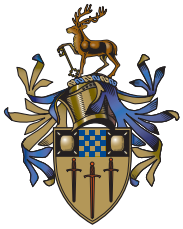
Applications of Neural Networks to Sleep Staging

Daniel Jorden

Supervisor: Prof. Anne Skeldon

Department of Mathematics
University of Surrey
Guildford GU2 7XH, United Kingdom



Submitted as a report for the Final Year Project for the M.Math. degree at the University of Surrey

May, 2020

Scientific abstract

Neural networks have a wide array of applications within the life science domain. There are many varying architectures that lend themselves to different problem solving approaches. One such architecture is the Long Short Term Memory (LSTM) neural network which mimics human memory formation and decay allowing for pattern recognition in time series data. A great deal of research has been done into sleep cycles and sleep stages within the life sciences. A sleeping human body outputs a lot of data that can be recorded and used to classify the sleep stages more accurately, for example electrical heart activity can be measured with an electrocardiogram (ECG). In the first part of this report we detail the underlying mathematics that goes into how neural networks process data and learn from it. We will cover the various tunable parameters and problem areas associated with our LSTM network. In the second part of this report we use an LSTM on ECG data to classify the various sleep stages. We manage an accuracy of 45% to 60% classification and discuss the limiting factors that contribute to such a result.

Contents

1 Introduction

Since their inception in the 1940s, neural networks have had a variety of applications in the modern scientific fields, from speech recognition to weather prediction. What started out as an oversimplified artificial model of the human brain has since developed to process a wide variety of data. One of the most significant areas of development has been the life sciences, using artificial intelligence to increase quality of life and medical capabilities, such as sleep dynamics. As we have progressed into the modern age our sleep has gradually suffered and new research into the nature of sleep and it's mechanics has been of keen interest to the scientific community. The basic sleep structure consists of several stages; rapid eye movement (REM), non-REM (N1, N2 ans N3) and wake. A helpful way of visualising the sleep stages is to graph the stages of sleep against time (hypnogram) as seen in figure 1.1.

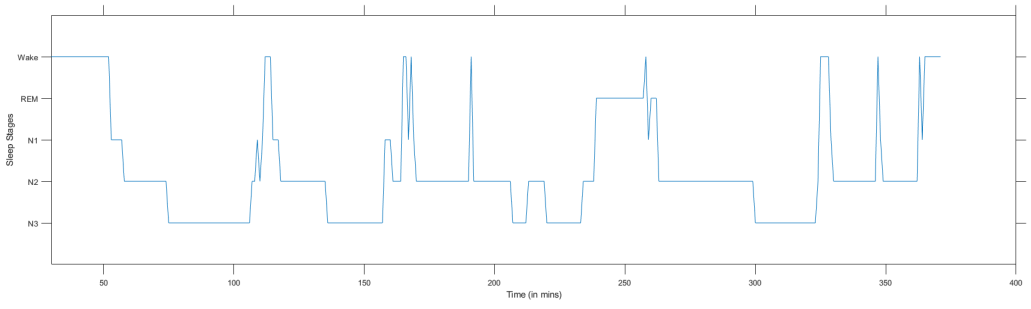


Figure 1.1: A Hypnogram showing 6 hours of sleep.

By applying neural networks to sleep staging data we can attempt to better recognise the several stages of sleep, when they begin, how long they last and how they cycle. [1]

1.1. Defining a Neural Network

From an application perspective a Neural Network is a tool that (using sufficient data) can solve an input-output problem, that is a problem that transforms a valued input into a valued output. There are dozens of different architectures that utilise varying structures to solve certain kinds of problems. Our focus will be on just three:

- Feed Forward Neural Networks (FFNNs): The most fundamental network. It takes an input and generates an output.
- Recurrent Neural Networks (RNNs): Takes a sequence of inputs one-by-one and generates an output typically for each input. They have special structure that allows them to retain information about the previous input. This makes them excellent for solving time sensitive problems, e.g. weather prediction.
- Long Short Term Memory Neural Networks (LSTMs): A special type of RNN that mimicks the human's memory system, with the ability to recall, forget, memorise and deliver an output in a time sensitive environment. They are particularly brilliant when it comes to recognising patterns in time series data, as such an LSTM is the architecture of choice when it comes to recognising sleep stages.

First, we'll need an understanding of a basic Feed Forward neural network and the underlying maths behind it.

2 Neural Networks for Time Series

Our first step is to understand what is meant by a neural network, the several types of architectures and how exactly one learns to solve problems.

2.1. Formal Introduction to Feed Forward Neural Networks (FFNNs)

The most primitive network is the feed forward neural network as seen in figure 2.1. Each network consists primarily of four elements; layers, nodes, weights and biases. Our example network has three layers each comprising a series of nodes. The first layer (consisting of two nodes) is our input layer and the last layer (consisting of four nodes) is out output layer. Any layers in between the input and output are considered to be hidden layers, in this case we have one hidden layer (consisting of three nodes). The purpose of layers is to stage out information processing, for example we could liken; the input layer to the human ear listening to a question, the hidden layer to the brain processing the information, and the output layer to the human mouth translating the information into a verbal answer.

Weights are thus responsible for carrying information from one layer to the next. An individual weight consists of three things; a starting node, a stopping node and a weight value, in our example the highlighted weight has a weight value of -0.76. Each node also has a bias associated with it (represented by a smaller number at the bottom of each node). To better highlight this information each weight and bias in the example figure has been graded with a colour, with red meaning more negative values (to a minimum of -2) and green meaning more positive values (to a maximum of $+2$).

Let us now run through an example to show how these different elements work together to generate an output. Our network is going to attempt to identify what quadrant a given

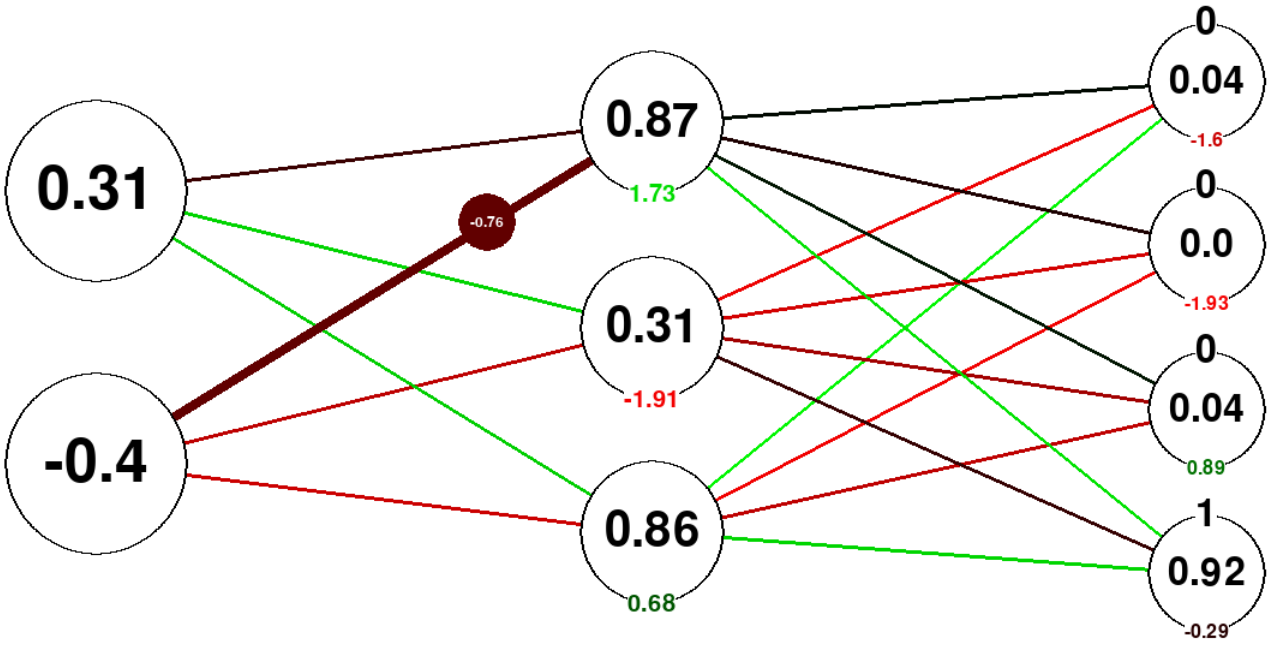


Figure 2.1: A standard FFNN.

two-dimensional (x, y) value is in. Our input vector will be the coordinates $\begin{pmatrix} x & y \end{pmatrix}$, our output vector will be the quadrants $\begin{pmatrix} First & Second & Third & Fourth \end{pmatrix}$, with each entry being a value between 0 and 1, with 0 indicating the coordinate is not in this quadrant and 1 indicating it is.

In our example we give the network an input vector of $\begin{pmatrix} 0.31 & -0.41 \end{pmatrix}$ in the input layer. This coordinate is in the fourth quadrant and so we want our output vector to be as close to $\begin{pmatrix} 0 & 0 & 0 & 1 \end{pmatrix}$ as possible. To generate the next layer (the hidden layer) we pass the current layer through the weights and biases, each weight and bias does this as follows:

1. First the weight takes the value in its starting node (e.g. -0.4).
2. Then multiplies it by its weight value to get a translated value (e.g. $-0.4 \times -0.76 = 0.304$).
3. Then all the translated values for a certain node are summed (e.g. $0.304 + -0.1395 = 0.1645$).

Step 4 Then the bias associated with that node is added (e.g. $0.1645 + 1.73 = 1.8945$).

Step 5 Then an activation function is applied to that sum (e.g. $\sigma(1.8945) = 0.8692$).

The activation function 'resizes' the summed output between 0 and 1 so as to stop the values from becoming too large, here we have used a sigmoid function; $\sigma(x) = \frac{1}{1+e^x}$. Now we have

generated the hidden layer we can generate the next layer (the output layer). The process of successively calculating each layer is called forward propagation. The output layer gives us our output vector $\begin{pmatrix}0.04 & 0.0 & 0.04 & 0.92\end{pmatrix}$, we can use this output vector to calculate how poorly the network performed. The function that translates an output vector into a scalar value is called a cost function. The cost function we use here is the squared error function, where we take each output node, subtract the value it should be, square the result and sum each result together, for example, $(0.04-0)^2+(0.0-0)^2+(0.04-0)^2+(0.92-1)^2=0.0096$. Clearly the network has performed well in this example with an error of just 0.0096.

Now, using the calculated error, we alter the weights and biases of the network accordingly starting with the output layer and working backwards to the input layer in a process called back propagation. Each change we make to a weight or bias is determined by its effect on the cost function, we can calculate these changes using derivatives however, to do so we need a more rigourous formulation of the networks elements.

2.1.1. Mathematical Formulation

Having seen how a neural network calculates an output vector we can now focus on how the network learns from its errors. To produce a more accurate response the network will need to minimise its error, having control over only its weights and biases and not the node values directly, thus a useful approach is to derive how the cost function changes with respect to each weight and bias.

We begin with a few definitions:

Definition 2.1. *L is the number of layers in the network. n_k is the number of nodes in the k^{th} layer with $k \in \{1,...,L\}$.*
In our previous example (Figure 2.1) we have; $L=3, n_1=2, n_2=3, n_3=4$.

Definition 2.2. *$N_i^{(k)}$ is the i^{th} node in the k^{th} layer, where $i \in \{1,...,n_k\}$.*
Note: The definition of a node is included purely to aid learnability, its purpose through this definition is to help define our other elements.

Definition 2.3. *$w_{ij}^{(k)}$ is the weight value with starting node $N_j^{(k-1)}$ and stopping node $N_i^{(k)}$. The collection of all weights in the k^{th} layer can be represented by a weight matrix $W^{(k)}$ with dimensions (n_k, n_{k-1}).*
Note: The layer of a weight is the layer of its stopping node, as such the input layer has no weights.

