Neural Networks: The Backpropagation Algorithm

Annette Lopez Davila
Math 400, College of William and Mary
Professor Chi-Kwong Li

Abstract

This paper illustrates how basic theories of linear algebra and calculus can be combined with computer programming methods to create neural networks.

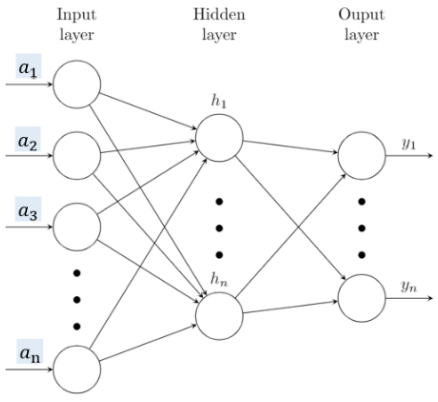
Keywords: Gradient Descent, Backpropagation, Chain Rule, Automatic Differentiation, Activation and Loss Functions

1 Introduction

As computers advanced in the 1950s, researchers attempted to simulate biologically inspired models that could recognize binary patterns. This led to the birth of machine learning, an application of computer science and mathematics in which systems have the ability to "learn" by improving their performance. Neural networks are algorithms that can learn patterns and find connections in data for classification, clustering, and prediction problems. Data including images, sounds, text, and time series are translated numerically into tensors, thus allowing the system to perform mathematical analysis.

In this paper, we will be exploring fundamental mathematical concepts behind neural networks including reverse mode automatic differentiation, the gradient descent algorithm, and optimization functions.

2 Neural Network Architecture



In order to understand Neural Networks, we must first examine the smallest unit in a system: the neuron. A neuron is a unit which holds a number; it is a mathematical function that collects information. These neurons are connected to each other in layers and are assigned an activation value; the higher the activation value, the greater the activation. Each activation number is multiplied with a corresponding weight which describes connection strength from node to node. A neural network has an architecture of input nodes, output nodes, and hidden layers. For each node in a proceeding layer, the weighted sum is computed:

`

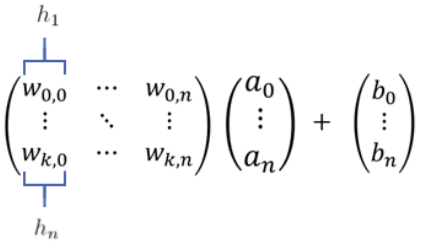
$$z_i = w_1 a_1 + w_2 a_2 + \cdots w_n a_n$$

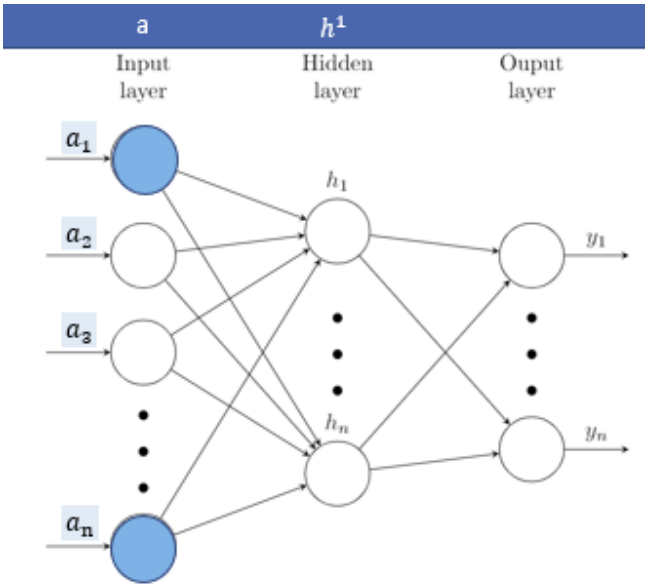
$$where\ i = [1,\ \#\ of\ neurons\ in\ hidden\ layer]\ and\ \ n = \#\ of\ \ activation\ \ numbers$$

The weighted inputs are added with a bias term in order for the output to be meaningfully active.

$$z_i = w_1 a_1 + w_2 a_2 + \cdots w_n a_n + b$$

A neural network's hidden layers have multiple nodes. For the first node in the hidden layer, we multiplied the corresponding weights and biases against the activation number. This must be repeated throughout the nodes in the hidden layer. The above equation can be consolidated into vectors in order to exemplify this:





Each row in matrix $\vec{w}$ represents the weights corresponding with each hidden layer, while the columns represent the weights corresponding to a particular activation number.

3 The Activation Function

The function z_i is linear in nature; thus, a nonlinear activation function is applied for more complex performance. Activation functions commonly used include sigmoid functions, piecewise functions, gaussian functions, tangent functions, threshold functions, or ReLu functions.

