A Simple 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 post 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 assisting in the application for this methodology to work within Quantum. 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. While it is unlikely that members of Quantum will need to build neural networks 'from scratch' in this way, let this work act as a theoretical primer to the method itself. Integrating neural net frameworks looks to be a central part of DataRobot's new custom code and compose ML features, and moreover will be essential in improving models and remaining at the cutting edge of machine learning. Often neural networks are considered a 'black box' or learning, but understand 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 appropriately.

I

Neural networks describe a class of machine learning predictors which take their influence from the biological architecture that constitutes the animal brain[1].

The neural network in figure 1 serves as primary example throughout, 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]Warren S. McCulloch; Walter Pitts. "A Logical Calculus of the Idea Immanent in Nervous Activity"

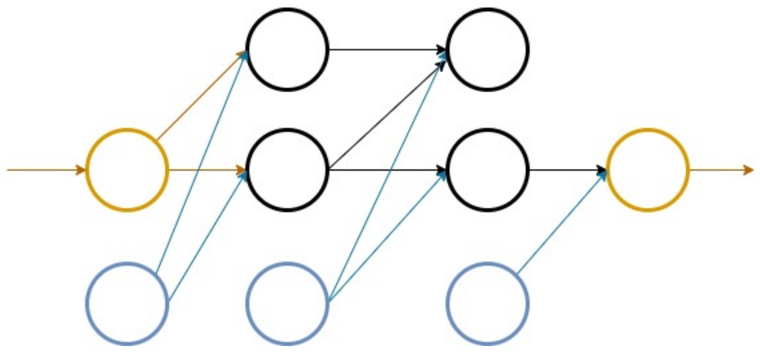

Figure 1: Schematic for a basic feed-forward neural network

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:

$$A \to B, \tag{1}$$

$$A \to C, \tag{2}$$

$$B \to C \tag{3}$$

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 (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 σ_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 σ_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, σ_1, which forwards it to neurons three, σ_3, and four, σ_4. Synchronously, σ_1 (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 σ_3 is then applied to them. The result of this process is:

$$\sigma_3(w_{1,3} \cdot x + w_{2,3} \cdot 1) \tag{4}$$

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

$$\sigma_4(w_{1,4} \cdot x + w_{2,4} \cdot 1) \tag{5}$$

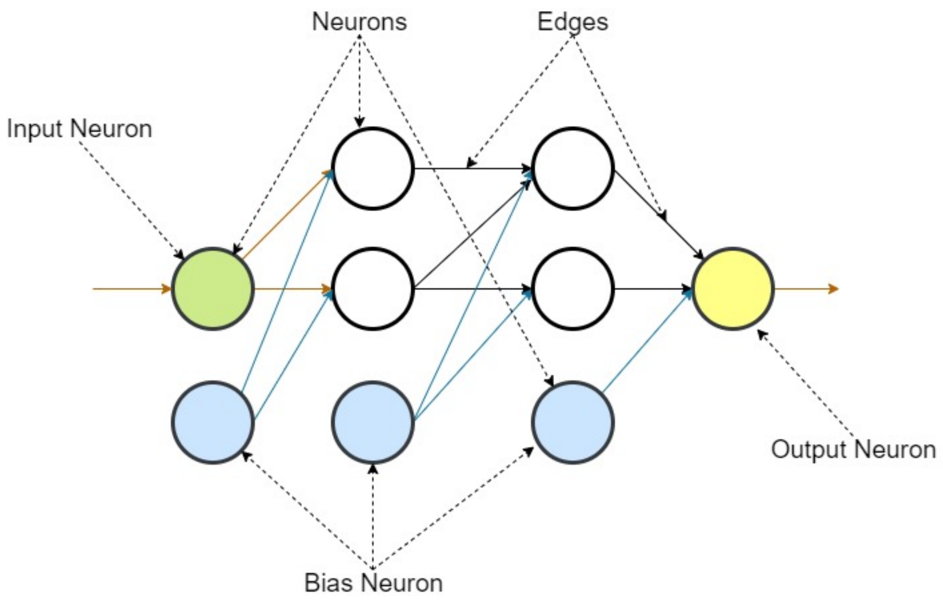

Figure 2: Labelled schematic for the feed-forward neural network

The third neuron outputs its value to the sixth neuron, the fourth outputs its value to both the sixth and the seventh, and the fifth neuron (the bias neuron) outputs the number 1 to both the sixth and the seventh neuron. Their weights are then applied, then their functions, and so on. Eventually, the ninth neuron receives a value, applies σ_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 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. Examining the sixth neuron, σ_6 in particular:

IMAGE NEURON 6 GOES HERE

Evidently, the input values are received, the weighted sum a_6 is computed, then σ_6 is applied to it, resulting in the output value z_6. Thus, we can generalise:

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

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 σ_6 is a bias neuron

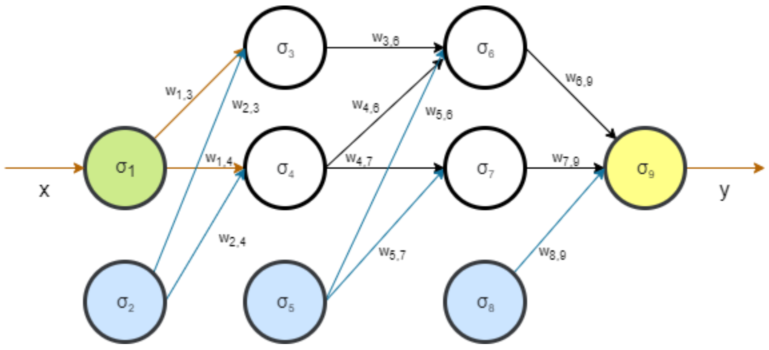

Figure 3: The feed-forward neural network with the layers depicted

with the constant value 1, therefore $w_{4,5} \cdot z_5$ is equivalent to $w_{4,5}$. Furthermore, note that the image shows the weights of the incoming edges, but not the weight of the outgoing edge: the weights of an edge should always be thought of as belonging to the neuron the edge acts as an input to.

Recalling that our neural network can be divided into layers, a useful process for purposes of evaluating the network, we can analyse further:

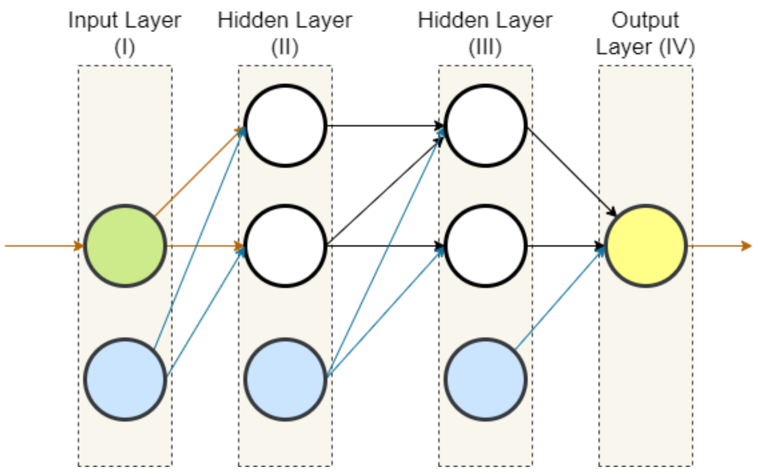

Figure 4: The feed-forward neural network with the respective layers depicted

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

$$z_k = x_k \tag{7}$$

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

$$z_k = \sigma_k \left(\sum_{i \in I_k} w_{i,k} z_i \right) \tag{8}$$

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

II

What are neural networks good for? Why do they work so well? We can start by addressing the first question.

If all values[2], are real numbers, then the total network is said to *implement a function*:

$$f : \mathbb{R}^n \to \mathbb{R}^m \tag{9}$$

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[3] and returns an element in $\mathbb{R}^m$, which is a vector of m numbers[4]. 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 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:

$$f : \{0, 1\}^{8 \cdot 200 \cdot 200} \to \{0, 1\} \tag{10}$$

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

$$f : \{0, 1\}^{8 \cdot 280} \to \{0, ..., 10\} \tag{11}$$

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.

[2]That is, the inputs, outputs, and the values sent between neurons.
[3]All the numbers fed to the input neurons
[4]All the numbers returned by the output neurons

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' (referring to the twitter example, that it labels new tweets accurately). 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.[5] One possible solution is to point towards the *expressibility theorems* on neural networks, most notably, the *universal approximation theorem*[6] – 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, the 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 – results asserting that results can be achieved with exponential resources are generally unsatisfying.

[5]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 her brain accomplishes this.

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