Definition 2.4. *$b_i^{(k)}$ is the bias associated with node $N_i^{(k)}$. The collection of all biases in the k^{th} layer can be represented by a bias vector $\mathbf{b}^{(k)}$ with dimension n_k.*

Definition 2.5. *σ is the activation function applied in step 5. σ' is its derivative.*

Definition 2.6. *$z_i^{(k)}$ is the value of $N_i^{(k)}$ after step 4 but before step 5,*

$$z_i^{(k)} = \sum_{r=1}^{n_k} w_{ir}^{(k)} a_r^{(k-1)} + b_i^{(k)}, \qquad \mathbf{z}^{(k)} = W^{(k)}\mathbf{a}^{(k-1)} + \mathbf{b}^{(k)}. \tag{2.1}$$

$a_i^{(k)}$ is the value of $N_i^{(k)}$ after step 5,

$$a_i^{(k)} = \sigma(z_i^{(k)}), \qquad \mathbf{a}^{(k)} = \sigma(\mathbf{z}^{(k)}). \tag{2.2}$$

Definition 2.7. *y_i is the expected result for the i^{th} output node, $N_i^{(L)}$. The collection of all expected results can be represented by the vector $\mathbf{y}$ with dimension n_L.*

Lastly we define our cost function. It is important to choose a function that best fits the system, for our purposes a simple squared error function will suffice as it is easily differentiable.

Definition 2.8. *The Cost Function for the 0^{th} training example,*

$$C_0 = \sum_{i=1}^{n_L} (a_i^{(L)} - y_i)^2, \qquad C_0 = |\mathbf{a}^{(L)} - \mathbf{y}|. \tag{2.3}$$

Now we have our definitions we can move on to properly derive the equations that facilitate back propagation.

2.1.2. Back Propagation (Output Layer, L)

The aim of this method is to find how sensitive the error is with respect to each weight and bias, as such we derive the partial differentials.

$$\frac{\partial C_0}{\partial w_{ij}^{(L)}} = \frac{\partial C_0}{\partial a_i^{(L)}} \frac{\partial a_i^{(L)}}{\partial z_i^{(L)}} \frac{\partial z_i^{(L)}}{\partial w_{ij}^{(L)}}. \tag{2.4}$$

Focusing on each definition outlined above we can clearly see;

$$\frac{\partial C_0}{\partial a_i^{(L)}} = 2(a_i^{(L)} - y_i). \qquad \frac{\partial a_i^{(L)}}{\partial z_i^{(L)}} = \sigma'(z_i^{(L)}), \qquad \frac{\partial z_i^{(L)}}{\partial w_{ij}^{(L)}} = a_j^{(L-1)}. \tag{2.5}$$

Substituting in each equation we get,

$$\frac{\partial C_0}{\partial w_{ij}^{(L)}} = 2(a_i^{(L)} - y_i)\sigma'(z_i^{(L)})a_j^{(L-1)}. \tag{2.6}$$

This formula tells us how much influence $w_{ij}^{(L)}$ is having on the cost function, by an almost identical formulation we can calculate the effect each bias has in the final layer has too;

$$\frac{\partial C_0}{\partial b_i^{(L)}} = 2(a_i^{(L)} - y_i)\sigma'(z_i^{(L)}). \tag{2.7}$$

2.1.3. Back Propagation (Hidden Layers)

But what about the previous layer $(L-1)$? The nodes in the previous layer can effect the cost function in multiple ways as seen in figure 2.2.

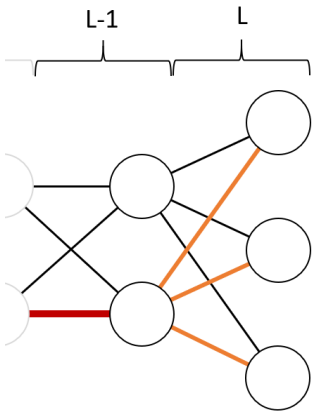


Figure 2.2: A single weight (red) affecting the output layer (L) through several weights (orange).

We must sum the changes over each of these weights.

$$\frac{\partial C_0}{\partial a_j^{(L-1)}} = \sum_{i=1}^{n_L} \frac{\partial z_i^{(L)}}{\partial a_j^{(L-1)}} \frac{\partial a_i^{(L)}}{\partial z_i^{(L)}}, \frac{\partial C_0}{\partial a_i^{(L)}} = \sum_{i=1}^{n_L} 2w_{ij}^{(L)}\sigma'(z_i^{(l)})(a_i^{(L)} - y_i). \tag{2.8}$$

Using this result, we can see how we'd like the nodes in the previous layer to change. We can use this result to determine how the weights in that layer should change too, and so we

propagate backwards through the network accordingly.

$$\begin{aligned}\frac{\partial C_0}{\partial w_{jk}^{(L-1)}} &= \frac{\partial z_j^{(L-1)}}{\partial w_{jk}^{(L-1)}}\frac{\partial a_j^{(L-1)}}{\partial z_j^{(L-1)}}\frac{\partial C_0}{\partial a_j^{(L-1)}} \\ &= \frac{\partial z_j^{(L-1)}}{\partial w_{jk}^{(L-1)}}\frac{\partial a_j^{(L-1)}}{\partial z_j^{(L-1)}}\sum_{i=1}^{n_L}\frac{\partial z_i^{(L)}}{\partial a_j^{(L-1)}}\frac{\partial a_i^{(L)}}{\partial z_i^{(L)}}\frac{\partial C_0}{\partial a_i^{(L)}} \\ &= a_k^{(L-2)}\sigma'(z_j^{(L-1)})\sum_{i=1}^{n_L}2w_{ij}^{(L)}\sigma'(z_i^{(L)})(a_i^{(L)}-y_i),\end{aligned} \tag{2.9}$$

$$\begin{aligned}\frac{\partial C_0}{\partial b_j^{(L-1)}} &= \frac{\partial z_j^{(L-1)}}{\partial b_j^{(L-1)}}\frac{\partial a_j^{(L-1)}}{\partial z_j^{(L-1)}}\frac{\partial C_0}{\partial a_j^{(L-1)}} \\ &= (1)\frac{\partial a_j^{(L-1)}}{\partial z_j^{(L-1)}}\sum_{i=1}^{n_L}\frac{\partial z_i^{(L)}}{\partial a_j^{(L-1)}}\frac{\partial a_i^{(L)}}{\partial z_i^{(L)}}\frac{\partial C_0}{\partial a_i^{(L)}} \\ &= \sigma'(z_j^{(L-1)})\sum_{i=1}^{n_L}2w_{ij}^{(L)}\sigma'(z_i^{(L)})(a_i^{(L)}-y_i).\end{aligned} \tag{2.10}$$

Using these nested summations we can even derive the equations for the $L-2$ layer.

$$\begin{aligned}\frac{\partial C_0}{\partial w_{ks}^{(L-2)}} &= \frac{\partial z_k^{(L-2)}}{\partial w_{ks}^{(L-2)}}\frac{\partial a_k^{(L-2)}}{\partial z_k^{(L-2)}}\sum_{j=1}^{n_{L-1}}\frac{\partial z_j^{(L-1)}}{\partial a_k^{(L-2)}}\frac{\partial a_j^{(L-1)}}{\partial z_j^{(L-1)}}\sum_{i=1}^{n_L}\frac{\partial z_i^{(L)}}{\partial a_j^{(L-1)}}\frac{\partial a_i^{(L)}}{\partial z_i^{(L)}}\frac{\partial C_0}{\partial a_i^{(L)}} \\ &= a_s^{(L-3)}\sigma'(z_k^{(L-2)})\sum_{j=1}^{n_{L-1}}w_{jk}^{(L-1)}\sigma'(z_j^{(L-1)})\sum_{i=1}^{n_L}2w_{ij}^{(L)}\sigma'(z_i^{(L)})(a_i^{(L)}-y_i),\end{aligned} \tag{2.11}$$

$$\begin{aligned}\frac{\partial C_0}{\partial b_k^{(L-2)}} &= \frac{\partial z_k^{(L-2)}}{\partial b_k^{(L-2)}}\frac{\partial a_k^{(L-2)}}{\partial z_k^{(L-2)}}\sum_{j=1}^{n_{L-1}}\frac{\partial z_j^{(L-1)}}{\partial a_k^{(L-2)}}\frac{\partial a_j^{(L-1)}}{\partial z_j^{(L-1)}}\sum_{i=1}^{n_L}\frac{\partial z_i^{(L)}}{\partial a_j^{(L-1)}}\frac{\partial a_i^{(L)}}{\partial z_i^{(L)}}\frac{\partial C_0}{\partial a_i^{(L)}} \\ &= \sigma'(z_k^{(L-2)})\sum_{j=1}^{n_{L-1}}w_{jk}^{(L-1)}\sigma'(z_j^{(L-1)})\sum_{i=1}^{n_L}2w_{ij}^{(L)}\sigma'(z_i^{(L)})(a_i^{(L)}-y_i).\end{aligned} \tag{2.12}$$

This manner of deriving the equations is rather cumbersome so instead we note that there is a recursive pattern.

$$\frac{\partial C_0}{\partial b_i^{(L)}} = 2\sigma'(z_i^{(L)})(a_i^{(L)}-y_i) = \delta(b_i^{(L)}),\quad \frac{\partial C_0}{\partial w_{ij}^{(L)}} = a_j^{(L-1)}\delta(b_i^{(L)}) = \delta(w_{ij}^{(L)}). \tag{2.13}$$

And so the general pattern is;

$$\delta(b_j^{(k-1)}) = \sigma'(z_j^{(k-1)})\sum_{i=1}^{n_k}w_{ij}^{(k)}\delta(b_i^{(k)}),\qquad \delta(w_{ij}^{(k)}) = a_j^{(k-1)}\delta(b_i^{(k)}). \tag{2.14}$$

2.1.4. Back Propagation (Updating the Network)

So now we can observe how the weights and biases in each layer should change to produce a more desirable outcome. However, it is not always best to immediately apply these changes, especially if the single training example we have back propagated on was not particularly representative of most cases. As such, we might like to record the changes for several training examples before applying an average of those recorded changes, we call this set of training examples a batch;

$$w_{ij}^{(L)+} = w_{ij}^{(L)} - \mu * \frac{1}{B}\sum_{r=1}^{B}\frac{\partial C_r}{\partial w_{ij}^{(L)}}, \qquad b_i^{(L)+} = b_i^{(L)} - \mu * \frac{1}{B}\sum_{r=1}^{B}\frac{\partial C_0}{\partial b_i^{(L)}}. \tag{2.15}$$

The learning rate (μ) represents how significantly a single batch affects the weights and biases. More generally it can be thought of as how large our steps towards an optimal solution are as seen in figure 2.3. Larger values of μ will cause large shifts in the weights and biases resulting in the network "jumping around" optimal solutions without effectively converging on towards one (left). Smaller values are thus typically used so that the network converges to an optimal solution more gradually (center), however too small a value increases the likelihood that the network will get stuck in a less than optimal solution (right).

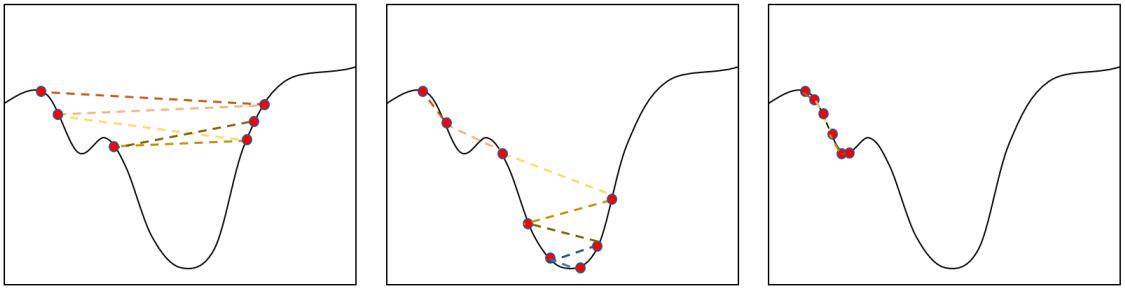


Figure 2.3: 2D visual representation of how varying μ values might affect convergence on an optimal solution. From left to right; large, normal, small.

