Mathematics of machine learning:

How does mathematics enable neural networks to “learn” to solve problems with real-world applications?

Mathematics

Word count: 3744

Contents

1 Introduction

In this essay, I will explore the structure of the backbone of machine learning by examining neural networks through a mathematical lens. As a field that is on the precipice between math and computer science, depending completely on the interdisciplinary knowledge and approach, it is personally very exciting as they have been my passions for a long time. I also believe it is worthy of discussion, both through formal mathematics but also what it intuitively means and further implies, as it is the one area which very obviously highlights the raw power that mathematics holds in solving real world problems when used in an intelligent way, and in combination with today's technology. A machine able to recognize handwriting, speech, or pictures of cats, was thought to be impossible not too long ago, but this relatively young field has made all of that, and more, possible in a way that people do not seem to fully comprehend just yet.

My hope is that the argumentation and discussion in this essay will be able to communicate the main ideas behind the topic of neural networks in a clear and coherent way; through examining their structure, function, and way of "learning", as well as what that even means for a machine. What follows is a conclusion reflecting on the surprisingly intuitive mathematical concepts behind neural networks, which represent the solution to many practical problems. In all, this would answer the question of what mathematics allow neural networks to learn and solve real-world problems.

2 Neural network architecture

In machine learning, artificial neural networks (ANNs) are computational learning systems comprised of “neurons”, objects that hold a numerical value, that comprise a network of functions to understand and translate a data input into a desired output (Neural). It is important to note that these systems differ from networks of neurons in, for example, the human brain, a “natural” neural network, because they are analog — holding any value between 0 and 1 — while the brain can only have a neuron fire (1) or not fire (0), making the signals it operates on effectively digital. The artificial neurons are then ordered into layers, and any neuron from one layer will be connected to every other neuron in a previous layer by a “synapse”, a number representing the relative weight of the connection between two neurons (Synapse). A system like this might consist of an input layer, one or more hidden layers, and an output layer:

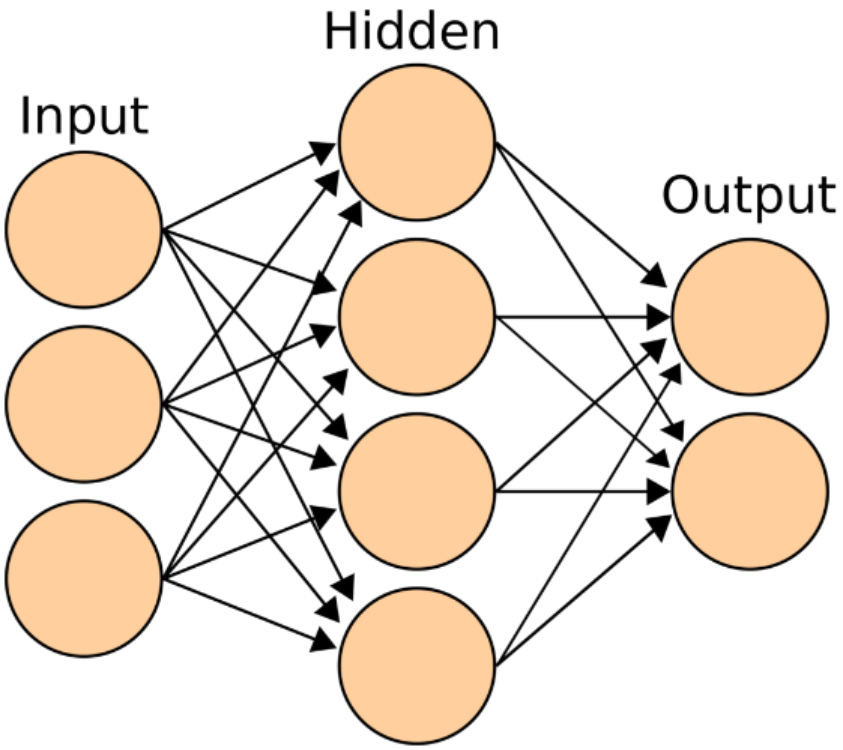

Figure 1: A basic neural network setup (Daityari)

Figure 1 shows a relatively basic ANN, as it has only one hidden layer. It still, however, has more than an input and output layer, and can thus be identified as a multilayer perceptron (MLP). Since a perceptron is “an algorithm used for supervised learning of binary classifiers” (Perceptron), an MLP is such an algorithm that trains binary classifiers (the neurons in the output layer; motivated to have as close to a discrete, binary value as possible, and each representing a class that the data it is being trained on can fall into) through multiple layers.

