loss'])
best_train_loss_epoch = (np.argmax(history.history['loss']) + 1)
best_test_loss = np.max(history.history['val_loss'])
best_test_loss_epoch = (np.argmax(history.history['val_loss']) + 1)

output = f"""best train acc: {best_train_acc} at epoch {best_train_acc_epoch}
best test acc: {best_test_acc} at epoch {best_test_acc_epoch}
best train loss: {best_train_loss} at epoch {best_train_loss_epoch}
best test loss: {best_test_loss} at epoch {best_test_loss_epoch}"""
```

```
# Plots
# summarize history for accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model Accuracy - {}'.format(optimizer_name))
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.grid(visible=True, axis='both', linewidth=.1)
plt.xlim(0, epochs-1)
plt.ylim(0,1)
plt.xticks(np.arange(0, epochs+1, 5))
plt.yticks(np.arange(0, 1.1, .1))
accuracy_plot = plt.gcf()
plt.show()
# summarize history for loss with y-axis range 0-2
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss - {}'.format(optimizer_name))
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.grid(visible=True, axis='both', linewidth=.1)
plt.xlim(0, epochs-1)
plt.ylim(0,2)
plt.xticks(np.arange(0, epochs+1, 5))
plt.yticks(np.arange(0, 2.2, .2))
loss_plot_a = plt.gcf()
plt.show()
# summarize history for loss with y-axis range 0-1.6
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss - {}'.format(optimizer_name))
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.grid(visible=True, axis='both', linewidth=.1)
plt.xlim(0, epochs-1)
plt.ylim(0,1.6)
plt.xticks(np.arange(0, epochs+1, 5))
plt.yticks(np.arange(0, 1.8, .2))
loss_plot_b = plt.gcf()
plt.show()

# Confusion matrix(s)
# note we put the original y_test integer encoded instead of y_test_onehot for confusion matrix to function
y_pred = np.array(list(map(lambda x: np.argmax(x), model.predict(X_test))))
cm = confusion_matrix(y_test, y_pred)
```

```
cm = confusion_matrix(y_test, y_pred)
nbr_of_labels = (len(y_pred)/3)
# Unnormalized
fig, ax = plt.subplots(figsize=(8,8))
sns.heatmap(cm, annot=True, cbar=True, fmt='g', vmin=0, vmax=nbr_of_labels, xticklabels=['Breath', 'Body', 'Gratitude'], yticklabels=['Breath', 'Body', 'Gratitude'], cmap="Blues")
plt.xlabel('Predicted Label', labelpad=10)
plt.ylabel('True Label', labelpad=10)
plt.title("Confusion Matrix - {}".format(optimizer_name))
confusion_matrix_unnormalized_plot = plt.gcf()
plt.show()
# Normalized
cmn = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis] #normalizes over the rows, percentage of correct labels per meditation
fig, ax = plt.subplots(figsize=(8,8))
sns.heatmap(cmn, annot=True, cbar=True, vmin=0, vmax=1, fmt='.2f', xticklabels=['Breath', 'Body', 'Gratitude'], yticklabels=['Breath', 'Body', 'Gratitude'], cmap="Blues")
plt.xlabel('Predicted Label', labelpad=10)
plt.ylabel('True Label', labelpad=10)
plt.title("Confusion Matrix - {}".format(optimizer_name))
confusion_matrix_normalized_plot = plt.gcf()
plt.show()


# classification report
# note we put the original y_test integer encoded instead of y_test_onehot for classification report to function
clr = classification_report(y_test, y_pred, target_names=['Breath', 'Body', 'Gratitude'])
print("Classification Report:\n----------------------\n", clr)
clr_dict = classification_report(y_test, y_pred, target_names=['Breath', 'Body', 'Gratitude'], output_dict=True)
clreport_df = pd.DataFrame(clr_dict).transpose()

# Saving model and plots, diagrams
model.save(model_name)

accuracy_plot.savefig(fname='/content/{}/{}_accuracy_plot'.format(model_name, model_name), dpi=300)
loss_plot_a.savefig(fname='/content/{}/{}_loss_plot_a'.format(model_name, model_name), dpi=300)
loss_plot_b.savefig(fname='/content/{}/{}_loss_plot_b'.format(model_name, model_name), dpi=300)
confusion_matrix_unnormalized_plot.savefig(fname='/content/{}/{}_confusion_matrix_unnormalized_plot'.format(model_name, model_name), dpi=300)
confusion_matrix_normalized_plot.savefig(fname='/content/{}/{}_confusion_matrix_normalized_plot'.format(model_name, model_name), dpi=300)
clreport_df.to_csv('/content/{}/{}_classification_report.csv'.format(model_name, model_name))

save_path = '/content/{}/{}_best_values'.format(model_name, model_name)
with open(save_path, 'w') as f:
  f.write(output)

return history, model
```

Run Model

```
[ ] train_model(n_hidden_nodes=16, dropout=0.4, lr=0.005, epochs=50, patience=50)

Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 16)                3072

 dense (Dense)               (None, 3)                 51

=================================================================
Total params: 3,123
Trainable params: 3,123
Non-trainable params: 0
_________________________________________________________________
None
Epoch 1/50
180/180 [==============================] - 46s 222ms/step - loss: 1.0428 - accuracy: 0.4315 - val_loss: 0.9820 - val_accuracy: 0.4859
Epoch 2/50
180/180 [==============================] - 31s 173ms/step - loss: 1.0274 - accuracy: 0.4594 - val_loss: 1.0572 - val_accuracy: 0.4371
Epoch 3/50
180/180 [==============================] - 32s 176ms/step - loss: 0.9920 - accuracy: 0.5021 - val_loss: 0.9805 - val_accuracy: 0.5000
Epoch 4/50
180/180 [==============================] - 32s 180ms/step - loss: 0.9705 - accuracy: 0.5178 - val_loss: 0.9828 - val_accuracy: 0.5184
Epoch 5/50
180/180 [==============================] - 31s 172ms/step - loss: 0.9716 - accuracy: 0.5181 - val_loss: 0.8444 - val_accuracy: 0.6052
Epoch 6/50
180/180 [==============================] - 31s 170ms/step - loss: 0.9035 - accuracy: 0.5755 - val_loss: 0.8099 - val_accuracy: 0.6448
Epoch 7/50
180/180 [==============================] - 34s 186ms/step - loss: 1.0335 - accuracy: 0.4862 - val_loss: 0.9816 - val_accuracy: 0.5346
Epoch 8/50
180/180 [==============================] - 33s 186ms/step - loss: 0.9303 - accuracy: 0.5520 - val_loss: 0.7949 - val_accuracy: 0.6518
Epoch 9/50
180/180 [==============================] - 31s 175ms/step - loss: 0.8647 - accuracy: 0.5941 - val_loss: 0.6448 - val_accuracy: 0.7175
Epoch 10/50
180/180 [==============================] - 31s 174ms/step - loss: 0.7906 - accuracy: 0.6501 - val_loss: 0.6849 - val_accuracy: 0.6822
Epoch 11/50
180/180 [==============================] - 32s 176ms/step - loss: 0.7610 - accuracy: 0.6719 - val_loss: 1.2773 - val_accuracy: 0.4767
Epoch 12/50
180/180 [==============================] - 32s 175ms/step - loss: 0.9627 - accuracy: 0.5403 - val_loss: 0.9423 - val_accuracy: 0.5141
Epoch 13/50
180/180 [==============================] - 32s 180ms/step - loss: 0.8128 - accuracy: 0.6255 - val_loss: 0.6700 - val_accuracy: 0.6900
Epoch 14/50
180/180 [==============================] - 31s 175ms/step - loss: 0.6931 - accuracy: 0.6956 - val_loss: 0.7288 - val_accuracy: 0.6730
Epoch 15/50
180/180 [==============================] - 31s 172ms/step - loss: 0.6527 - accuracy: 0.7263 - val_loss: 0.5765 - val_accuracy: 0.7782
Epoch 16/50
180/180 [==============================] - 31s 174ms/step - loss: 0.5934 - accuracy: 0.7585 - val_loss: 0.6168 - val_accuracy: 0.7542
Epoch 17/50
180/180 [==============================] - 31s 172ms/step - loss: 0.5732 - accuracy: 0.7648 - val_loss: 0.6469 - val_accuracy: 0.7613
Epoch 18/50
```

```
180/180 [==============================] - 31s 174ms/step - loss: 0.6158 - accuracy: 0.7476 - val_loss: 0.6833 - val_accuracy: 0.7027
Epoch 19/50
180/180 [==============================] - 31s 172ms/step - loss: 0.5704 - accuracy: 0.7605 - val_loss: 0.6508 - val_accuracy: 0.7564
Epoch 20/50
180/180 [==============================] - 31s 173ms/step - loss: 0.5429 - accuracy: 0.7816 - val_loss: 0.4828 - val_accuracy: 0.8340
Epoch 21/50
180/180 [==============================] - 32s 175ms/step - loss: 0.5162 - accuracy: 0.7941 - val_loss: 0.5289 - val_accuracy: 0.8058
Epoch 22/50
180/180 [==============================] - 31s 172ms/step - loss: 0.5050 - accuracy: 0.7943 - val_loss: 0.5354 - val_accuracy: 0.8072
Epoch 23/50
180/180 [==============================] - 31s 175ms/step - loss: 0.4633 - accuracy: 0.8101 - val_loss: 0.6233 - val_accuracy: 0.7719
```

Download Files

