**Building Neural Networks: A Hands-On Journey from Scratch with Python**

[[](https://medium.com/@thisislong?source=post_page-----a789bec70b29--------------------------------)](https://medium.com/@thisislong?source=post_page-----a789bec70b29--------------------------------)

[Long Nguyen](https://medium.com/@thisislong?source=post_page-----a789bec70b29--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F33712ab9fe37&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40thisislong%2Fbuilding-a-neural-network-from-scratch-with-backpropagation-a789bec70b29&user=Long+Nguyen&userId=33712ab9fe37&source=post_page-33712ab9fe37----a789bec70b29---------------------post_header-----------)

11 min read

·

Nov 18

98

2

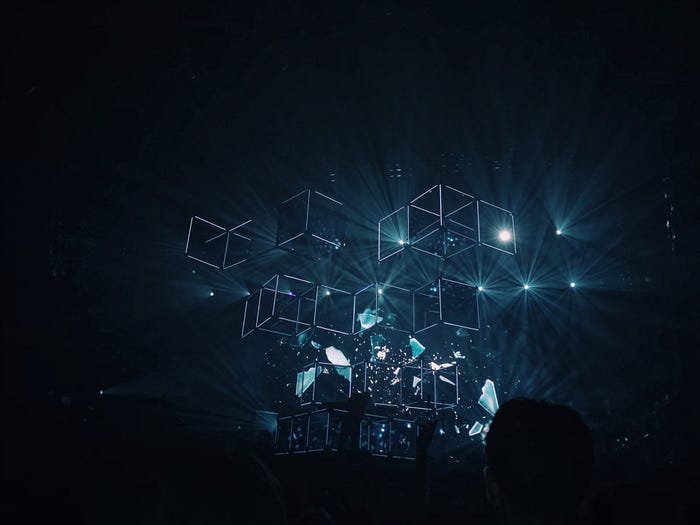


Photo by [fabio](https://unsplash.com/@fabioha?utm_source=medium&utm_medium=referral" \t "_blank) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

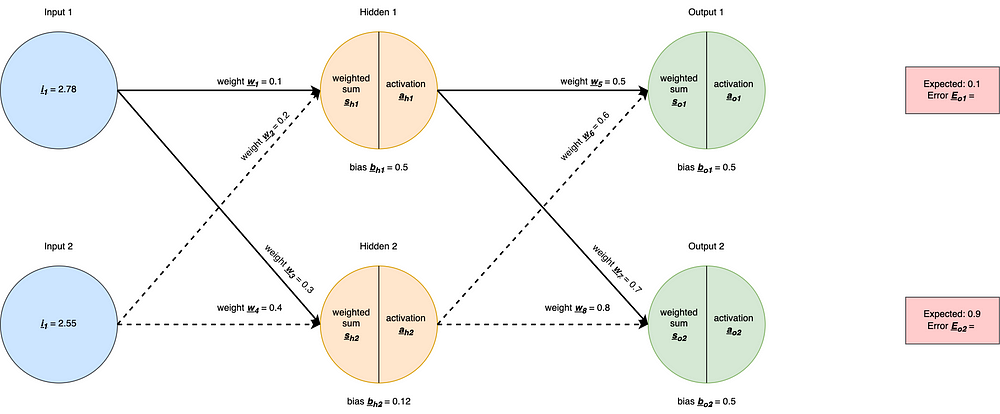
In this blog post, we will explore the fundamentals of neural networks, understand the intricacies of forward and backward propagation, and implement a neural network from the ground up with Python in 3 levels!

* Level 1: Without using external libraries
* Level 2: With [numpy](https://numpy.org/" \t "_blank)
* Level 3: With [Tensorflow](https://www.tensorflow.org/" \t "_blank)

**I. Forward and Backward Propagation Walkthrough**

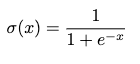
But first, let’s use an example neural network and work out the mathematical calculation one neuron at a time to understand what’s happening behind the scene!

