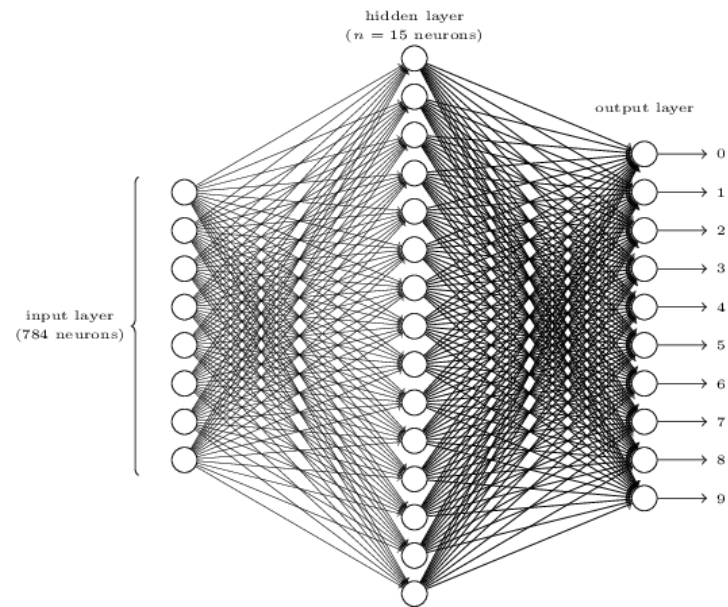


The logo for 'enlight' is displayed in a white, serif font against a solid blue background. The background is decorated with a pattern of small, light blue dots and thin white lines, resembling a starry sky or a neural network diagram.

enlight

Build a Neural Network



Login to Download Project & Start Coding

By Samay Shamdasani

OCTOBER 04, 2017 // 8745 VIEWS

What is a Neural Network?

Before we get started with the *how* of building a Neural Network, we need to understand the *what* first.

Neural networks can be intimidating, especially for people new to machine learning. However, this tutorial will break down how exactly a neural network works and you will have a working flexible neural network by the end. Let's get started!

Understanding the process

With approximately 100 billion neurons, the human brain processes data at speeds as fast as 268 mph! In essence, a neural network is a collection of **neurons** connected by **synapses**. This collection is organized into three main layers: the input layer, the hidden layer, and the output layer. You can have many hidden layers, which is where the term **deep learning** comes into play. In an artificial neural network, there are several inputs, which are called **features**, and produce a single output, which is called a **label**.

