# A Brief Introduction to Neural Networks

Jack Adam Collins

This paper will discuss what neural networks are (part I), why they are an effective means of prediction (part II), and how they are trained (part III). The central aim of this paper is two-fold. Firstly, to reinforce what I have learned about neural nets via research and development within the machine learning team. Secondly, to share this knowledge more widely for the purposes of finding appropriate applications for this methodology to work within the *Quantum* faculty space. This introduction is written to serve as a very general presentation, specifically to a simple *feed-forward* example with the first two sections requiring only minimal mathematical notation. Accompanying this paper is a repository of jupyter notebooks with instructions on building a neural network 'from scratch' without the use of an external deep learning framework[1]. While it is unlikely that members of Quantum will need to build neural networks in this way, let this paper, and accompanying notebooks, act as a theoretical primer to the methodology itself. While neural networks are often considered 'black box', understanding the fundamentals of building simple neural networks without a pre-existing framework allows for a better understanding of the methodology and an intuition as to the scale required for a neural network to fulfil a given task adequately.

## I

Neural networks describe a class of machine learning predictors which take their influence from the biological architecture that constitutes the animal brain[2].

The neural network in figure 1 serves as the primary example throughout this paper, beginning by defining the specific components visualised in the diagram. The circles represent individual *neurons*, and the interconnecting arrows are *edges*. In combination, they define the *graph* underlying the neural network – a cluster of nodes with linking arrows between them. If the graph has no cycles, that is, edges looping around in a circle, then the neural net is considered *feed-forward*, which for the purposes of simplicity will be the focus of the neural networks presented. In figure 2, the green neuron is the *input neuron*, the yellow neuron is the *output neuron*, and the blue neurons are *bias neurons*.

If the possibility exists of dividing the entire set of neurons into an ordered list of subsets, such that each neuron *only* points towards neurons in the subset next in line,

[1]Access to the repo can be provided upon request: `https://stash.aviva.co.uk/projects/TPA/repos/mlt_laboratory/`

[2]Warren S. McCulloch; Walter Pitts. "A Logical Calculus of the Idea Immanent in Nervous Activity".

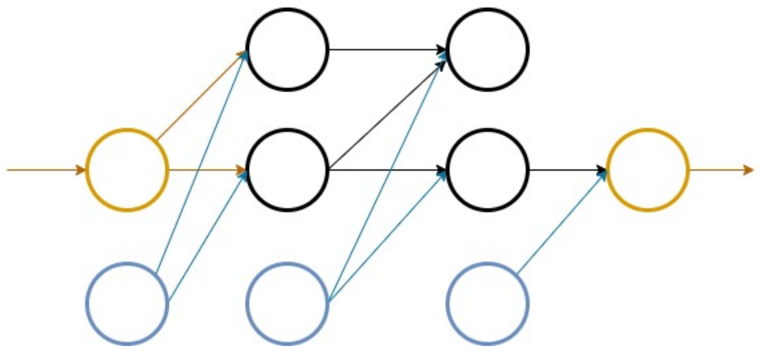

Figure 1: Schematic for a basic feed-forward neural network

then the network is considered *layered*. While the focus here is layered networks, non-layered networks do in fact exist. For example, a neural network with the given edges:

$$A \to B,$$

$$A \to C,$$

$$B \to C$$

cannot be divided into layers.

The graph underlying the neural network illustrates the way in which each neuron points only towards neurons within the next layer. This depiction is simplified but can be expanded upon by incorporating adequate notation, as illustrated in figure 3.

In the schematic, *weights* are given by $w_{i,j}$. The weights determine the relative importance of the value of the incoming edge. *Functions* are given by $\sigma_x$, in which functions are considered in the classical sense: a value is taken as input before being output as something else. If we are to discuss, say, the $3^{rd}$ neuron, then we shall herein use the notation $\sigma_3$. Referring specifically to figure 3, we can now walk through the particular operations of the network. An input value $x$ is given to the input neuron, $\sigma_1$, which forwards it to neurons three, $\sigma_3$, and four, $\sigma_4$. Synchronously, $\sigma_2$ (the first bias neuron) sends a constant value, 1, to neurons three and four (for simplicity, all our bias neurons will be outputing the value of 1). At this point, the third neuron has received two values – it now multiplies them with the weights on the respective edges; multiplying the $x$ input from the first neuron with $w_{1,3}$ and the number 1 coming from the bias neuron with $w_{2,3}$. The values are then added together and the function $\sigma_3$ is then applied to them. The result of this process is[3]:

$$\sigma_3(w_{1,3} \cdot x + w_{2,3} \cdot 1) \tag{1}$$

The $4^{th}$ neuron also receives two values, performing the same operation (using the weights $w_{1,4}$ and $w_{2,4}$), leading to the term:

$$\sigma_4(w_{1,4} \cdot x + w_{2,4} \cdot 1) \tag{2}$$

[3]Note that, typically multiplying by 1 would be simplified in a given equation, but to illustrate the role the bias plays in the calculations, the value remains shown in both equations (1) and (2).

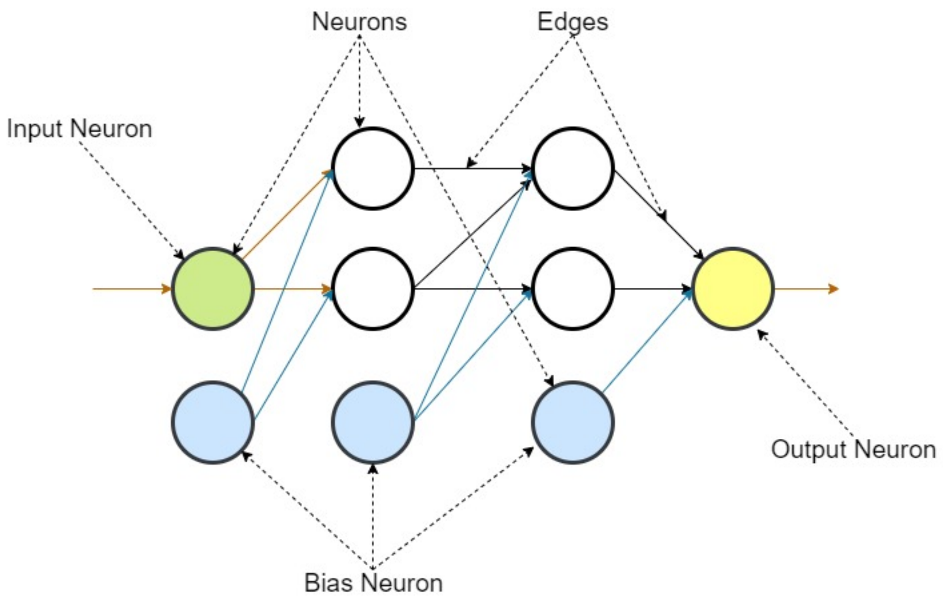

Figure 2: Labelled schematic for the feed-forward neural network

The 3rd neuron outputs its value to the 6th neuron, the 4th outputs its value to both the 6th and the 7th, and the 5th neuron (the bias neuron) outputs the number 1 to both the 6th and the 7th neuron. Their weights are then applied, then their functions, and so on. Eventually, the 9th neuron receives a value, applies $\sigma_9$ to it, and outputs the result, which is the resulting output of the entire network.

Recall that the blue neurons depicted in the schema are named bias neurons – having no incoming edges. Instead of computing something, they always output a given number and thus bias the weighted sum, which arrives at the neurons in the next layer, by a constant factor. Each hidden layer has exactly one bias neuron as does the input layer. In this way, each neuron, excluding the input neurons, has an incoming edge from a bias neuron. Also note that the input neuron doesn't apply any function to the input it receives (but all other neurons do).

To describe this process more concisely, additional notation is required in order to express the value incoming to a given neuron, in particular, the weighted sum of the values on the incoming edges, and the final value which comes out after the application of the function. In particular, we can examine the 6th neuron, $\sigma_6$, (shown in figure 4).

