PYIMAGESEARCH

DEEP LEARNING (HTTPS://PYIMAGESEARCH.COM/CATEGORY/DEEP-LEARNING/)

TUTORIALS (HTTPS://PYIMAGESEARCH.COM/CATEGORY/TUTORIALS/)

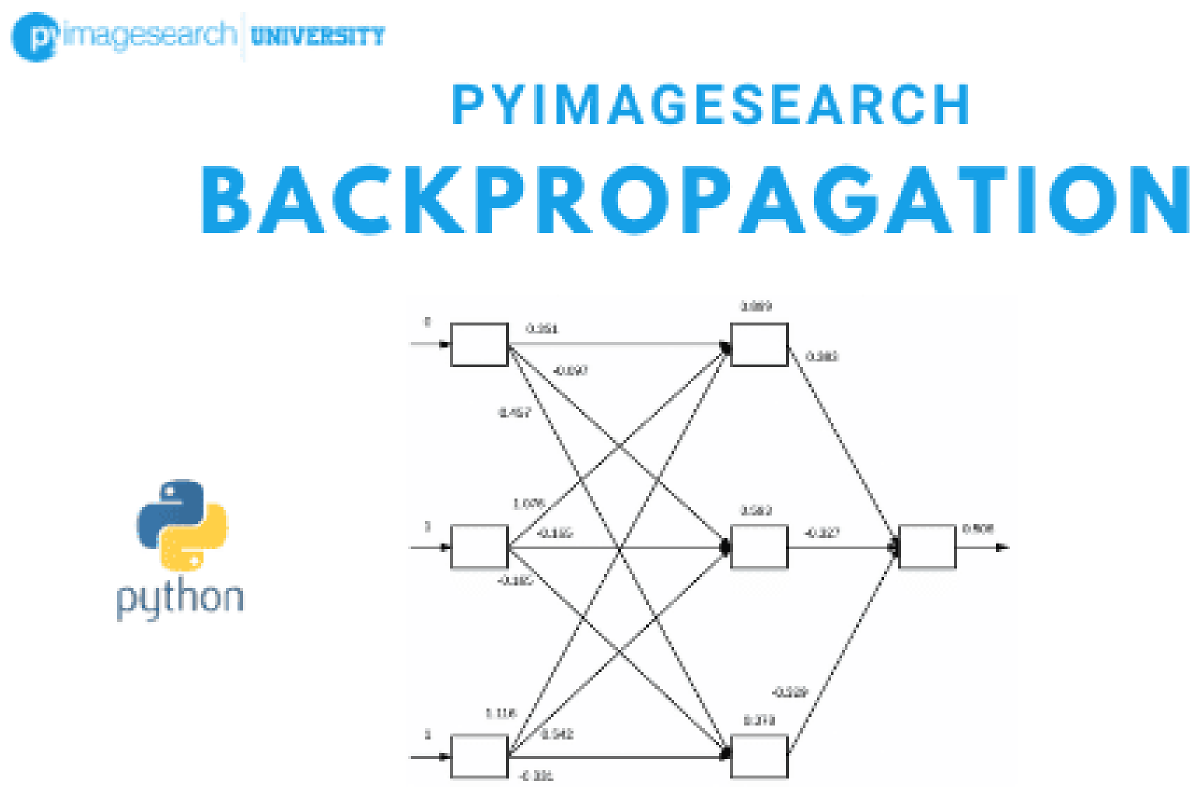
Backpropagation from scratch with Python

by **Adrian Rosebrock (https://pyimagesearch.com/author/adrian/)** *on* May 6, 2021

Click here to download the source code to this post

6:57

Backpropagation is arguably *the* most important algorithm in neural network history — without (efficient) backpropagation, it would be *impossible* to train deep learning networks to the depths that we see today. Backpropagation can be considered the cornerstone of modern neural networks and deep learning cornerstone of modern neural networks and deep learning.



The original incarnation of backpropagation was introduced back in the 1970s, but it wasn't until the seminal 1988 paper, *Learning representations by back-propagating errors* by **Rumelhart, Hinton, and Williams (https://dl.acm.org/doi/10.5555/65669.104451)**, were we able to devise a faster algorithm, more adept to training deeper networks.

There are quite literally hundreds (if not thousands) of tutorials on backpropagation available today. Some of my favorites include:

1. **Andrew Ng's (https://www.coursera.org/learn/machine-learning)** discussion on backpropagation inside the Machine Learning course by Coursera.

2. The heavily mathematically motivated Chapter 2 — *How the backpropagation algorithm works* from *Neural Networks and Deep Learning* by **Michael Nielsen (http://neuralnetworksanddeeplearning.com/chap2.html)**.

3. **Stanford's cs231n (http://cs231n.stanford.edu/)** exploration and analysis of backpropagation.

4. **Matt Mazur's (https://mattmazur.com/2015/%2003/17/a-step-by-step-backpropagation-example/)** excellent concrete example (with actual worked numbers) that demonstrates how backpropagation works.

As you can see, there are no shortage of backpropagation guides — instead of regurgitating and reiterating what has been said by others hundreds of times before, I'm going to take a different approach and do what makes PyImageSearch publications special:

Construct an intuitive, easy to follow implementation of the backpropagation algorithm using the Python language.

Inside this implementation, we'll build an actual neural network and train it using the back propagation algorithm. By the time you finish this section, you'll understand how backpropagation works — and perhaps more importantly, you'll have a stronger understanding of how this algorithm is used to train neural networks from scratch.

Looking for the source code to this post?

JUMP RIGHT TO THE DOWNLOADS SECTION →

Backpropagation

The backpropagation algorithm consists of two phases:

1. The *forward pass* where our inputs are passed through the network and output predictions obtained (also known as the *propagation* phase).
2. The *backward pass* where we compute the gradient of the loss function at the final layer (i.e., predictions layer) of the network and use this gradient to recursively apply the chain rule to update the weights in our network (also known as the *weight update* phase).

We'll start by reviewing each of these phases at a high level. From there, we'll implement the backpropagation algorithm using Python. Once we have implemented backpropagation we'll want to be able to make *predictions* using our network — this is simply the forward pass phase, only with a small adjustment (in terms of code) to make the predictions more efficient.

Finally, I'll demonstrate how to train a custom neural network using backpropagation and Python on both the:

1. XOR dataset
2. MNIST dataset

The Forward Pass

The purpose of the forward pass is to propagate our inputs through the network by applying a series of dot products and activations until we reach the output layer of the network (i.e., our predictions). To visualize this process, let's first consider the XOR dataset (**Table 1**, *left*).

x_0	x_1	y	x_0	x_1	x_2
0	0	0	0	0	1
0	1	1	0	1	1
1	0	1	1	0	1
1	1	0	1	1	1

Table 1: *Left:* The bitwise XOR dataset (including class labels). *Right:* The XOR dataset design matrix with a bias column i nserted (excluding class labels for brevity).

Here, we can see that each entry *X* in the design matrix (*left*) is 2-dim — each data point is represented by *two* numbers. For example, the first data point is represented by the feature vector *(0, 0)*, the second data point by *(0, 1)*, etc. We then have our output values *y* as the right column. Our target output values are the *class labels*. Given an input from the design matrix, our goal is to correctly predict the target output value.

To obtain perfect classification accuracy on this problem we'll need a feedforward neural network with at least a single hidden layer, so let's go ahead and start with a *2−2−1* architecture (**Figure 1**, *top*). This is a good start; however, we're forgetting to

include the bias term. There are two ways to include the bias term ***b*** in our network. We can either:

1. Use a separate variable.

2 Treat the bias as a trainable parameter *within* the weight matrix by inserting a column of 1’s into the feature vectors.