We call the entire dataset we are training on an epoch. We might want to train the network on the entire dataset multiple times to increase accuracy. Training for 30 epochs would simply mean, training on the entire dataset 30 times. The training examples included in each batch are typically selected at random so that the network doesn't associate any given order to its learning.

2.2. Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) are used when there is a temporal element to the problem, i.e. E.g. Weather patterns, music composition, time series anomaly detection.

This architecture accepts a sequence of input vectors and provides either a sequence of output vectors (pairing each input vector to an output vector) or a single output vector (assigning the entire sequence one output vector). We wish to assign each sleep interval with its own sleep stage, so we shall focus on the former structure.

2.2.1. Structure

Since we now need to account for a sequence of data we need a way of preserving information about the previous time step/s. Lets look at a very simplified example structure in figure 2.4.

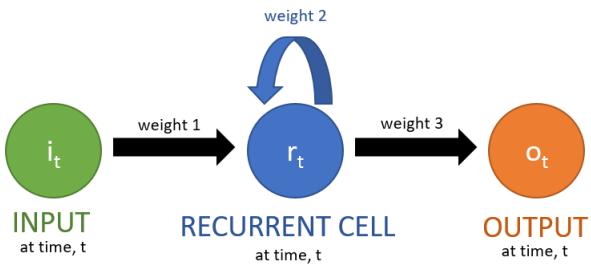


Figure 2.4: A very basic RNN.

At every time step the network is given a new input which (as usual) is fed forward to the next layer, in this case each layer has exactly one cell/node. However, unlike the FFNN the reccurent cell has a weight connecting its current state at $t = t_0$ to its next state at $t = t_0 + 1$. To aid in this understanding we can visually "unfold" the network through time as shown in figure 2.5.

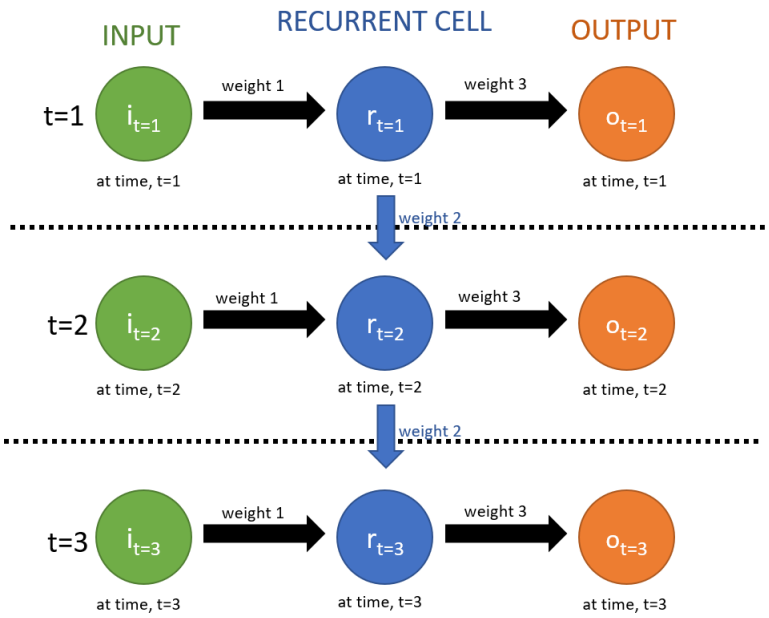


Figure 2.5: An unfolded (in time) RNN.

This recurrent weight allows processed information from the previous time step to influence the next timestep and so on, thus creating an artificial sense of memory. Note that in the unfolded network, although we have added more nodes to express the time dimensionality of the network the weights and biases do not change with respect to time, hence the identical labelling of "weight 1/2/3" for each time step.

2.2.2. Running Example

Let us setup a very basic example. We will use the structure from figure 2.4 and won't include any biases or activation functions for the sake of simplicity. Let us have our network attempt to output a running sum of our inputs, for example an input sequence of $[(1), (2), (3)]$ should ouput a sequence of $[(1), (1+2), (1+2+3)] = [(1), (3), (6)]$.

We start at $t = 0$ with our weights randomised as seen in figure 2.6. Before we input any data the value of every cell/node is set to zero, this is so our recurrent weight passes forward a zero to the $t = 1$ recurrent cell.

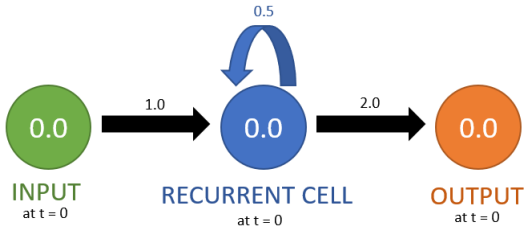


Figure 2.6: The initial RNN before any input vector.

Now we need to pass the RNN our sequence $[(1), (2), (3)]$, starting with (1) as seen in figure 2.7. At $t = 0$ we pass the RNN our 1-dimensional input vector (1) (top). Then we calculate the $t = 1$ recurrent cell value, it has two inputs; the weight from the $t = 1$ input node and the weight from the $t = 0$ reccurent cell. So the resulting value is: $1.0 \times 1.0 + 0.5 \times 0.0 = 1.0$ (middle). Finally we calculate the output node: $2.0 \times 1.0 = 2.0$ (bottom).

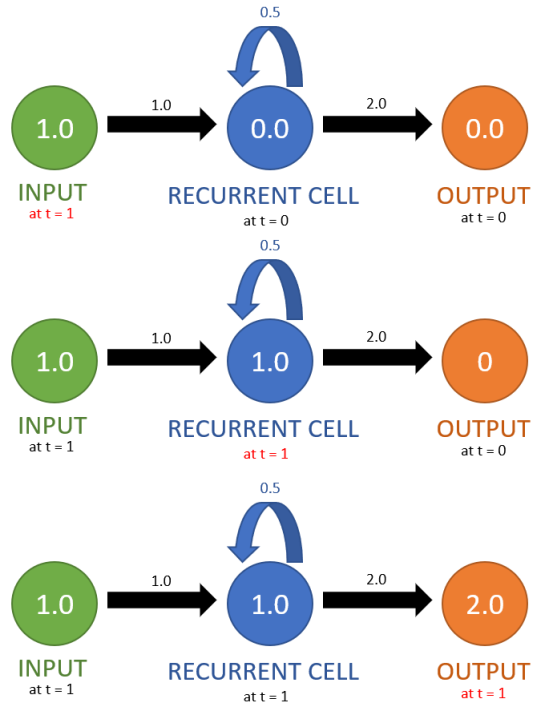


Figure 2.7: Performing a pass forward through the RNN at $t = 1$.

We can see our first output vector for our sequence in (2.0). Now we repeat this process for $t = 2$ where we pass the RNN our next input vector of (2) and then $t = 3$ for the input vector of (3) as seen in figure 2.8.

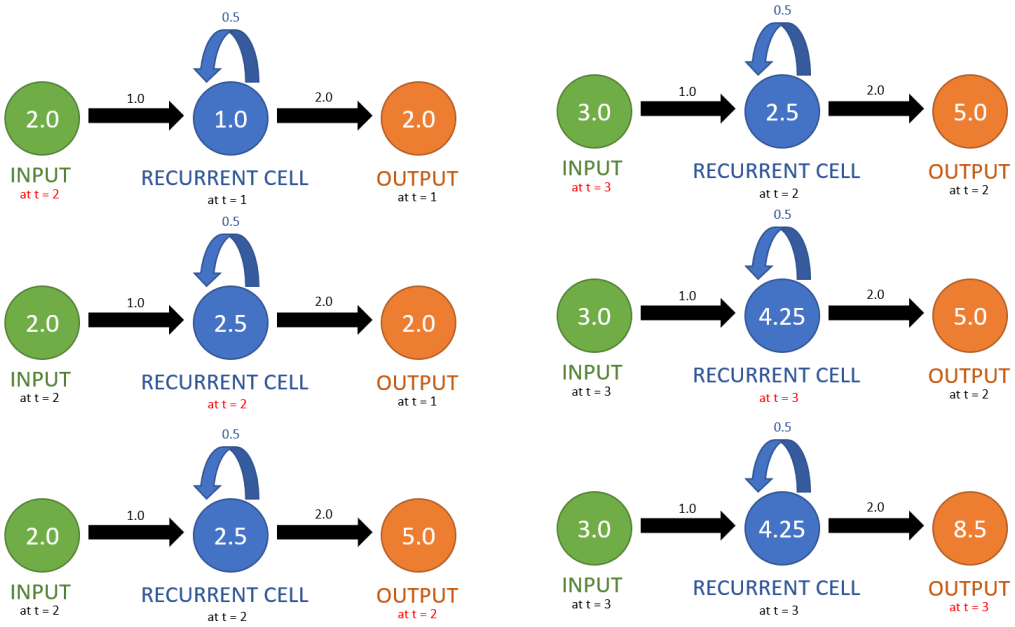


Figure 2.8: Performing a pass forward through the RNN at $t = 2, 3$.

So our final output sequence is $[(2.0), (5.0), (8.5)]$ as opposed to our desired output of $[(1.0), (3.0), (6.0)]$. The error calculation is identical only now we must also average over each timestep. Back propagation is derived in a similar manner; the output weight directly affects the output sequence, the recurrent weight affects the output sequence through the output weight and itself depending on the number of timesteps, and so on. We can see a clearer image of the relations for back propagation by using the unfolded visual as seen in figure 2.9.

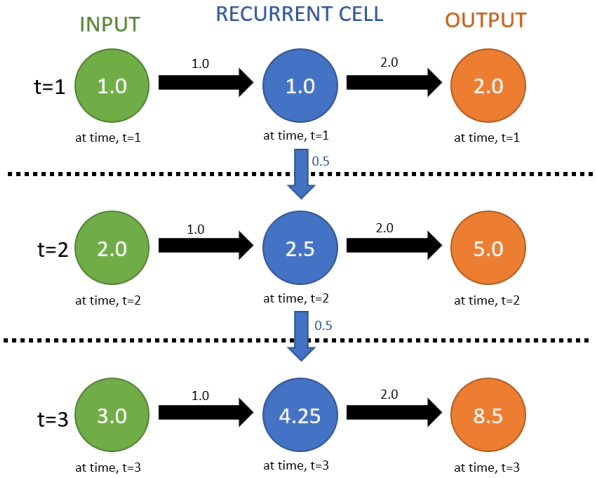


Figure 2.9: Unfolded training example.

2.3. Long Short Term Memory Neural Networks (LSTMs)

The LSTM consists of four sections each responsible for a different function within the network. See figure 2.10.

- Forget - This Network determines what can be forgotten based upon the previous output and the current input.
- Input - This Network determines what information should be added to the current memory based upon the previous output and the current input.
- Memory - This Network generates a new memory based upon the previous output and the current input.
- Output - This Network determines what parts of the memory cell should be the output.

