An Introduction to the Backpropagation Algorithm and Deep Neural Networks

James Fitzpatrick

0.1 The Artificial Neural Network

In the last twenty years, machine learning has exploded in popularity, and from behind the scenes, it drives much of the modern economy. Machine learning encompasses a broad range of methods, but perhaps the most well known is the artificial neural network and, in particular, deep learning, which employs the use of extremely large neural networks for data classification, analysis and forecasting. Neural networks surfaced in force the 1980's and had 'died' by the end of the decade. Two major problems beset them: the lack of computing power and the absence of huge sets of labelled and unlabelled data. Such problems no longer exist and so they now present realistic solutions to many problems that are usually extremely difficult for computing machines.

0.1.1 What Neural Networks Do

One way of thinking about the function of an artificial neural network is as a method of curve fitting for a set of noisy data. This idea is easily visualised for two dimensional numerical data where, for example, we could be trying to find a function that, given a set of inputs, would reliably reproduce a set of experimental results. Experimental results come with inherent measurement errors so it often makes sense to perform a least squares fitting in order to find the curve that best describes the data.

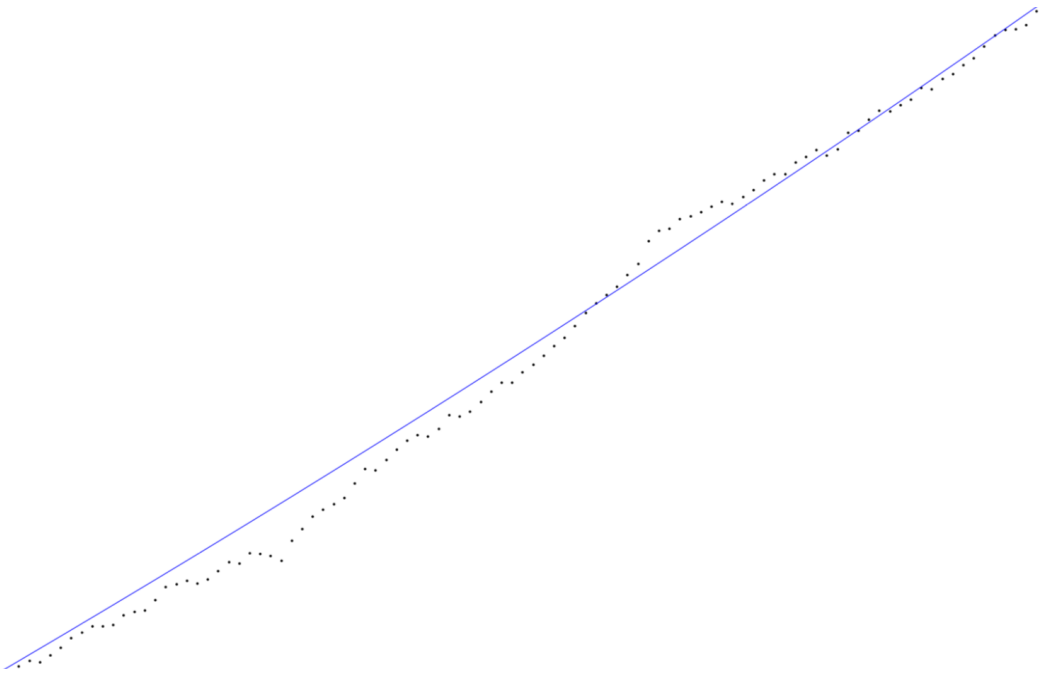

Figure 1: Typically the datapoints will not lie along the line of best fit, but they will lie very close to it. Reduction in experimetnal error can bring them closer but this is often difficult.

If we train the network for long enough, it is possible that we can come arbitrarily close to matching the datapoints exactly. Such training often takes a *very* long

time, however. It is also equally likely that we might not, that we might get trapped in a situation where we do not come close to a good fit, especially if the function describing the data is a highly non-linear function of its arguments.