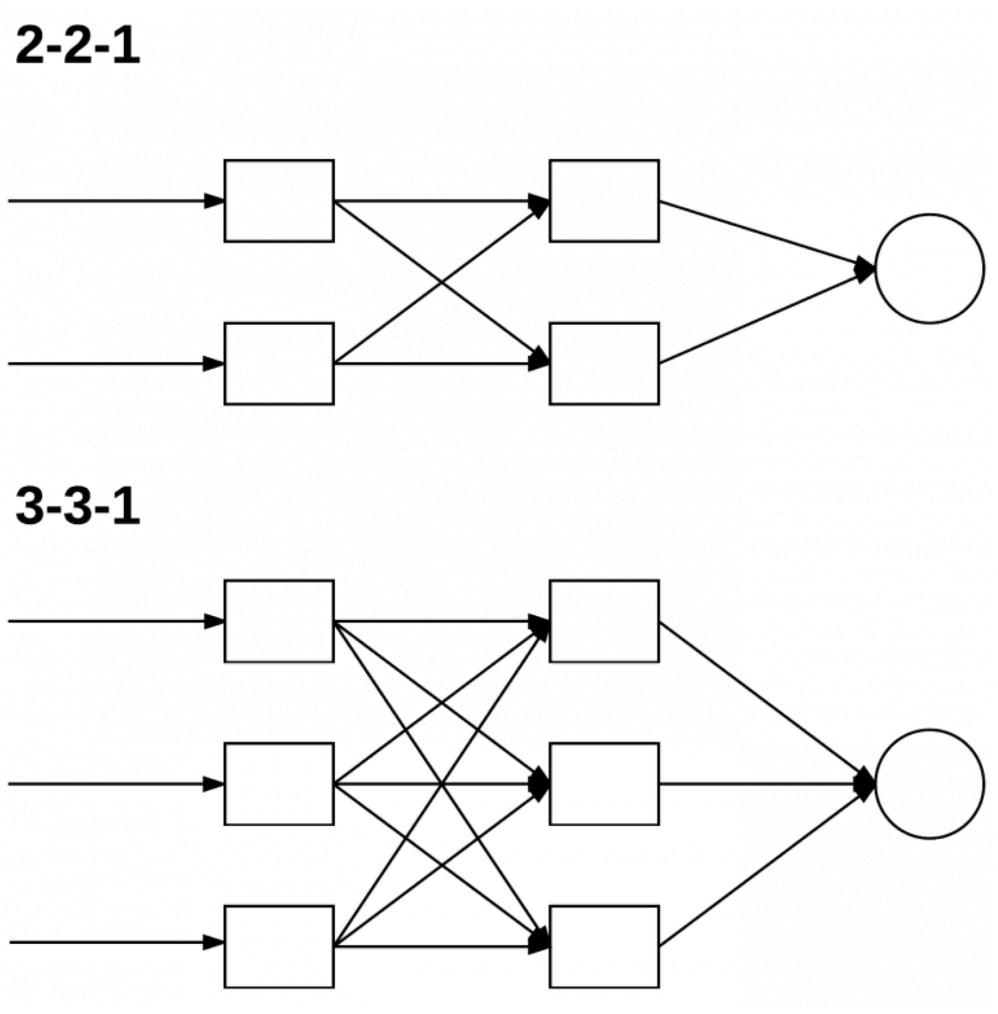


Figure 1: *Top:* To build a neural network to correctly classify the XOR dataset, we’ll need a network with two input nodes, two hidden nodes, and one output node. This gives rise to a *2−2−1* architecture. *Bottom:* Our actual internal network architecture representation is *3−3−1* due to the bias trick. In the vast majority of neural network implementations this adjustment to the weight matrix happens internally and is something that you do not need to worry about; however, it’s still important to understand what is going on under the hood.

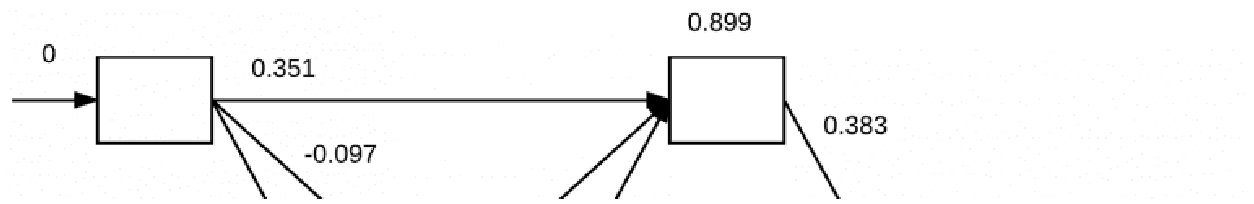
Inserting a column of 1’s into our feature vector is done programmatically, but to ensure we understand this point, let’s update our XOR design matrix to explicitly see this taking place (**Table 1**, *right*). As you can see, a column of 1’s have been added to our feature vectors. In practice you can insert this column anywhere you like, but we typically place vectors. In practice you can insert this column anywhere you like, but we typically place it either as (1) the first entry in the feature vector or (2) the last entry in the feature vector.

Since we have changed the size of our input feature vector (normally performed *inside* neural network implementation itself so that we do not need to explicitly modify our design matrix), that changes our (perceived) network architecture from *2−2−1* to an (internal) *3−3−1* (**Figure 1**, *bottom*).

We'll still refer to this network architecture as *2−2−1*, but when it comes to implementation, it's actually *3−3−1* due to the addition of the bias term embedded in the weight matrix.

Finally, recall that both our input layer and all hidden layers require a bias term; however, the final output layer *does not* require a bias. The benefit of applying the bias trick is that we do not need to explicitly keep track of the bias parameter any longer — it is now a trainable parameter *within* the weight matrix, thus making training more efficient and substantially easier to implement.

To see the forward pass in action, we first initialize the weights in our network, as in **Figure 2**. Notice how each arrow in the weight matrix has a value associated with it — this is the *current* weight value for a given node and signifies the amount in which a given input is amplified or diminished. This weight value will then be *updated* during the backpropagation phase.



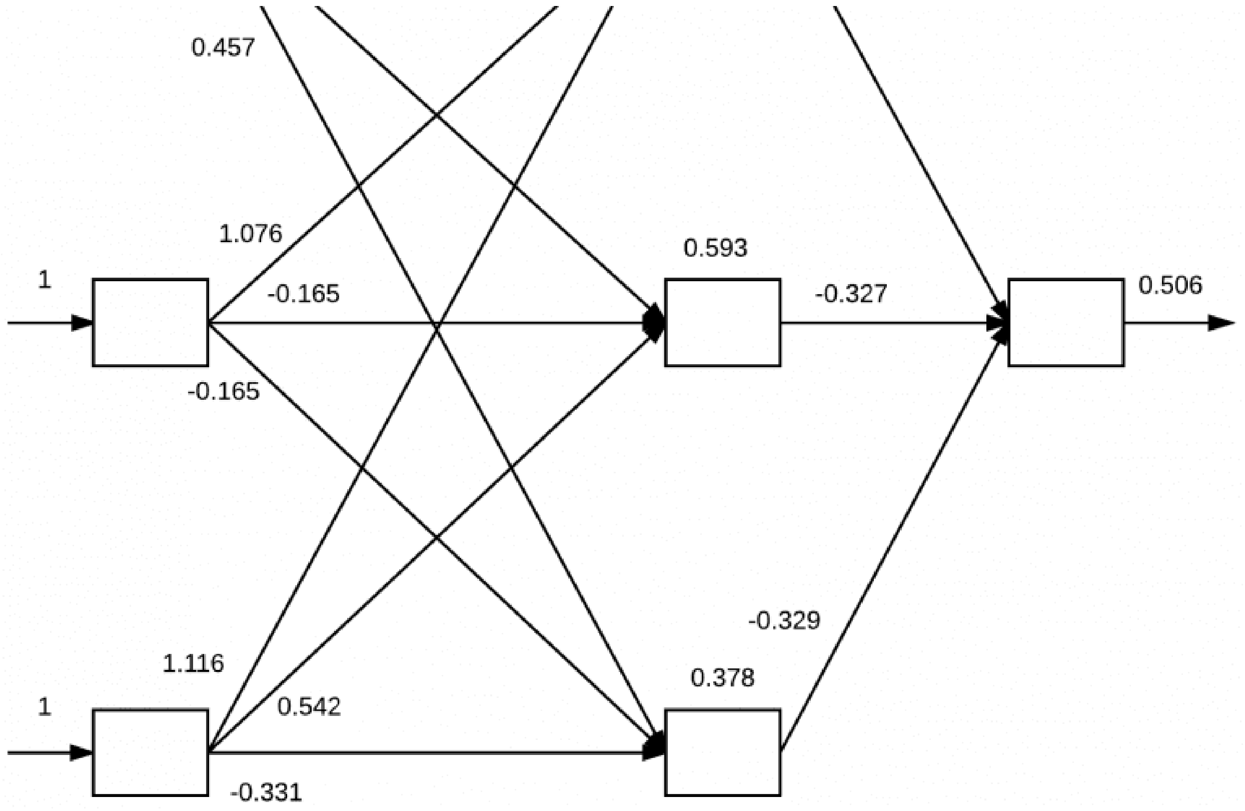


Figure 2: An example of the forward propagation pass. The input vector *[0,1,1]* is presented to the network. The dot product between the inputs and weights are taken, followed by applying the sigmoid activation function to obtain the values in the hidden layer (*0.899*, *0.593*, and *0.378*, respectively). Finally, the dot product and sigmoid activation function is computed for the final layer, yielding an output of *0.506*. Applying the step function to *0.506* yields *1*, which is indeed the correct target class label.