Our sample neural network will consist of: 2 input neurons, 1 hidden layer with 2 neurons and an output layer with 2 neurons. Some initial weights and bias values have been provided to help with the calculation. Assume the expected output is 0.1 and 0.9:



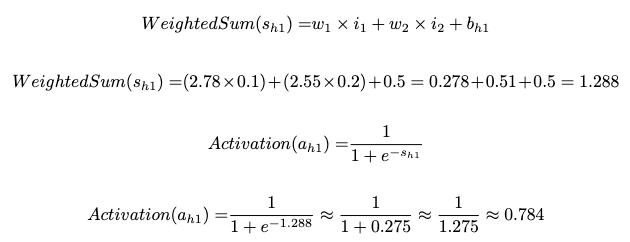
**1. Forward Propagation**

Note: I’m using sigmoid as the activation function

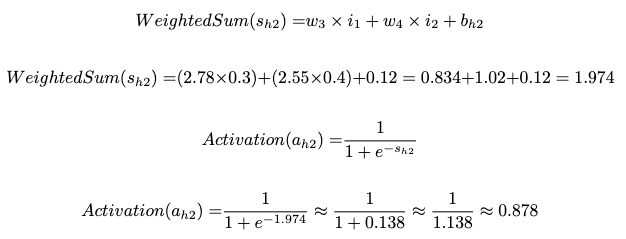


**Hidden Layer**

Hidden Neuron 1:

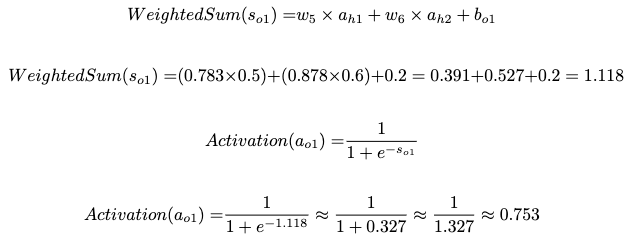


Hidden Neuron 2:

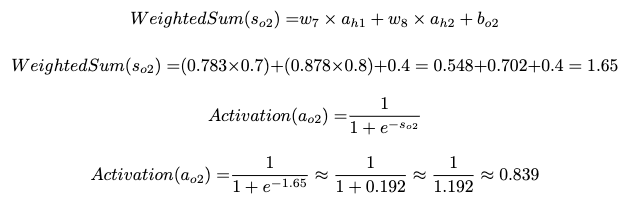


**Output Layer**

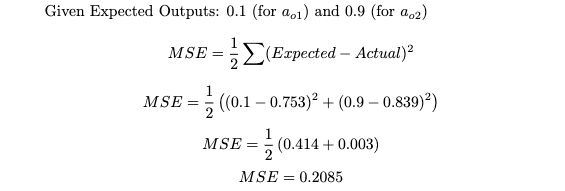
Output Neuron 1:



Output Neuron 2:



**Mean Squared Error (MSE) Calculation**



So, the Mean Squared Error (MSE) is approximately 0.2085. This is a measure of the difference between the expected and actual outputs. A lower MSE indicates a better fit of the model to the given data.

**2. Backward Propagation**

Once predictions are obtained, we need to train the network by adjusting weights and biases based on prediction errors. This is achieved through backward propagation.

Assuming we use the learning rate of 0.5



With backpropagation, we want to understand the sensitivity of the error function, which represents the disparity between actual and expected values, to a small adjustment (“nudge”) in a particular weight, such as *w5*. I found a lot of value in revising the basics of calculus and derivatives (especially the chain rule) which has helped me grasp how backpropagation works easier. This video does a great job of explaining the intuition <https://www.youtube.com/watch?v=tIeHLnjs5U8>.

Then, the objective is to diminish the error function by reducing its gradient, thereby facilitating a ‘descent’ along the gradient — “gradient descent”.

Let’s start from the output layer and work backwards.

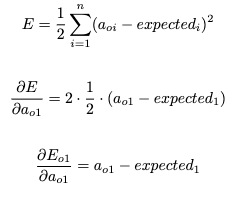
**Output Layer**

Applying the chain rule to get the formula to calculate the change in error function with respect to a small change in weight *w5*

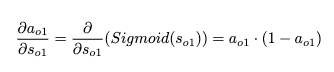


Let’s work out what each component maps to.

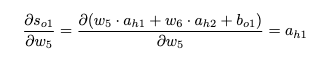
First — We’ve got the error function and its derivative with respect to *ao1*



Second — the derivative of the activation over the weighted sum, aka the derivative of sigmoid function



Lastly — the derivative of the weighted sum with respect to *w5*gives you *ah1* which is the output of the *h1* neuron in the previous layer



