A Mathematical Overview of Forward and Backward Propagation in Artificial Neural Networks

Harrison Green

July 2025

1 Introduction

First formalized in 1943 by Warren McCulloch and Walter Pitts, the artificial neural network (ANN) has served as an integral tool and foundational component to the development of modern artificial intelligence[1]. Modeled after the human brain, these man-made connections of neurons have served as a highly effective basis for modern technological innovation.

This paper presents an introductory mathematical breakdown of the forward and backward propagation processes applied by the most widely used contemporary ANN model, including the ReLU, softmax and cross-entropy loss functions.

2 Input and Initial Processing

An ANN begins its training by converting input to a numerical form. For instance, suppose an ANN is being trained to distinguish between 8x8, grayscale images of circles and squares. If one neuron is assigned to each pixel, we have 64 initiating neurons—collectively referred to as the input layer.

In our example, each neuron will represent the brightness of the pixel within the interval $[0, 1]$. A pure white pixel will be interpreted as 1, a black pixel as 0, and all other gray pixels will fall elsewhere in the interval.

Suppose that this network is being trained on the following image. The assignment of neurons to pixels would be as follows, where $a_1, a_2, ..., a_{64}$ are the values of the neurons in the input layer:

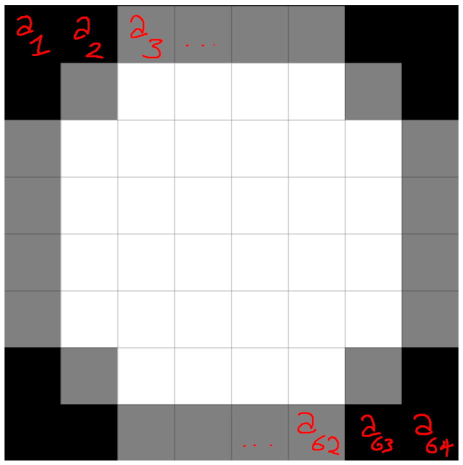

Figure 1: The assignment of the neurons to pixels.

And the values of the neurons, within the interval $[0, 1]$, would be:

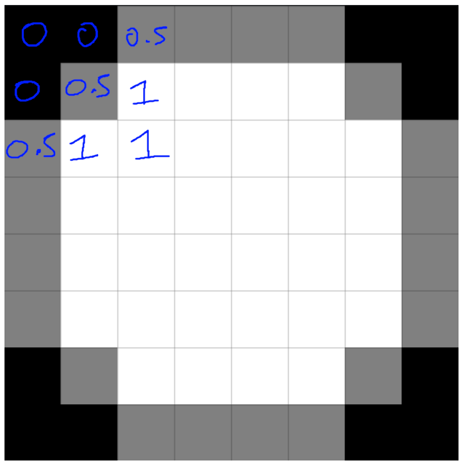

Figure 2: The assignment values of the assigned neurons.

With the data received and fully processed, training can begin. Note that input of all kind can be processed by such networks, including colored images with RGB vectors for pixel values, text, or even audio bytes.

3 Forward Propagation

Below is a visualization of a neural network with 3 layers:

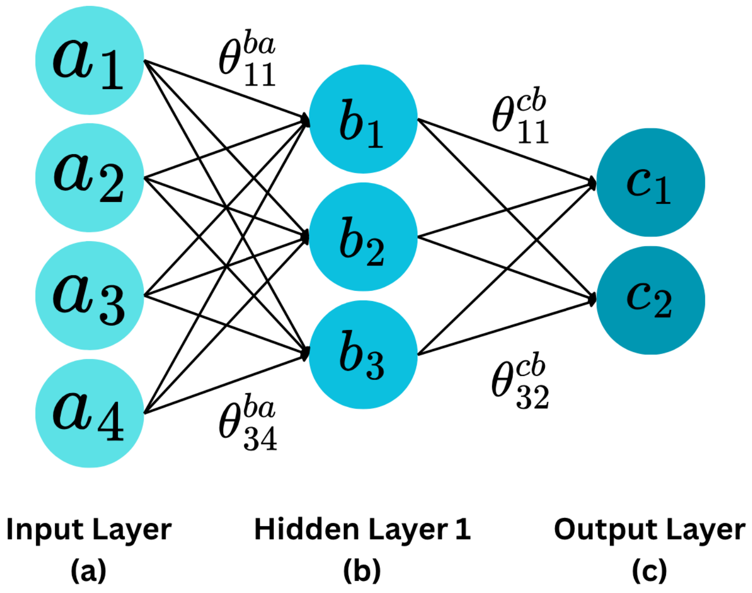

Figure 3: A simplified neural network demonstrating all connections between neurons.