Convergence to a good solution is not guaranteed and even if it is, it might take so long that we may know the outcome of a process that we wanted to predict before the network can predict it. It therefore makes sense to ask ourselves: why exactly are neural networks so useful and successful if we could use older, quicker methods to find these best fit curves? The answer is that although neural networks are computationally expensive to train, they can quickly and cheaply produce predictions when they have been trained well. They can be made to be extremely insensitive to noisy data and so can predict outcomes that they have never experienced to a high degree of accuracy. Neural networks are also extremely good at picking up on non-linear relationships that humans find difficult to see and so can tell us about aspects of data that would otherwise go unnoticed. Perhaps the most important use of the neural network is that they can process numerical representations of non-numerical data and identify patterns. This concept extends to the idea of natural language forecasting, image recognition, image colorisation and video editing. Indeed, even for purely numerical data, neural networks deal with high-dimensional data with little more difficulty than they do with data we can represent in two dimensions.

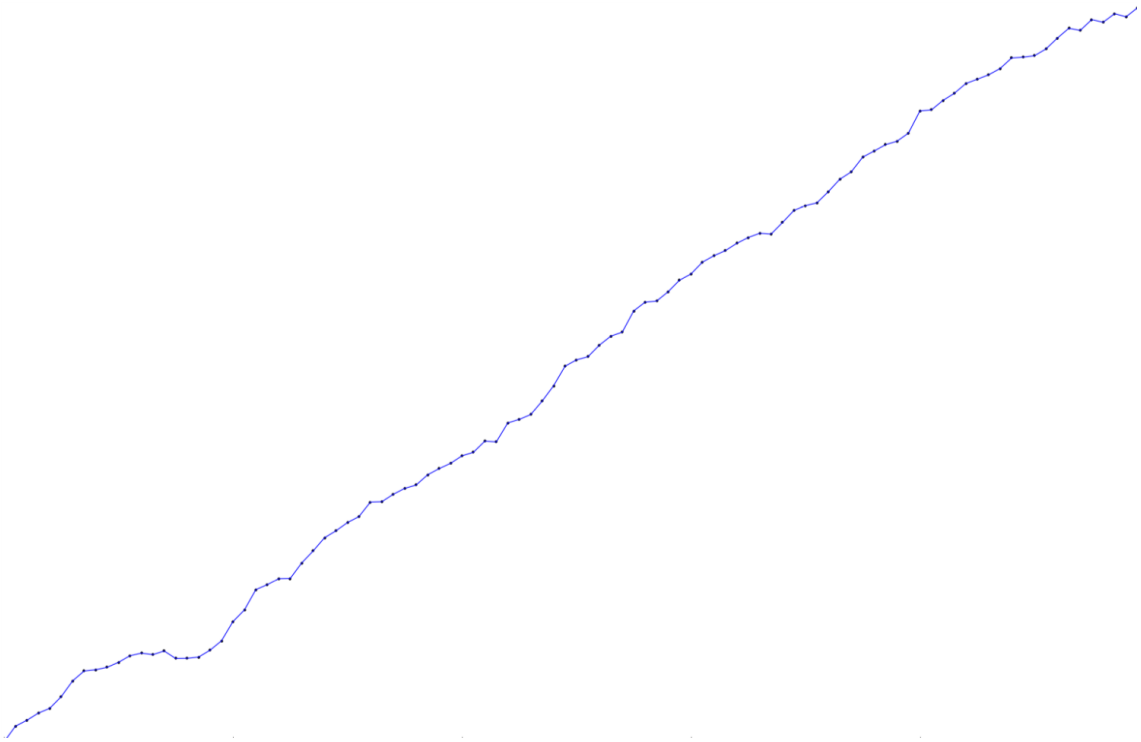

Figure 2: An example of a network overfitting to a data set. The fit may represent this particular set well, but it will not in general be able to give good predictions for values that it has never encountered which lie between the known ones. This occurs when a neural network has been trained for too long.

Of course, we have to be careful. Neural networks have been popularised in the me-

dia as the general solution to a very broad class problems and the answer to general artificial intelligence. Although they are highly adaptable systems, deep artificial neural networks are plagued with many problems and it is a veritable skill to employ them, to structure the information that the network needs to learn in a way that it *can* learn it. We noted before that these networks converge painfully slowly and this problem is exacerbated further when the networks become deeper and deeper. One has to be careful too that the networks are not over-trained, for if they are, we meet the problem of over-fitting.

These problems are only the beginning. Many different structures of neural network have been invented to overcome the various shortcomings of what we will see as the *feedforward* network. Such structures allow us to analyse different types of data. Recurrent Neural Networks, for example, often deal with time-series data, whereas Convolutional Neural Networks were constructed to efficiently process images.