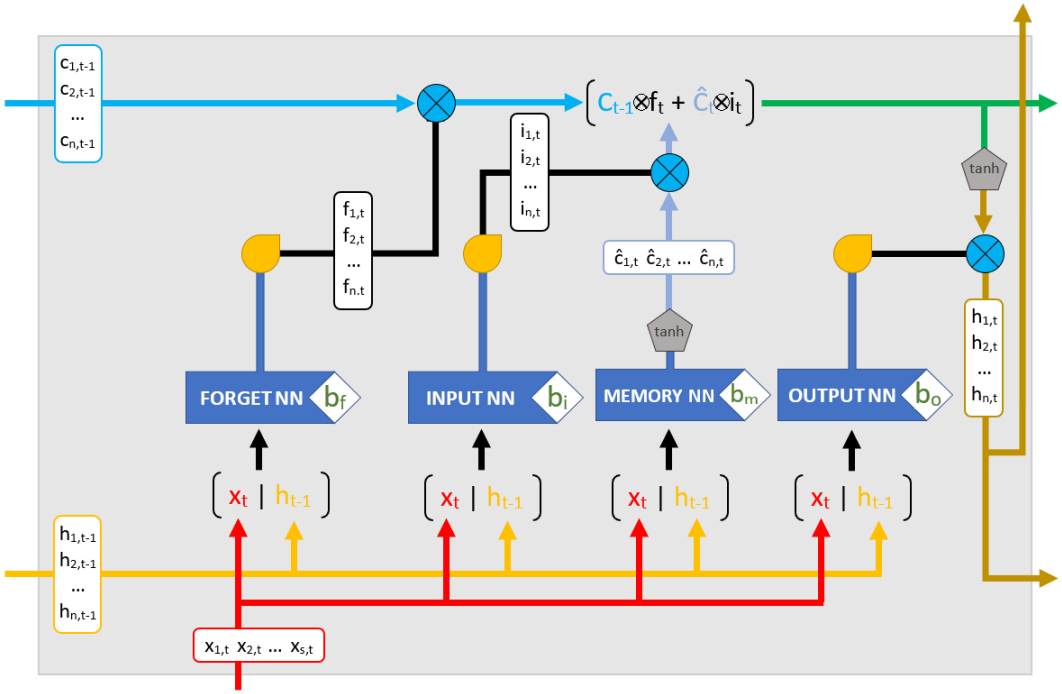


Figure 2.10: LSTM Structure (folded).

To understand what is occurring at each step of the LSTM process we're going to introduce some important notation.

Definition 2.9. *Vector Concatenation;*

$$[\mathbf{a}, \mathbf{b}] = [(a_1, ..., a_m), (b_1, ..., b_n)] = (a_1, ..., a_m, b_1, ..., b_n). \tag{2.16}$$

Element-wise Vector Multiplication;

$$\mathbf{a} \otimes \mathbf{b} = (a_1, ..., a_k) \otimes (b_1, ..., b_k) = (a_1 \times b_1, ..., a_k \times b_k). \tag{2.17}$$

Note the dimensionality of the second identity; dim(A) = dim(B) = k.

Definition 2.10. $\mathbf{x}_t$ *is the input vector at the t timestep with dimension* n_x*.*

Definition 2.11. $\mathbf{h}_t$ *is the output vector at the t timestep with dimension* n_h*.*

Definition 2.12. $\mathbf{c}_t$ *is the updated memory cell/vector at the t timestep with dimension* n_h*.* Note: the output vector and the memory cell are the same dimensionality intentionally since the memory cell serves as a long term output memory.

Definition 2.13. *Weights and Biases.*

- W_f *and* $\mathbf{b}_f$ *are the weight matrix and bias vector associated with the Forget NN, with dimensions* $(n_h, n_x + n_h)$ *and* (n_h) *respectively.*
- W_i *and* $\mathbf{b}_i$ *are the weight matrix and bias vector associated with the Input NN, with dimensions* $(n_h, n_x + n_h)$ *and* (n_h) *respectively.*
- W_m *and* $\mathbf{b}_m$ *are the weight matrix and bias vector associated with the Memory NN, with dimensions* $(n_h, n_x + n_h)$ *and* (n_h) *respectively.*
- W_o *and* $\mathbf{b}_o$ *are the weight matrix and bias vector associated with the Output NN, with dimensions* $(n_h, n_x + n_h)$ *and* (n_h) *respectively.*

Note: Here we're using no hidden layer within each Neural Network, the input layer fully connects to the output layer.

2.3.1. Forget NN

The Forget NN takes the concatenation of the last output vector $\mathbf{h}_{t-1}$ and the current input vector $\mathbf{x}_t$ and generates a Forget vector $\mathbf{f}_t$ of values ranging from 0 (forget) to 1 (remember) using a sigmoid activation function.

$$\mathbf{f}_t = \sigma(W_f \times [\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}_f) \tag{2.18}$$

This vector is then element-wise vector multiplied onto the previous memory cell: $\mathbf{c}_{t-1}$. This has the desired effect of removing (forgetting) certain information from the memory cell. We can liken the forget vector to a series of water valves, and the memory cell to flowing water. Each memory cell value is multiplied by its corresponding forget vector value, a closed valve lets no water through which is the equivalent of a forget vector value of 0, a half open valve lets half the water flow ($f_{i,t} = 0.5$), etc.

2.3.2. Memory NN

Now we want to generate a new memory from the current input and previous output so that it can be added to our current memory cell, since it is a memory cell we use a tanh activation function here. The Memory NN can be though of as a translator, converting the input vectors into memory form.

$$\hat{\mathbf{c}}_t = \tanh\left(W_m \times [\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}_m\right) \tag{2.19}$$

Before its added to the current memory it must first pass through the Input gate.

2.3.3. Input NN

The Input NN functions identically to the Forget NN, it can be thought of as the Forget NN for the Memory NN, as such it takes the concatenation of the last output vector $\mathbf{h}_{t-1}$ and the current input vector $\mathbf{x}_t$ and generates an Input vector $\mathbf{i}_t$ (not to be confused with the input vector $\mathbf{x}_t$) of values ranging from 0 (forget) to 1 (remember) using a sigmoid activation function. The Input NN is essentially deciding what parts of the new memory are unimportant and removing them.

$$\mathbf{i}_t = \sigma(W_i \times [\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}_i) \tag{2.20}$$

The Input vector $\mathbf{i}_t$ is then element-wise vector multiplied onto the new memory cell, $\hat{\mathbf{c}}_t$. This has the desired effect of removing certain information from the new memory cell.

2.3.4. Updating the Memory Cell

Using the previous networks we can now update the memory cell.

$$\mathbf{c}_t = \mathbf{f}_t \otimes \mathbf{c}_{t-1} + \mathbf{i}_t \otimes \hat{\mathbf{c}}_t. \tag{2.21}$$

2.3.5. Output NN

Finally we determine the output of the LSTM. First, we run the new memory cell $\mathbf{c}_t$ through a tanh activation function to make the output values within the desired range $[-1, 1]$. The Output NN then also functions identically to the Forget NN. It can be thought of as the Forget NN for the updated memory cell, as such it takes the concatenation of the last output vector $\mathbf{h}_{t-1}$ and the current input vector $\mathbf{x}_t$ and generates an Ouput vector $\mathbf{i}_t$ (not to be confused with the output vector $\mathbf{h}_t$) of values ranging from 0 (forget) to 1 (remember) using a sigmoid activation function. The Output NN is essentially deciding what parts of the updated memory are to be the output.

$$\mathbf{o}_t = \sigma(W_o \times [\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}_o) \tag{2.22}$$

This vector is then element-wise vector multiplied onto the tanh updated memory cell, $\tanh(\mathbf{c}_{t-1})$.

$$\mathbf{h}_t = \mathbf{o}_t \otimes \tanh(\mathbf{c}_t) \tag{2.23}$$

The resulting product is the output vector of the LSTM. Both $\mathbf{h}_t$ (short term memory) and $\mathbf{c}_t$ (long term memory) are carried forward.

3

Layers, Nodes and Tuning Parameters

Now we understand the various parameters and features of a networks underlying structure, we can more closely examine how each feature affects the networks performance, as well as some problem areas we shall need to account for.

3.1. Layers

One of the most important features of a neural network is the number of layers. In almost all cases we have an input layer, an output layer and then any number of hidden layers. A network with one hidden layer is called a shallow network, a network with more than one hidden layer is called a deep network.

3.1.1. Shallow Learning

Generally speaking a single layer is capable of learning the vast majority of problem solutions. This ability of a single hidden layer is summarised as the Universal Approximation theorem.

Theorem 3.1. *[2] Let $\varphi : \mathbb{R} \to \mathbb{R}$ be a nonconstant, bounded, and continuous function (called the activation function). Let I_m denote the m-dimensional unit hypercube $[0,1]^m$. The space of real-valued continuous functions on I_m is denoted by $C(I_m)$. Then, given any $\varepsilon > 0$ and any function $f \in C(I_m)$, there exist an integer N, real constants $v_i, b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}^m$ for $i = 1, \ldots, N$, such that we may define:*

$$F(x) = \sum_{i=1}^{N} v_i \varphi\left(w_i^T x + b_i\right) \tag{3.1}$$

as an approximate realization of the function f; that is,

$$|F(x) - f(x)| < \varepsilon \tag{3.2}$$

for all $x \in I_m$.

So a single layer can approximate any function that contains a continuous mapping from one finite space to another [3]. However the theorem speaks nothing of the learnability of such a solution, only its existance. For problems that are particularly troublesome to learn, we can add more layers.

3.2. Nodes

Another important feature of a network is the number of nodes in each layer. Clearly the input layer size is dictated by the data and the output layer size is dictated by the labels, however the real question is how many nodes should we have in our hidden layer/s.

3.2.1. Large Layers and Overfitting

More nodes increases the degree to which the network can be precise. Precision is obviously of benefit to any model however a problem arises when the network becomes too specific to the training data and thus loses it ability to predict general patterns it hasn't been trained on. See figure 3.1.

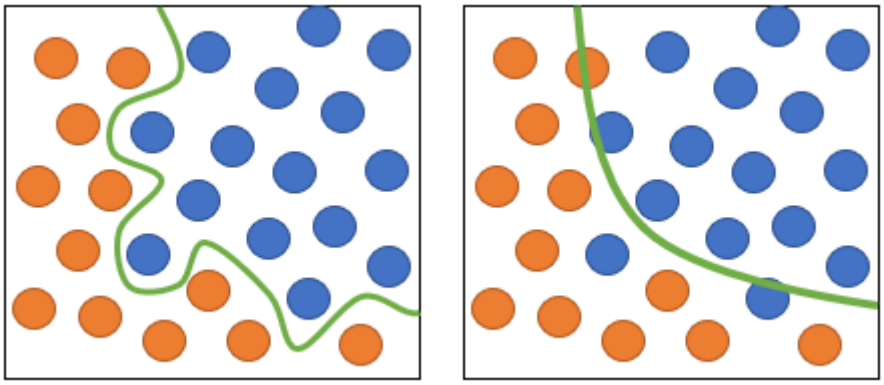


Figure 3.1: A visual representation of overfitting.

3.2.2. Small Layers and Underfitting

Fewer nodes decreases the learning time. Faster training is obviously very helpful when tuning a network however the tradeoff is a less capable network that generalises too much. See figure 3.2.

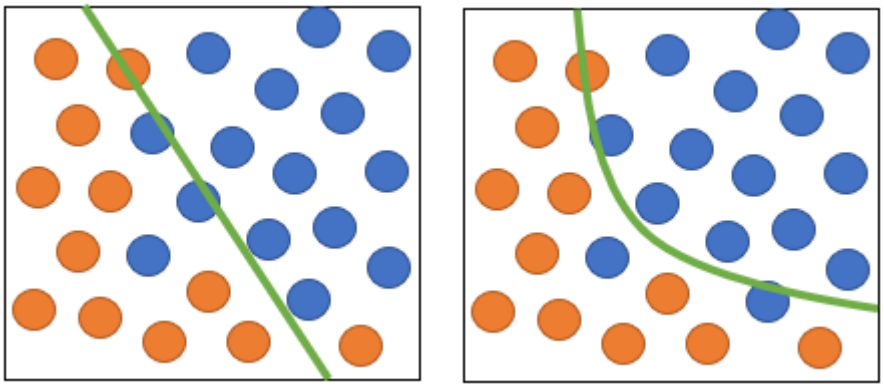


Figure 3.2: A visual representation of underfitting.

3.3. Problem Areas and Solutions

There are a few issues that can arise when constructing and training a network, fortunately there are solutions to these problems.