The input layer, represented by a_1, a_2, a_3, a_4, has four neurons. The output layer, denoted by c_1, c_2, has only two neurons, implying two possible outputs. It is important to note that most neural networks will include several hidden layers; this network has only one for simplicity.

In order to calculate the value of a neuron in a hidden or output layer, we must add the products of the previous neurons with their corresponding weights, add the bias, and apply an activation function (in our case, ReLU). The value of b_1, for instance, can be expressed by the following equation:

$$b_1 = \max\left\{0, \sum_{j=1}^{4} a_j \theta_{1j}^{ba} + \beta_1^b\right\}$$

We start defining a neuron by summing the values of the previous neurons and weights. In this example, the weights between layers b and a are denoted as $\theta_{11}^{ba}, \theta_{12}^{ba}, \theta_{...}^{ba}, \theta_{34}^{ba}$, where θ_{mn}^{ba} refers to the weight connecting b_m and a_n.

In run/step 0, before the AI has received training data, weights are assigned randomly. Common methods of distribution are uniformly randomizing values within an interval, and Gaussian distribution. More advanced methods include the Xavier initialization[2].

After summing our weights, we must add our bias values, β. These bias values are constants that are each assigned to one neuron. They interact with the activation function to provide overall stronger learning by amplifying the impact of neurons with small values—we will go into more depth momentarily[3]. Due to the naming convention, we can effortlessly represent the value of the hidden layer in terms of the input layer with matrices:

$$\begin{bmatrix} b_1^{raw} \\ b_2^{raw} \\ b_3^{raw} \end{bmatrix} = \begin{bmatrix} \theta_{11} & \theta_{12} & \theta_{13} & \theta_{14} \\ \theta_{21} & \theta_{22} & \theta_{23} & \theta_{24} \\ \theta_{31} & \theta_{32} & \theta_{33} & \theta_{34} \end{bmatrix} \cdot \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} + \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix}$$

Our current values, labeled "raw" have yet to be processed by our activation function, defined as:

$$\text{ReLU}(b_n^{raw}) = \max\{0, b_k^{raw}\} = b_n$$

The ReLU activation function takes the larger value between 0 and our raw neuron. In other words, if our neuron value after weight and bias calculation is less than 0, we set it to 0. Activation functions are in place in order to create a curve; without an activation function, neural networks could be simplified to linear equations. As efficient as linear networks are, capacity for learning is significantly restricted, and so activation functions are typically implemented, by convention.

Our activation function, ReLU, acts as a filter, only allowing the neuron values above 0 to continue through the network.

We apply our activation function:

$$\begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} \max\{0, b_1^{raw}\} \\ \max\{0, b_2^{raw}\} \\ \max\{0, b_3^{raw}\} \end{bmatrix}$$

And continue our simplified network to the output layer.

$$\begin{bmatrix} c_1^{raw} \\ c_2^{raw} \end{bmatrix} = \begin{bmatrix} \theta_{11} & \theta_{12} & \theta_{13} \\ \theta_{21} & \theta_{22} & \theta_{23} \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} + \begin{bmatrix} \beta_1 \\ \beta_2 \end{bmatrix}$$

Apply our activation function:

$$\begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} \max\{0, c_1^{raw}\} \\ \max\{0, c_2^{raw}\} \end{bmatrix}$$

Finally, we have the values of the neurons in our output layer. We, however, are not done operating on these values. In their current form, we can call these neurons logits: raw scores that are yet to be converted into probability.

A neural network works by interpreting data, then classifying it by its resemblance to another object (or set of objects), such as in the aforementioned ANN being trained to distinguish circles from squares.. To convert these logits to probabilities, we apply the softmax function. We will call our final probabilities p_1, p_2.

$$p_k = \frac{e^{c_k}}{\sum_{j=1} e^{c_j}}$$

Here, the sum j iterates up to the number of classes, or neurons in the output layer (in our case, 2). These exponential functions are implemented for the sake of smoothness, exaggerating difference, and working well with later functions. Moreover, the softmax function will always produce values $\in [0, 1]$, which sum to 1, allowing for functional probability values.