We can see that the input values are received, the weighted sum $a_6$ is computed, then $\sigma_6$ is applied to it, resulting in the output value $z_6$. Thus, we can generalise:

$$z_6 = \sigma_6(a_6) = \sigma_6(w_{3,6} \cdot z_3 + w_{4,6} \cdot z_4 + w_{5,6})$$

Hence, the output of a given neuron is expressed in terms of the outputs of the neurons in the previous layer; thus, the equation can give a description of the output of any neuron within the network. Note, the value $z_5$ is missing – since $\sigma_5$ is a bias neuron with the constant value 1, therefore $w_{4,5} \cdot z_5$ is equivalent to $w_{4,5}$. Furthermore, note that

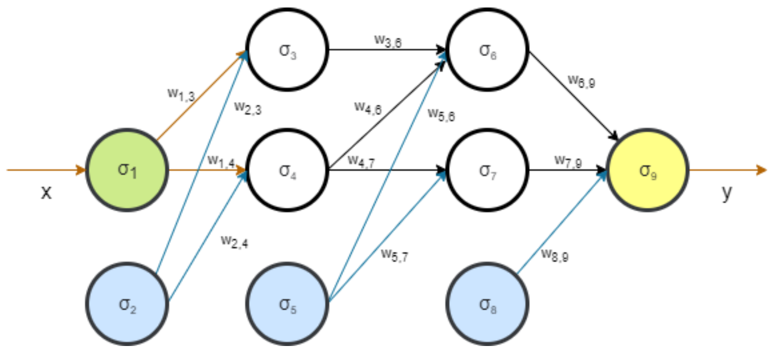

Figure 3: The feed-forward neural network with the layers depicted

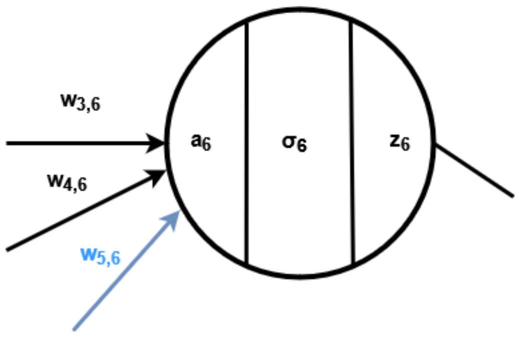

Figure 4: Closer inspection of the sixth neuron.

the image shows the weights of the incoming edges, but not the weight of the outgoing edge[4].

Recalling that our neural network can be divided into layers, a useful process for purposes of evaluating the network, we can analyse further by looking to figure 5.

To begin, the network is fed the input value, or values, $x_k$, setting:

$$z_k = x_k$$

for each neuron, $k$, in the input layer. Assuming all output values for some layer, $\ell$, are known, it is possible to then compute the output values for each neuron, $k$, in layer $\ell + 1$ by setting

$$z_k = \sigma_k \left( \sum_{i \in I_k} w_{i,k} z_i \right)$$

where $I_k$ is the set of indices of neurons with an edge pointing towards neuron $k$. As the equation applies for any layer, the entire network can be evaluated, a single layer

[4]The weights of an edge should be thought of as belonging to the neuron the edge acts as an input to.

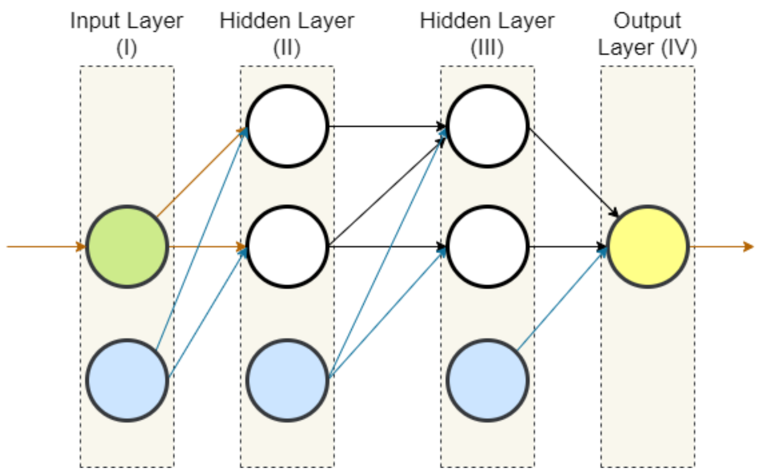

Figure 5: The feed-forward neural network with the respective layers depicted

at a time. The output values $z_k$ of the final layers will be the output values $y_k$ of the network.

## II

Now that we know what a neural network is and how to compute one, we can ask what are neural networks good for? Why do they work so well? We can start by addressing the first question.