On the *far left* of **Figure 2**, we present the feature vector *(0, 1, 1)* (and target output value 1 to the network). Here we can see that *0*, *1*, and *1* have been assigned to the three input nodes in the network. To propagate the values through the network and obtain the final classification, we need to take the dot product between the inputs and the weight values, followed by applying an activation function (in this case, the *sigmoid* function, *σ*).

Let's compute the inputs to the three nodes in the hidden layers:

1 $\sigma((0\times0.351) + (1\times1.076) + (1\times1.116)) = 0.899$

2 $\sigma((0\times -0.097) + (1\times -0.165) + (1\times0.542)) = 0.593$

3 $\sigma((0\times0.457) + (1\times -0.165) + (1\times -0.331)) = 0.378$

Looking at the node values of the hidden layers (**Figure 2**, *middle*), we can see the nodes have been updated to reflect our computation.

We now have our *inputs* to the hidden layer nodes. To compute the output prediction, we once again compute the dot product followed by a sigmoid activation:

(1) $\sigma((0.899 \times 0.383) + (0.593 \times -0.327) + (0.378 \times -0.329)) = 0.506$

The output of the network is thus *0.506*. We can apply a step function to determine if this output is the correct classification or not:

$$f(net) = \begin{cases} 1 & if\ net > 0 \\ 0 & otherwise \end{cases}$$

Applying the step function with *net = 0.506* we see that our network predicts *1* which is, in fact, the correct class label. However, our network is *not very confident* in this class label — the predicted value *0.506* is very close to the threshold of the step. Ideally, this prediction should be closer to *0.98−0.99*, implying that our network has truly learned the underlying pattern in the dataset. In order for our network to actually "learn," we need to apply the backward pass.

The Backward Pass

To apply the backpropagation algorithm, our activation function must be *differentiable* so that we can compute the *partial derivative* of the error with respect to a given weight $w_{i,j}$, loss (E), node output o_j, and network output net_j.

(2) $\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{i,j}}$

As the calculus behind backpropagation has been exhaustively explained many times in previous works (see **Andrew Ng (https://www.coursera.org/learn/machine-learning)**, **Michael Nielsen (http://neuralnetworksanddeeplearning.com/chap2.html)**, and **Matt**

Mazur (https://mattmazur.com/2015/ 03/17/a-step-by-step-backpropagation-example/)), I'm going to skip the derivation of the backpropagation chain rule update and instead explain it via code in the following section.

For the mathematically astute, please see the references above for more information on the chain rule and its role in the backpropagation algorithm. By explaining this process in code, my goal is to help readers understand backpropagation through a more intuitive, implementation sense.

Implementing Backpropagation with Python

Let's go ahead and get started implementing backpropagation. Open a new file, name it `neuralnetwork.py` , store it in the `nn` submodule of `pyimagesearch` (like we did with `perceptron.py`), and let's get to work:

→ **Launch Jupyter Notebook on Google Colab**

Backpropagation from scratch with Python

```
 1. | # import the necessary packages
 2. | import numpy as np
 3. |
 4. | class NeuralNetwork:
 5. |    def __init__(self, layers, alpha=0.1):
 6. |        # initialize the list of weights matrices, then store the
 7. |        # network architecture and learning rate
 8. |        self.W = []
 9. |        self.layers = layers
10. |        self.alpha = alpha
```

On **Line 2,** we import the only required package we'll need for our implementation of back propagation — the NumPy numerical processing library.

Line 5 then defines the constructor to our `NeuralNetwork` class. The constructor requires a single argument, followed by a second optional one:

- `layers` : A list of integers which represents the actual *architecture* of the feedforward network. For example, a value of *[2, 2, 1]* would imply that our first input layer has two nodes, our hidden layer has two nodes, and our final output layer has one node.

- `alpha` : Here we can specify the learning rate of our neural network. This value is applied during the weight update phase.

Line 8 initializes our list of weights for each layer, `W` . We then store `layers` and `alpha` on **Lines 9 and 10**.

Our weights list `W` is empty, so let's go ahead and initialize it now:

→ **Launch Jupyter Notebook on Google Colab**

```
Backpropagation from scratch with Python
12.         # start looping from the index of the first layer but
13.         # stop before we reach the last two layers
14.         for i in np.arange(0, len(layers) - 2):
15.             # randomly initialize a weight matrix connecting the
16.             # number of nodes in each respective layer together,
17.             # adding an extra node for the bias
18.             w = np.random.randn(layers[i] + 1, layers[i + 1] + 1)
19.             self.W.append(w / np.sqrt(layers[i]))
```

On **Line 14,** we start looping over the number of layers in the network (i.e., `len(layers)`), but we stop before the final two layers (we'll find out exactly why later in the explanation of this constructor).

Each layer in the network is randomly initialized by constructing an *M×N* weight matrix by sampling values from a standard, normal distribution (**Line 18**). The matrix is *M×N* since we wish to connect every node in the *current layer* to every node in the *next layer*.

For example, let's suppose that `layers[i] = 2` and `layers[i + 1] = 2` . Our weight matrix would, therefore, be *2×2* to connect all sets of nodes between the layers. However, we need to be careful here, as we are forgetting an important component — *the bias term*. To account for the bias, we add one to the number of `layers[i]` and *the bias term*. To account for the bias, we add one to the number of `layers[i]` and `layers[i + 1]` — doing so changes our weight matrix `w` to have the shape *3×3* given *2+1* nodes for the current layer and *2+1* nodes for the next layer. We scale `w` by dividing by the square root of the number of nodes in the current layer, thereby normalizing the variance of each neuron's output (**http://cs231n.stanford.edu/ (http://cs231n.stanford.edu/)**) (**Line 19**).

The final code block of the constructor handles the special case where the input connections need a bias term, but the output does not:

→ **Launch Jupyter Notebook on Google Colab**

```
Backpropagation from scratch with Python
        # the last two layers are a special case where the input
        # connections need a bias term but the output does not
        w = np.random.randn(layers[-2] + 1, layers[-1])
        self.W.append(w / np.sqrt(layers[-2]))
```

Again, these weight values are randomly sampled and then normalized.

The next function we define is a Python "magic method" named `__repr__` — this function is useful for debugging:

→ **Launch Jupyter Notebook on Google Colab**

```
Backpropagation from scratch with Python
    def __repr__(self):
        # construct and return a string that represents the network
        # architecture
        return "NeuralNetwork: {}".format(
            "-".join(str(l) for l in self.layers))
```

In our case, we'll format a string for our `NeuralNetwork` object by concatenating the integer value of the number of nodes in each layer. Given a `layers` value of `(2, 2, 1)`, the output of calling this function will be:

→ **Launch Jupyter Notebook on Google Colab**

```
Backpropagation from scratch with Python
>>> from pyimagesearch.nn import NeuralNetwork
>>> nn = NeuralNetwork([2, 2, 1])
>>> print(nn)
NeuralNetwork: 2-2-1
```

Next, we can define our sigmoid activation function:

→ **Launch Jupyter Notebook on Google Colab**

Backpropagation from scratch with Python

```
    def sigmoid(self, x):
        # compute and return the sigmoid activation value for a
        # given input value
        return 1.0 / (1 + np.exp(-x))
```

As well as the *derivative* of the sigmoid which we'll use during the backward pass:

→ **Launch Jupyter Notebook on Google Colab**

Backpropagation from scratch with Python

```
    def sigmoid_deriv(self, x):
        # compute the derivative of the sigmoid function ASSUMING
        # that x has already been passed through the 'sigmoid'
        # function
        return x * (1 - x)
```

Again, note that whenever you perform backpropagation, you'll always want to choose an activation function that is *differentiable*.