4 Loss Calculation

We will now compute the loss of the network. The loss is a value that we use to quantify the error of the ANN, for the sake of effective, efficient training. In our model, we will use the cross-entropy loss function:

$$\text{Loss} = -\sum_{j=1} r_j \left(\log\left(p_j\right)\right)$$

Where r_1, r_2 are the true probabilities of each scenario.

Let us return to our initial neural network, where we are attempting to distinguish between circles and squares. Assume we run our circular figure (see Fig. 1) through the neural network, and it returns the following matrix of probabilities:

$$\begin{bmatrix} p_\circ \\ p_\square \end{bmatrix} = \begin{bmatrix} 0.6 \\ 0.4 \end{bmatrix}$$

That is, that there is a 0.6 (60%) chance of the image being a circle, and a 0.4 (40%) chance of a square. To a human, the shape in the image is quickly identifiable as a circle. So, we know that the true probability of each class, denoted as r, is:

$$\begin{bmatrix} r_\circ \\ r_\square \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

If we assign $r_\circ = r_1 = 1$, $p_\circ = p_1 = 0.6$, and similarly $r_\square = r_2 = 0$ and $p_\square = p_2 = 0.4$, we can use our values with the cross-entropy function:

$$\text{Loss} = -\sum_{j=1} r_j \left(\log\left(p_j\right)\right) = -(1 \log(0.6) + 0 \log(0.4)) = -\log(0.6) \approx 0.22$$

Thus, our loss value is $-\log(0.6)$.

5 Backward Propagation

With our loss value, we will use backward propagation to modify our weights and biases, in hope that future iterations will feature better performance.

We start by calculating how much each weight contributed to the loss, defined as error. To do this, we will employ the partial derivative of the loss, with respect to the weights between these layers. The weight connecting hidden neuron b_n to output neuron c_m is denoted as θ^{cb}_{mn}. The partial derivative is then:

$$\frac{\partial \text{Loss}}{\partial \theta^{cb}_{mn}} = \delta^c_m \cdot b_n$$

δ^c_m denotes the error of neuron c_m, and we use this formula to get the error of the weight. Since we are computing the error between our final two layers, we can apply the softmax and cross-entropy functions to simply define the error of the neurons in the output layer:

$$\frac{\partial \text{Loss}}{\partial \theta_{mn}} = (p_m - r_m) \cdot b_n$$

In order to calculate weight error in earlier layers, we are unable to simplify the derivative with softmax and cross-entropy in the same way we could for hidden $\rightarrow$ output. So, we use a recursive definition:

$$\delta_n^b = \left(\sum_{m=1}^{M} \delta_m^c \cdot \theta_{mn}^{cb} \right) \cdot \mathbf{1}_{b_n^{\text{raw}} > 0}$$

Where M denotes the total number of neurons in layer c.

Here, $\mathbf{1}_{b_m^{\text{raw}} > 0}$ is an activation function:

$$\mathbf{1}_{b_m^{\text{raw}} > 0} = \begin{cases} 1 & b_m^{\text{raw}} > 0, \\ 0 & b_m^{\text{raw}} \leq 0 \end{cases}$$

The process to compute bias error is similar, albeit slightly simpler. Since biases correspond to individual neurons, we can define bias error as equal to our neuron error:

$$\frac{\partial \text{Loss}}{\partial \beta_m^c} = (p_m - r_m)$$

And between hidden layers:

$$\frac{\partial \text{Loss}}{\partial \beta_m^c} = \delta_m^c$$

With our errors computed, we can proceed in updating each parameter using the following formulas:

$$\theta := \theta - \eta \cdot \frac{\partial \text{Loss}}{\partial \theta} \quad \text{and} \quad \beta := \beta - \eta \cdot \frac{\partial \text{Loss}}{\partial \beta}$$

Here, η is the learning rate: a tunable constant that controls the size of updates. The most commonly used tuning rate is 10^{-2} (0.01), as it provides the perfect balance of speed and stability; this tuning rate allows for fast growth without being so fast that the network oscillates around optimal values, counter-intuitively sacrificing efficiency[4].