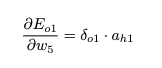
Putting them together



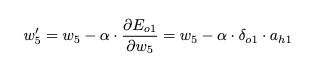
Usually, we can define a delta as



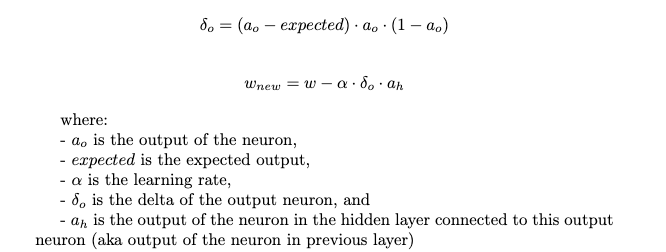
Then the formula can be shortened to



This is the gradient of the error function — applying gradient descent to get a new value of weight *w5* by reducing the weight it by the learning rate times gradient



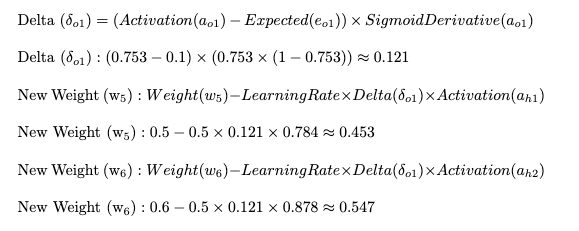
Let’s generalise the formulas for the delta of an output neuron, and the formula to update the weight in an output layer.



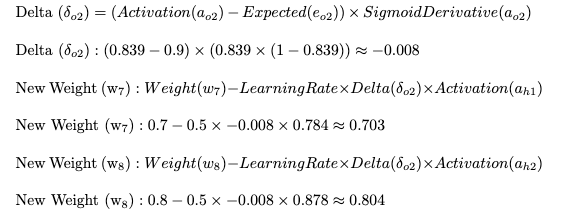
From this exercise, you should be able to then derive the formula for updating bias on your own — it’s very similar to updating weights. Hint: the final result doesn’t involve previous layer neuron’s output.

Now let’s apply real numbers from the example to those equations to calculate new weights *w5*, *w6, w7, w8*

Output Neuron 1:

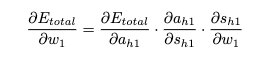


Output Neuron 2:



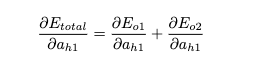
**Hidden Layer**

Applying the chain rule again to get the formula to calculate the change in error function with respect to a small change in weight *w1*

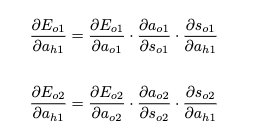


This formula will be a little bit more complicated as we’re further away from the output, so a lot more “chaining” of functions will happen so take your time to go through this.

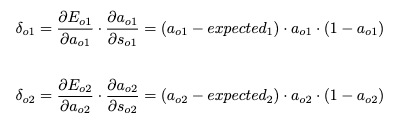
Looking at the first derivative — derivative of total errors with respect to *ah1.*Because total error equals sum of *Eo1* and *Eo2*, using the sum rule we’ve got



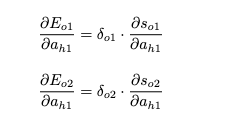
Applying the chain rule to each element



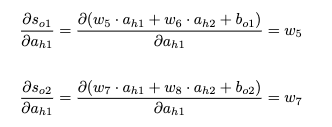
Since we calculated Delta(Eo1) and Delta (Eo2) previously



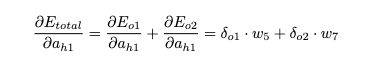
We can substitute those in



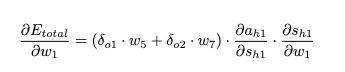
Derivative of the weighted sum with respect to previous layer neuron‘s output is basically just the corresponding weight



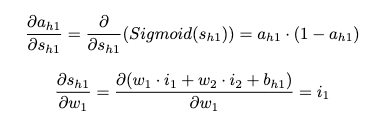
The derivative of total error with respect to weight *ah1* now looks like



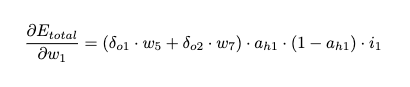
Substituting this back to the initial formula



Derivative of *ah1* over *sh1* is the derivative of the sigmoid function, and the derivative of *sh1* over *w1* is the output of the previous layer neuron (which is the input layer neuron as we only have 1 hidden layer in this example)



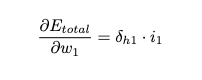
Putting it together



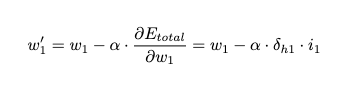
Let’s group the weighted sum of the deltas in the next layer (output layer) with the sigmoid derivative and call it the Delta(*h1*)