If all values[5], are real numbers, then the total network is said to *implement a function*:

$$f : \mathbb{R}^n \to \mathbb{R}^m$$

where $n$ and $m$ represent the number of neurons in the input layer and output layer respectively, which in the case of the network presented is given by $n = m = 1$. As we can see, $f$ always takes an element in $\mathbb{R}^n$, a vector of $n$ numbers[6] and returns an element in $\mathbb{R}^m$, which is a vector of $m$ numbers[7]. This happens for any possible input, that is, for any $n$ input numbers, we see a return of $m$ output numbers. In this way, *implementing a function* allows for abstraction: the behaviour is fully described by $f : \mathbb{R}^n \to \mathbb{R}^m$. This is beneficial as functions are better understood objects than neural networks.

Evidently, the utility of the neural network is that it allows an evaluation of the function $f$, which can be especially of interest if $f$ does something useful. For example, suppose that $f$ takes the source of an image as input and then outputs a single bit indicating whether or not the image contains a cat. In this example, if we suppose the

[5]That is, the inputs, outputs, and the values sent between neurons.
[6]All the numbers fed to the input neurons.
[7]All the numbers returned by the output neurons.

image is gray-scale with 1 byte per pixel and the image has dimensions of 200 by 200 pixels, then the function is of the form:

$$f : \{0,1\}^{8\cdot 200\cdot 200} \to \{0,1\}$$

We see that the function takes a vector of $8 \cdot 200 \cdot 200$ bits as input (that is, the image), and outputs a single bit, 1 or 0, determining "cat" or "not cat" respectively. A similar example is in practice at the social media company, Twitter, where the function takes the form:

$$f : \{0,1\}^{8\cdot 280} \to \{0,...,10\}$$

The vector here encodes a 'tweet' (280-character limit, each encoded as a single byte with special characters ignored), and then outputs a value associated to the likelihood of the tweet in violating the platform's terms of service.

*How* a neural network is trained is expanded on in section III, but for simplicity we should note that this works based on training data. That is, we have some sequence of examples $(x_1, y_1), ..., (x_n, y_n)$ where $x_i$ are inputs to the neural network and $y_i$ are outputs. The network is then trained to do well on the training data, with the intention that this will lead it to also do well in the 'real world'[8]. However, this description is not specific to neural networks – it applies to all instances of supervised learning and is likely a familiar concept. So what is it about neutral networks in particular that make them so powerful?

Evidently, neural networks, or *deep learning*, works remarkably well. It's development has helped improve the cutting edge in areas ranging from speech recognition and visual object recognition to genomics and automatic game playing. Yet, it is still not fully understood why deep learning works so well. Many algorithms using artificial neural networks are understood only at a heuristic level, where we empirically know that certain training protocols employing large data sets will result in excellent performance.[9] One possible solution is to point towards the *expressibility theorems* on neural networks, most notably, the *universal approximation theorem*[10] – which states that a feed-forward neural network with only a single hidden layer – recall that, in the earlier example, we had two hidden layers – can approximate any continuous and bounded function under some given reasonable conditions. However, this proof largely consists of brute-forcing the approximation. In addition the neural networks constructed in the proof have exponentially many neurons in the hidden layer – conclusions asserting that results can be achieved with exponential resources are generally unsatisfying.

A simpler result which can be rigorously proved with less resource states that a neural network, utilising only a single hidden layer, can implement any boolean function of the form:

[8]Referring to the twitter example, that it labels new tweets accurately.

[9]Much like its biological analog, the situation with neural nets is reminiscent of the situation with human brains: we know that if we train a child according to a certain curriculum, she will learn certain skills — but we lack a deep understanding of *how exactly* her brain accomplishes this.

[10]Cybenko, G. (1989) "Approximations by superpositions of sigmoidal functions".

$$f : \{0,1\}\, n \to \{0,1\} \tag{3}$$

which we can think of as taking $n$ bits as input and outputting a binary value. This theorem states that there is an architecture, or graph, for a given neural network such that, for each possible function of the binary form given by (3), there is a set of weights such that the neural network defined by that particular architecture and that set of weights implements that function. Furthermore, every neuron within the network applies the same function, $\sigma$. However, in this theorem, the number of neurons in the hidden layer of this architecture is, again, forced to have an exponential number of neurons in $n$, therefore, conclusions are also unsatisfying in terms of explainability.