We'll draw inspiration from the scikit-learn library and define a function named `fit` which will be responsible for actually training our `NeuralNetwork`:

→ **Launch Jupyter Notebook on Google Colab**

Backpropagation from scratch with Python

```
    def fit(self, X, y, epochs=1000, displayUpdate=100):
        # insert a column of 1's as the last entry in the feature
        # matrix -- this little trick allows us to treat the bias
        # as a trainable parameter within the weight matrix
        X = np.c_[X, np.ones((X.shape[0]))]

        # loop over the desired number of epochs
        for epoch in np.arange(0, epochs):
            # loop over each individual data point and train
            # our network on it
            for (x, target) in zip(X, y):
                self.fit_partial(x, target)

```

```
56. |             # check to see if we should display a training update
57. |             if epoch == 0 or (epoch + 1) % displayUpdate == 0:
58. |                 loss = self.calculate_loss(X, y)
59. |                 print("[INFO] epoch={}, loss={:.7f}".format(
60. |                     epoch + 1, loss))
```

The `fit` method requires two parameters, followed by two optional ones. The first, `X`, is our *training data*. The second, `y`, is the corresponding class labels for each entry in `X`. We then specify `epochs`, which is the number of epochs we'll train our network for. The `displayUpdate` parameter simply controls how many *N* epochs we'll print training progress to our terminal.

On **Line 47,** we perform the bias trick by inserting a column of 1's as the last entry in our feature matrix, `X`. From there, we start looping over our number of `epochs` on **Line 50**. For each epoch, we'll loop over each individual data point in our training set, make a prediction on the data point, compute the backpropagation phase, and then update our weight matrix (**Lines 53 and 54**). **Lines 57-60** simply check to see if we should display a training update to our terminal.

The actual heart of the backpropagation algorithm is found inside our `fit_partial` method below:

→ **Launch Jupyter Notebook on Google Colab**

```
Backpropagation from scratch with Python
62. |     def fit_partial(self, x, y):
63. |         # construct our list of output activations for each layer
64. |         # as our data point flows through the network; the first
65. |         # activation is a special case -- it's just the input
66. |         # feature vector itself
67. |         A = [np.atleast_2d(x)]
```

The `fit_partial` function requires two parameters:

- `x` : An individual data point from our design matrix.
- `y` : The corresponding class label.

We then initialize a list, `A`, on **Line 67** — this list is responsible for storing the output activations for each layer as our data point `x` forward propagates through the network activations for each layer as our data point `x` forward propagates through the network. We initialize this list with `x`, which is simply the input data point.

From here, we can start the forward propagation phase:

→ **Launch Jupyter Notebook on Google Colab**

Backpropagation from scratch with Python

```
69.         # FEEDFORWARD:
70.         # loop over the layers in the network
71.         for layer in np.arange(0, len(self.W)):
72.             # feedforward the activation at the current layer by
73.             # taking the dot product between the activation and
74.             # the weight matrix -- this is called the "net input"
75.             # to the current layer
76.             net = A[layer].dot(self.W[layer])
77. 
78.             # computing the "net output" is simply applying our
79.             # nonlinear activation function to the net input
80.             out = self.sigmoid(net)
81. 
82.             # once we have the net output, add it to our list of
83.             # activations
84.             A.append(out)
```

We start looping over every layer in the network on **Line 71**. The *net input* to the current `layer` is computed by taking the dot product between the activation and the weight matrix (**Line 76**). The *net output* of the current layer is then computed by passing the net input through the nonlinear sigmoid activation function. Once we have the net output, we add it to our list of activations (**Line 84**).

Believe it or not, this code is the *entirety* of the forward pass — we are simply looping over each of the layers in the network, taking the dot product between the activation and the weights, passing the value through a nonlinear activation function, and continuing to the next layer. The final entry in `A` is thus the output of the last layer in our network (i.e., the *prediction*).

Now that the forward pass is done, we can move on to the slightly more complicated backward pass:

→ **Launch Jupyter Notebook on Google Colab**

Backpropagation from scratch with Python

```
86.  		# BACKPROPAGATION
87.  		# the first phase of backpropagation is to compute the
88.  		# difference between our *prediction* (the final output
89.  		# activation in the activations list) and the true target
90.  		# value
91.  		error = A[-1] - y
92.  
93.  		# from here, we need to apply the chain rule and build our
94.  		# list of deltas 'D'; the first entry in the deltas is
95.  		# simply the error of the output layer times the derivative
96.  		# of our activation function for the output value
97.  		D = [error * self.sigmoid_deriv(A[-1])]
```

The first phase of the backward pass is to compute our `error`, or simply the difference between our *predicted* label and the *ground-truth* label (**Line 91**). Since the final entry in the activations list `A` contains the output of the network, we can access the output prediction via `A[-1]`. The value `y` is the target output for the input data point `x`.

Remark: When using the Python programming language, specifying an index value of `-1` indicates that we would like to access the *last* entry in the list. You can read more about Python array indexes and slices in this tutorial: **http://pyimg.co/6dfae (http://pyimg.co/6dfae)**.

Next, we need to start applying the chain rule to build our list of deltas, `D`. The deltas will be used to update our weight matrices, scaled by the learning rate `alpha`. The first entry in the deltas list is the error of our output layer multiplied by the derivative of the sigmoid for the output value (**Line 97**).

Given the delta for the final layer in the network, we can now work backward using a `for` loop:

→ **Launch Jupyter Notebook on Google Colab**

Backpropagation from scratch with Python

```
 99.  		# once you understand the chain rule it becomes super easy
100.  		# to implement with a 'for' loop -- simply loop over the
101.  		# layers in reverse order (ignoring the last two since we
102.  		# already have taken them into account)
103.  		for layer in np.arange(len(A) - 2, 0, -1):
104.  			# the delta for the current layer is equal to the delta
```

```
105. |             # of the *previous layer* dotted with the weight matrix
106. |             # of the current layer, followed by multiplying the delta
107. |             # by the derivative of the nonlinear activation function
108. |             # for the activations of the current layer
109. |             delta = D[-1].dot(self.W[layer].T)
110. |             delta = delta * self.sigmoid_deriv(A[layer])
111. |             D.append(delta)
```

On **Line 103,** we start looping over each of the layers in the network (ignoring the previous two layers as they are already accounted for in **Line 97**) in *reverse order* as we need to work *backward* to compute the delta updates for each layer. The `delta` for the current layer is equal to the delta of the previous layer, `D[-1]` dotted with the weight matrix of the current layer (**Line 109**). To finish off the computation of the `delta` , we multiply it by passing the activation for the `layer` through our derivative of the sigmoid (**Line 110**). We then update the deltas `D` list with the `delta` we just computed (**Line 111**).

Looking at this block of code we can see that the backpropagation step is iterative — we are simply taking the delta from the *previous layer*, dotting it with the weights of the *current layer*, and then multiplying by the derivative of the activation. This process is repeated until we reach the first layer in the network.

Given our deltas list `D` , we can move on to the weight update phase:

→ **Launch Jupyter Notebook on Google Colab**

Backpropagation from scratch with Python

```
113. |         # since we looped over our layers in reverse order we need to
114. |         # reverse the deltas
115. |         D = D[::-1]
116. |
117. |         # WEIGHT UPDATE PHASE
118. |         # loop over the layers
119. |         for layer in np.arange(0, len(self.W)):
120. |             # update our weights by taking the dot product of the layer
121. |             # activations with their respective deltas, then multiplying
122. |             # this value by some small learning rate and adding to our
123. |             # weight matrix -- this is where the actual "learning" takes
124. |             # place
125. |             self.W[layer] += -self.alpha * A[layer].T.dot(D[layer])
```

Keep in mind that during the backpropagation step we looped over our layers in *reverse* order. To perform our weight update phase, we'll simply *reverse* the ordering of entries in `D` so we can loop over each layer sequentially from *0* to *N*, the total number of layers in the network (**Line 115**).

Updating our actual weight matrix (i.e., where the actual "learning" takes place) is accomplished on **Line 125**, which is our gradient descent. We take the dot product of the current `layer` activation, `A[layer]` with the deltas of the current `layer`, `D[layer]` and multiply them by the learning rate, `alpha`. This value is added to the weight matrix for the current `layer`, `W[layer]`.

We repeat this process for all layers in the network. After performing the weight update phase, backpropagation is officially done.