3 Layer to layer transition and neuron activation

Besides the neurons in the input layer, whose value is assigned by the sample we are attempting to classify, the process of neuron activation between the layers begins with a dot product between a $1 \times n$ parameter matrix (containing the weights on the synapses connecting the neuron getting activated to every neuron from the previous layer) and an n-dimensional neuron value vector (presented as an $n \times 1$ matrix, making the dot product defined, and containing the values that each neuron from the previous layer holds). In practice, this equates to taking the weighted sum of the values of all the neurons in the previous layers as they pertain to one neuron in the next layer through the aforementioned parameters, however, the rationale behind writing the activation in matrix form is that it makes, what would usually be long and tedious equations especially in large networks, very easy to condense and transform into efficient code, since many libraries in programming languages optimize matrix operations to a great extent (Sanderson). While this already makes the process of neuron activation a function in and of itself, the result is further processed by evaluating it in a formal “activation function”, such as the well-known logistic sigmoid function, adding non-linearity to the process. The equation for the logistic sigmoid function is (Wood):

$$S(x) = \frac{1}{1 + e^{-x}} = \frac{e^{x}}{e^{x} + 1}$$

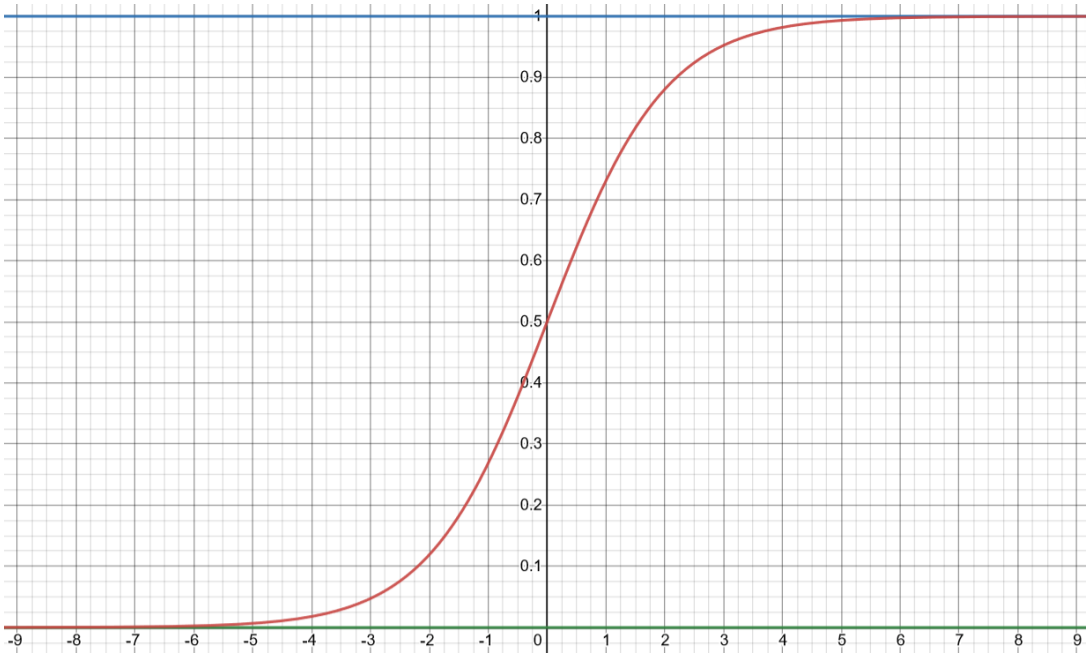

Figure 2: Corresponding graph of logistic sigmoid function clearly demonstrating its non-linearity and horizontal asymptotes at y=1 and y=0; created in Desmos (Desmos)

This non-linearity is essential in adding a layer of complexity that avoids the entire network computing what "could be re-factored to a simple linear operation or matrix transformation on its inputs" (Wood), which is exactly what the dot product from the previous step is doing. This function is also convenient as it bounds the output between 0 and 1, which can be proven by evaluating the limits as the input approaches ±∞:

$$\lim_{x\to+\infty} \frac{e^x}{e^x+1} = \lim_{x\to+\infty} \left(\frac{e^x}{e^x+1}\right) \div \frac{e^x}{e^x}, \qquad e^x \neq 0$$

$$= \lim_{x\to+\infty} \frac{1}{1+e^{-x}}, \qquad and\ since \lim_{x\to+\infty} 1+e^{-x} = 1\ then$$

$$\frac{\lim_{x\to+\infty} 1}{\lim_{x\to+\infty} 1+e^{-x}} = \frac{1}{1} = 1$$

and

$$\lim_{x\to-\infty} \frac{e^x}{e^x+1}, \qquad since \lim_{x\to-\infty} e^x+1 = 1\ then$$

$$\frac{\lim_{x\to-\infty} e^x}{\lim_{x\to-\infty} e^x + 1} = \frac{0}{1} = 0$$

The limit of the quotient is equivalent to the quotient of the limits if and only if the denominator is not equal to 0 as the limit is approached (Guichard 42) — quotient rule of limits.