If the *universal approximation theorem* is unable to provide satisfying conclusions, then a better explanation might be found in a paper titled *Why does deep and cheap learning work so well?*[11], in which properties encountered in physics, such as symmetry and locality, can be used to expand upon the manner in which we attempt to understand how neural networks[12] approximate arbitrary functions. This approach begins by looking towards simple tasks, such as image-classification, and pointing out that these kinds of tasks are, in some sense, not possible to solve by way of a neural network. Referring back to the example given earlier, in which a 200 by 200 bit grey-scale image, with a single byte for each pixel defines an input space given by:

$$f : \{0,1\}^{8 \cdot 200 \cdot 200}$$

As we can see, this space consists of $2^{320000}$ various encodings for images, which is roughly $10^{10000}$. We need not be modest however; we know that neural networks are, in fact, used to classify much larger images. If, in fact, the images were 1000 by 1000 pixels with 4 bytes per pixel, then the space has $2^{32000000}$elements, which is around $10^{10000000}$. On the other hand, a neural network with, say, 1000 neurons only has $n^{1000}$ possible combinations, where $n$ is the number of different values possible for each weight. Even if we are very liberal and assume that each weight can lead to a 100 meaningfully different configurations, that would only provide us with the ability to differentiate between at most $10^{10000}$ different cases – not enough for very small gray-scale images, and certainly nowhere near enough for decent sized images. This follows because it is impossible for a neural network to differentiate more images than it has different configurations. Moreover, this point does nothing to address the amounts of training data that are available in practice, which are nowhere near as large.

The suggestion then, is that the problem space is generated by physical processes that are much more simple to define than the state space itself and, furthermore, even simpler than the neural network. To refer back to an earlier example, cats - or even images of them - are complicated objects mathematically, yet the process that came to generate cats (i.e, evolution) is a simple mechanism relatively. Likewise, the instruction

[11]Lin, H W., Tegmark, M, Rolnick, D. (2016) "Why does deep and cheap learning work so well?"

[12]A brief note on terminology: 'deep learning' refers to neural networks with many hidden layers rather than just one or two.

to 'draw a picture of a cat', which we might provide to a class of children, is simpler than the pictures that they would come to create. So, while it is the case that a neural network comprised of around one thousand neurons is only capable of providing reasonable responses to a miniscule subset of images within 'image-space', this is acceptable since most of the images we are interested in are contained within that subset[13]. Evidently, that deep learning has had great success in practice presents a clear opposition to the *universal approximation theorem.* So we might ask, why deep learning rather than any other methodology? Lin et al. assert that nature is, at a fundamental level, hierarchical:

"*One of the most striking features of the physical world is its hierarchical structure. Spatially, it is an object hierarchy: elementary particles form atoms which in turn form molecules, cells, organisms, planets, solar systems, galaxies, etc. Causally, complex structures are frequently created through a distinct sequence of simpler steps.*"[14]

Evidently, it is not simply that the small complexity of the generation process makes it feasible for neural networks to accomplish tasks such as facial recognition. Rather, the neural net may learn by reversing this generative process and, since this process occurs in multiple steps, the most effective neural networks will themselves perform their computations in multiple steps. We can show that multiple steps translates into multiple layers by observing that, even if a network possesses multiple input nodes, it is still considered to be modifying a single object, that is, a vector whereby $\mathbf{x} \in \mathbb{R}_n$, where $n$ represents the number of input nodes. Therefore the function, $f$, which computes the network can be given by:

$$f = \sigma_\ell \circ A_\ell \circ \cdots \circ \sigma_2 \circ A_2$$

where $\ell$ is the number of layers, $A_i$ are linear functions respectively, $\sigma_i$ are the functions implemented by the neurons, coordinate-wise. The input-vector $\mathbf{x}$ enters the input neurons and passes them unchanged, they are then scaled by the values of the weights between the first and second laters, which corresponds to a linear transformation, $A_2$. The neurons in the second layer, the first hidden layer, apply their respective functions, corresponding to the application component-wise of $\sigma_2$[15]. The outputs are then scaled by the weights between the second and third layers, also a linear transformation, $A_3$, and so on. Finally, the output layer applies the function $\sigma_\ell$ component-wise. So we can see, the network does apply a pair of transformations for each layer.