Once our network is trained on a given dataset, we'll want to make predictions on the testing set, which can be accomplished via the `predict` method below:

→ **Launch Jupyter Notebook on Google Colab**

Backpropagation from scratch with Python

```
    def predict(self, X, addBias=True):
        # initialize the output prediction as the input features -- this
        # value will be (forward) propagated through the network to
        # obtain the final prediction
        p = np.atleast_2d(X)

        # check to see if the bias column should be added
        if addBias:
            # insert a column of 1's as the last entry in the feature
            # matrix (bias)
            p = np.c_[p, np.ones((p.shape[0]))]

        # loop over our layers in the network
        for layer in np.arange(0, len(self.W)):
            # computing the output prediction is as simple as taking
            # the dot product between the current activation value 'p'
            # and the weight matrix associated with the current layer,
            # then passing this value through a nonlinear activation
            # function
            p = self.sigmoid(np.dot(p, self.W[layer]))

        # return the predicted value
        return p
```

The `predict` function is simply a glorified forward pass. This function accepts one required parameter followed by a second optional one:

required parameter followed by a second optional one:

- `X` : The data points we'll be predicting class labels for.
- `addBias` : A boolean indicating whether we need to add a column of *1*'s to `X` to perform the bias trick.

On **Line 131**, we initialize `p`, the output predictions as the input data points `X`. This value `p` will be passed through every layer in the network, propagating until we reach the final output prediction.

On **Lines 134-137,** we make a check to see if the bias term should be embedded into the data points. If so, we insert a column of 1's as the last column in the matrix (exactly as we did in the `fit` method above).

From there, we perform the forward propagation by looping over all layers in our network on **Line 140**. The data points `p` are updated by taking the dot product between the current activations `p` and the weight matrix for the current `layer`, followed by passing the output through our sigmoid activation function (**Line 146**).

Given that we are looping over all layers in the network, we'll eventually reach the final layer, which will give us our final class label prediction. We return the predicted value to the calling function on **Line 149**.

The final function we'll define inside the `NeuralNetwork` class will be used to calculate the loss across our *entire* training set:

→ **Launch Jupyter Notebook on Google Colab**

Backpropagation from scratch with Python

```
    def calculate_loss(self, X, targets):
        # make predictions for the input data points then compute
        # the loss
        targets = np.atleast_2d(targets)
        predictions = self.predict(X, addBias=False)
        loss = 0.5 * np.sum((predictions - targets) ** 2)

        # return the loss
        return loss
```

The `calculate_loss` function requires that we pass in the data points `X` along with their ground-truth labels, `targets`. We make predictions on `X` on **Line 155** and then compute the sum squared error on **Line 156**. The loss is then returned to the calling function on **Line 159**. As our network learns, we should see this loss decrease.

Backpropagation with Python Example #1: Bitwise XOR

Now that we have implemented our `NeuralNetwork` class, let's go ahead and train it on the bitwise XOR dataset. As we know from our work with the Perceptron, this dataset is *not* linearly separable — our goal will be to train a neural network that can model this nonlinear function.

Go ahead and open a new file, name it `nn_xor.py`, and insert the following code:

→ **Launch Jupyter Notebook on Google Colab**

Backpropagation from scratch with Python

```
 1. | # import the necessary packages
 2. | from pyimagesearch.nn import NeuralNetwork
 3. | import numpy as np
 4. |
 5. | # construct the XOR dataset
 6. | X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
 7. | y = np.array([[0], [1], [1], [0]])
```

Lines 2 and 3 import our required Python packages. Notice how we are importing our newly implemented `NeuralNetwork` class. **Lines 6 and 7** then construct the XOR dataset.

We can now define our network architecture and train it:

→ **Launch Jupyter Notebook on Google Colab**

Backpropagation from scratch with Python

```
 9. | # define our 2-2-1 neural network and train it
10. | nn = NeuralNetwork([2, 2, 1], alpha=0.5)
11. | nn.fit(X, y, epochs=20000)
```

On **Line 10**, we instantiate our `NeuralNetwork` to have a *2−2−1* architecture, implying there is:

1. An input layer with two nodes (i.e., our two inputs).
2. A single hidden layer with two nodes.
3. An output layer with one node.

Line 11 trains our network for a total of 20,000 epochs.

Once our network is trained, we'll loop over our XOR datasets, allow the network to predict the output for each one, and display the prediction on our screen:

→ **Launch Jupyter Notebook on Google Colab**

Backpropagation from scratch with Python

```
13. # now that our network is trained, loop over the XOR data points
14. for (x, target) in zip(X, y):
15.     # make a prediction on the data point and display the result
16.     # to our console
17.     pred = nn.predict(x)[0][0]
18.     step = 1 if pred > 0.5 else 0
19.     print("[INFO] data={}, ground-truth={}, pred={:.4f}, step={}".format(
20.         x, target[0], pred, step))
```

Line 18 applies a step function to the sigmoid output. If the prediction is *> 0.5*, we'll return *one*, otherwise, we will return *zero*. Applying this step function allows us to binarize our output class labels, just like the XOR function.

To train our neural network using backpropagation with Python, simply execute the following command:

→ **Launch Jupyter Notebook on Google Colab**

Backpropagation from scratch with Python

```
$ python nn_xor.py
[INFO] epoch=1, loss=0.5092796
[INFO] epoch=100, loss=0.4923591
[INFO] epoch=200, loss=0.4677865
...
[INFO] epoch=19800, loss=0.0002478
[INFO] epoch=19900, loss=0.0002465
[INFO] epoch=20000, loss=0.0002452
```

```
[INFO] epoch=20000, loss=0.0002452
```

A plot of the squared loss is displayed below (**Figure 3**). As we can see, loss slowly decreases to approximately zero over the course of training. Furthermore, looking at the final four lines of the output we can see our predictions:

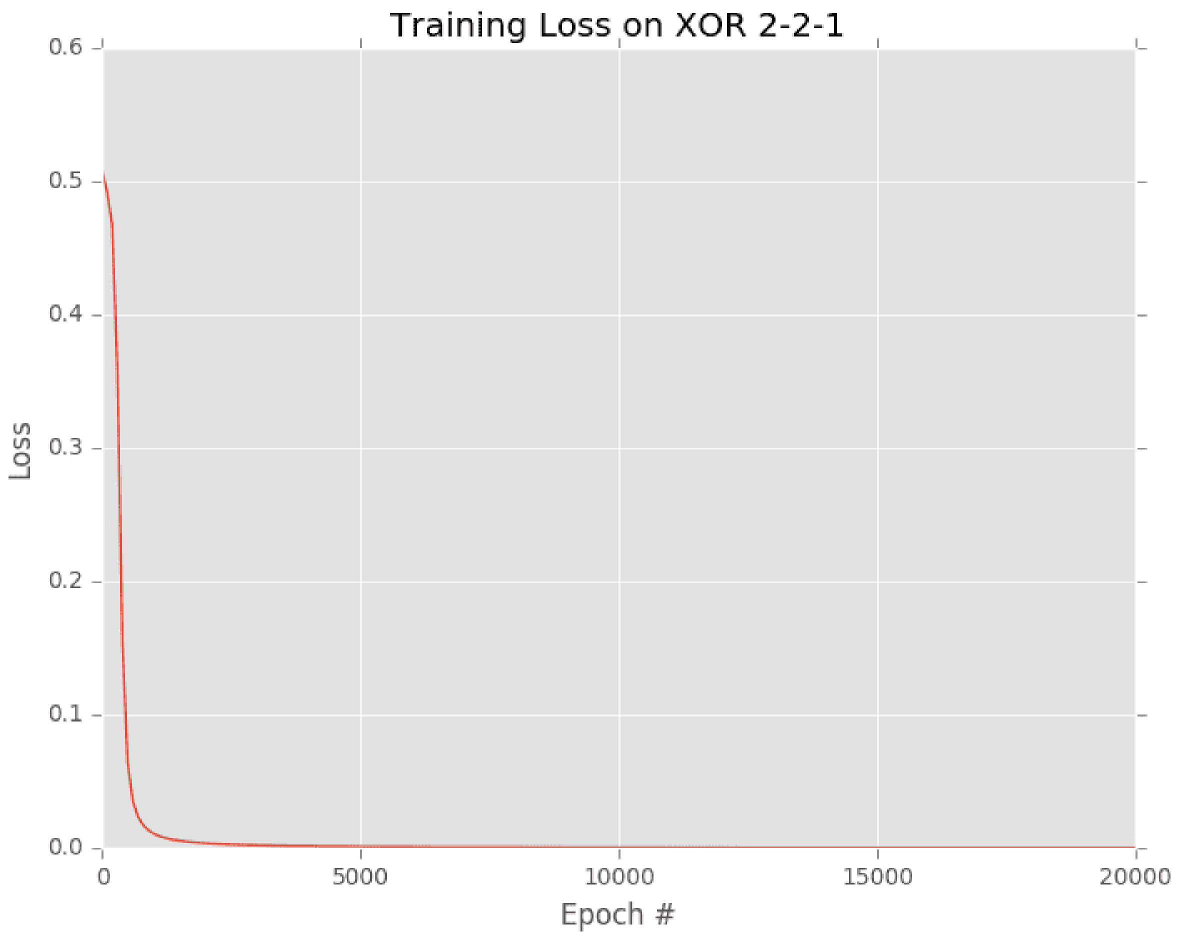


Figure 3: Loss over time for our *2−2−1* neural network.