Progress in the field of neural networks has, in a sense, slowed once more and there is no guarantee that it will start again. Of course there are now more machine learning papers being published than ever before, and, even if the progress for now is slow and incremental, there have been great strides made in the research in recent years.

0.1.2 How (Small) Neural Networks Work

The best place to start is with the concept of a perceptron. Perceptrons were the earliest model of an artificial neural network, indeed they are the representation of the 'neuron' in the neural network. The classical idea of a perceptron is of an object that takes a weighted sum of values as an input. If the input reaches a particular threshold, then the neuron will 'activate' or 'fire'. The neuron therefore has two choice, to fire or not fire, which lends itself well to a binary representation.
We say then, that the neuron is a function $f(x)$ that fires if the sum of the inputs B_j, which are each assigned a 'weight' a_j, a number that designates how 'important' that input is, is greater than some threshold γ. That is:

$$f\left(\sum_{i}^{n} a_i B_i\right) = f(\alpha) = \begin{cases} 1 & \alpha \geq \gamma \\ 0 & \alpha < \gamma. \end{cases} \tag{1}$$

A good example I have seen to explain this is essentially the decision a person would take whether or not to go out and buy an ice cream. Consider that we have three inputs, whether or not it is raining, how expensive the ice cream is and whether we have recently broken up with someone. Some factors will be more important than others. Imagine that it is not raining, and we have not recently broken up with someone, but we don't like spending money at all. Then the perceptron would assign a large weight to the cost and we would not buy the ice cream. Now consider that ice creams are expensive and it is still not raining, but we have recently been broken up with. Perhaps this break-up is the most important factor so we assign it extremely high importance, more than the cost of the ice cream, then when we sum up how important each of these is, it exceeds the threshold and we buy the ice cream.

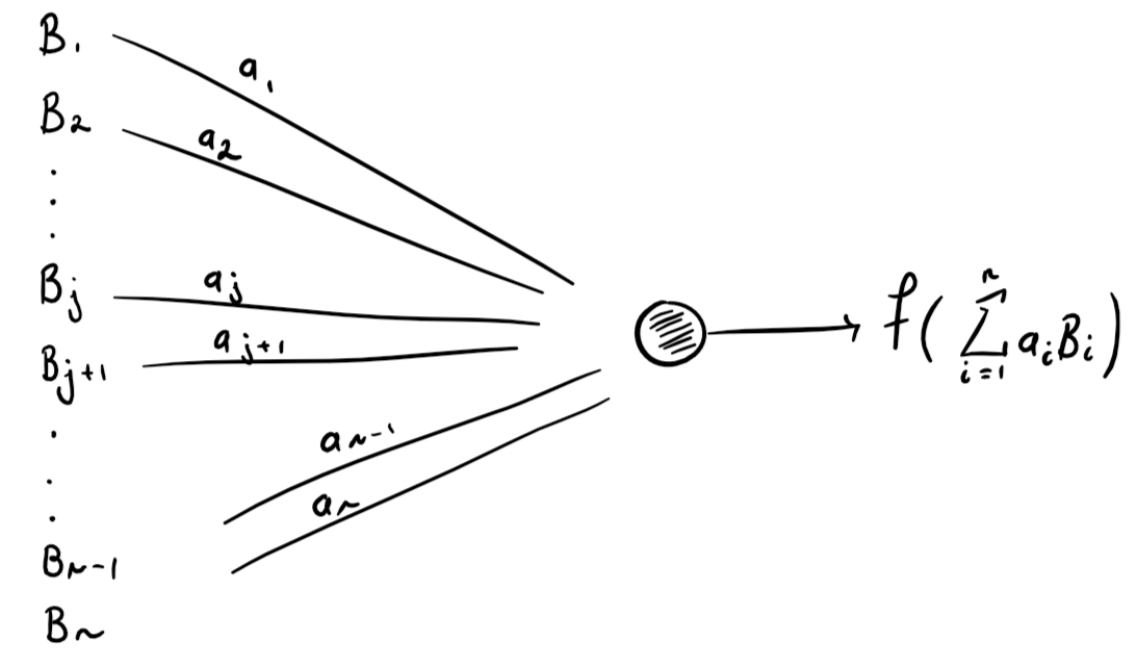

Figure 3: We can think of the perceptron as a machine that can make a binary choice depending on the information it is fed.

We can make factors very unimportant by giving them large negative weight values and we can assign them high importance by assigning them large, positive weight values. The perceptron as a decision function is a step function. Only if we exceed the threshold will it activate, otherwise it will always be zero.