3.3.1. Overfitting and Underfitting

As covered in the previous section, overfitting and underfitting can result is poor preformance. The most common solution to this kind of problem is pruning.

Pruning is a trial and error method of reaching the most effective layer sizes. You run the network with a certain configuration, change the parameters slightly and run the network again to see if the change produces a more accurate result. Hoever, there are some general rules-of-thumb that have been observed to work most effectively when it comes to the number of nodes in a layer [3]:

- Nodes should be between input and output size.
- Nodes should be equal to two thirds of the input size + output size.
- Nodes should be less than twice the input layer size.

3.3.2. Learning Rate

Recall that the learning rate (μ) is the amount a single training example affects the changes to the weights and biases.

If the learning rate is too large, the network treats each new piece of information like a trauma, overwriting more of the previously learned behaviour. This results in a network that makes large "jumps" in accuracy and inaccuracy with very little progress made. See figure 3.3.

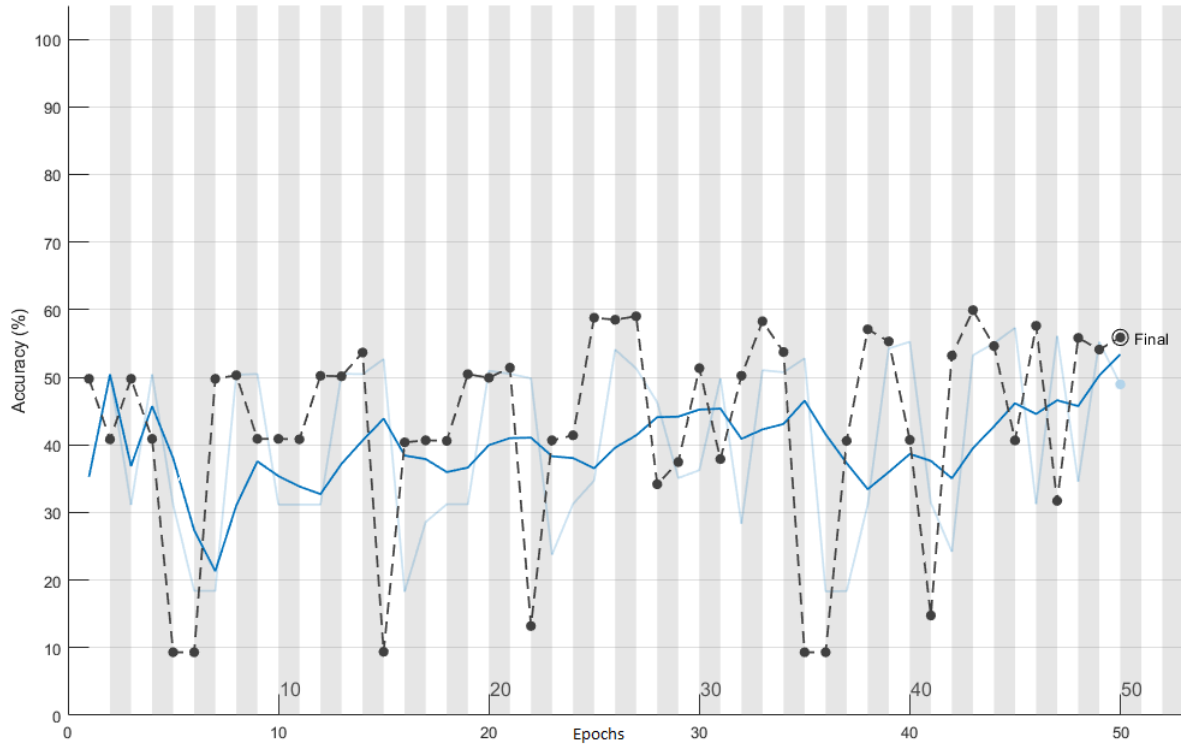


Figure 3.3: Training our LSTM (as later detailed in section 4.3.) on sleep data with a high learning rate ($\mu = 0.1$).

If the learning rate is too small, the network treats each new piece of information with very little regard, barely impacting the network state. This behaviour results in flatlining and weak learning gradients. See figure 3.4.

3.3.3. Exploding and Vanishing Gradients

As we saw in our formulation of a feed forward neural network back in subsection 2.1.3., we derived the recurrsive formula for back propagation.

$$\delta(b_j^{(k-1)}) = \sigma'(z_j^{(k-1)}) \sum_{i=1}^{n_k} w_{ij}^{(k)} \delta(b_i^{(k)}), \qquad \delta(w_{ij}^{(k)}) = a_j^{(k-1)} \delta(b_i^{(k)}). \tag{3.3}$$

We can see the change each layer we propagate through is a series of summations and multiplications. If the weights and nodes of the network are less than 1, then the changes to the made to the weights and biases become smaller the further we propagate backwards (vanishing

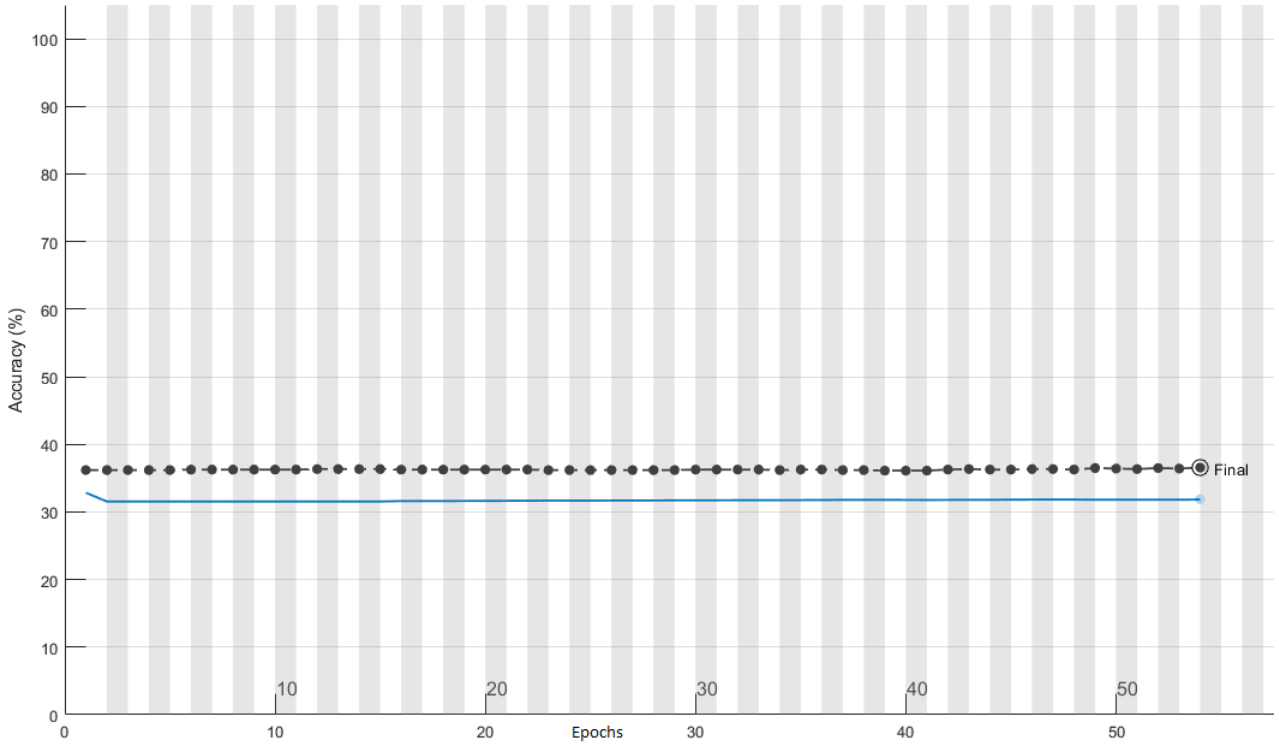


Figure 3.4: Training our LSTM (as later detailed in section 4.3.) on sleep data with a low learning rate ($\mu = 0.0001$).

gradient). Likewise, if the weights and nodes of the network are greater than 1, then the changes made to the weights and biases become larger the further we propagate backwards (exploding gradient). This is of course primarily a problem for deep networks (networks with more than one hidden layer). A vanishing gradient will fail to affect the earlier layer weights, thus impacting learning potential, and an exploding gradient will affect the early layer weights too greatly, thus negating the effect of later layers.

One of the best solutions to prevent vanishing gradient is to use activation functions that don't compress node values into the $[0, 1]$ range, such as a ReLu function. Exploding gradients are usually handled with a gradient threshold; an upper limit to the value of a weight and bias. The most common gradient threshold used is two (2) as it allows for weights to potentially double a starting nodes value into a stoppiung node.

4 Sleep Staging with Neural Networks

In this chapter we define and train a LSTM network on Electrocardiogram (ECG) sleep data in an attempt to accurately classify the sleep stages. ECG data is a measure of the electrical activity of the heart [4]. We will need to look at the data, prepare it for analysis, convert it into a series of input vectors (and their associated labels) and feed it into our network.

4.1. The Data

Our data takes the form of five ECG profiles provided by the Surrey Sleep Research Centre. Each of these profiles details one night of sleep taken over a 12 hour period, with each 30 second interval being scored as one of five sleep stage labels (N1, N2, N3, Wake and REM).

Each profile has three text documents associated with it; the timed ECG data,the timed labels and the markers, signalling the start and end of the ECG recording, as displayed in figure 4.1. We first need to extract this data into a more accessible program.

4.1.1. Extracting the Data

Looking at the label text document (center) we can see when the first label is applied (07.03.2017 23:02:00,000; Wake). It is clear we will need to take our ECG signal (left) from the start of this 30 second interval till the end of the last labelled interval.

We export this text document into MATLAB and make use of the import function to extract the full column containing ECG signal and the associated labels. Doing this for each profile we can now proceed to examine our data.

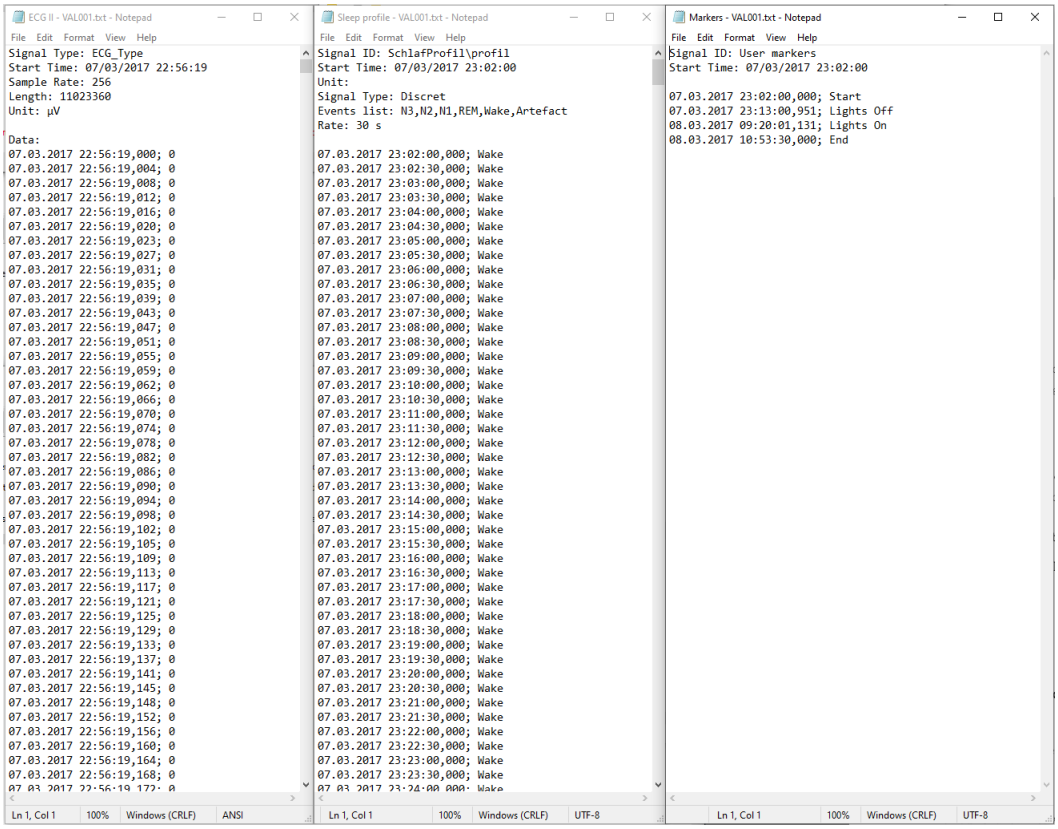


Figure 4.1: Text Files from left to right: ECG data, Labels and Markers

Let us inspect the raw ECG signal from the first individuals profile covering a 12 hour period in figure 4.2 as well as the corresponding hypnogram in figure 4.3. Here colour is used to distinguish the separate sleep stages. (Note that we can group N1, N2 and N3 into NREM (non-REM) (bottom), this is to create a simpler classification problem should five stages be too complicated for our network.) We can see our first major problem area; noise. Due to the methodology of certain peak detection algorithms the giant peaks in the ECG signal will interfere with our ability to process the data in an accurate manner, so the raw signal will need to undergo some form of normalisation process prior to feature extraction. Feature extraction is a process by which certain features from the data (such as heartrate) are extracted and fed into our network instead of the entire raw ECG signal, this helps the network focus on important elements that may have a more direct relation to the sleep stages as well as reducing training time.