→ **Launch Jupyter Notebook on Google Colab**

Backpropagation from scratch with Python

```
[INFO] data=[0 0], ground-truth=0, pred=0.0054, step=0
[INFO] data=[0 1], ground-truth=1, pred=0.9894, step=1
[INFO] data=[1 0], ground-truth=1, pred=0.9876, step=1
[INFO] data=[1 1], ground-truth=0, pred=0.0140, step=0
```

For each and every data point, our neural network was able to correctly learn the XOR pattern, demonstrating that our multi-layer neural network is capable of learning pattern, demonstrating that our multi-layer neural network is capable of learning nonlinear functions.

To demonstrate that least one hidden layer is required to learn the XOR function, go back to **Line 10** where we define the *2−2−1* architecture:

→ **Launch Jupyter Notebook on Google Colab**

Backpropagation from scratch with Python

```
10. # define our 2-2-1 neural network and train it
11. nn = NeuralNetwork([2, 2, 1], alpha=0.5)
12. nn.fit(X, y, epochs=20000)
```

And change it to be a *2-1* architecture:

→ **Launch Jupyter Notebook on Google Colab**

Backpropagation from scratch with Python

```
10. # define our 2-1 neural network and train it
11. nn = NeuralNetwork([2, 1], alpha=0.5)
12. nn.fit(X, y, epochs=20000)
```

From there, you can attempt to retrain your network:

→ **Launch Jupyter Notebook on Google Colab**

Backpropagation from scratch with Python

```
$ python nn_xor.py
...
[INFO] data=[0 0], ground-truth=0, pred=0.5161, step=1
[INFO] data=[0 1], ground-truth=1, pred=0.5000, step=1
[INFO] data=[1 0], ground-truth=1, pred=0.4839, step=0
[INFO] data=[1 1], ground-truth=0, pred=0.4678, step=0
```

No matter how much you fiddle with the learning rate or weight initializations, you'll never be able to approximate the XOR function. This fact is why multi-layer networks with nonlinear activation functions trained via backpropagation are so important — they enable us to learn patterns in datasets that are otherwise nonlinearly separable.

Backpropagation with Python Example: MNIST Sample

As a second, more interesting example, let's examine a subset of the MNIST dataset (**Figure 4**) for handwritten digit recognition. This subset of the MNIST dataset is built-into the scikit-learn library and includes 1,797 example digits, each of which are *8×8* grayscale images (the original images are *28×28*). When flattened, these images are represented by an *8×8* = *64*-dim vector.

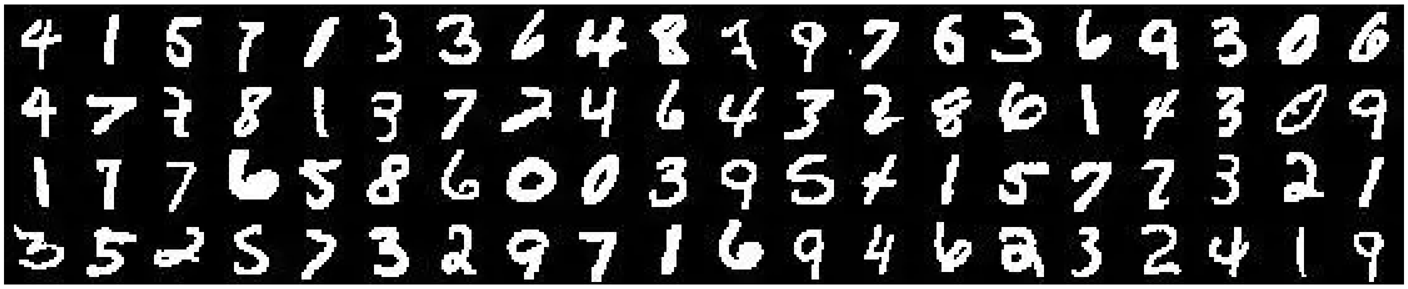


Figure 4: A sample of the MNIST dataset. The goal of this dataset is to correctly classify the handwritten digits, *0−9*.

Let's go ahead and train our `NeuralNetwork` implementation on this MNIST subset now. Open a new file, name it `nn_mnist.py`, and we'll get to work:

→ **Launch Jupyter Notebook on Google Colab**

Backpropagation from scratch with Python

```
1. # import the necessary packages
2. from pyimagesearch.nn import NeuralNetwork
3. from sklearn.preprocessing import LabelBinarizer
4. from sklearn.model_selection import train_test_split
5. from sklearn.metrics import classification_report
6. from sklearn import datasets
```

We start on **Lines 2-6** by importing our required Python packages.

From there, we load the MNIST dataset from disk using the scikit-learn helper functions:

→ **Launch Jupyter Notebook on Google Colab**

Backpropagation from scratch with Python

```
 8. # load the MNIST dataset and apply min/max scaling to scale the
 9. # pixel intensity values to the range [0, 1] (each image is
10. # represented by an 8 x 8 = 64-dim feature vector)
11. print("[INFO] loading MNIST (sample) dataset...")
12. digits = datasets.load_digits()
13. data = digits.data.astype("float")
14. data = (data - data.min()) / (data.max() - data.min())
15. print("[INFO] samples: {}, dim: {}".format(data.shape[0],
16.    data.shape[1]))
```

We also perform min/max normalizing by scaling each digit into the range *[0, 1]* **(Line 14)**.

Next, let's construct a training and testing split, using 75% of the data for testing and 25% for evaluation:

→ **Launch Jupyter Notebook on Google Colab**

Backpropagation from scratch with Python

```
# construct the training and testing splits
(trainX, testX, trainY, testY) = train_test_split(data,
	digits.target, test_size=0.25)

# convert the labels from integers to vectors
trainY = LabelBinarizer().fit_transform(trainY)
testY = LabelBinarizer().fit_transform(testY)
```

We'll also encode our class label integers as vectors, a process called *one-hot encoding*.

From there, we are ready to train our network:

→ **Launch Jupyter Notebook on Google Colab**

Backpropagation from scratch with Python

```
# train the network
print("[INFO] training network...")
nn = NeuralNetwork([trainX.shape[1], 32, 16, 10])
print("[INFO] {}".format(nn))
nn.fit(trainX, trainY, epochs=1000)
```

Here, we can see that we are training a `NeuralNetwork` with a *64−32−16−10* architecture. The output layer has ten nodes due to the fact that there are ten possible output classes for the digits *0-9*.

We then allow our network to train for 1,000 epochs. Once our network has been trained, we can evaluate it on the testing set:

→ **Launch Jupyter Notebook on Google Colab**

Backpropagation from scratch with Python

```
# evaluate the network
print("[INFO] evaluating network...")
```

```
34. predictions = nn.predict(testX)
35. predictions = predictions.argmax(axis=1)
36. print(classification_report(testY.argmax(axis=1), predictions))
```

Line 34 computes the output `predictions` for every data point in `testX`. The `predictions` array has the shape `(450, 10)` as there are 450 data points in the testing set, each of which with ten possible class label probabilities.

To find the class label with the *largest probability* for *each data point*, we use the `argmax` function on **Line 35** — this function will return the index of the label with the highest predicted probability. We then display a nicely formatted classification report to our screen on **Line 36**.