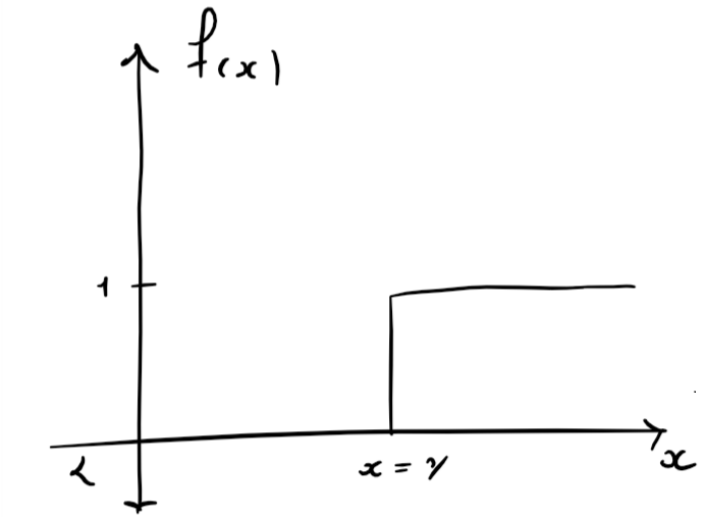

Figure 4: The traditional perceptron step activation function.

We can take this idea further, by using new 'activation functions'. The activation functions are important, since they make the network non-linear, which allows it to create non-linear approximations to functions that describe the input data. In most practical applications the data will be extremely non-linear and even if it is not, non-linear activation functions can describe linear data. One of the first alternatives to the step activation function that was proposed was the sigmoid activation function, which is given by the expression:

$$f(x) = \frac{1}{1 + e^{-x}}. \tag{2}$$

There are a variety of reasons that this function was chosen, but most of all because it resembles the step function, it has an easily computable derivative and it forces the output to lie in the range $[0, 1]$. This is extremely useful because now we are able to do more than just classify data into two categories. We can now use the perceptron to learn to reproduce any normalised output. Other choices exist, such as the hyperbolic tangent activation function and the now standard ReLU, or rectified linear unit, activation function. The latter is one of the innovations that was developed to combat the problem of vanishing gradient in deep neural networks that we mentioned earlier.

But now what do the outputs of the perceptron even mean? Well, it's up to us how we interpret them and it will depend on how the input data is structured, but let us think back to the ice cream example from before. The perceptron output could now represent the *probability* that given a set of conditions we go out and buy an ice cream. This is perfectly reasonable because probability values can only take values between 0 and 1. We must now choose weights to reflect these probabilities. Probability is fundamental to forecasting, so it is no surprise that we chose an activation function to cast our input space into the range $[0, 1]$.

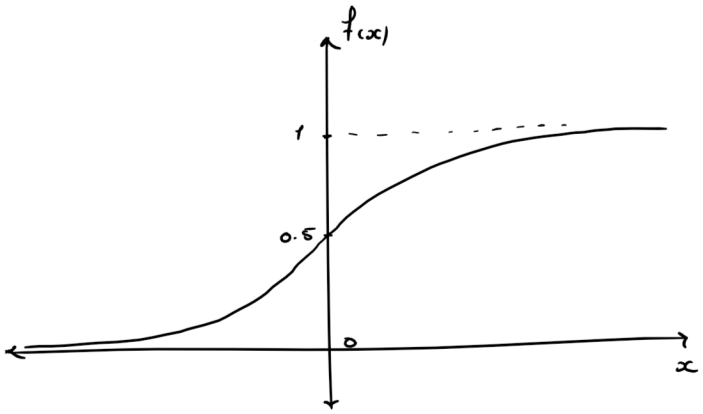

Figure 5: The sigmoid activation function.

Now if we think about it, we've reached a problem. Consider that somehow we know how likely it is that we'll buy an ice cream, given specific inputs, for all possible combinations of conditions in the input space, how do we know what the values of the weights should be? This is the where the *learning* comes into play. We must find a way to teach the perceptron to teach itself the correct weights. To do this, we start with random initial weights and teach the network to correct them until it correctly guesses the probability that we buy the ice cream.