Most of the “activating”, output of ~0 changing to ~1, happens around $x = 0$, and knowing this, it allows for a greater control over individual neuron activations, as we can introduce a bias for each of them before computing in an activation function. Suppose, for example, that it favors an application of our network if a particular neuron activates meaningfully, that is above 0.5, only if the aforementioned dot product is greater than 5, we would then define the bias to be -5, and would add it to the result before applying the activation function. This would make it so that neurons which would initially have a meaningful activation, now do not:

$$S(1) \approx 0.7311, \text{ but } S(1 + (-5)) \approx 0.0180$$

In extreme cases, however, when the dot product value is very large, or very small, the bias has a negligible effect on the final activation, conserving the impact of extreme results, and further demonstrating the usefulness of non-linearity in activation functions:

$$S(-5) \approx 0.0067 \text{ and } S(-5 + (-5)) \approx 0 \text{ or } S(10) \approx 1 \text{ and } S(10 + (-5)) \approx 0.9933$$

The sigmoid is not the only activation function widely used, and there are simpler ones such as the ReLU, or rectified linear unit, in the form of a linear piecewise function (Wood):

$$f(x) = \begin{cases} 0, & x \leq 0 \\ x, & x > 0 \end{cases} \text{ or } f(x) = \max(0, x)$$

The ReLU and the family of other rectified linear unit functions give the bias a slightly different role in the whole network. Namely, since there is a strict cutoff for when the

function has a value, at 0, any value that is below it will immediately result in a 0, making the bias a sort of activation threshold:

$$f(3) = 3, \text{ but } f\big(3 + (-5)\big) = 0 \text{ or } f\big(4.99 + (-5)\big) = 0$$

$$\text{and even } f(-4) = 0 \text{ but } f(-4 + 5) = 1$$

Their simplicity is also the main reason one might want to use a ReLU instead of a sigmoid function and is seen when we imagine the cumulative computational cost associated with manipulating these functions in any way many times. This is especially the case in taking their derivatives, which are clearly much easier to compute for a ReLU, and are an important aspect to consider when training a neural network which is a process that takes computing thousands of derivatives that go through the activation functions.

So, if we label any neuron as n_m^l where l is the number of the layer, starting with layer 1, the input layer, and m is the number of the neuron from top to bottom, any corresponding weight as $w_{m,p}$ where p is the number of the neuron from the previous layer it is connected to, and any bias associated with a specific neuron's activation as b_m^l, we can, in general, write the neuron's activation function, $a(n_m^l)$, as:

$$a(n_m^l) = S\left(\begin{bmatrix} w_{m,1} & w_{m,2} & \cdots & w_{m,p} \end{bmatrix} \cdot \begin{bmatrix} a\big(n_1^{l-1}\big) \\ a\big(n_2^{l-1}\big) \\ \vdots \\ a\big(n_p^{l-1}\big) \end{bmatrix} + b_m^l \right)$$

$$= S\big(w_{m,1}a\big(n_1^{l-1}\big) + w_{m,2}a\big(n_2^{l-1}\big) \ldots + w_{m,p}a\big(n_p^{l-1}\big) + b_m^l\big)$$

Since the result obtained from the dot product is a 1×1 matrix, or rather a scalar (Nykamp, Dot product), the bias will also be a scalar. It is useful to think of the individual elements of the network in this way because we can extend the parameter matrix from being $1 \times p$ to being $m \times p$ and thus taking into account the weights between all of the neurons from the previous layer to all the neurons in the next

layer, ordered by rows correspondingly, and the bias from being 1×1 to being $m \times 1$ and thus taking into account the bias on each neuron of the next layer. This then results in a matrix-vector multiplication of the extended parameter matrix and the original neuron value vector giving finally an $m \times 1$ matrix, by definition (Nykamp, Multiplying matrices), that can be added to the bias matrix, and to that result applied the activation function. This process incredibly results in a $m \times 1$ matrix which contains the activation values for neurons of the entire layer, regardless of its size, all at once; in general, L_l:

$$L_l = S\left(\begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,p} \\ w_{2,1} & w_{2,2} & \dots & w_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \cdots & w_{m,p} \end{bmatrix}\begin{bmatrix} a(n_1^{l-1}) \\ a(n_2^{l-1}) \\ \vdots \\ a(n_p^{l-1}) \end{bmatrix} + \begin{bmatrix} b_1^l \\ b_2^l \\ \vdots \\ b_m^l \end{bmatrix}\right)$$

$$= S\left(\begin{bmatrix} w_{1,1}a(n_1^{l-1}) + w_{1,2}a(n_2^{l-1}) \dots + w_{1,p}a(n_p^{l-1}) \\ w_{2,1}a(n_1^{l-1}) + w_{2,2}a(n_2^{l-1}) \dots + w_{2,p}a(n_p^{l-1}) \\ \vdots \\ w_{m,1}a(n_1^{l-1}) + w_{m,2}a(n_2^{l-1}) \dots + w_{m,p}a(n_p^{l-1}) \end{bmatrix} + \begin{bmatrix} b_1^l \\ b_2^l \\ \vdots \\ b_m^l \end{bmatrix}\right)$$

$$= S\left(\begin{bmatrix} w_{1,1}a(n_1^{l-1}) + w_{1,2}a(n_2^{l-1}) \dots + w_{1,p}a(n_p^{l-1}) + b_1^l \\ w_{2,1}a(n_1^{l-1}) + w_{2,2}a(n_2^{l-1}) \dots + w_{2,p}a(n_p^{l-1}) + b_2^l \\ \vdots \\ w_{m,1}a(n_1^{l-1}) + w_{m,2}a(n_2^{l-1}) \dots + w_{m,p}a(n_p^{l-1}) + b_m^l \end{bmatrix}\right)$$