From this, an observation that supports Lin et al., can be found by implementing identity functions in the neurons. In this configuration, the network decomposes the transformations, given by:

$$f = A_l \circ \cdots \circ A_2$$

[13]Notably, given the set of constraints imposed on locality, symmetry and polynomial order, the number of parameters within the Hamiltonian of the standard model is limited to just *32*.

[14]Ibid.

[15]Assuming that all the neurons within the layer implement the same function.

and concatenating functions in this way leads to a linear function, that is, the network *only* applies a single matrix, $A_\ell \cdots A_2$, to the input-vector. Supposing that the elements of the input comprise a matrix, $N$, and objective of the network is to multiply the given input with another matrix, $M$. This can be achieved trivially using simply the input and output layers, without need of hidden layers, setting $A_\ell = A_2 = M$. This should be recognisable as the standard way of performing matrix multiplication, requiring a number of operations, $n^3$, but there are also more sophisticated algorithms with unusual time complexities, such as $O(n^{2.3425667})$ – the point here is that they're faster for very large matrices. Hence, as neural networks are universal approximators, there exists a network that implements matrix multiplication with similar time complexity. This is not something that can be replicated by way of a shallow network[16]. Lin et al. have refered to these results as *no-flattening theorems*, asserting that it is possible to prove, rigourously, that some specific neural networks are not able to be flattened without a loss in efficiency[17]. Is there an entirely satisying to the question "*why do they work?*" The answer is clearly not in the affirmative, and suggests that we are perhaps a long way from a satisfactory conclusion. However, this is an area of active research and, in fact, is a topic of interest in some of the research undertaken by Aviva in collaboration with Cambridge University[18].

## III

To understand *how* a neural network learns, we first need to specify which particular part of the network we want to learn. While there are many differing approaches, typically, the underlying graph[19] and the functions, $\sigma_k$, are treated as fixed while the task is then to find the set of weights, $w_k$, which perform the best. All of these specifications we refer to as the *architecture* of the network.

We start by assuming that we are operating in the mode of *supervised learning*, in which we have access to a sequence of training examples given by:

$$S = ((x_1, y_2, \cdots, (x_m, y_m))$$

Here, $x_i$ is an input to the network, while $y_i$ is the corresponding correct output. The network is then trained to predict well on $S$, the training data, with the intention that this will also perform effectively in real-world applications. How is this process performed? By selecting some finite encoding for real numbers and performing an exhaustive search over all possible combinations for the weight values, testing the network with each configuration before finally choosing the best. To illustrate this process, if we

[16]On a related note, proofs in linear algebra often argue by way of decomposing a matrix into smaller matrices.

[17]Additionally, we should add that there is a version of the universal approximation theorem which restricts the width of a network in favour of additional hidden layers.

[18]This work is being undertaken by Jonathan Crabbe whose supervisor, Mihaela Van Der Schaar, has written extensively on explainability.

[19]Recall that the graph specifies all neurons and edges that comprise the network.

specify a 64-bit encoding for the set of real numbers, then we will find that there are $2^{64\cdot 12}$ combinations of values, and therefore, this approach will require this number of rounds for testing on all $S$. Evidently, a more efficient approach is required and, as such, we will look toward *backpropagation* by way of stochastic gradient descent.

Supposing a network of the form given in figure 3, with some given preliminary weights so that the network is fully defined, we might then focus on some training point, $(x_i, y_i)$. If $x_i$ serves as the input fed into the network, we will then get some number as output, $f(x_i)$ This prediction will be good should $f(x_i)$ be in proximity of $y_i$ and will be considered poor otherwise, since $y_i$ is the known, correct output for $x_i$. Since we can compare the output of the network to the actual value, we can therefore see how 'wrong' it is. We can then use this to correct the weights in our network so that, next time, it is less wrong. Thus, the difference between the prediction and the correct value will be used to measure the quality of prediction by squaring the difference, $(y_i - f(x_i))^2$, to gain a measure for the performance of the network on this point. This difference is dependent on all the values of the weights, $w_{\bullet,\bullet}$, since all of these values contribute in order to compute $f(x_i)$. Suppose we were to pick out a weight arbitrarily, $w_{i,j}$ and treat all other weights as fixed, then consider the difference, $(y_i - f(x_i))^2$ , as a function of the weight in question. We call this the *loss function*:

$$\ell : \mathbb{R} \to \mathbb{R}$$

This loss function takes a value, $x$, for the weight, $w_{i,j}$ and outputs the value, $(y_i - f(x_i))^2$, while all other weights remain fixed. Computing this function we can then calculate the partial derivative, $\frac{\partial \ell}{\partial w_{i,j}}$. If this value is positive, then we can infer that $(y_i - f(x_i))^2$ will increase should the value of the weight, $w_{i,j}$, be increased. If the value is negative, then $(y_i - f(x_i))^2$ will decrease if the value of the weight, $w_{i,j}$, is increased. In both cases, we are able to determine exactly how to modify the weight $w_{i,j}$ in order to decrease the error. This can then be performed for each separate weight, resulting in an iteration through the neural network in which all the weights are improved. Then, starting with the new weights, this process is repeated for the next training point which will conclude by updating the weights, and the process is repeated for a third training point and so on. The output of the algorithm will be the final weights after updating over all the training points. The specific size of adjustment in changing each weight's value is determined by a parameter called the *step size.*

Recall that the network is evaluated layer by layer. Similiarly, we calculate the partial derivative with respect to each weight in a layer by layer process, however this is in reverse: hence, *backpropagation.* We can refer back to figure 3 and take as our example an edge directed into the final layer: $w_{6,9}$. Inspecting the $9^{th}$ neuron more closely, we can see in figure 6 that there are 3 elements required for differentiation; the input or weighted sum of the incoming edges ($a_6$), the function of the neuron ($\sigma_6$), and the output after the application of the function ($z_6$).

As we are interested in the term $\frac{\partial \ell}{\partial w_{6,9}}$, via the chain rule, we can write:

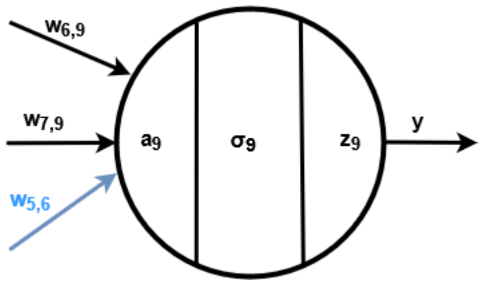

Figure 6: Closer inspection neuron 9.

$$\frac{\partial \ell}{\partial w_{6,9}} = \frac{\partial \ell}{\partial z_9} \cdot \frac{\partial z_9}{\partial a_9} \cdot \frac{\partial a_9}{\partial w_{6,9}}$$

where the first term on the right-hand side is:

$$\frac{\partial \ell}{\partial z_9} = \frac{\partial (y - z_9)^2}{\partial z_9} = 2z_9$$

and the second term on the right-hand side is the output of the 9th neuron with regard to its input; dependent on the function $\sigma_9$. Lastly, the final term on the right-hand side is simple; we have:

$$\frac{\partial a_9}{\partial w_{6,9}} = \frac{\partial [w_{6,9} z_6 + w_{7,9} z_7 + w_{8,9}]}{\partial w_{6,9}} = z_6$$[20]

Though the value $\frac{\partial \ell}{\partial w_{6,9}}$ is of central interest, we instead drop the third term in the chain rule above and instead focus on the derivative with regard to the input value of the neuron in the final layer, rather than with regard to the weights associated with the incoming edges. Thus, we can then process the *whole network* by way of the following steps:

1. Compute the derivative with regards to the input neurons of the final layer.
2. For each layer, $k$, compute the derivative with regards to the inputs of the neurons of layer $k$ in terms of the derivatives with regards to the inputs of the neurons of layer $k+1$.
3. For each weight, $w_{i,j}$, compute $\frac{\partial \ell}{\partial w_{i,j}} = \frac{\partial \ell}{\partial a_j} \cdot \frac{\partial a_j}{\partial w_{i,j}} = \frac{\partial \ell}{\partial a_j} \cdot z_i$

1 was addressed earlier on, while 3 is fully described by the given equation. In order to demonstrate 2, we suppose that layers three and four have been processed with the second layer remaining for processing. Given that we know the derivatives for $a_6$, $a_7$, and $a_8$ from above, we can use these computations in order to calculate the derivative with regards to an input neuron within the second layer. As an example, selecting $a_4$ and, again, inspect the relevant part of the network more closely in figure 7.