Function Name	*Function*
Sigmoid/Logistic	$f(x) = \frac{1}{1+e^{-\beta x}}$
Piecewise Linear	$f(x) = \begin{cases} 0 \; if \; x \leq x_{min} \\ mx + b \; if \; x_{\max} > x > x_{min} \\ 1 \; if \; x \geq x_{\max} \end{cases}$
Gaussian	$f(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$
Threshold/Unit Step	$f(x) = \begin{cases} 0 \; if \; 0 > x \\ 1 \; if \; x \geq 0 \end{cases}$
ReLu	$f(x) = max(0, x)$
Tanh	$f(x) = \tanh(x)$

Activation function choice depends on the range needed for the data, error, and speed. Without an activation function, the neural network behaves like a linear regression model. The need for an activation function comes from the definition of linear functions and transformations. Previously we discussed the linear algebra from the input step to the hidden layer. The solution of the function would resolve as a matrix of weighted sums. In order to calculate an output, the weighted sums matrix becomes the "new" activation layer. These activation numbers have their own sets of weights and biases. When we substitute the activation matrix for the weighted sums matrix, we see that a composition of two linear functions is a linear function itself. Hence, an activation function is needed.

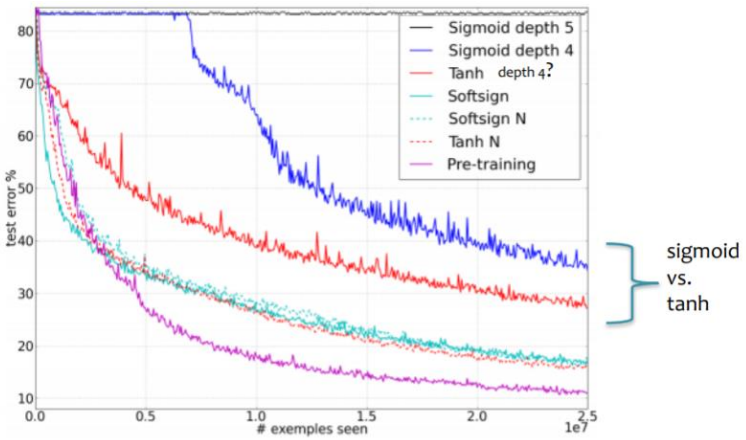


Figure from Glorot & Bentio (2010)

Proof: Composition of Linear Functions

$$\vec{z}_1 = \vec{w}_1\vec{a} + b_1$$

$$\vec{z}_2 = \vec{w}_2\vec{z}_1 + b_2$$

$$\vec{z}_2 = \vec{w}_2(\vec{w}_1\vec{a} + b_1) + b_2$$

$$\vec{z}_2 = [\vec{w}_2\vec{w}_1]\vec{a} + [\vec{w}_2\vec{b}_1 + \vec{b}_2]$$

$$If\ W = [\vec{w}_2\vec{w}_1]\ and\ B = [\vec{w}_2\vec{b}_1 + \vec{b}_2], then\ \vec{z}_2 = W\vec{a} + B, which\ is\ also\ a\ linear\ function$$

With the activation function, the new weighted sum becomes:

$$h_i = \sigma(z_i) = \sigma(w_1a_1 + w_2a_2 + \cdots w_na_n + b)$$

$$\overrightarrow{h^1} = \sigma(\vec{w}\vec{a} + \vec{b})$$

4 The Cost/Loss Function

A neural network may have thousands of parameters. Some combinations of weights and biases will produce better output for the model. For example, in a binary classification problem, the algorithm will classify some input as one of two things. The output node with the highest activation number will determine how the input is classified. In a binary classification problem, there are two labels. For example, an image can be determined to be a cat or dog; the feature "cat" is given the label of 0 and "dog" is given label 1. Different weights and biases will produce different output. How can we determine which combination of parameters will be most accurate?

In order to measure error, a loss function is necessary. The loss function tells the machine how far away the combination of weights and biases is from the optimal solution. There are many loss

functions that can be used in neural networks; Mean Squared Error and Cross Entropy Loss are two of the most common.

$$MSE\ Cost = \Sigma 0.5(y - \hat{y})^2$$

$$Cross\ Entropy\ Cost = \Sigma(\hat{y}\ log(y) + (1 - \hat{y})\ log(1 - y))$$

The loss function contains every weight and bias in the neural network. That can be a very big function!