To train our custom `NeuralNetwork` implementation on the MNIST dataset, just execute the following command:

→ **Launch Jupyter Notebook on Google Colab**

Backpropagation from scratch with Python

```
$ python nn_mnist.py
[INFO] loading MNIST (sample) dataset...
[INFO] samples: 1797, dim: 64
[INFO] training network...
[INFO] NeuralNetwork: 64-32-16-10
[INFO] epoch=1, loss=604.5868589
[INFO] epoch=100, loss=9.1163376
[INFO] epoch=200, loss=3.7157723
[INFO] epoch=300, loss=2.6078803
[INFO] epoch=400, loss=2.3823153
[INFO] epoch=500, loss=1.8420944
[INFO] epoch=600, loss=1.3214138
[INFO] epoch=700, loss=1.2095033
[INFO] epoch=800, loss=1.1663942
[INFO] epoch=900, loss=1.1394731
[INFO] epoch=1000, loss=1.1203779
[INFO] evaluating network...
             precision    recall  f1-score   support

          0       1.00      1.00      1.00        45
          1       0.98      1.00      0.99        51
          2       0.98      1.00      0.99        47
          3       0.98      0.93      0.95        43
          4       0.95      1.00      0.97        39
          5       0.94      0.97      0.96        35
          6       1.00      1.00      1.00        53
          7       1.00      1.00      1.00        49
          8       0.97      0.95      0.96        41
          9       1.00      0.96      0.98        47

avg / total       0.98      0.98      0.98       450
```

I have included a plot of the squared loss as well (**Figure 5**). Notice how our loss starts off very high, but quickly drops during the training process. Our classification report demonstrates that we are obtaining *≈98%* classification accuracy on our testing set; however, we are having some trouble classifying digits `4` and `5` (95% and 94% accuracy, respectively). Later in this book, we'll learn how to train Convolutional Neural Networks on the *full* MNIST dataset and improve our accuracy further.

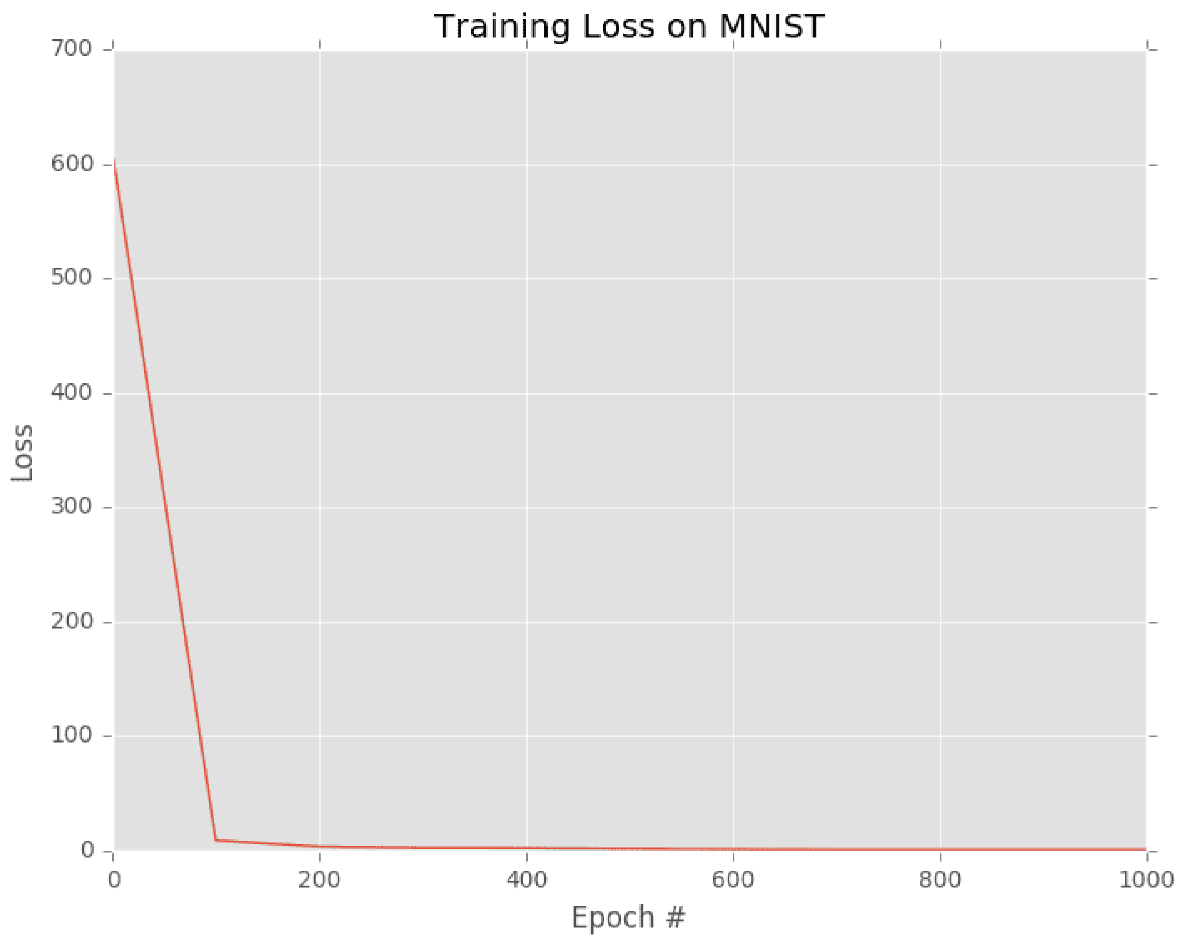
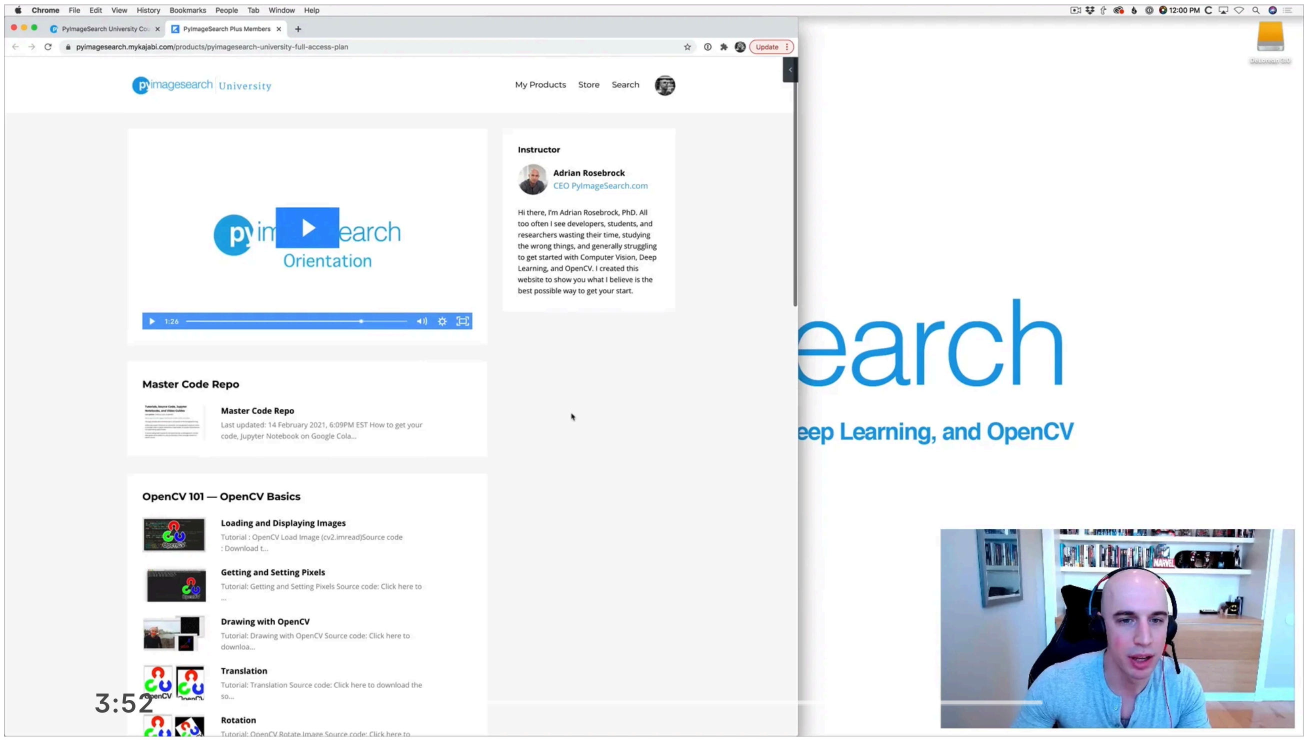


Figure 5: Plotting training loss on the MNIST dataset using a *64−32−16−10* feedforward neural network.

What's next? We recommend PyImageSearch University

(https://pyimagesearch.com/pyimagesearch-university/?utm_source=blogPost&utm_medium=bottomBanner&utm_campaign=What%27s%20next%3F%20I%20recommend).



Course information:
86+ total classes • 115+ hours hours of on-demand code walkthrough videos • Last updated: January 2025
★★★★★ 4.84 (128 Ratings) • 16,000+ Students Enrolled

I strongly believe that if you had the right teacher you could *master* computer vision and deep learning.

Do you think learning computer vision and deep learning has to be time-consuming, overwhelming, and complicated? Or has to involve complex mathematics and equations? Or requires a degree in computer science?