0.1.3 Perceptron Learning

It is a little more difficult to develop the method in the case of the sigmoid activation function, as opposed to the Heaviside activation function, but it is perfectly manageable and will give us a better starting point for proper backpropagation. Consider then that we have a sigmoid perceptron with n inputs and m training vectors, that is m sets of outcomes for which we know the probabilities. Then if we initialise

the weights randomly using the Gaussian distribution $\mathcal{N}(0,1)$, the output of the j^{th} input vector is:

$$f\left(\sum_i^n w_i a_i^j\right) = \alpha^j \qquad \text{where } \beta^j \text{ represents the actual probability.} \tag{3}$$

We now use the standard notation w_i to refer to the i^{th} weight and a_i the i^{th} input value. In order to quantify how close we are to the correct answer, we need to define what is called a *cost function*. The most common choice is the mean square error function, for it has a convenient derivative:

$$E = \frac{1}{2}(\alpha^j - \beta^j)^2 \tag{4}$$

Now that we know how close we are to the correct solution, we need to establish how much a change in the value of the weight would change the value of the output. This will give us an idea of how to alter the weights and, importantly, whether to make them bigger or smaller. Another way of looking at this is that this allows us to see how important or unimportant certain inputs are to the result of the output. We now use the method of gradient descent. Using Taylor's theorem, we investigate how a small perturbation of the activation function affects the error. Up to first order:

$$E(w_k + \Delta w_k) = E(w_k) + \frac{dE(w_k)}{dw_k}\Delta w_k \tag{5}$$

Hence we aim to evaluate the derivative of the cost function with respect to the weight. To do this we will need to use the chain rule.

$$\frac{dE(w_k)}{dw_k} = \frac{dE(w_k)}{f(w_k)}\frac{df(w_k)}{dw_k} = \frac{dE(w_k)}{f(w_k)}\frac{df(w_k)}{dg(w_k)}\frac{dg(w_k)}{dw_k}, \quad g(w_k) = \sum_i^n w_i a_i^j \tag{6}$$

$$\frac{dE(w_k)}{dw_k} = \Big(f(w_k) - \beta\Big)\Big(f(w_k)(1 - f(w_k))\Big)\Big(a_k^j\Big) \tag{7}$$

where we obtain the term in the last bracket by performing the differentiation:

$$\frac{d}{dw_k}\sum_i^n w_i a_i^j = \sum_i^n \delta_{ki} a_i^j$$

Each of the three bracketed terms are easily computed. Indeed, we chose the sigmoid activation function because, as we see in the second bracket, its derivative is simply a function of itself. This afford easy implementation without the need to worry about numerical differentiation. We mentioned earlier that problems arise with over-fitting and indeed they can obviously arise from under-fitting. It is actually the case that for good accuracy we want the network to train relatively slowly, so we now introduce a training rate ϵ to slow down our improvement. This can help avoid both fitting problems. Finally then, we update the weights according to:

$$w_k \to w_k - \epsilon \frac{dE(w_k)}{dw_k} = w_k - \epsilon \delta_k a_k^j \tag{8}$$

Typically a good initial value for the training rate is 0.01 although varying it may be necessary to achieve good convergence.

Using these basic principles we'll develop the Backpropagation algorithm for updating deep neural network weights. Stated rather simply, we train a sigmoid perceptron as follows:

Algorithm 1 Sigmoid Perceptron Training Algorithm

1: Initialise weights according to $\mathcal{N}(0, 1)$.
2: Push data vector through the perceptron.
3: Compute δ_k.
4: Compute the new weight $w_k \to w_k - \epsilon \delta_k a_k$
5: Repeat with each input vector until required accuracy is achieved.

0.2 Deep Neural Networks

When we make decisions about things, we often make decisions based on decisions we have already concluded given some collection of information. This is where the power of neural computing becomes clear. We more often call the sigmoid perceptron a *neuron*. Each neuron can make a single decision when it is fed finitely many pieces of information. Consider the ice cream situation again, where now a separate neuron will use the same information to decide where we get the ice cream from. Perhaps if it is raining we will be more likely to go to the nearest shop if it is raining, but maybe if the person we have recently broken up with works there we would prefer to avoid it. Now we can make two decisions based on the information that we originally had. Now, some shops have a better selection of ice creams than others so let us now feed our two decisions into another neuron. This neuron might tell us how likely we are to be somewhat happy by the end of the day. If we go to the nearest shop we can get our favourite ice cream for sure so perhaps we will be very happy, especially if we have not recently broken up. Perhaps if it is raining very heavily, and we do not want an ice cream, then maybe we won't be very happy. We structure these decisions with the following network:

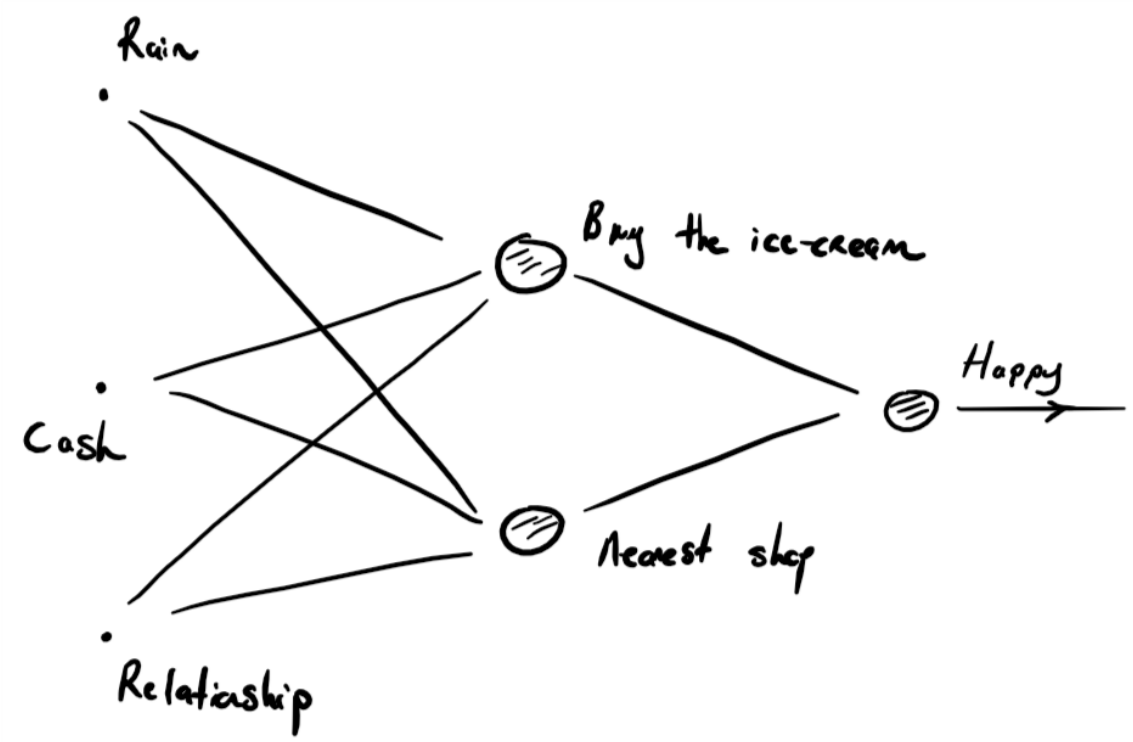

Figure 6: A neural network with three layers. We consider the inputs to be the outputs of unseen neurons, which the two neurons in the middle layer take as inputs. The last layer has a single neuron.

This neural network then has three layers. There are two layers of computing neurons, one with two neurons and one with a single neuron that takes as inputs the outputs of the previous two neurons. We consider the original three inputs to be neurons in some sense, but there are no weights that relate them to a previous layer, so we use dots instead to highlight that they are different. Now that we relate multiple neurons in one layer to multiple neurons in the next, we must use a matrix to contain the weights. A network can have any number of layers that we'd like, limited of course in practicality by the computing power that we have available.

Consider then that a deep neural network has m layers and the i^{th} layer of the network has n_i neurons. We store the weights between relate the outputs α^i of the i^{th} layer to the k^{th} input a^{i+1} of the $(i+1)^{th}$ layer with an $n_i \times n_{i+1}$ matrix of random weights W^{i+1}.