[20]These are all known parameters – $z_\bullet$ is obtained by evaluating the neural network on the input $x_i$.

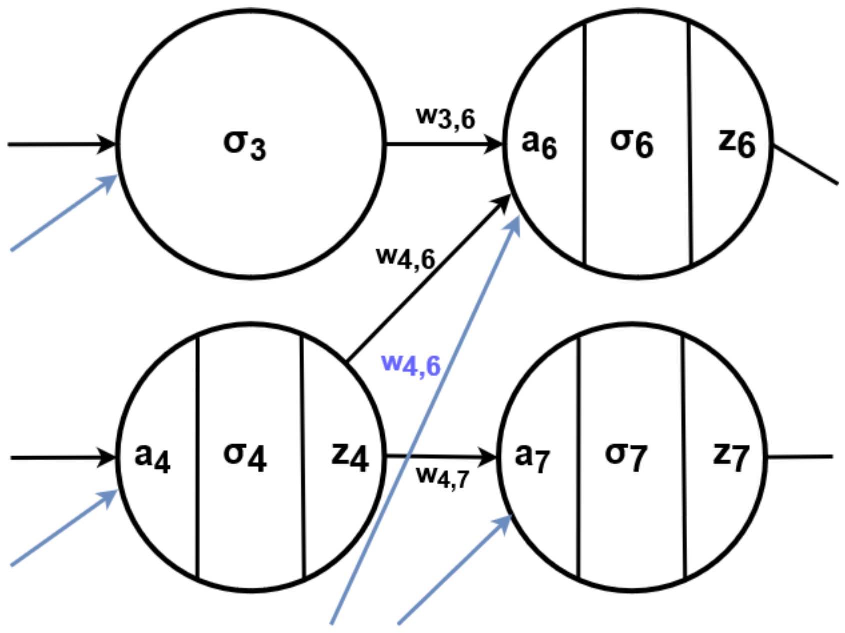

Figure 7: A closer inspection of some of the neurons within the network.

Thus, we have:

$$\frac{\partial \ell}{\partial a_4} = \frac{\partial \ell}{\partial z_4} \cdot \frac{\partial z_4}{\partial a_4}$$

Notice that the second term on the right-hand side is dependent on $\sigma_4$, but what of the first? Here we can try to understand a final complication. The manner in which $z_4$ impacts the error cannot be reduced to any single derivate in the third layer since the $4^{th}$ neuron feeds in to both the $6^{th}$ and $7^{th}$ neurons. Therefore, we need apply the advanced chain rule:

$$\frac{\partial \ell}{\partial z_4} = \frac{\partial \ell}{\partial a_6} \cdot \frac{\partial a_6}{\partial z_4} + \frac{\partial \ell}{\partial a_7} \cdot \frac{\partial a_7}{\partial z_4}$$

As we can see, both $\frac{\partial \ell}{\partial a_6}$ and $\frac{\partial \ell}{\partial a_7}$ are terms that have been computed in previous rounds, while the remaining terms are easy: $\frac{\partial a_6}{\partial z_4} = w_{4,6}$ and $\frac{\partial a_7}{\partial z_4} = w_{4,7}$.

At this point, we might object that the learning process will get stuck in a local minimum since the loss-function is non-convex. The practical solution is to simply run the entire backpropagation algorithm a number of times starting from different initial weights. The aim here then is that, if each run finds some local optimum, we then take the best of these since the result will likely be very good. Repeating the entire algorithm 1000 times may seem inefficient, but recall earlier that the alternative brute-force approach would require upwards of $2^n$ steps, where $n$ is the number of neurons. Even for large neural networks, running backpropagation in the number of a million times would be vastly more efficient than utilising a brute force search.

It is worth emphasising that neural networks are popular due to the fact that they perform well in practice, not because of theoretical results. Given their importance, neural networks continue to be a worthy area of pursued research.