```
[ ] # check model_name at top of document and change model_X
    !zip -r /content/analysis_6_adam.zip /content/analysis_6_adam
    files.download('/content/analysis_6_adam.zip')
```

```
updating: content/analysis_6_adam/ (stored 0%)
updating: content/analysis_6_adam/analysis_6_adam_confusion_matrix_normalized_plot.png (deflated 25%)
updating: content/analysis_6_adam/analysis_6_adam_confusion_matrix_unnormalized_plot.png (deflated 27%)
updating: content/analysis_6_adam/analysis_6_adam_classification_report.csv (deflated 52%)
updating: content/analysis_6_adam/analysis_6_adam_accuracy_plot.png (deflated 10%)
updating: content/analysis_6_adam/assets/ (stored 0%)
updating: content/analysis_6_adam/analysis_6_adam_loss_plot_b.png (deflated 9%)
updating: content/analysis_6_adam/keras_metadata.pb (deflated 86%)
updating: content/analysis_6_adam/analysis_6_adam_best_values (deflated 41%)
updating: content/analysis_6_adam/variables/ (stored 0%)
updating: content/analysis_6_adam/variables/variables.index (deflated 60%)
updating: content/analysis_6_adam/variables/variables.data-00000-of-00001 (deflated 13%)
updating: content/analysis_6_adam/analysis_6_adam_loss_plot_a.png (deflated 9%)
updating: content/analysis_6_adam/saved_model.pb (deflated 90%)
```

Bibliography

[1] Alexander Amini. Mit 6.s191: Recurrent neural networks. *YouTube*, Feb 2021. https://www.youtube.com/watch?v=5tvmMX8r_OM&list=PLtBw6njQRU-rwp5__7C0oIVt26ZgjG9NI&index=3.

[2] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[3] George B Dantzig. Linear programming. *History of mathematical programming*, pages 19–31, 1991.

[4] Pierre de Fermat. Pierre de fermats abhandlungen über maxima und minima (1629). *Akademische verlagsgesellschaft mbh*, (238), 1934.

[5] Timothy Dozat. Incorporating nesterov momentum into adam. *Open Review*, 2016.

[6] Ding-Zhu Du, Panos M. Pardalos, and Weili Wu. History of optimization. *Encyclopedia of optimization*, 2008.

[7] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[9] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. 2012. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[10] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[11] Raimi Karim. 10 gradient descent optimisation algorithms. *Medium*, May 2020. https://towardsdatascience.com/10-gradient-descent-optimisation-algorithms-86989510b5e9.

[12] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[13] Kiprono Elijah Koech. Cross-entropy loss function. *Medium*, Nov 2021. https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e.

[14] Merriam-Webster. Optimum vs optimal. *Merriam-Webster*, 2022. https://www.merriam-webster.com/words-at-play/optimum-vs-optimal,.

[15] Andrew Ng. Gradient descent with momentum (c2w2l06). 2017. https://www.youtube.com/watch?v=k8fTYJPd3_I.

[16] Christopher Olah. Understanding lstm networks. *Understanding LSTM Networks – colah's blog*, Aug 2015. https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[17] Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr computational mathematics and mathematical physics*, 4(5):1–17, 1964.

[18] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

[19] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[20] Robin M Schmidt. Recurrent neural networks (rnns): A gentle introduction and overview. *arXiv preprint arXiv:1912.05911*, 2019.

[21] Claude Elwood Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

[22] Bismark Singh and Mark Eisner. A brief history of optimization and mathematical programming. *Informs*. https://www.informs.org/Explore/History-of-O.R.-Excellence/O.R.-Methodologies/Optimization-Mathematical-Programming.

[23] Ava Soleimany. Mit 6.s191: Recurrent neural networks. *YouTube*, Feb 2021. https://www.youtube.com/watch?v=qjrad0V0uJE&list=PLtBw6njQRU-rwp5__7C0oIVt26ZgjG9NI&index=2.

[24] Ilya Sutskever. Training recurrent neural networks. 2013. https://www.cs.utoronto.ca/~ilya/pubs/ilya_sutskever_phd_thesis.pdf.

[25] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. *International conference on machine learning*, pages 1139–1147, 2013.

[26] Greg Van Houdt, Carlos Mosquera, and Gonzalo Nápoles. A review on the long short-term memory model. *Artificial Intelligence Review*, 53(8):5929–5955, 2020.

[27] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[28] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. Dive into deep learning. *arXiv preprint arXiv:2106.11342*, 2021.

[29] Tehseen Zia and Usman Zahid. Long short-term memory recurrent neural network architectures for urdu acoustic modeling. *International Journal of Speech Technology*, 22(1):21–30, 2019.