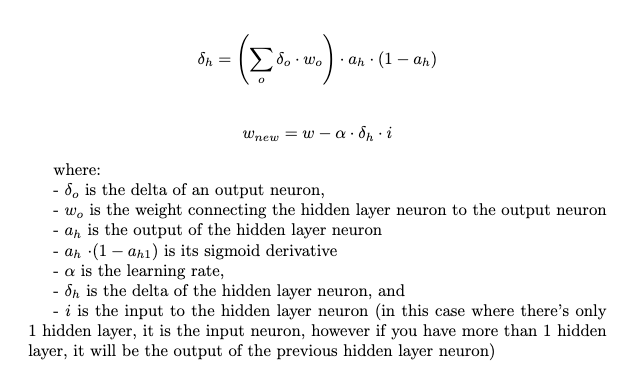
Rewrite the formula of the gradient of error function with respect to *w1*



Applying gradient descent and updating *w1* with learning rate *alpha*

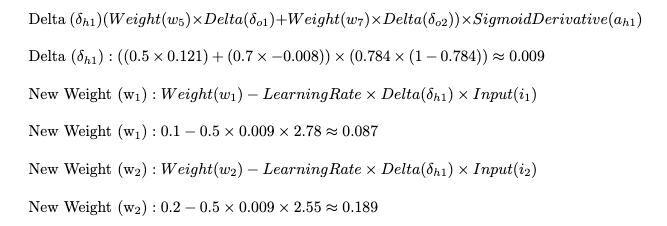


Let’s generalise the formulas for the delta of a neuron in a hidden layer and the formula to update the weight in hidden layer

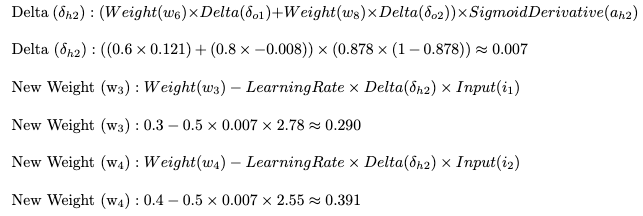


Now let’s apply real numbers from the example to those equations to calculate new weights *w1*, *w2, w3, w4*

Hidden Neuron 1:



Hidden Neuron 2:



That’s it! All of our weights have been updated — and that was just 1 iteration (epoch). Imagine if we run it thousands or millions of times, the error will become smaller and smaller, hence increasing the accuracy of the network’s prediction.

There were a lot of math formulas and calculations and variables so errors are quite likely to occur. If you notice something that is incorrect, please let me know!

**II. Level 1: Building A Neural Network Without Using External Libraries**

Now that we’ve covered the math, let’s dive into the first level of building a neural network: Without using external libraries (like numpy or PyTorch or Tensorflow)

First, let’s define the 2 functions for the sigmoid activation function and its derivative. These 2 will be reused throughout the exercise.

Now let’s build a class for our Neuron:

* Weights (weights): Neurons receive input signals, each associated with a weight. These weights determine the importance of each input.
* Bias (bias): Similar to the intercept in a linear equation, the bias allows the neuron to adjust its output independently of the input.
* Delta (delta): This is used during the backpropagation process for adjusting weights (you see the knowledge from the walkthrough we’ve done earlier is coming into our code). It represents the error derivative with respect to the weighted sum.
* Output (output): The result of the neuron's activation function.

The sigmoid function introduces non-linearity to the model, enabling it to learn complex patterns.

Neurons are then organised into layers — here’s the Layer class. Layers organise neurons into meaningful groups. Neurons in the same layer share the same input and output dimensions.

Now, the bulk of the logic is in the Network class. It represents the neural network itself and orchestrates its training and prediction processes.

Key properties:

* Hidden Layers (hidden\_layers): A list containing hidden layers, each represented by the Layer class.
* Output Layer (output\_layer): The output layer of the network, also represented by the Layer class.
* Learning Rate (learning\_rate): A hyperparameter determining the step size at each iteration during the training process.

Key functions:

* The feed\_forward method conducts the forward pass, activating each neuron in sequence, starting from receiving the inputs and progressing through hidden layers to the output layer.
* The back\_propagate method performs the backpropagation algorithm, calculating and updating the deltas of neurons in each layer. Then it calls update\_weights\_for\_all\_layers to update the weights after delta calculation is done.
* The train method trains the neural network for a specified number of epochs using the provided training set and expected outputs. The expected list uses one-hot encoding to indicate the expected output.