That's *not* the case.

All you need to master computer vision and deep learning is for someone to explain things to you in *simple, intuitive* terms. *And that's exactly what I do*. My mission is to change education and how complex Artificial Intelligence topics are taught.

If you're serious about learning computer vision, your next stop should be PyImageSearch University, the most comprehensive computer vision, deep learning, and OpenCV course online today. Here you'll learn how to *successfully* and *confidently* apply computer vision to your work, research, and projects. Join me in computer vision mastery.

Inside PyImageSearch University you'll find:

✓ **86+ courses** on essential computer vision, deep learning, and OpenCV topics

✓ **86 Certificates** of Completion

✓ **115+ hours hours** of on-demand video

✓ **Brand new courses released *regularly***, ensuring you can keep up with state-of-the-art techniques

✓ **Pre-configured Jupyter Notebooks in Google Colab**

✓ Run all code examples in your web browser — works on Windows, macOS, and Linux (no dev environment configuration required!)

Backpropagation Summary

Today, we learned how to implement the backpropagation algorithm from scratch using Python. Backpropagation is a generalization of the gradient descent family of algorithms that is specifically used to train multi-layer feedforward networks.

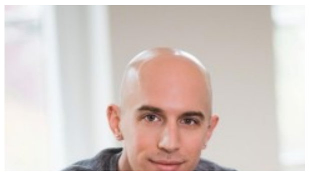
The backpropagation algorithm consists of two phases:

1. The forward pass where we pass our inputs through the network to obtain our output classifications.
2. The backward pass (i.e., weight update phase) where we compute the gradient of the loss function and use this information to iteratively apply the chain rule to update the weights in our network.

Regardless of whether we are working with simple feedforward neural networks or complex, deep Convolutional Neural Networks, the backpropagation algorithm is still used to train these models. This is accomplished by ensuring that the activation functions inside the network are *differentiable*, allowing the chain rule to be applied functions inside the network are *differentiable*, allowing the chain rule to be applied. Furthermore, any other layers inside the network that require updates to their weights/parameters, must also be compatible with backpropagation as well.

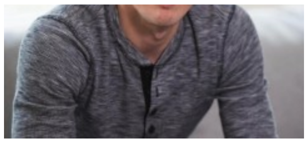
We implemented our backpropagation algorithm using the Python programming language and devised a multi-layer, feedforward `NeuralNetwork` class. This implementation was then trained on the XOR dataset to demonstrate that our neural network is capable of learning nonlinear functions by applying the backpropagation algorithm with at least one hidden layer. We then applied the same backpropagation + Python implementation to a subset of the MNIST dataset to demonstrate that the algorithm can be used to work with image data as well.

In practice, backpropagation can be not only challenging to implement (due to bugs in computing the gradient), but also hard to make efficient without special optimization libraries, which is why we often use libraries such as Keras, TensorFlow, and mxnet that have *already* (correctly) implemented backpropagation using optimized strategies.



About the Author

Hi there, I'm Adrian Rosebrock, PhD. All too often I see developers, students, and researchers wasting their time,



developers, students, and researchers wasting their time, studying the wrong things, and generally struggling to get started with Computer Vision, Deep Learning, and OpenCV. I created this website to show you what I believe is the best possible way to get your start.

Previous Article:

Understanding weight initialization for neural networks

(https://pyimagesearch.com/2021/05/06/understanding-weight-initialization-for-neural-networks/)

Next Article:

OpenCV Eigenfaces for Face Recognition

(https://pyimagesearch.com/2021/05/10/opencv-eigenfaces-for-face-recognition/)

Comment section

Hey, Adrian Rosebrock here, author and creator of PyImageSearch. While I love hearing from readers, a couple years ago I made the tough decision to no longer offer 1:1 help over blog post comments.

At the time I was receiving 200+ emails per day and another 100+ blog post comments. I simply did not have the time to moderate and respond to them all, and the sheer volume of requests was taking a toll on me.

Instead, my goal is to *do the most good* for the computer vision, deep learning, and OpenCV community at large by focusing my time on authoring high-quality blog posts, tutorials, and books/courses.

If you need help learning computer vision and deep learning, I suggest you refer to my full catalog of books and courses (https://pyimagesearch.com/books-and-courses/) — they have helped tens of thousands of developers, students, and researchers *just like yourself* learn Computer Vision, Deep Learning, and OpenCV.

Click here to browse my full catalog. (https://pyimagesearch.com/books-and-courses/)

Similar articles

DLIB FACE APPLICATIONS TUTORIALS

Face detection with dlib (HOG and CNN)

April 19, 2021

(https://pyimagesearch.com/2021/04/19/face-detection-with-dlib-hog-and-cnn/)

DEEP LEARNING DL4CV

macOS Mojave: Install TensorFlow and Keras for Deep Learning

January 30, 2019

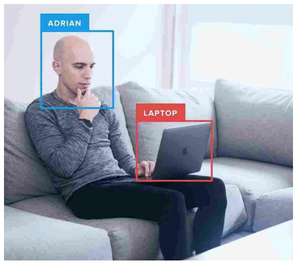
(https://pyimagesearch.com/2019/01/30/macos-mojave-install-tensorflow-and-keras-for-deep-learning/)

BUILDING A POKEDEX EXAMPLES OF IMAGE SEARCH ENGINES TUTORIALS

Building a Pokedex in Python: OpenCV and Perspective Warping (Step 5 of 6)

May 5, 2014

(https://pyimagesearch.com/2014/05/05/building-pokedex-python-opencv-perspective-warping-step-5-6/)



You can learn Computer Vision, Deep Learning, and OpenCV.

Get your FREE 17 page Computer Vision, OpenCV, and Deep Learning Resource Guide PDF. Inside you'll find our hand-picked tutorials, books, courses, and libraries to help you master CV and DL.

Topics

Deep Learning (https://pyimagesearch.com/category/deep-learning/)

Dlib Library (https://pyimagesearch.com/category/dlib/)

Embedded/IoT and Computer Vision (https://pyimagesearch.com/category/embedded/)

Face Applications (https://pyimagesearch.com/category/faces/)

Image Processing (https://pyimagesearch.com/category/image-processing/)

Interviews (https://pyimagesearch.com/category/interviews/)

Keras & Tensorflow (https://pyimagesearch.com/category/keras-and-tensorflow/)

OpenCV Install Guides (https://pyimagesearch.com/opencv-tutorials-resources-guides/)

Machine Learning and Computer Vision (https://pyimagesearch.com/category/machine-learning/)

Medical Computer Vision (https://pyimagesearch.com/category/medical/)

Optical Character Recognition (OCR) (https://pyimagesearch.com/category/optical-character-recognition-ocr/)

Object Detection (https://pyimagesearch.com/category/object-detection/)

Object Tracking (https://pyimagesearch.com/category/object-tracking/)

OpenCV Tutorials (https://pyimagesearch.com/category/opencv/)

Raspberry Pi (https://pyimagesearch.com/category/raspberry-pi/)

Books & Courses

PyImageSearch University (https://pyimagesearch.com/pyimagesearch-university/)

FREE CV, DL, and OpenCV Crash Course (https://pyimagesearch.com/free-opencv-computer-vision-deep-learning-crash-course/)

Practical Python and OpenCV (https://pyimagesearch.com/practical-python-opencv/)

Deep Learning for Computer Vision with Python (https://pyimagesearch.com/deep-learning-computer-vision-python-book/)

PyImageSearch Gurus Course (https://pyimagesearch.com/pyimagesearch-gurus/)

Raspberry Pi for Computer Vision (https://pyimagesearch.com/raspberry-pi-for-computer-vision/)

PyImageSearch

Affiliates (https://pyimagesearch.com/affiliates/)

Get Started (https://pyimagesearch.com/start-here/)

About (https://pyimagesearch.com/about/)

Consulting (https://pyimagesearch.com/consulting-2/)

Coaching (https://pyimagesearch.com/consult-adrian/)

FAQ (https://pyimagesearch.com/faqs/)

YouTube (https://pyimagesearch.com/youtube/)

Blog (https://pyimagesearch.com/topics/)

Contact (https://pyimagesearch.com/contact/)

Privacy Policy (https://pyimagesearch.com/privacy-policy/)

(https://www.facebook.com/pyimagesearch) (https://twitter.com/PyImageSearch) (https://www.linkedin.com/company/pyimagesearch/) (https://www.youtube.com/channel/UCoQK7OVcIVy-nV4m-SMCk_Q/videos)