$$= \begin{bmatrix} S(w_{1,1}a(n_1^{l-1}) + w_{1,2}a(n_2^{l-1}) \dots + w_{1,p}a(n_p^{l-1}) + b_1^l) \\ S(w_{2,1}a(n_1^{l-1}) + w_{2,2}a(n_2^{l-1}) \dots + w_{2,p}a(n_p^{l-1}) + b_2^l) \\ \vdots \\ S(w_{m,1}a(n_1^{l-1}) + w_{m,2}a(n_2^{l-1}) \dots + w_{m,p}a(n_p^{l-1}) + b_m^l) \end{bmatrix} = \begin{bmatrix} a(n_1^l) \\ a(n_2^l) \\ \vdots \\ a(n_m^l) \end{bmatrix}$$

As previously mentioned, this entire process and all the matrix operations in it can be written in a condensed form, and if we label the parameter matrix as $W_{l,l-1}$ and the bias matrix as B_l, the first line of the preceding calculation becomes simply:

$$L_l = S(W_{l,l-1}L_{l-1} + B_l)$$

To make this process clearer, we can take the network from Figure 1 as an example:

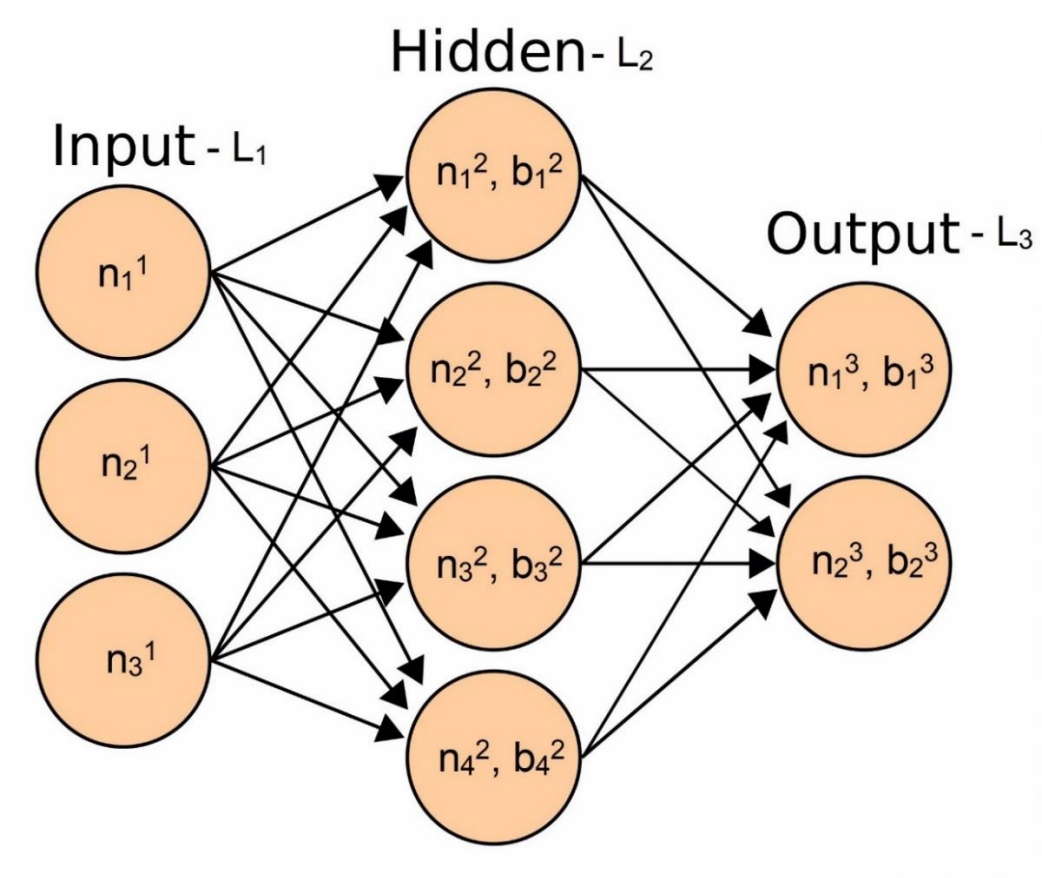

Figure 3: Annotated network from Figure 1; showing layers, and neurons and their bias

Annotating weights on Figure 3 was omitted due to how many of them there are even in a very small network such as the one pictured. However, knowing where each one is positionally is not really the point of neural networks, it is enough for us to know that each neuron from a layer is connected to each one from the previous layer by one of them, but it exemplifies the magnitude of these structures and justifies what they are able to achieve. For example, just to recognize handwritten digits, one of the examples mentioned in the introduction, on a $28x28$ pixel grid, a network needs to have $28 \times 28 = 784$ input neurons, one for each pixel to show whether there is something written on it, and 10 binary classifiers — neurons in the output layer — classifying each digit from 0-9. If there are a conservative number of 2 hidden layers (since many contemporary and much more advanced structures contain tens) with 16 neurons each (such as in the example from Grant Sanderson

of 3Blue1Brown), then the total number of weights and biases such a network requires is 13002 (Sanderson)! This is also why there is a constant strive for optimization in this field, as even though the individual calculations are simple for a computer, the sheer number of them slows the process down, especially taking into consideration the computational demand that the “learning” process takes.