Now it’s time to try to run this code on some sample data. I’ve reused the data from this [tutorial](https://machinelearningmastery.com/implement-backpropagation-algorithm-scratch-python/).

This function creates a sample dataset and initialise the network with 1 hidden layer (with 2 neurons) and 1 output layer (with 2 neurons). Then, training is run for 40 epochs with learning rate of 0.5.

The neurons’ weights are randomised initially and updated as training goes on.

If you run the code, you should get something similar to this

That concludes level 1 — building a neural network without using external libraries. As you can see, most of the math formula we derived from the initial walkthrough is used extensively in the code, so it really helps to do all of the calculation manually before you start implementing the code.

Now, the code is obviously quite lengthy and somewhat complex — let’s try to simplify that by using numpy!

**III. Level 2: Building A Neural Network With Numpy**

Since you are now familiar with the flow of the network, I’ll give you all of the code at once:

Using numpy helps us shorten the code a little bit — you can imagine that it’s doing “bulk” calculation by utilising matrices instead of looping one neuron and one layer at a time as our previous implementation.

However, you’d need to have a pretty good mental model of the dimensions of the matrices in each step in order to understand and write the correct calculation, which can be a bit challenging. I’ve put comments in the code about the dimensions expected for most of the calculations (based on the test case).

The Layer class is a data class that encapsulates the parameters and attributes associated with a layer in the neural network. We don’t need a Neuron class anymore since it will just be an element in the numpy array/matrix.

I’ve added a static method createwhich creates a network with random weights and biases based on the specified number of neurons in each layer. The rest of the functions are the same, except the calculation is done with matrix multiplications instead of manually multiplying each neuron’s data.

There are obviously many ways of implementing this — one might simplify this further and remove the Layerclass completely and represents the whole network with nested arrays. However I find that approach a bit hard to wrap my head around with all of the multiple dimensions so I went with this approach for now.

Let’s try running the code with the same dataset

The dimensions of input and expected output are obviously slightly changed to fit with numpy n-D arrays, but the data stay the same.

This is a sample output

**III. Level 3: Building A Neural Network With Tensorflow**

In this level, we’ve transitioned from a detailed, 200-line implementation of a neural network to just a few concise lines using TensorFlow. The power of TensorFlow allows us to express neural network architectures with ease.

However, I won’t go into details about this code, and Tensorflow in general as our aim in this blog is not to delve into the intricacies of TensorFlow but to comprehend the fundamental workings of a neural network. TensorFlow abstracts away many of the underlying details, making it an efficient tool for practical applications but potentially not the best way to learn.

This is merely to demonstrate that neural network is complex and to fully understand it, it’s recommended to attempt to build one from scratch. Starting from the basics lays a solid foundation, enabling a deeper understanding of the complexities involved. While libraries like TensorFlow offer convenience, diving into their usage without a fundamental understanding of neural networks can hinder comprehensive learning.

Sample output

In this blog journey, we took a dive into the behind the scene of neural networks, starting from the basic walkthrough with math calculation and then moving into code implementation with Python. We built these networks step by step, first with plain Python, then with the help of a handy library called numpy, and finally, we peeked into the powerful realm of TensorFlow.

As we wrap up, I invite you to try this out for yourself. Coding is a journey of discovery, and building a neural network from scratch is like a backstage tour. So, grab your coding gear, start tinkering, and enjoy the adventure of learning. Happy coding!

References and resources:

* [Gradient descent, how neural networks learn](https://www.youtube.com/watch?v=IHZwWFHWa-w)
* [What is backpropagation really doing?](https://www.youtube.com/watch?v=Ilg3gGewQ5U&t=33s)
* [The essence of calculus](https://www.youtube.com/watch?v=WUvTyaaNkzM&list=PLZHQObOWTQDN52m7Y21ePrTbvXkPaWVSg&index=10)
* [Backpropagation calculus](https://www.youtube.com/watch?v=tIeHLnjs5U8)
* [A Step by Step Backpropagation Example](https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/)
* [How to Code a Neural Network with Backpropagation In Python (from scratch)](https://machinelearningmastery.com/implement-backpropagation-algorithm-scratch-python/)
* [Difference between numpy dot() and Python 3.5+ matrix multiplication](https://stackoverflow.com/a/34142617)
* [CHAPTER 2 — How the backpropagation algorithm works](http://neuralnetworksanddeeplearning.com/chap2.html)