To further demonstrate the process of backward propagation, suppose we are trying to update the weights and biases in the following network:

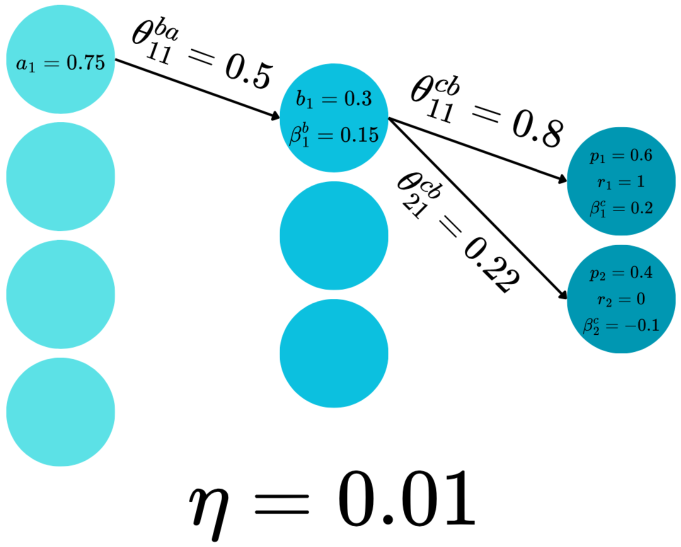

Figure 4: A neural network with labeled values, pre-backward propagation

To calculate hidden $\rightarrow$ output weight errors:

$$\frac{\partial \text{Loss}}{\partial \theta_{11}^{cb}} = \delta_1^c \cdot b_1 = (-0.4)(0.3) = -0.12$$

$$\frac{\partial \text{Loss}}{\partial \theta_{21}^{cb}} = \delta_1^c \cdot b_1 = (0.4)(0.3) = 0.12$$

Hidden $\rightarrow$ hidden weight errors:

$$\frac{\partial \text{Loss}}{\partial \theta_{11}^{ba}} = \delta_1^b \cdot a_1 = ((-0.4)(0.8) + (0.4)(0.22)) \cdot (0.75) = -0.174$$

And bias weight errors:

$$\frac{\partial \text{Loss}}{\partial \beta_1^c} = \delta_1^c = -0.4$$

$$\frac{\partial \text{Loss}}{\partial \beta_2^c} = \delta_2^c = 0.4$$

$$\frac{\partial \text{Loss}}{\partial \beta_1^b} = \delta_1^b = -0.232$$

Finally, update weights:

$$\theta_{11}^{cb} := 0.8 - 0.01(-0.12) = 0.8012$$
$$\theta_{21}^{cb} := 0.22 - 0.01(0.12) = 0.2188$$
$$\theta_{11}^{ba} := 0.5 - 0.01(0.174) = 0.50174$$

And biases:

$$\beta_1^c := 0.2 - 0.01(-0.4) = 0.204$$
$$\beta_2^c := -0.1 - 0.01(0.4) = -0.104$$
$$\beta_1^b := 0.15 - 0.01(-0.232) = 0.15232$$

Thus, the backward propagation process is complete, and all weight and bias parameters have been updated appropriately.

6 Further Training

Further training iterations will follow the same steps. Notably, it is necessary that diverse data is used for training and testing; feeding an ANN the same data more than once does not modify the weights and biases in a way that is helpful to the ANN's learning.

In most cases, ANNs are trained with batches of data at a time, without modifying weights or biases between individual iterations in order to maximize efficiency. With batch training, weights and biases are instead adjusted with the loss averages of batches.

7 Conclusion

This paper has provided a mathematical exploration of the forward and backward propagation processes that are at the center of the training of artificial neural networks. By following how the data transforms and shifts from input to output, we demonstrated how ANNs learn through weight and bias updates through gradient descent. Key components such as the ReLU and softmax functions, along with the cross-entropy loss, were discussed to describe learning computational level.

References

[1] *Neural Nets—History and People*, University of Washington, accessed July 23, 2025. https://faculty.washington.edu/seattle/Neural-Nets/History.html

[2] J. Brownlee, *Weight Initialization for Deep Learning Neural Networks*, MachineLearningMastery.com, February 2, 2021. https://www.machinelearningmastery.com/weight-initialization-for-deep-learning-neural-networks/

[3] *Importance of Neural Network Bias and How to Add It*, Turing.com, September 29, 2022. https://www.turing.com/kb/necessity-of-bias-in-neural-networks

[4] *The Learning Rate: A Hyperparameter That Matters*, Medium.com, May 28, 2023. https://mohitmishra786687.medium.com/the-learning-rate-a-hyperparameter-that-matters-b2f3b68324ab