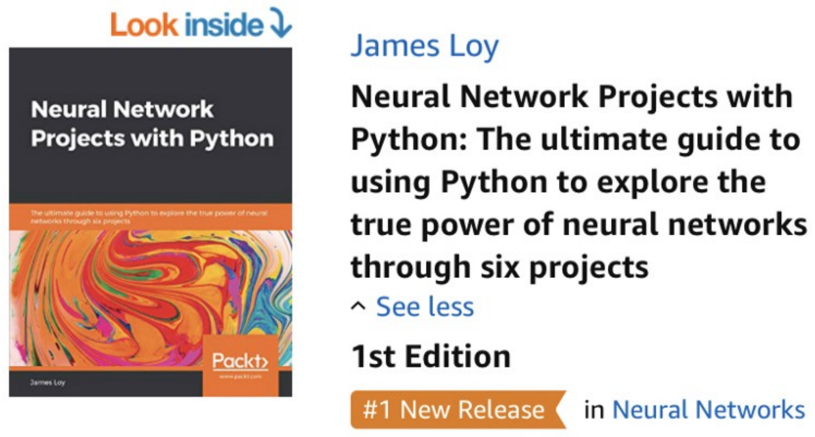
The book is a continuation of this article, and it covers end-to-end implementation of neural network projects in areas such as face recognition, sentiment analysis, noise removal etc. Every chapter features a unique neural network architecture, including Convolutional Neural Networks, Long Short-Term Memory Nets and Siamese Neural Networks. If you're looking to create a strong machine learning portfolio with deep learning projects, do consider getting the book!

You can get the book from Amazon: **Neural Network Projects with Python**



James Loy

Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects

^ See less

1st Edition

#1 New Release in Neural Networks

As always, feel free to reach out to me on LinkedIn!

. . .

Motivation: As part of my personal journey to gain a better understanding of Deep Learning, I've decided to build a Neural Network from scratch without a deep learning library like TensorFlow. I believe that understanding the inner workings of a Neural Network is important to any aspiring Data Scientist.

This article contains what I've learned, and hopefully it'll be useful for you as well!

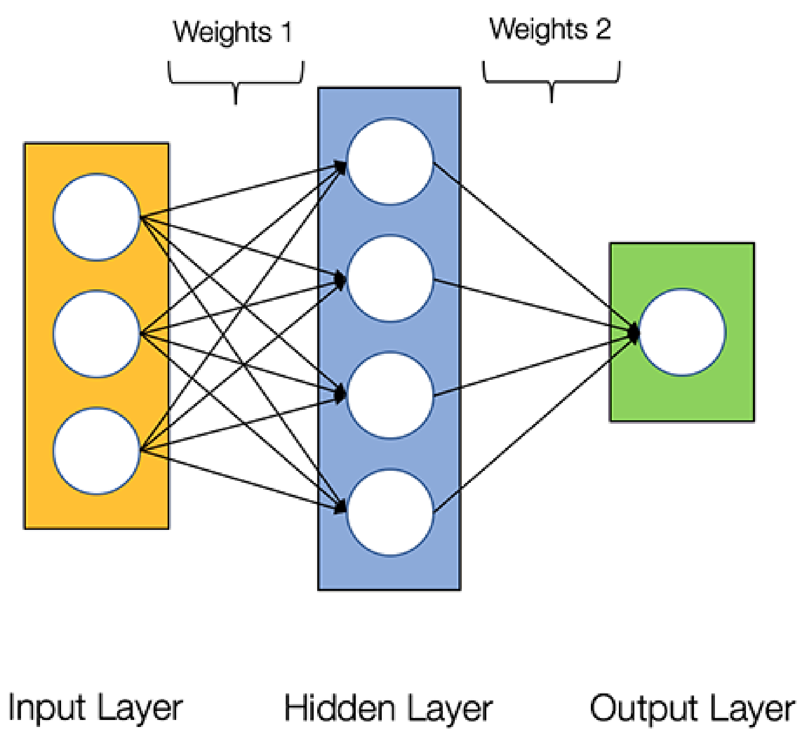
What's a Neural Network?

Most introductory texts to Neural Networks brings up brain analogies when describing them. Without delving into brain analogies, I find it easier to simply describe Neural Networks as a mathematical function that maps a given input to a desired output.

Neural Networks consist of the following components

- An **input layer**, $\boldsymbol{x}$
- An arbitrary amount of **hidden layers**
- An **output layer**, $\hat{y}$
- A set of **weights** and **biases** between each layer, ***W and b***
- A choice of **activation function** for each hidden layer, $\boldsymbol{\sigma}$. In this tutorial, we'll use a Sigmoid activation function.