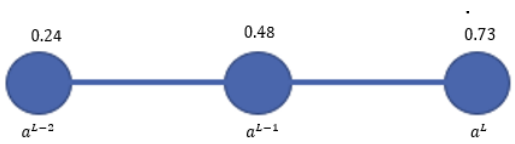
$$C(w_1, w_2, \dots, w_h, b_1, \dots, b_i)$$

5 The Backpropagation Algorithm

The objective of machine learning involves the optimization of the chosen loss function. With every epoch, the machine "learns" by adapting the weights and biases to minimize the loss. Optimization theory centers itself on calculus. For neural networks in particular, reverse-mode automatic differentiation serves a core role.

In order to minimize the cost function, one must determine which weights and biases to adjust. Computing the gradient with respect to the parameters can help us do just that, as by definition the gradient is a vector of partial derivatives of $C(w_1, w_2, \dots, w_h, b_1, \dots, b_i)$. As we recall, derivatives measure the change of a function's output with respect to its input. The gradient of the cost function tells us in which direction $C(w_1, w_2, \dots, w_h, b_1, \dots, b_i)$ decreases most quickly. This is often known as Gradient Descent. With each epoch, the machine converges towards the local minimum. Automatic differentiation combines the chain rule with massive computational power in order to derive the gradient from a potentially massive, complex model. In reverse, this algorithm is better known as Backpropagation. Backpropagation is recursively done through every single layer of the neural network.

In order to understand the basic workings of backpropagation, let us look at the simplest example of a neural network: a network with only one node per layer.



We have derived the equations for cost, weighted sum, and activated weighted sum:

$$z^L = w^L a^{L-1} + b^L$$

$$a^L = \sigma(z^L)$$

$$C = (a^L - y)^2 *$$ [1]

[1] The cost function is simplified for proof of concept

We can determine how sensitive the cost function is to changes in a single weight. Beginning from the output, we can apply the chain rule to every activation layer. For a weight between the hidden layer and output layer, our derivative is:

$$\frac{\delta C_k}{\delta w^L} = \frac{\delta z^L}{\delta w^L}\frac{\delta a^L}{\delta z^L}\frac{\delta C_k}{\delta a^L}$$

With the definition of the functions, we can easily solve for the partial derivatives:

$$\frac{\delta C_k}{\delta a} = 2(a^L - y)$$

$$\frac{\delta a^L}{\delta z^L} = \sigma'(z^L)$$

$$\frac{\delta z}{\delta w^L} = a^{L-1}$$

$$\frac{\delta C_k}{\delta w^L} = a^{L-1}\sigma'(z^L)\ 2(a^L - y)$$

This method is iterated through every weight, activation number, and bias in the system. Previously, we calculated the derivative of one particular cost function with one variable. However, in order to account for every weight in that layer, the average of the derivatives is taken:

$$\frac{\delta C}{\delta w^L} = \frac{1}{n}\sum_{k=0}^{n-1}\frac{\delta C_k}{\delta w^L}$$

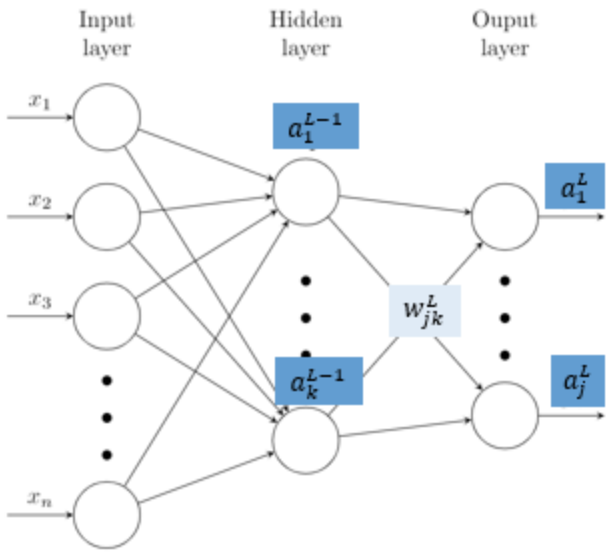
Similarly, we can calculate the sensitivity of the cost function with respect to a single bias between the hidden layer and the output layer and the derivative accounting for every bias in a layer:

$$\frac{\delta C_k}{\delta b^L} = \frac{\delta z^L}{\delta b^L}\frac{\delta a^L}{\delta z^L}\frac{\delta C}{\delta a^L} = \sigma'(z^L)\ 2(a^L - y) \qquad \frac{\delta C}{\delta b^L} = \frac{1}{n}\sum_{k=0}^{n-1}\frac{\delta C_k}{\delta b^L}$$

What happens when we go beyond the output layer and the preceding hidden layer? The chain rule is applied once more, and the derivative changes in account to its partials. For example, the derivative below accounts for the partials of the cost function with respect to an input activation number.