Our network, however, only has 4 neurons in its only hidden layer, and 2 neurons in the output layer, so if we randomly generate a set of weights, biases, and neuron activation values for the hidden layer, we can show the activation calculation for the output layer:

$$W_{3,2} = \begin{bmatrix} -1.48 & -2.49 & 2.72 & 3.46 \\ -1.81 & 2.84 & -0.438 & -4.80 \end{bmatrix}, L_2 = \begin{bmatrix} 0.978 \\ 0.322 \\ 0.315 \\ 0.201 \end{bmatrix}, B_3 = \begin{bmatrix} 0.884 \\ -0.226 \end{bmatrix}$$

$$L_3 = S(W_{3,2}L_2 + B_3)$$

$$L_3 = S\left(\begin{bmatrix} -1.48 \times 0.978 + (-2.49 \times 0.322) + 2.72 \times 0.315 + 3.46 \times 0.201 \\ -1.81 \times 0.978 + 2.84 \times 0.322 + (-0.438 \times 0.315) + (-4.80 \times 0.201) \end{bmatrix} + \begin{bmatrix} 0.884 \\ -0.226 \end{bmatrix}\right)$$

$$L_3 = S\left(\begin{bmatrix} -0.69696 \\ -1.95847 \end{bmatrix} + \begin{bmatrix} 0.884 \\ -0.226 \end{bmatrix}\right) = S\left(\begin{bmatrix} 0.18704 \\ -2.18447 \end{bmatrix}\right) = \begin{bmatrix} S(0.18704) \\ S(-2.18447) \end{bmatrix} \approx \begin{bmatrix} 0.547 \\ 0.101 \end{bmatrix}$$

The choice to randomly assign weights and biases might seem unusual, especially since we know what they do and could use them to influence the network by, in natural language processing applications for example, assigning a greater weight to wavelength inputs associated with different sounds we might wish to ultimately classify, but even for industrial applications this is not done due to, again, the sheer size of the networks and how many connections there potentially are, all of which would require a value to be set by hand. In the worst of cases, this is physically impossible, but it is always pointless due to the nature of the mathematics behind what makes these machines intelligent, and what ultimately motivates them — the neurons, weights, and biases, and the equations that train and correct them, do not

care about, nor can they “consciously” distinguish between, different wavelengths of our inputs, or the edges and curves on a handwritten digit. Mathematicians and computer scientists, however, can and do, which is why they frame these tasks in a purely mathematical way that a system like this can follow.

On its own, the result obtained in the preceding calculation is a small part of the whole mathematical structure and process that a neural network very much is. It represents only one step out of potentially many that processing a single sample input might take, and only one sample input out of many thousands that are usually needed, whatever the problem we are optimizing for might be.

4 System motivation, training, and optimization

It follows, naturally, that powerful mathematics, capable of taking so many connections and their influence on the overall network into account, is behind the training and optimization of such a complex system. This is by far the most important aspect of these structures, and the one that gives the field they are associated with the name machine “learning”.

What it means for a machine to learn, in the context that the math done previously established, is getting as close to the optimal values for the many weights and biases of the network, starting from random initiation, similar to our example. This is done through the following, largely intuitive steps (Nielsen 1). First recording the outputs of our network when we feed it data for which we know what the output should be (“labelled” data). With this, we can obtain insight into the success of our network by defining a “cost function”, one type of which involves taking the sum of the squares of the differences between the network’s outputs, and what we know the output should be. Since every weight and bias affects what our network outputs, and those value are used in computing the cost function, this effectively means that these weights and biases are its inputs, which is important for further steps. Optimizing the network, then, means minimizing the cost function (since that would imply that the outputs are close to what the label for our data is expecting), and minimizing the cost function necessitates a change in the weights and biases resulting in the outputs, in a way that is favorable.

The mathematical concept facilitating these changes is gradient descent. It involves computing the gradient of the cost function, a vector containing the partial derivatives of a function with respect to its inputs, effectively compiling the relative magnitude of the rates at which changes to individual inputs would move the function to a

maximum (Kreyszig 310). Since the goal is to instead minimize the cost function (whose inputs are all the weights and biases, as mentioned), we would instead apply small, proportional, and opposite changes to the weights and biases of our network from where they correspond in the gradient vector, therefore descending it. We choose a small negative factor, the learning rate (Nielsen 1), to multiply the gradient vector by, before adding that resultant vector to the vector of current weights and biases, therefore changing them by a step in the right direction towards minimizing the cost function and taking into account the relative effect each of them has on the cost (their magnitude in the gradient vector), without nuking them with the entire gradient vector. It is important to note, however, that since we want to train our network to classify inputs we feed it among all the possible binary classifiers, we need to train it on labelled data sets that are ideally large and include a similar number of examples for each output option. This ensures that we do not end up with gradients preferential to optimizing for one specific output neuron and allows us to instead descend a gradient obtained by averaging the individual gradients for each training example, while being confident that it will contain the actual values that we can use to change the weights and biases in a way that will optimize the system for any input we give it. In practical applications, “stochastic gradient descent” is employed instead, since the regular gradient descent process implies the calculation of thousands of gradients, one for each training example, which is simply too computationally expensive (Sanderson). They are not, however, fundamentally different, so in stochastic gradient descent we only add another averaging step, this time in the cost functions, that is the values of the output layer, of a set number of training examples, which we use to get one gradient for all of them, that we average with the other gradients from a different set of training examples. What this ultimately