The diagram below shows the architecture of a 2-layer Neural Network (*note that the input layer is typically excluded when counting the number of layers in a Neural Network*)



Architectu re of a 2-layer Neu ral Network

Creating a Neural Network class in Python is easy.

```
class NeuralNetwork:
    def __init__(self, x, y):
        self.input      = x
        self.weights1   = np.random.rand(self.input.shape
        self.weights2   = np.random.rand(4,1)
```

Training the Neural Network

The output $\hat{y}$ of a simple 2-layer Neural Network is:

$$\hat{y} = \sigma(W_2\sigma(W_1x + b_1) + b_2)$$

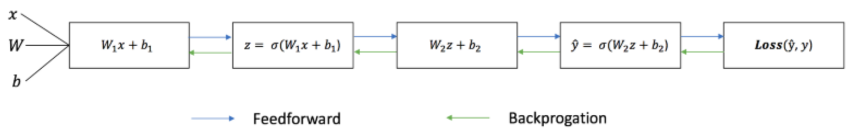
You might notice that in the equation above, the weights W and the biases b are the only variables that affects the output $\hat{y}$.

Naturally, the right values for the weights and biases determines the strength of the predictions. The process of fine-tuning the weights and biases from the input data is known as **training the Neural Network.**

Each iteration of the training process consists of the following steps:

- Calculating the predicted output $\hat{y}$, known as **feedforward**
- Updating the weights and biases, known as **backpropagation**

The sequential graph below illustrates the process.



Feedforward

As we've seen in the sequential graph above, feedforward is just simple calculus and for a basic 2-layer neural network, the output of the Neural Network is:

$$\hat{y} = \sigma(W_2\sigma(W_1x + b_1) + b_2)$$

Let’s add a feedforward function in our python code to do exactly that. Note that for simplicity, we have assumed the biases to be 0.

```python
class NeuralNetwork:
    def __init__(self, x, y):
        self.input      = x
        self.weights1   = np.random.rand(self.input.shap
        self.weights2   = np.random.rand(4,1)
        self.y          = y
        self.output     = np.zeros(self.y.shape)

```

However, we still need a way to evaluate the “goodness” of our predictions (i.e. how far off are our predictions)? The **Loss Function** allows us to do exactly that.

Loss Function

There are many available loss functions, and the nature of our problem should dictate our choice of loss function. In this tutorial, we’ll use a simple **sum-of-sqaures error** as our loss function.

$$Sum - of - Squares\ Error = \sum_{i=1}^{n} (y - \hat{y})^2$$

That is, the sum-of-squares error is simply the sum of the difference between each predicted value and the actual value. The difference is squared so that we measure the absolute value of the difference.

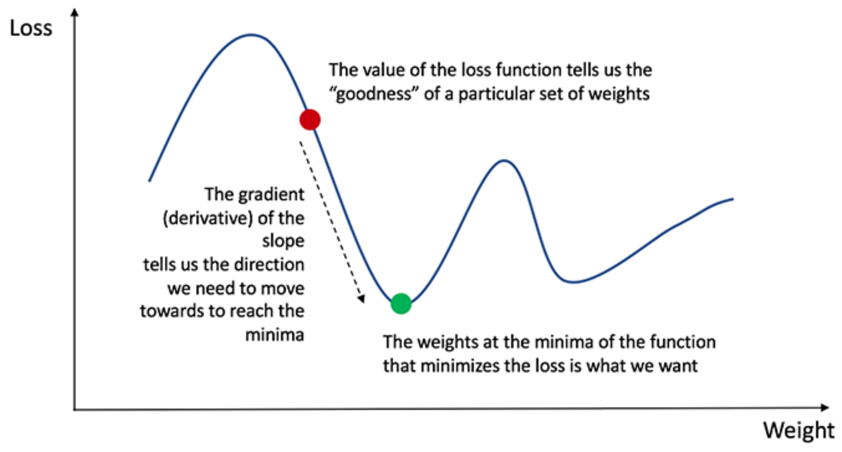
Our goal in training is to find the best set of weights and biases that minimizes the loss function.

Backpropagation

Now that we’ve measured the error of our prediction (loss), we need to find a way to **propagate** the error back, and to update our weights and biases.

In order to know the appropriate amount to adjust the weights and biases by, we need to know the **derivative of the loss function with respect to the weights and biases**.

Recall from calculus that the derivative of a function is simply the slope of the function.


Loss
The value of the loss function tells us the "goodness" of a particular set of weights
The gradient (derivative) of the slope tells us the direction we need to move towards to reach the minima
The weights at the minima of the function that minimizes the loss is what we want
Weight

Gradient descent algorithm

If we have the derivative, we can simply update the weights and biases by increasing/reducing with it(refer to the diagram above). This is known as **gradient descent**.