$$\sum_{j=1}^{n_i} W_{jk}^{i+1} \alpha_j^i = a_k^{i+1} \tag{9}$$

We can improve this relationship a little bit more by introducing something called the bias. The bias is a another weight that we introduce for each layer. It functions much like the constant term in a linear equation by allowing us to shift the value of the weighted sum up or down. This is extremely important for convergence to a good solution, so we must always remember to include the bias. There is one bias neuron for each layer. The output of the bias term is always one, so the input for each

neuron in the i^{th} layer has an extra constant b_i. We then update our relationship to:

$$\sum_{j=1}^{n_i} W_{jk}^{i+1} \alpha_j^i + b^i = a_k^{i+1}. \tag{10}$$

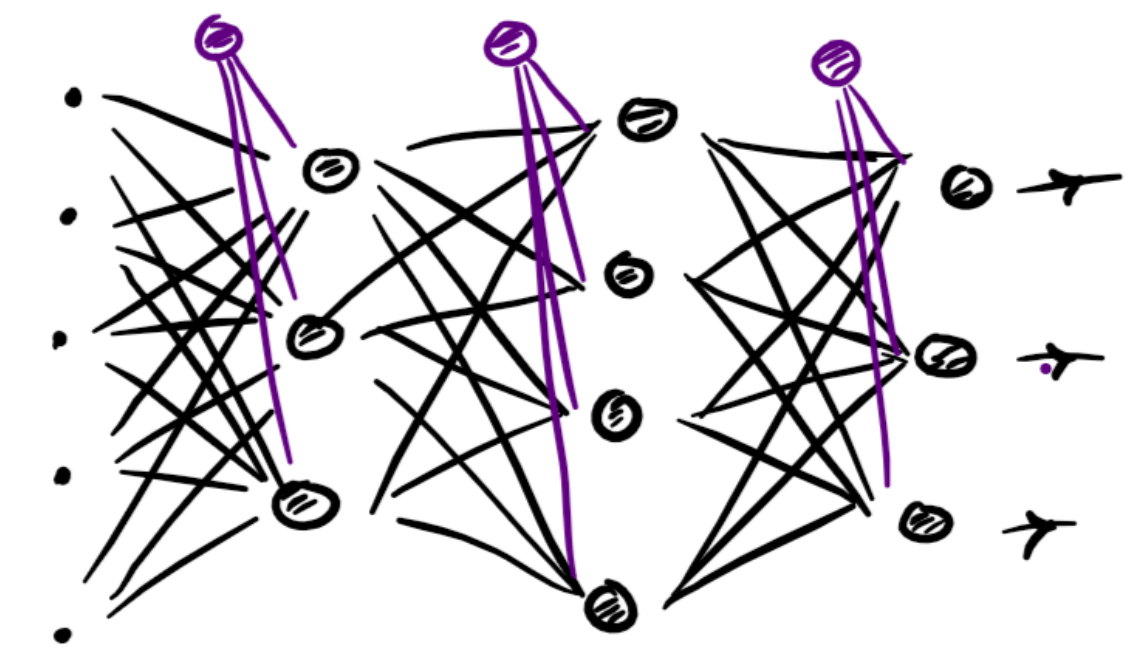

Figure 7: We can have many, many layers in a neural network. Each layer has a single bias neuron which contributes once to the input of each neuron in that layer. This is represented by the purple neuron in the image above.

We should now be careful to use the correct terminology. We call the first layer the input layer, the last layer the output layer and everything in between is called a "hidden layer". When we add neurons to a network, we have two choices, we can add neurons to a new layer or we can add them to an existing one. Consider that we have a three-layer neural network, so one hidden layer. We might want to improve the accuracy of the network, since adding more neurons allows us to identify more patterns. How do we choose where to put them? Well putting them in the one existing hidden layer will help us 'remember' more patterns but putting them in new hidden layers will help us to identify more abstract patterns, but the convergence will be much slower. We can decide to do one, or both of these procedures but we must be careful. If we only have three layers, adding many neurons to the one hidden layer encourages the network to only learn what it has been given and make it unable to produce good results when it encounters new inputs. Adding too many layers can be counter-productive, since highly abstract features may not really exist in the data. In such a case we add a lot of computing time for little gain.

In physics, one always tries to reduce the number of free parameters in a model. Indeed the Standard Model has been reduced to a mere nineteen parameters nec-

essary to measure experimentally. Neural networks are then anomalous in a sense, since they rely on maximising the number of free parameters. Many modern neural network models now employ massively deep and dense networks with millions of neurons and hence millions of free parameters, the weights, that must be tuned. It's easy to see how they can quickly become black boxes, the meaning of their operations difficult or even impossible to decipher. Training these networks is therefore very time consuming.