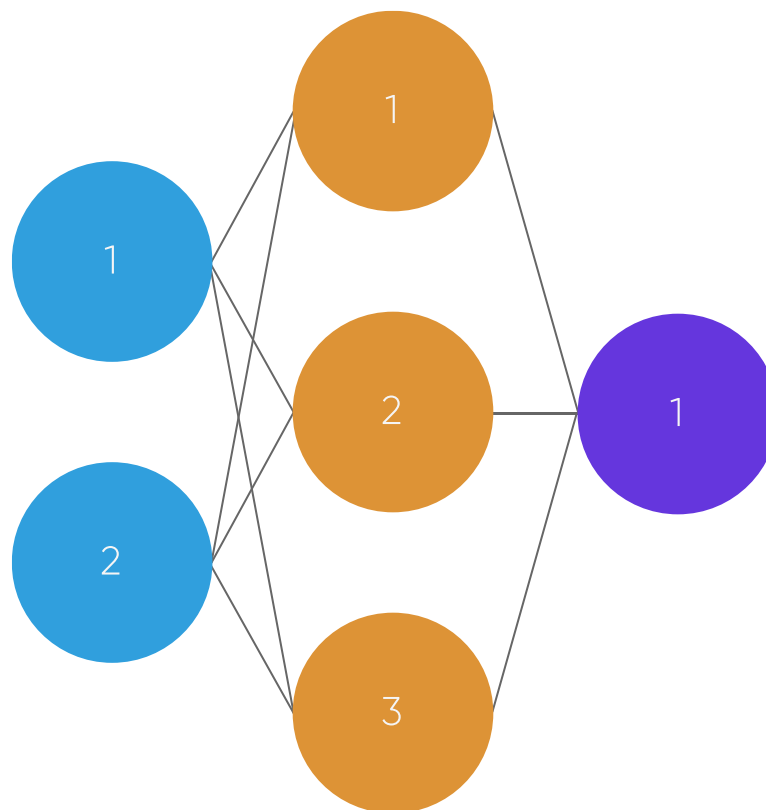


Image via [Kabir Shah](#)

The circles represent neurons while the lines represent synapses. The role of a synapse is to multiply the inputs and **weights**. You can think of weights as the “strength” of the connection between neurons. Weights primarily define the output of a neural network. However, they are highly flexible. After, an activation function is applied to return an output.

Here’s a brief overview of how a simple feedforward neural network works:

1. Takes inputs as a matrix (2D array of numbers)
2. Multiplies the input by a set weights (performs a [dot product](#) aka matrix multiplication)
3. Applies an activation function
4. Returns an output
5. Error is calculated by taking the difference from the desired output from the data and the predicted output.
This creates our gradient descent, which we can use to alter the weights
6. The weights are then altered slightly according to the error.
7. To train, this process is repeated 1,000+ times. The more the data is trained upon, the more accurate our outputs will be.

At its core, neural networks are simple. They just perform a dot product with the input and weights and apply an activation function. When weights are adjusted via the gradient of loss function, the network adapts to the changes to produce more accurate outputs.

Our neural network will model a single hidden layer with three inputs and one output. In the network, we will be predicting the score of our exam based on the inputs of how many hours we studied and how many hours we slept the day before. Our test score is the output. Here's our sample data of what we'll be training our Neural Network on:

Hours Studied, Hours Slept (input)	Test Score (output)
------------------------------------	---------------------

2, 9	92
------	----

1, 5	86
------	----

3, 6	89
------	----

4, 8	?
------	---

Original example via [Welch Labs](#)

As you may have noticed, the `?` in this case represents what we want our neural network to predict. In this case, we are predicting the test score of someone who studied for four hours and slept for eight hours based on their prior performance.

Forward Propagation

Let's start coding this bad boy! Open up a new python file. You'll want to import `numpy` as it will help us with certain calculations.

First, let's import our data as numpy arrays using `np.array`. We'll also want to normalize our units as our inputs are in hours, but our output is a test score from 0-100. Therefore, we need to scale our data by dividing by the maximum value for each variable.

```
import numpy as np

# X = (hours sleeping, hours studying), y = score on test
X = np.array([[2, 9], [1, 5], [3, 6]], dtype=float)
y = np.array([[92], [86], [89]], dtype=float)

# scale units
X = X/np.amax(X, axis=0) # maximum of X array
y = y/100 # max test score is 100
```