However, we can't directly calculate the derivative of the loss function with respect to the weights and biases because the equation of the loss function does not contain the weights and biases. Therefore, we need the **chain rule** to help us calculate it.

$$Loss(y,\hat{y}) = \sum_{i=1}^{n}(y - \hat{y})^2$$

$$\frac{\partial\, Loss(y,\hat{y})}{\partial W} = \frac{\partial Loss(y,\hat{y})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z} * \frac{\partial z}{\partial W} \qquad \text{where } z = Wx + b$$

$$= 2(y - \hat{y}) * \text{derivative of sigmoid function} * x$$

$$= 2(y - \hat{y}) * \text{z(1-z)} * x$$

Chain ru le for calcu lating derivative of the loss fu nction with respect to the weights. Note that for simplicity, we have only displayed the partial derivative assu ming a 1-layer Neu ral Network.

Phew! That was ugly but it allows us to get what we needed—the derivative (slope) of the loss function with respect to the weights, so that we can adjust the weights accordingly.

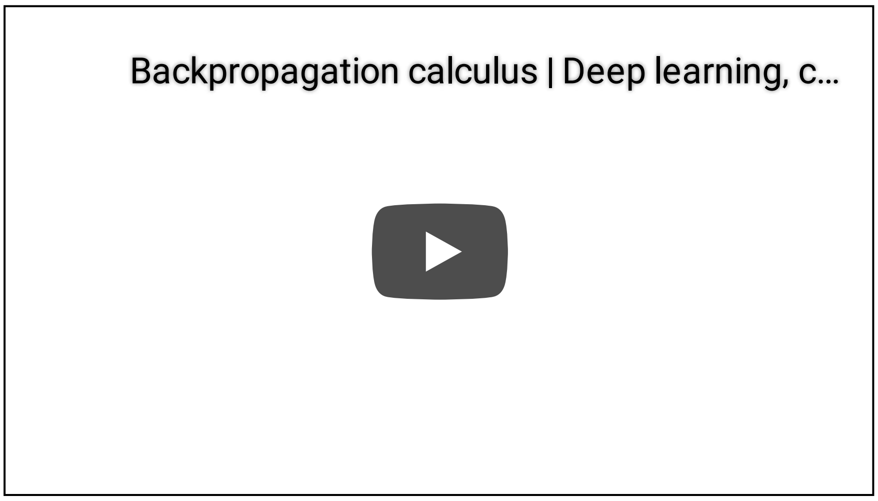
Now that we have that, let’s add the backpropagation function into our python code.

```python
class NeuralNetwork:
    def __init__(self, x, y):
        self.input      = x
        self.weights1   = np.random.rand(self.input.shap
        self.weights2   = np.random.rand(4,1)
        self.y          = y
        self.output     = np.zeros(self.y.shape)

    def feedforward(self):
        self.layer1 = sigmoid(np.dot(self.input, self.we
        self.output = sigmoid(np.dot(self.layer1, self.w

    def backprop(self):
        # application of the chain rule to find derivati
```

For a deeper understanding of the application of calculus and the chain rule in backpropagation, I strongly recommend this tutorial by 3Blue1Brown.


Backpropagation calculus | Deep learning, c...

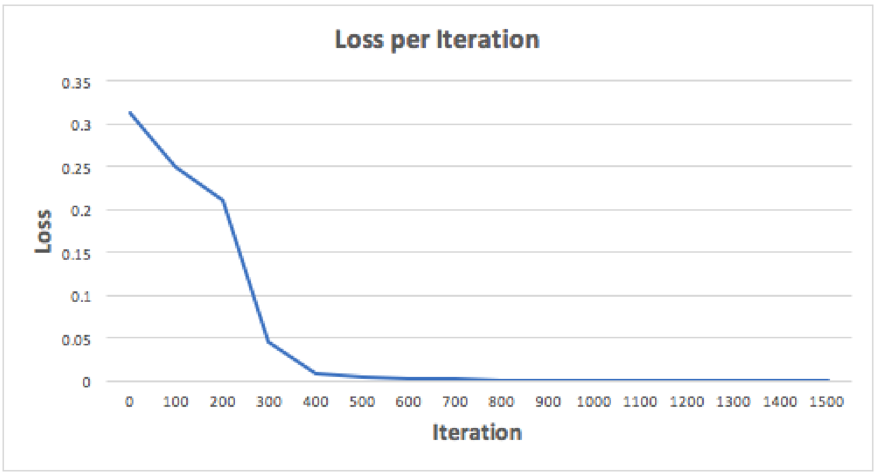
Putting it all together

Now that we have our complete python code for doing feedforward and backpropagation, let’s apply our Neural Network on an example and see how well it does.

X1	X2	X3	Y
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	0

Our Neural Network should learn the ideal set of weights to represent this function. Note that it isn't exactly trivial for us to work out the weights just by inspection alone.

Let's train the Neural Network for 1500 iterations and see what happens. Looking at the loss per iteration graph below, we can clearly see the loss **monotonically decreasing towards a minimum.** This is consistent with the gradient descent algorithm that we've discussed earlier.



Let's look at the final prediction (output) from the Neural Network after 1500 iterations.

Prediction	Y (Actual)
0.023	0
0.979	1
0.975	1
0.025	0

Predictions after 1500 training iterations