Up close in figure 4.4 we can see the structure of the ECG signal more clearly. The high peaks are called R peaks and their frequency and variance are two of the most commonly extracted features in modern studies [5] [6], this is due to the high correlation to the sleep stages [7]. We

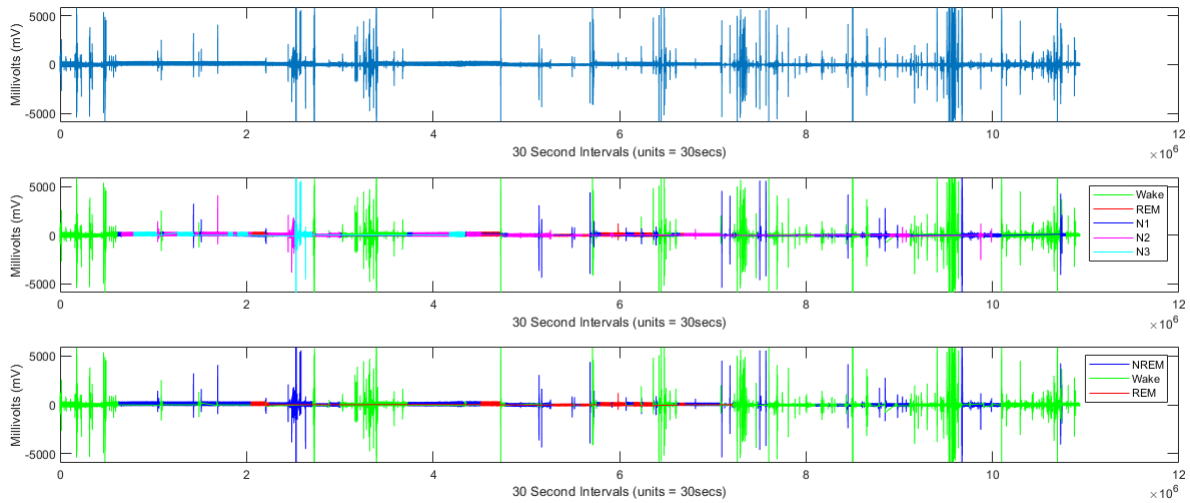


Figure 4.2: The first profiles raw ECG signal (12 hour interval). Top: Raw ECG signal. Middle: Raw ECG signal colour coded by 5 sleep stages. Bottom: Raw ECG signal colour coded by 3 sleep stages.

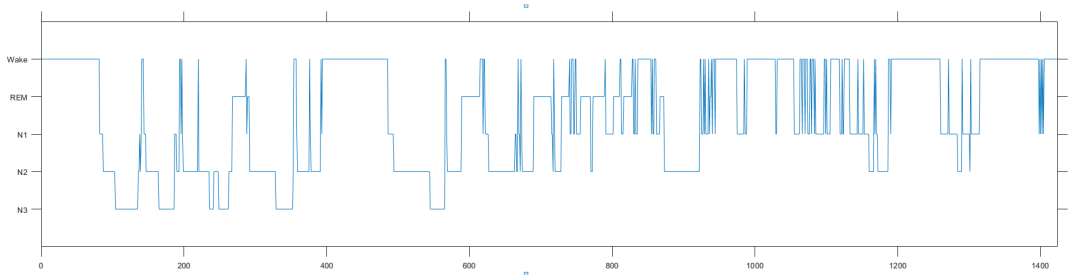


Figure 4.3: The first profiles hypnogram in 30 second time intervals.

can see how large spikes in the data create sizable differences between these R peaks in figure 4.5, these differences cause certain algorithms to label only the large peaks or to ignore them entirely which is undesirable.

4.1.2. Artifacts

The fifth profile contains data labelled as “Artifact” at many points throughout the sleep profile. I tried to add a sixth category but this vastly decreased the accuracy of the other profile scores, so in the end (mostly due to time constraints) I decided to ignore the profile altogether. Depsite this there are many ways around this kind of issue, for example, I could train a separate network

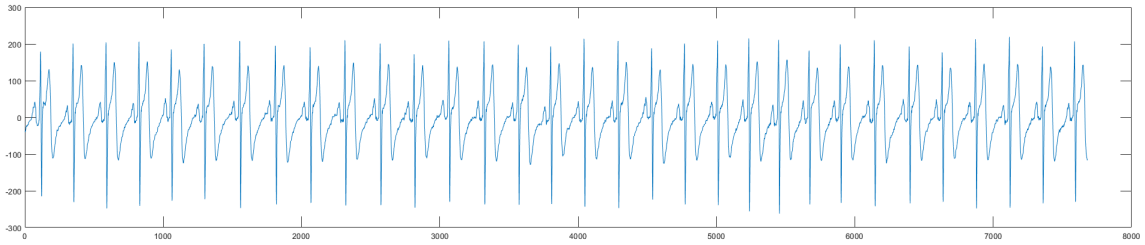


Figure 4.4: A typical 30 second interval of the raw ECG signal.

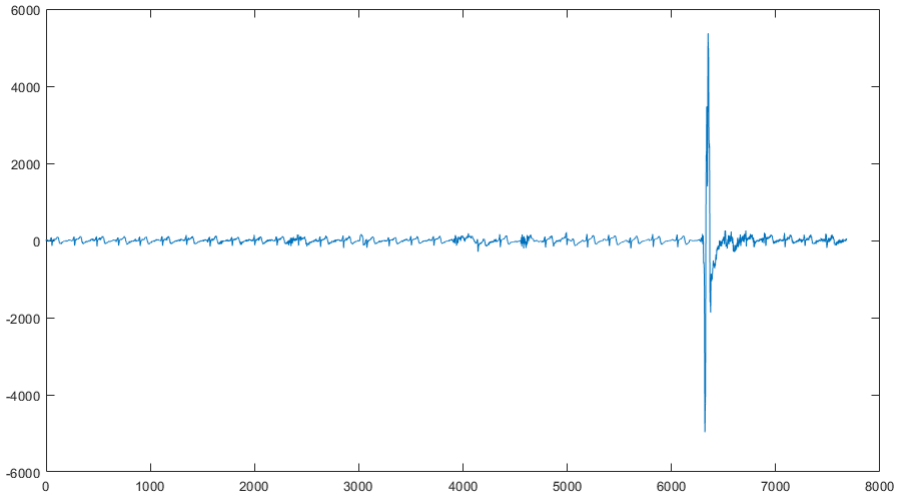


Figure 4.5: A 30 second interval with a noisy peak.

to detect and omit artifacts from unlabelled data.

Average	Variance	Label
0.9191	2.9784	Wake
0.8024	8.9512	Wake
0.9211	2.8909	Wake
0.9499	0.2144	N1
0.9536	0.8725	N1
...	...	...

Table 4.1: A small example of the dataset.

4.2. Data Preprocessing

Firstly we must convert the raw ECG signal into a form the network can work with as using the entire raw signal would be a lengthy and ineffecient endeavor (due to the size of the data set) with no guarantee that the network will abstract the underlying features of each sleep stage. We take two different approaches:

- Average R to R Interval and Variance
- Wavelets

4.2.1. R to R Intervals

In this approach we feed the network the mean R to R Interval (in seconds) and variance (in 256^{th}s of a second) as a two feature input for each 30 second segment. The network will label each time step as a sequence as seen in table 4.1.

Note: The unit difference for average and variance is to ensure the variance values are not too small to be accounted for.

However, we must first extract the R peaks from the raw ECG signal, we do this by normalising the signal (to remove noise, etc.) and then using peak detection. There are three normalisation approaches we take:

- Outlier Fill and Rescale; A MATLAB fill outliers algorithm fills the outliers before a rescale between -1 and 1.
- Fixed Window Average; Each 30 second interval is rescaled between -1 and 1.
- Moving Window Average; A fixed interval "window" is passed through the entire signal, each point is an average of the normalisations applied to it.

Below, in figure 4.6, we can see a comparison of the methods of normalisation. First are the fixed/moving window averages.

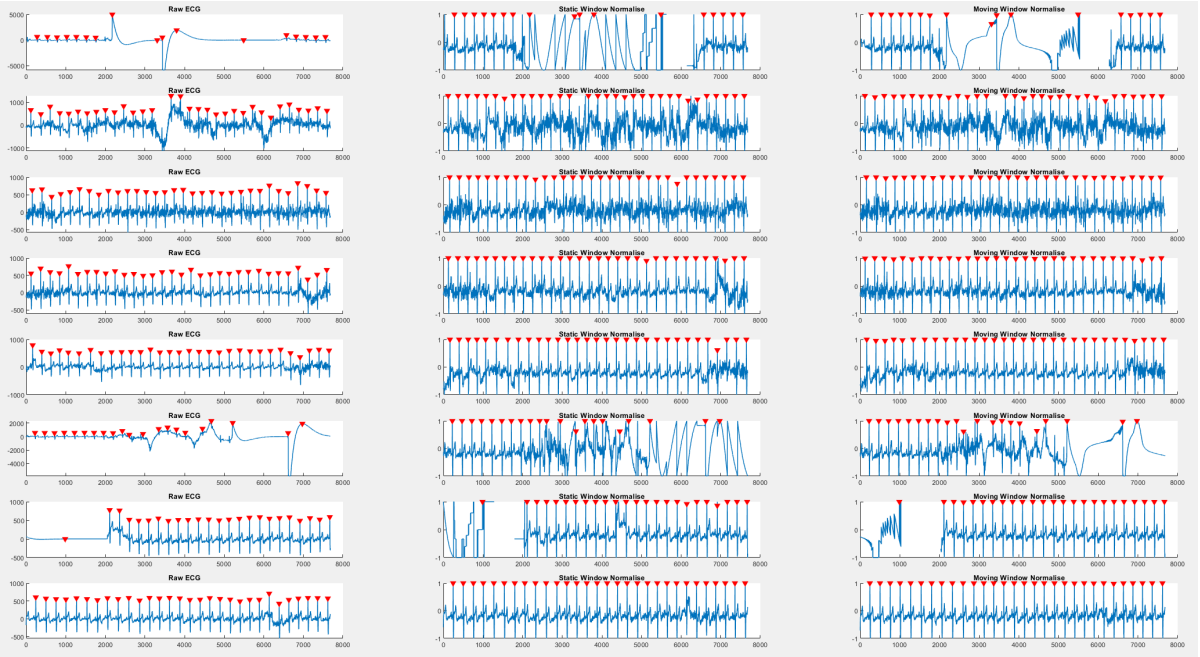


Figure 4.6: Signals from left to right: Raw ECG, Static Window and Moving Window

As we can see, both methods do a fairly good job at identifying R peaks. The advantage of the moving window average however, is that is does a better job of reducing baseline wander [8] (the gradual shifting of the peaks center). Now let us look at MATLAB's outlier fill and rescale method in figure 4.7.