means for the network is that it might not move in the most optimal direction to minimize the overall cost every time, but it will on average, and the time taken for a computer to get through all the data will be significantly shorter (Sanderson). After getting through this process once, the overall gradient will serve as the one which dictates the changes we make to each weight and bias before doing this process again, with different sets, of different data until our network can accurately classify new, to it unknown, data to a satisfactory degree (sometime above 99% for more sophisticated neural networks (Nielsen 6), which is why they can be used for so many real-world applications, so reliably).

The way we compute any gradient vector, that is the partial derivatives in it, is at the heart of this entire process and is called reverse-mode differentiation, commonly referred to as the “backpropagation” algorithm in the context of neural networks (Olah). It is essentially the chain rule; the product of partial derivatives at intermediary steps needed to get to the connection we wish to ultimately calculate the effect of, and is best understood through an example, showing the raw math behind this “artificial intelligence” and the way it learns, and concluding the body of this essay:

Suppose, for example, we want to know what change should be made to the weight connecting the first neurons in the hidden and output layer in the network from Figure 3, $W_{1,1} = 0.15$:

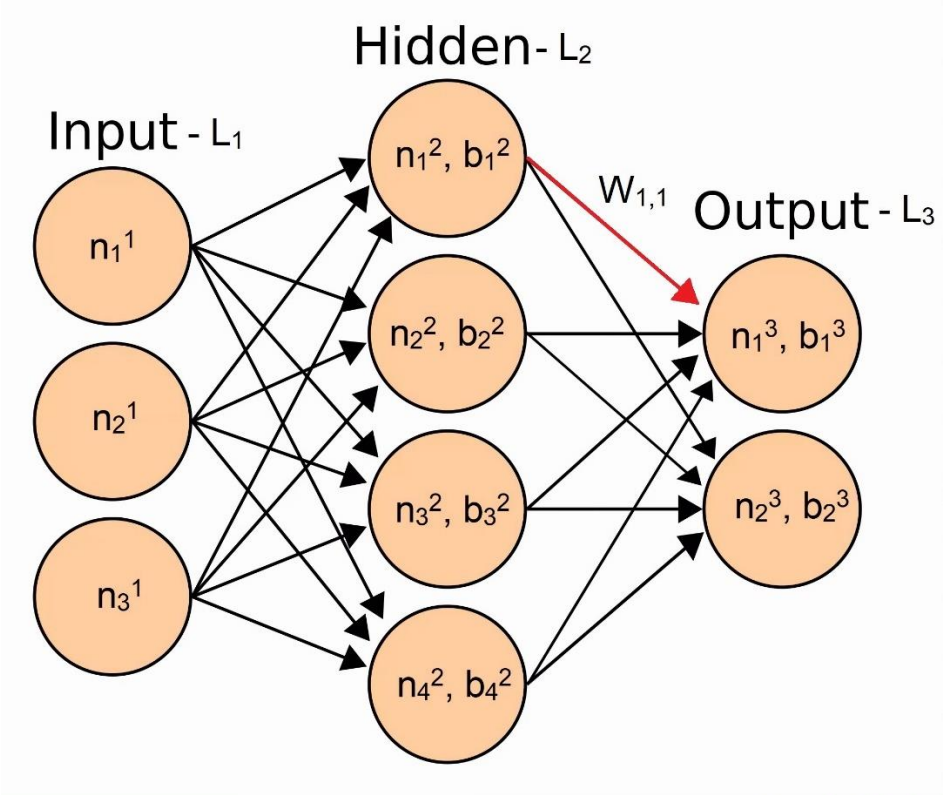

Figure 4: Showing the network and weight optimized in following calculations

We need to calculate the individual partial derivative for this weight from the gradient vector of the cost function, which will in this example be treated as the final gradient vector, with gradient descent applied on one hypothetical training example. This is done for simplicity's sake, since, as mentioned already, many training examples are used to train actual networks, whose gradients are then averaged and that result descended.