0.2.1 The Backpropagation Algorithm

It was mentioned earlier that interest in neural networks had died by the end of the eighties. This was largely the result of the inability to train networks with more than two layers. It was obvious how to train multiple Heaviside perceptrons when the input was connected directly to the output through one layer of functions, but no one knew how to correct the weights in the previous layers. The reason for this is obvious: we know what we want the output to be, since we want the network to reproduce some data that we know. It is a subtler problem however, to adjust the weights for a hidden layer because we don't actually know what the outputs of the hidden layer should be. When the backpropagation algorithm was invented, which allowed us to perform exactly that kind of training for any number of layers. Fortunately it is, in concept, extremely simple. All we need to know is how to compute the necessary weight changes for the output layer, and how to use the chain rule recursively.

Now, computing the necessary change for the outer layer is a little bit different this time, since we will no longer assume the presence of a sigmoid activation function or that the output layer has only one neuron. We will also account for other cost measures.

$$\frac{dE(W^m_{jk})}{dW^m_{jk}} = \frac{dE(W^m_{jk})}{df(W^m_{jk})}\frac{df(W^m_{jk})}{dg(W^m_{jk})}\frac{dg(W^m_{jk})}{dW^m_{jk}} \tag{11}$$

$$= \frac{dE(W^m_{jk})}{df(W^m_{jk})}\frac{df(W^m_{jk})}{dg(W^m_{jk})}\frac{d}{dW^m_{jk}}\left(\sum_{i=1}^{n_m} W^m_{ik}\alpha^{m-1}_i + b^{m-1}_i\right)$$

$$= \frac{dE(W^m_{jk})}{df(W^m_{jk})}\frac{df(W^m_{jk})}{dg(W^m_{jk})}\alpha^{m-1}_j = \delta^m_k \alpha^{m-1}_j$$

Now that we have the δ^m_k, which is a measure of error for the outer layer, we can calculate the $\delta^{m-1}_k, \delta^{m-2}_k, ..., \delta^1_k$ recursively in that order. Since we know the functional dependence of all of the outputs of one layer on all of the outputs on the previous layer, we need merely use the chain rule. Consider then that we know the error of the k^{th} neuron in the $(l+1)^{th}$ layer, δ^{l+1}_k. Then we relate to it the error of the i^{th} in the l^{th} layer, δ^l_i, using:

$$\delta^l_i = \frac{dE(W^{l+1}_{ik})}{df(W^{l+1}_{ik})}\frac{df(W^{l+1}_{ik})}{dg(W^l_{ji})} = \frac{dE(W^{l+1}_{ik})}{dg(W^l_{ji})} = \frac{dE(W^{l+1}_{ik})}{dg(W^{l+1}_{ik})}\frac{dg(W^{l+1}_{ik})}{dg(W^l_{ji})} \tag{12}$$

$$= \frac{dE(W_{ik}^{l+1})}{dg(W_{ik}^{l+1})}\left(\frac{d}{dg(W_{ji}^{l})}g(W_{ik}^{l+1}, f(g(W_{ji}^{l})))\right) = \frac{dE(W_{ik}^{l+1})}{dg(W_{ik}^{l+1})}\left(W_{jk}^{l+1}\frac{df(W_{ji}^{l})}{dW_{ji}^{l}}\right) \quad (13)$$

$$\implies \delta_i^l = \delta_k^{l+1}\left(W_{jk}^{l+1}\frac{df(W_{ji}^{l})}{dW_{ji}^{l}}\right) \quad (14)$$

Now that we can compute the error for any given layer, we may compute the necessary change of weight for the neurons in each layer to train the network. We state the backpropagation algorithm as follows:

Algorithm 2 Backpropagation Algorithm

1: Initialise weights according to $\mathcal{N}(0, 1)$.
2: Feed data through the network.
3: Compute the outer layer error δ_k..
4: Compute the hidden layer errors δ_i^l.
5: Compute the gradient of the error $\delta_i^{l+1}a_k^l$.
6: Determine the change in the value of the weights $W_{ki}^l \to W_{ki}^l - \epsilon\delta_i^{l+1}a_k^l$.
7: Determine the change in the value of the bias weight $b_j^l \to\to b_j^l - \epsilon\delta_j^l$.
8: Repeat until required accuracy, or maximum accuracy, is achieved.

The backpropagation algorithm is most useful for collections of independent data vectors and is most effective for shallower networks. By using stochastic annealing and dropout methods, among others, one can achieve better convergence. For time series we would achieve better results using what are called Recurrent Neural Networks and their particularisation, the Long Short-Term Memory network.