The method clearly is able to label R peaks with some consistency, however the noise reduction is almost non existant and as such there tends to be some questionable R peak labels. After some inspection of the various profiles the moving window method proved to label R peaks with more consistency and accuracy due to its ability to handle the noisy raw ECG data. The moving window also preserves the underlying shape of the data better since each point is an average of all local neighbourhoods surrounding it.

4.2.2. Wavelets

In this approach we take a wavelet of each 30 second interval, as displayed in figure 4.8. The issue we have with feeding our raw data into the network is that the dataset is too large, a wavelet is a reduced dataset that preserves the structure of the original signal (like a low res-

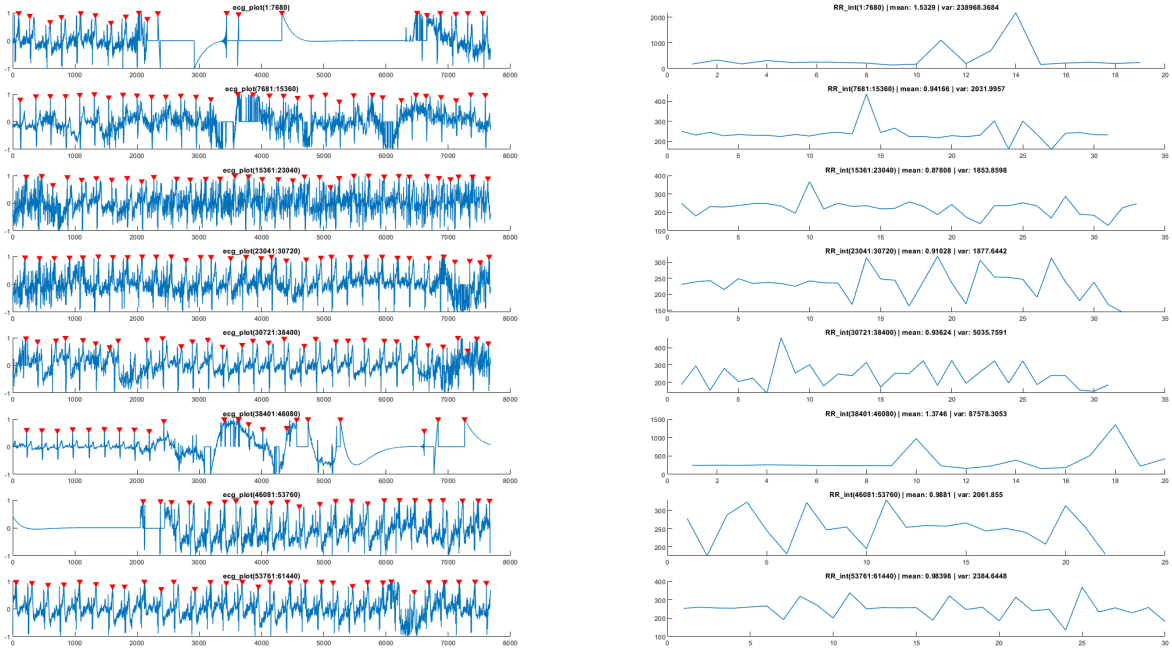


Figure 4.7: On the left: Raw ECG with labelled R peaks. On the right: A plot of each peaks intensity.

olution image). The wavelet transform takes a time series and transforms it in to two sets of coefficients, average and detail. Unlike the previous approach, we feed the network the wavelet and it is assigned a single label as a separate entity (that is, it treats each 30 second wavelet as being disconnected from the one prior). The idea here is to see if the network can correctly label an individual 30 second interval given no other information about the rest of the ECG data (e.g. what label came before).

Each wavelet can be described using its detail coefficient. The higher the detail coefficient the lower the resolution. Its most feasible to train with detail 2 and 3 coefficients as detail 1 is still far to sizeable and anything past detail 3 doesn't give the R peaks enough resolution to be accounted for by the network (the peaks are too "blurry").

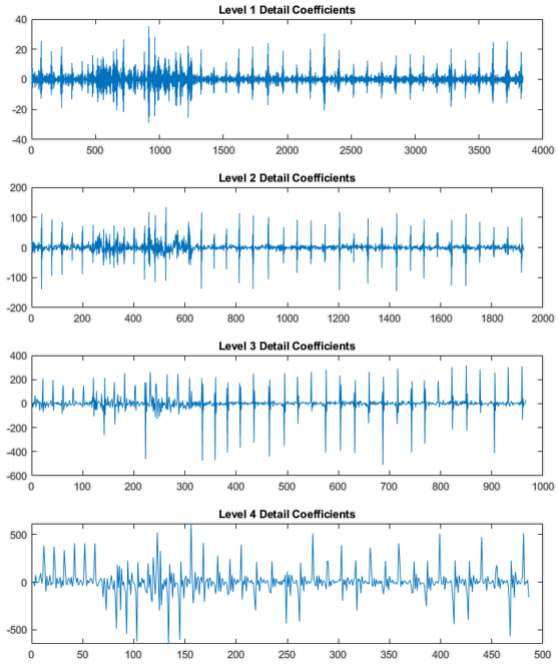


Figure 4.8: Wavelets of the Raw ECG signal in incremental detail coefficients.

4.3. The Architectures

So now given all we know about LSTMs and the various architectures lets define the LSTM architectures we intend on using.

4.3.1. Layers

For our network the layer structure is as follows:

- Input Layer: Takes input data.
- LSTM Layer: We use for sequence labelling.
- Fully Connected Layer: Our output layer.
- Softmax Layer: Our activation function for classification.
- Classification Layer: Uses Cross Entropy to determine Loss.

4.3.2. Nodes

The node configuration is as follows:

- Input Layer: 2 (R to R) or 1 (Wavelets) node/s depending upon the data set.
- LSTM Layer: 200 Nodes
- Fully Connected Layer: 5 Nodes (for all labels) or 3 Nodes (for reduced labels)

Here we leave space to train either all labels (Wake, N1, N2, N3, REM) or reduced labels (Wake, NREM, REM).

4.3.3. Parameters

The parameter configuration is as follows:

- Learning Rate: 0.001 (found via pruning the network)
- Gradient Threshold: 2 (Allows for weight doubling without exploding gradient)

4.4. Results

For our results we use MATLAB's deep learning toolbox and neural net training plot to display our training progress. The blue line represents the networks accuracy on its training set, the thicker black line represents the accuracy of the testing set.

4.4.1. R to R Intervals

When training the network to classify all five sleep stages with R to R interval average and variance, it became clear that the network would consistently classify only three labels (N2, REM and Wake) as seen in figure 4.9. As such, we train on the reduced label set (NREM, REM and Wake).

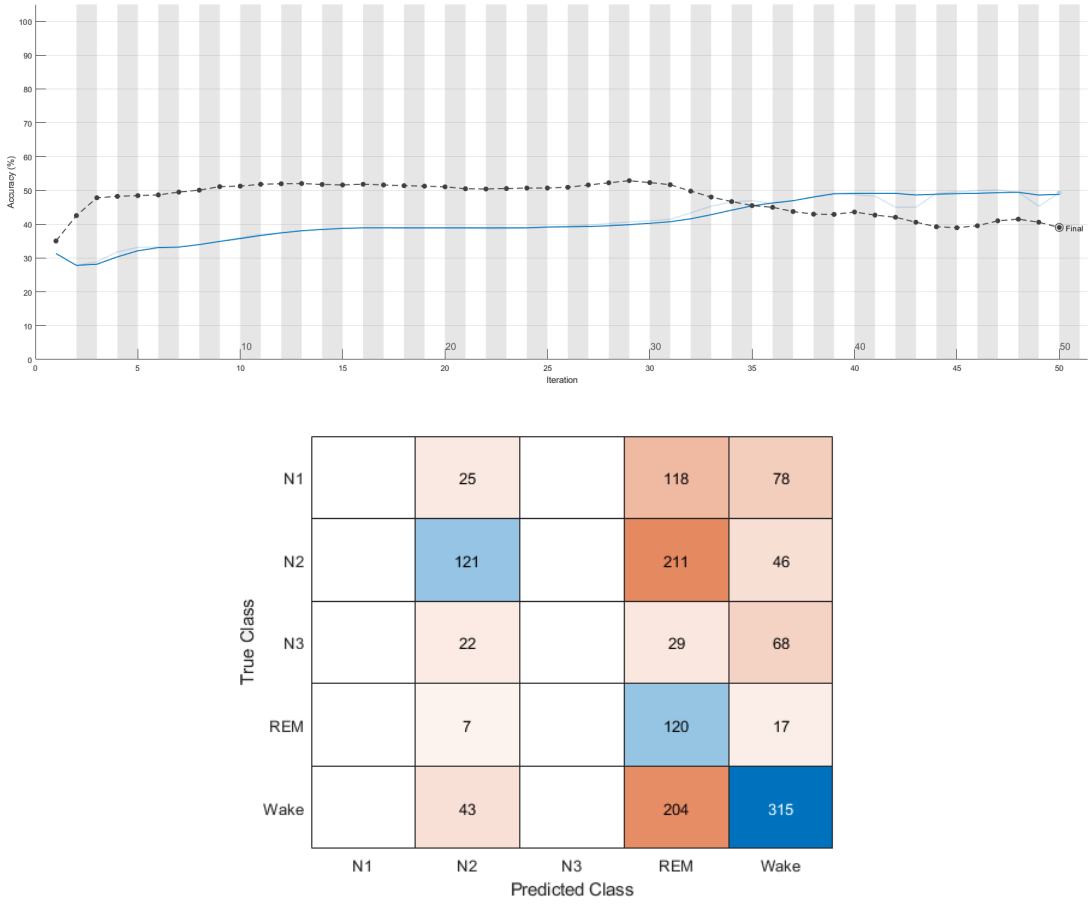


Figure 4.9: Training on all five labels with profile 1.

Firstly, we cover the results for the R to R interval average and variance. In each result we train on three profiles and test on a fourth.

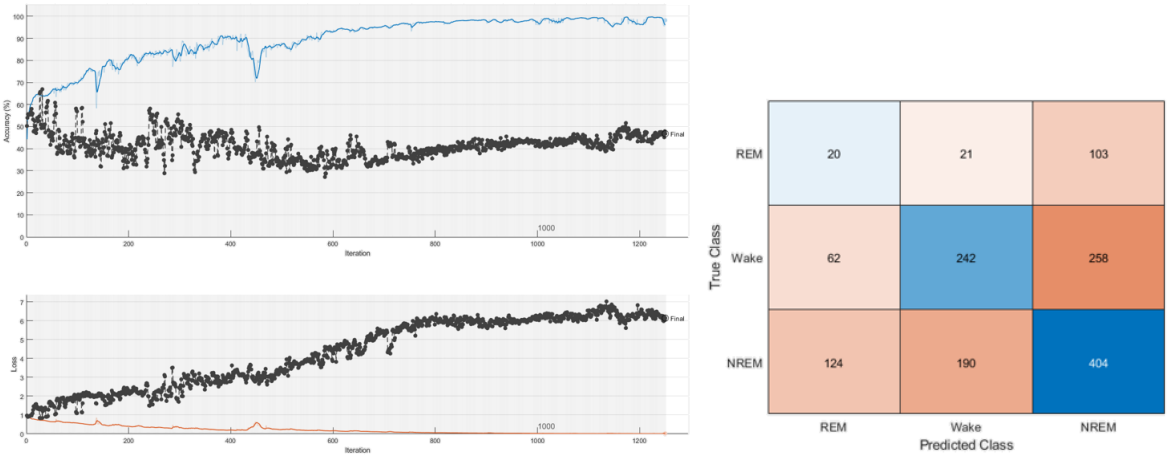


Figure 4.10: Test on Profile 1.