$$\frac{\delta C_k}{\delta a^{L-1}} = \frac{\delta z^L}{\delta a^{L-1}}\frac{\delta a^L}{\delta z^L}\frac{\delta C}{\delta a^L} = w^L\sigma'(z^L)\ 2(a^L - y)$$

Neural Networks tend to have several thousand inputs, outputs, and nodes; the above equations seem highly oversimplified. Although adding complexity changes the formulas slightly, the concepts remain the same, as seen below:



$$C_m = \sum_{j=0}^{n_L-1} \left(a_j^L - y_j\right)^2$$

$$a_j = \sigma\left(z_j^L\right)$$

$$z_j^L = \cdots + w_{jk}^L a_k^{L-1} + \cdots$$

$$\frac{\delta C_m}{\delta {w_{jk}}^L} = \frac{\delta {z_j}^L}{\delta {w_{jk}}^L}\frac{\delta {a_j}^L}{\delta {z_j}^L}\frac{\delta C_m}{\delta {a_j}^L}$$

$$\frac{\delta C_m}{\delta a^{L-1}} = \sum_{j=0}^{n_L-1} \frac{\delta {z_j}^L}{\delta {a_k}^{L-1}}\frac{\delta {a_j}^L}{\delta {z_j}^L}\frac{\delta C_m}{\delta {a_j}^L}$$

By calculating every derivative of each weight and bias, the gradient vector can be found. Although one could try to compute the gradient of a neural network by hand, the vector will usually be in complex dimensions unfathomable for us to decipher. Thus, with computational help, our neural network can perform such intricate calculations, and repeat them hundreds, if not thousands of times until the minimum is reached.

$$\nabla C = \begin{bmatrix} \frac{\delta C}{\delta w^1} \\ \frac{\delta C}{\delta b^1} \\ \vdots \\ \frac{\delta C}{\delta w^L} \\ \frac{\delta C}{\delta b^L} \end{bmatrix}$$

6 Applications and Further Research

Automatic differentiation has many applications other than in machine learning such as in Data Assimilation, Design Optimization, Numerical Methods, and Sensitivity Analysis. It is efficient, stable, precise, and known to be a better choice than other types of computer-based differentiation. Backpropagation has been called into question recently, as it does not learn continuously. For example, our brains learn continuously; they do not forget information when we learn something new. Because of this, backpropagation may be sidelined in Machine Learning in the future.

Applications of Neural Networks trained with Backpropagation vary greatly. Such applications include sonar target recognition, text recognition, network controlled steering of cars, face recognition software, remote sensing, and robotics.

7 Works Cited

Images

(n.d.). Retrieved September 03, 2020, from https://www.bing.com/images/search?view=detailV2,fashion mnist shoe

(n.d.). Retrieved September 03, 2020, from https://www.bing.com/images/search?view=detailV2,Gradient Descent Animation 3D

(n.d.). Retrieved September 03, 2020, from https://www.bing.com/images/search?view=detailV2,loss vs accuracy function neural network

(n.d.). Retrieved September 03, 2020, from https://www.bing.com/images/search?view=detailV2,neural netwrok diagram

(n.d.). Retrieved September 03, 2020, from https://www.bing.com/images/search?view=detailV2,neural netwrok diagram

(n.d.). Retrieved September 03, 2020, from https://www.bing.com/images/search?view=detailV2,sigmoid function

(n.d.). Retrieved September 03, 2020, from https://www.saedsayad.com/artificial_neural_network.htm

Sources

Gajawada, S. (2019, November 19). The Math behind Artificial Neural Networks. Retrieved September 03, 2020, from https://towardsdatascience.com/the-heart-of-artificial-neural-networks-26627e8c03ba

Kostadinov, S. (2019, August 12). Understanding Backpropagation Algorithm. Retrieved September 03, 2020, from https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd

Repetto, A. (2017, August 19). The Problem with Back-Propagation. Retrieved September 03, 2020, from https://towardsdatascience.com/the-problem-with-back-propagation-13aa84aabd71

Silva, S. (2020, March 28). The Maths behind Back Propagation. Retrieved September 03, 2020, from https://towardsdatascience.com/the-maths-behind-back-propagation-cf6714736abf

Skalski, P. (2020, February 16). Deep Dive into Math Behind Deep Networks. Retrieved September 03, 2020, from https://towardsdatascience.com/https-medium-com-piotr-skalski92-deep-dive-into-deep-networks-math-17660bc376ba

Victor Zhou. (n.d.). Machine Learning for Beginners: An Introduction to Neural Networks. Retrieved September 03, 2020, from https://victorzhou.com/blog/intro-to-neural-networks/