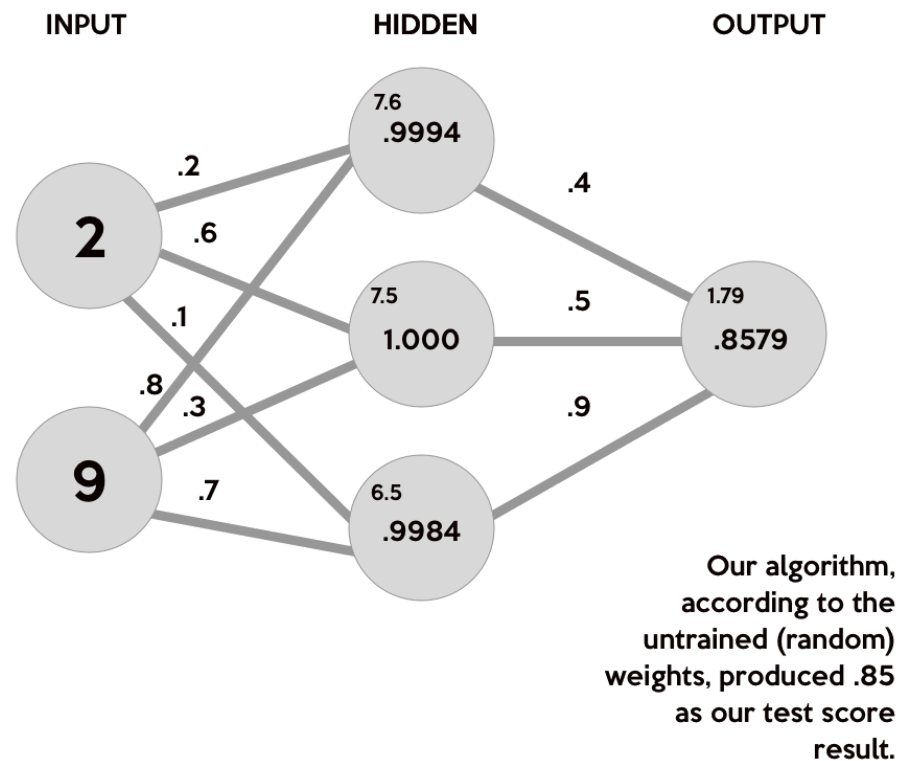
Next, let's define a python `class` and write an `init` function where we'll specify our parameters such as the input, hidden, and output layers.

```
class Neural_Network(object):
    def __init__(self):
        #parameters
        self.inputSize = 2
        self.outputSize = 1
        self.hiddenSize = 3
```

It is time for our first calculation. Remember that our synapses perform a [dot product](#), or matrix multiplication of the input and weight. Note that weights are generated randomly and between 0 and 1.

The calculations behind our network

In the data set, our input data, x , is a 3x2 matrix. Our output data, y , is a 3x1 matrix. Each element in matrix x needs to be multiplied by a corresponding weight and then added together with all the other results for each neuron in the hidden layer. Here's how the first input data element (2 hours studying and 9 hours sleeping) would calculate an output in the network:

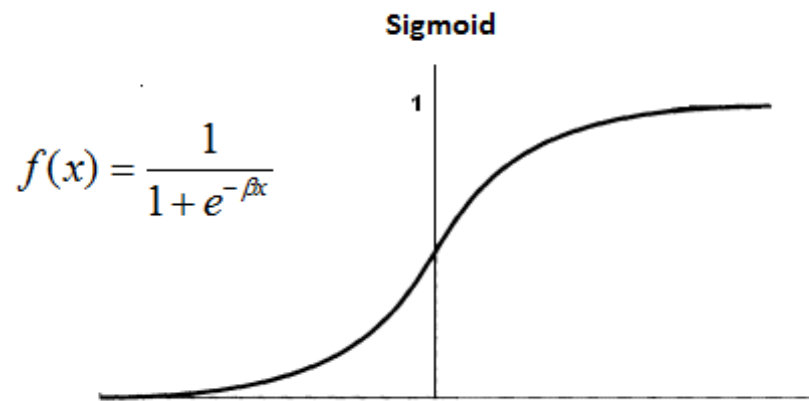


This image breaks down what our neural network actually does to produce an output. First, the products of the random generated weights ($.2$, $.6$, $.1$, $.8$, $.3$, $.7$) on each synapse and the corresponding inputs are summed to arrive as the first values of the hidden layer. These sums are in a smaller font as they are not the final values for the hidden layer.

$$\begin{aligned}(2 * .2) + (9 * .8) &= 7.6 \\(2 * .6) + (9 * .3) &= 3.9 \\(2 * .1) + (9 * .7) &= 6.5\end{aligned}$$

To get the final value for the hidden layer, we need to apply the [activation function](#). The role of an activation function is to introduce nonlinearity. An advantage of this is that the output is mapped from a range of 0 and 1, making it easier to alter weights in the future.