Since the cost function after one training example for this network is:

$$C = (a(n_1^3) - E_1)^2 + (a(n_2^3) - E_2)^2$$

With E_1 and E_2 being the expected values for the respective neurons in the output layer, say 1 for both, meaning we wanted to see both output neurons fully activated. We can see that C depends on $a(n_1^3)$ and $a(n_2^3)$, but since only $a(n_1^3)$ is connected to $W_{1,1}$, this is the value we will used in the chain rule expression for the partial derivative we are looking for. Further, $a(n_1^3)$ is of the form:

$$a(n_1^3) = \mathrm{S}(w_{1,1}a(n_1^2) + w_{1,2}a(n_2^2) + w_{1,3}a(n_3^2) + w_{1,4}a(n_4^2) + b_1^3)$$

Meaning it is dependent on the activation sum, the value inside the activation function, which we can label as AS_m^l for easier calculation, in this case AS_1^3:

$$AS_1^3 = w_{1,1}a(n_1^2) + w_{1,2}a(n_2^2) + w_{1,3}a(n_3^2) + w_{1,4}a(n_4^2) + b_1^3$$

Which is finally dependent on the weight $w_{1,1}$ among all the other weights, neuron activations and bias present, but which are used in calculating their own dependency on C and not in this specific example.

Thus, the partial derivative which we are looking for is:

$$\frac{\partial C}{\partial W_{1,1}} = \frac{\partial C}{\partial a(n_1^3)}\frac{\partial a(n_1^3)}{\partial AS_1^3}\frac{\partial AS_1^3}{\partial W_{1,1}}$$

If we give some value to $a(n_1^2)$ and AS_1^3, that we might record after a hypothetical training example, we can calculate $a(n_1^3)$, and have all the relevant variables to carry out the above calculation:

$$a(n_1^2) = 0.8 \; and \; AS_1^3 = 0.5 \; so \; a(n_1^3) = S(0.5) \approx 0.6225$$

So, treating all other variables beside the one that the partial derivative is written with respect to, as constants, we can evaluate the constituent partial derivatives (Partial):

$$\frac{\partial C}{\partial a(n_1^3)} = \frac{\partial}{\partial a(n_1^3)}((a(n_1^3) - E_1)^2 + (a(n_2^3) - E_2)^2)$$

$$\frac{\partial C}{\partial a(n_1^3)} = \frac{\partial}{\partial a(n_1^3)}((a(n_1^3) - E_1)^2) + \frac{\partial}{\partial a(n_1^3)}((a(n_2^3) - E_2)^2)$$

$$\frac{\partial C}{\partial a(n_1^3)} = \frac{\partial}{\partial a(n_1^3)}((a(n_1^3) - E_1)^2) + 0$$

$$\frac{\partial C}{\partial a(n_1^3)} = 2(a(n_1^3) - E_1)\frac{\partial}{\partial a(n_1^3)}(a(n_1^3) - E_1) + 0$$

$$\frac{\partial C}{\partial a(n_1^3)} = 2(a(n_1^3) - E_1) \cdot 1 + 0 = 2(0.6225 - 1) = -0.755$$

$$\frac{\partial a(n_1^3)}{\partial AS_1^3} = \frac{\partial}{\partial AS_1^3}(S(AS_1^3)) = \frac{\partial}{\partial AS_1^3}\left(\frac{1}{1+e^{-AS_1^3}}\right)$$

$$\frac{\partial a(n_1^3)}{\partial AS_1^3} = \frac{\partial}{\partial AS_1^3}((1+e^{-AS_1^3})^{-1})$$

$$\frac{\partial a(n_1^3)}{\partial AS_1^3} = -\frac{1}{(1+e^{-AS_1^3})^2}\frac{\partial}{\partial AS_1^3}(1+e^{-AS_1^3})$$

$$\frac{\partial a(n_1^3)}{\partial AS_1^3} = -\frac{1}{\left(1+e^{-AS_1^3}\right)^2}e^{-AS_1^3}\frac{\partial}{\partial AS_1^3}(-AS_1^3)$$

$$\frac{\partial a(n_1^3)}{\partial AS_1^3} = -\frac{1}{\left(1+e^{-AS_1^3}\right)^2}\cdot\left(-e^{-AS_1^3}\right) = \frac{e^{-0.5}}{(1+e^{-0.5})^2} \approx 0.235$$

$$\frac{\partial AS_1^3}{\partial W_{1,1}} = \frac{\partial}{\partial W_{1,1}}(w_{1,1}a(n_1^2) + w_{1,2}a(n_2^2) + w_{1,3}a(n_3^2) + w_{1,4}a(n_4^2) + b_1^3)$$

$$\frac{\partial AS_1^3}{\partial W_{1,1}} = \frac{\partial}{\partial W_{1,1}}(w_{1,1}a(n_1^2)) + \frac{\partial}{\partial W_{1,1}}(w_{1,2}a(n_2^2)) + \frac{\partial}{\partial W_{1,1}}(w_{1,3}a(n_3^2)) + \frac{\partial}{\partial W_{1,1}}(w_{1,4}a(n_4^2)) + \frac{\partial}{\partial W_{1,1}}(b_1^3))$$

$$\frac{\partial AS_1^3}{\partial W_{1,1}} = a(n_1^2) + 0 + 0 + 0 + 0 = 0.8$$

Therefore:

$$\frac{\partial C}{\partial W_{1,1}} = -0.755 \cdot 0.235 \cdot 0.8 \approx -0.142$$

This partial derivative is an element in the cost function gradient vector, and as such tells us by what amount to change $W_{1,1}$ so as to maximize the cost function. Since this is opposite of what the goal of our network is, we can instead obtain the opposite value by scaling the gradient by the aforementioned learning rate, $-r, r > 0$, which might have a value of -0.05. This ensures we are correcting our networks parameters, the weights and biases, in a way that minimizes the network's overall cost making it better at identifying unknown examples since that is how our cost function is defined, thus achieving "learning".