Test on Profile 1 in figure 4.10 Result: 50%. We see here that the biggest problem areas for this network are distinguishing between Wake and NREM, this is a common theme amongst the other profiles.

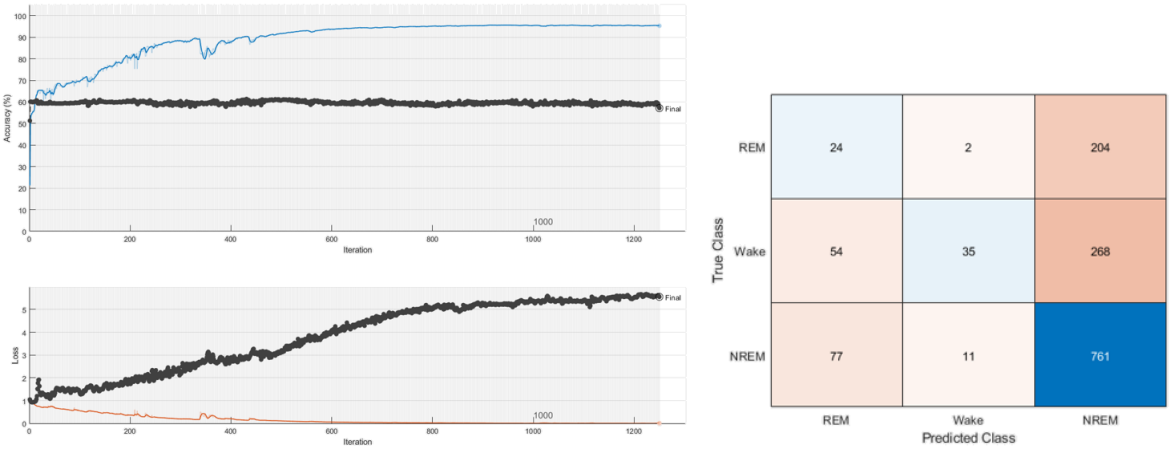


Figure 4.11: Test on Profile 2.

Test on Profile 2 in figure 4.11 Result: 60%. The network here has labelled almost all entries as NREM, hence the very unchanging testing line. This is common amongst networks that have a high volume of a certain kind of label; the network concludes that the best strategy is a blanket label with very little complexity.

Test on Profile 3 in figure 4.12 Result: 45%. Two points of interest with this result. Wake is most accurately labelled (rare occurrence) and instead of mislabelling Wake and NREM the

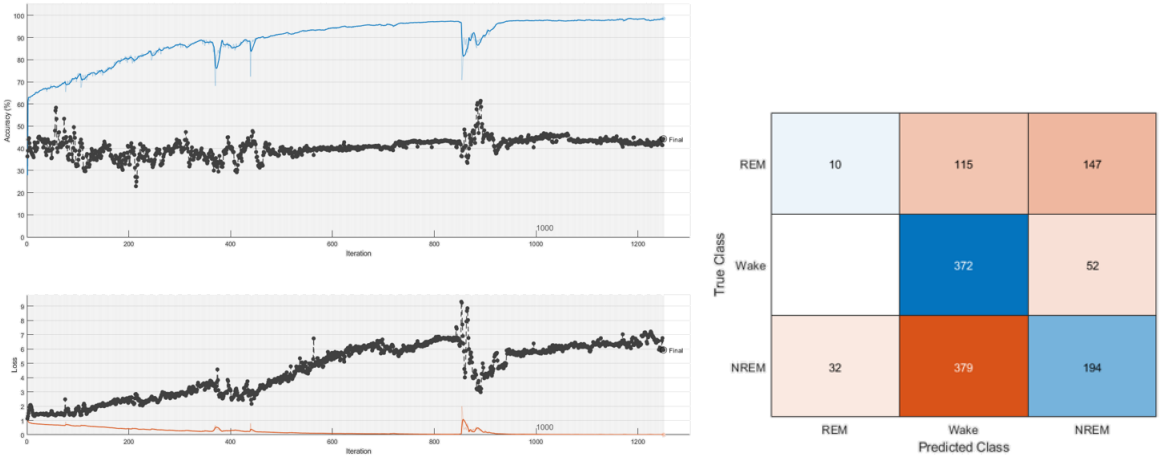


Figure 4.12: Test on Profile 3.

reverse mislabelling has occurred very frequently.

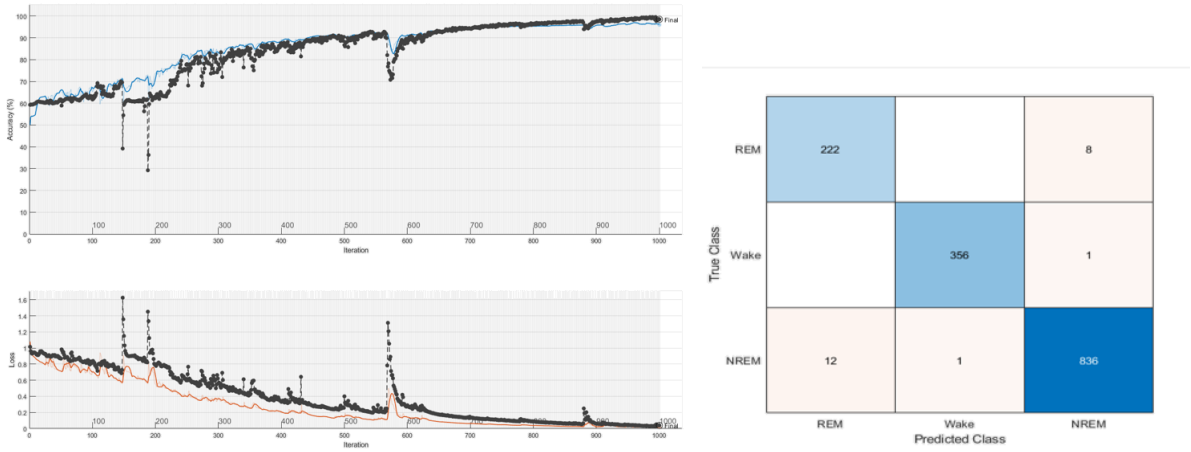


Figure 4.13: Test on Profile 4.

Test on Profile 4 in figure 4.13 Result: 98%. This is a very suprising result given the accuracy scores of all other profiles. One explanation could be that this individuals ECG is very typical of the average persons.

4.4.2. Wavelets

Lastly, the results for the Wavelets. In each result we train on a selection from profile 1 and test on the unselected.

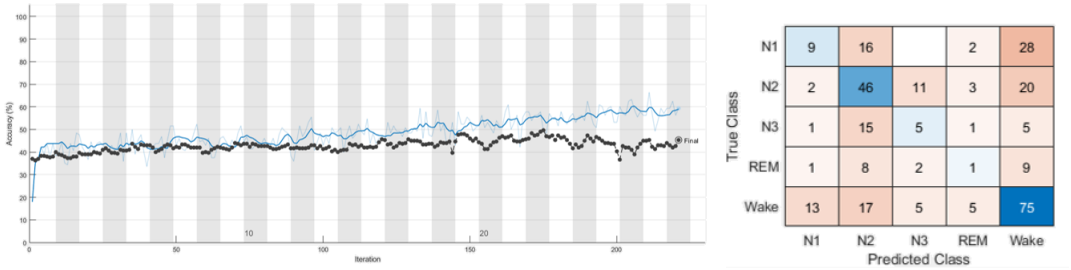


Figure 4.14: Test on Profile 1.

Test on Profile 1 in figure 4.14 w/ 2nd Degree Result: 45%. We can see that the network finds labelling Wake and N2 easiest.

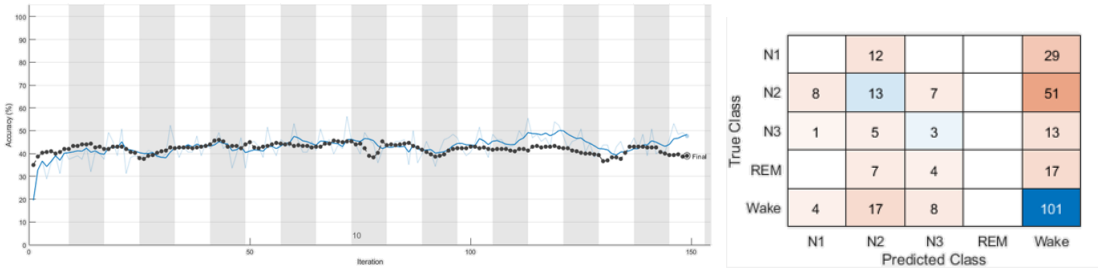


Figure 4.15: Test on Profile 1.

Test on Profile 1 in figure 4.15 w/ 3rd Degree Result: 40%. As with the previous result REM is so rare that the network finds it optimal to ignore the label altogether.

5 Conclusions and Outlook

It is clear from the results that each profile varies quite significantly from the next. As such the greatest limitation faced when training is the lack of avaliable data, for comparison one recent study in 2019 achieved a 95% confidence interval using 8682 polysomnograms [6], our report uses four raw ECG signals. This limitation prevents the network from abstracting a greater insight into what determines a sleep stage, therefore the network takes the very few profiles provided and places too much emphasis upon them, hence why the network testing accuracy (black line) decreases as the training accuracy (blue line) increases, this effect is called overfitting.

Another limitation is of course the algorithmic labelling of R peaks as there is much progress to be made in the creation of algorithms that perform accurately with the most common default being the Pan-Tompkins algorithm [9]. Processing time also proved quite a large obstacle as pruning a network that requires a lot of processing time is slightly unfeasible.

If provided more time and data it would have been of interest to explore several areas:

- The specific mathematics behind back propagation for a RNN/LSTM.
- The analytical solutions to problem areas such as the vanishing/exploding gradient.
- Bidirectional LSTMs and the reasons behind their improved accuracy ratings.
- A working implementation of the Pan-Tompkins R peak detection algorithm for more accurate R peak labelling.
- Training and testing on multiple nights of sleep from the same subject, as an ECG signal is quite unique to each individual.

Overall, I believe this report has served as a good introduction to the mathematical formulation of LSTMs and their applications to sleep stages. As the researching process continued

I found myself spending increasing amounts of time handling data preprocessing rather than focusing on the more fundamental aspects of the neural network itself. In many ways this report has served as an excellent tool for my growth and understanding of the neural networks involved in time series analysis as well as how to go about explain the process of researching a subject matter.

References

[1] A. Burkov, *The Hundred-Page Machine Learning Book,* .

[2] G. Cybenko, *Approximation by superpositions of a sigmoidal function,* Math. Control Signal Systems 2(1989) 303–314.

[3] J. Heaton, *The Number of Hidden Layers,* .

[4] D. L. Kulick, *Electrocardiogram (ECG or EKG),* .

[5] M. A. Radha M, Fonseca P et al., *Sleep stage classification from heart-rate variability using long short-term memory neural networks.,* .

[6] W. G. Haoqi Sun et al., *Sleep staging from electrocardiography and respiration with deep learning,* .

[7] G. A. Raetz SL, Richard CA and H. RM., *Dynamic characteristics of cardiac R-R intervals during sleep and waking states.,* .

[8] H. Chen and S. Chen, *A Moving Average based Filtering System with its Application to Real-time QRS Detection .,* .

[9] J. Pan and W. J. Tompkins, *A Real-Time QRS Detection Algorithm,* IEEE Transactions on Biomedical Engineering B **ME-32** (1985) 230.