There are many activation functions out there. In this case, we'll stick to one of the more popular ones - the sigmoid function.



```
S(7.6) = 0.999499799  
S(7.5) = 1.000553084  
S(6.5) = 0.998498818
```

Now, we need to use matrix multiplication again, with another set of random weights, to calculate our output layer value.

```
(.9994 * .4) + (1.000 * .5) + (.9984 * .9) = 1.79832
```

Lastly, to normalize the output, we just apply the activation function again.

```
S(1.79832) = .8579443067
```

And, there you go! Theoretically, with those weights, our neural network will calculate `.85` as our test score! However, our target was `.92`. Our result wasn't poor, it just isn't the best it can be. We just got a little lucky when I chose the random weights for this example.

How do we train our model to learn? Well, we'll find out very soon. For now, let's continue coding our network.

If you are still confused, I highly recommend you check out [this](#) informative video which explains the structure of a neural network with the same example.

Implementing the calculations

Now, let's generate our weights randomly using `np.random.randn()`. Remember, we'll need two sets of weights. One to go from the input to the hidden layer, and the other to go from the hidden to output layer.

```
#weights
self.W1 = np.random.randn(self.inputSize, self.hiddenSize) # (3x2) weight matrix from input to hidden layer
self.W2 = np.random.randn(self.hiddenSize, self.outputSize) # (3x1) weight matrix from hidden to output layer
```

Once we have all the variables set up, we are ready to write our `forward` propagation function. Let's pass in our input, `x`, and in this example, we can use the variable `z` to simulate the activity between the input and output layers. As explained, we need to take a dot product of the inputs and weights, apply an activation function, take another dot product of the hidden layer and second set of weights, and lastly apply a final activation function to receive our output:

```
def forward(self, X):  
    #forward propagation through our network  
    self.z = np.dot(X, self.W1) # dot product of X (input) and first set of 3x2 weights  
    self.z2 = self.sigmoid(self.z) # activation function  
    self.z3 = np.dot(self.z2, self.W2) # dot product of hidden layer (z2) and second set of 3x1 weights  
    o = self.sigmoid(self.z3) # final activation function  
    return o
```

Lastly, we need to define our sigmoid function:

```
def sigmoid(self, s):  
    # activation function  
    return 1/(1+np.exp(-s))
```

And, there we have it! A (untrained) neural network capable of producing an output.

```

import numpy as np

# X = (hours sleeping, hours studying), y = score on test
X = np.array([[2, 9], [1, 5], [3, 6]], dtype=float)
y = np.array([[92], [86], [89]], dtype=float)

# scale units
X = X/np.amax(X, axis=0) # maximum of X array
y = y/100 # max test score is 100

class Neural_Network(object):
    def __init__(self):
        #parameters
        self.inputSize = 2
        self.outputSize = 1
        self.hiddenSize = 3

        #weights
        self.W1 = np.random.randn(self.inputSize, self.hiddenSize) # (3x2) weight matrix from input to hidden
        self.W2 = np.random.randn(self.hiddenSize, self.outputSize) # (3x1) weight matrix from hidden to output

    def forward(self, X):
        #forward propagation through our network
        self.z = np.dot(X, self.W1) # dot product of X (input) and first set of 3x2 weights
        self.z2 = self.sigmoid(self.z) # activation function
        self.z3 = np.dot(self.z2, self.W2) # dot product of hidden layer (z2) and second set of 3x1 weights
        o = self.sigmoid(self.z3) # final activation function
        return o

    def sigmoid(self, s):

```

```
# activation function
return 1/(1+np.exp(-s))

NN = Neural_Network()

#defining our output
o = NN.forward(X)

print "Predicted Output: \n" + str(o)
print "Actual Output: \n" + str(y)
```

As you may have noticed, we need to train our network to calculate more accurate results.