Finally, we can show the corrected value for $W_{1,1}$, $\Delta W_{1,1}$, explicitly:

$$\Delta W_{1,1} = W_{1,1} - r\frac{\partial C}{\partial W_{1,1}}$$

$$\Delta W_{1,1} = 0.15 - 0.05 \cdot (-0.142) = 0.1271$$

However, this correction step is done through operations with the parameters of the system as a whole, so if we define a parameter vector, P, which contains all of the weights and biases of our network as such:

$$P = \begin{bmatrix} \vdots \\ W_{1,1} \\ \vdots \end{bmatrix}$$

And we follow the same procedure, propagating backwards to find how the cost function depends on each parameter, the cost function gradient vector, ∇C, is:

$$\nabla C = \begin{bmatrix} \vdots \\ \frac{\partial C}{\partial W_{1,1}} \\ \vdots \end{bmatrix}$$

Then the overall equation for making a correction to all of the parameters of our network after one training sample is:

$$\Delta P = P - r\nabla C = \begin{bmatrix} \vdots \\ W_{1,1} \\ \vdots \end{bmatrix} + \begin{bmatrix} \vdots \\ -0.05 \cdot \frac{\partial C}{\partial W_{1,1}} \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots \\ \Delta W_{1,1} \\ \vdots \end{bmatrix}$$

These corrected parameters are then used for processing the following training sample (or set of training samples) with the eventual goal of perfecting the efficiency of the network, that is the number of unknown training samples classified correctly vs. incorrectly.

5 Conclusion

Neural networks are a modern technology that has quickly expanded its influence far beyond the theoretical and into the mainstream and consumer spaces with applications in processing speech (natural language processing), handwritten objects, and even facial recognition for securing the phones we glance at so often throughout the day. Their structure allows them to “translate” complex problems into mathematics that a computer understands, ultimately being motivated through the interaction between all of its constituent elements. Mathematical concepts from many different fields such as linear algebra, calculus, logic and computation, all come together in an elegant way to form systems that solve very real, very practical problems thought impossible for a computer only decades ago. This is done in a surprisingly intuitive way, as every operation from the way that neural networks operate and learn, no matter how confusing it might seem at first glance, can essentially be described verbally without the need for complex terminology. Coming back to the original research question, the way in which mathematics manifest in neural networks is by facilitating the learning and optimization process at the core of these structures in a way outlined in the essay’s argument, leading to a final product which is effective to a great extent in solving various real-world problems that affect basically everyone alive today.

Perhaps the main limitation that this essay faced is the lack of both space and high-level understanding of the programming side of neural networks, and the code which would have potentially provided a much clearer picture of their functionality. The cutting edge of this field is also incredibly innovative and worthy of exploration, as it looks to provide a breakthrough that would make notoriously difficult problems

solved in many ways, similar to what current neural networks did for problems that are now history.

Works cited

Daityari, Shaumik. “A Beginner’s Guide to Keras: Digit Recognition in 30 Minutes.” Sitepoint, 07 Jul. 2021, www.sitepoint.com/keras-digit-recognition-tutorial/.

Guichard, David. *Whitman Calculus.* Whitman College, 2010.

Kreyszig, Erwin. *Advanced Engineering Mathematics.* Third ed., Wiley, 1972.

“Neural Network.” DeepAI Machine Learning Glossary, deepai.org/machine-learning-glossary-and-terms/neural-network.

Nielsen, Michael A. “Neural Networks and Deep Learning.” Determination Press, 2015, neuralnetworksanddeeplearning.com.

Nykamp, Duane Q. “Dot product in matrix notation.” Math Insight, mathinsight.org/dot_product_matrix_notation.

---. “Multiplying matrices and vectors.” Math Insight, mathinsight.org/matrix_vector_multiplication.

Olah, Christopher. “Calculus on Computational Graphs: Backpropagation.” colah’s blog, 31 Aug. 2015, colah.github.io/posts/2015-08-Backprop.

“Partial derivative.” Encyclopedia of Mathematics, 20 Jan. 2022, encyclopediaofmath.org/index.php?title=Partial_derivative&oldid=48132.

“Perceptron.” DeepAI Machine Learning Glossary, deepai.org/machine-learning-glossary-and-terms/perceptron.

Sanderson, Grant. “Neural Networks.” 3Blue1Brown, 05 Oct. 2017, www.3blue1brown.com/lessons/neural-networks.

“Synapse.” DeepAI Machine Learning Glossary, deepai.org/machine-learning-glossary-and-terms/synapse.

Wood, Thomas. “Activation Function.” DeepAI Machine Learning Glossary, deepai.org/machine-learning-glossary-and-terms/activation-function.