Backpropagation


The “learning” of our network

Since we have a random set of weights, we need to alter them to make our inputs equal to the corresponding outputs from our data set. This is done through a method called backpropagation.

Backpropagation works by using a **loss** function to calculate how far the network was from the target output.

Calculating error

One way of representing the loss function is by using the **mean sum squared loss** function:

 In this function, \hat{y} is our predicted output, and y is our actual output. Now that we have the loss function, our goal is to get it as close as we can to 0. That means we will need to have close to no loss at all. As we are training our network, all we are doing is minimizing the loss.

To figure out which direction to alter our weights, we need to find the rate of change of our loss with respect to our weights. In other words, we need to use the derivative of the loss function to understand how the weights affect the input.

In this case, we will be using a partial derivative to allow us to take into account another variable.

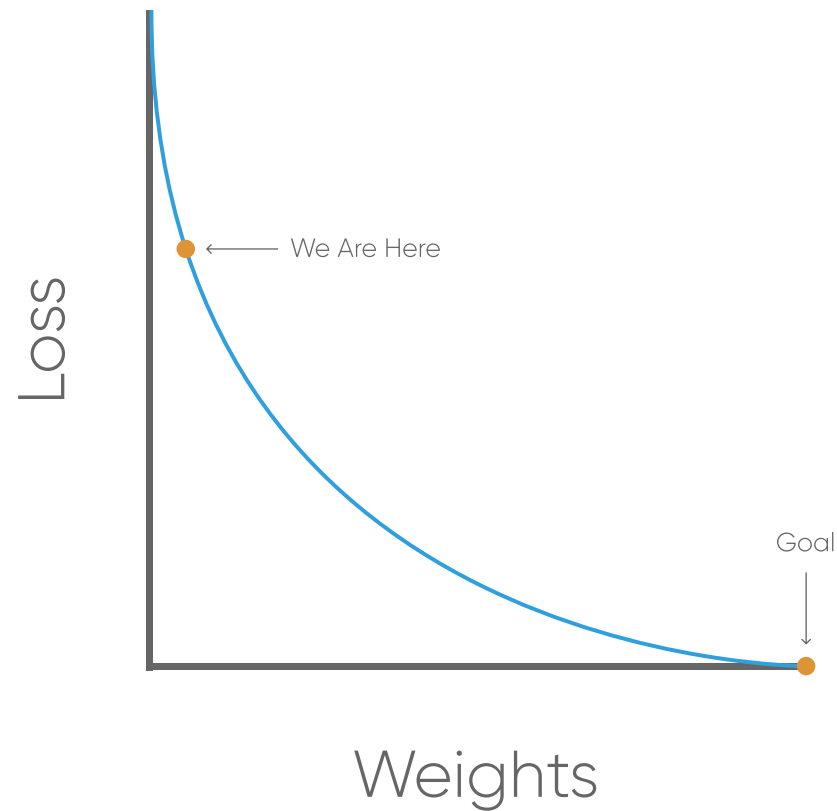


Image via [Kabir Shah](#)

This method is known as **gradient descent**. By knowing which way to alter our weights, our outputs can only get more accurate.

Here's how we will calculate the incremental change to our weights:

1. Find the **margin of error** of the output layer (o) by taking the difference of the predicted output and the actual output (y)

2. Apply the derivative of our sigmoid activation function to the output layer error. We call this result the **delta output sum**.
3. Use the delta output sum of the output layer error to figure out how much our z^2 (hidden) layer contributed to the output error by performing a dot product with our second weight matrix. We can call this the z^2 error.
4. Calculate the delta output sum for the z^2 layer by applying the derivative of our sigmoid activation function (just like step 2).
5. Adjust the weights for the first layer by performing a **dot product of the input layer** with the **hidden (z^2) delta output sum**. For the second layer, perform a dot product of the hidden(z^2) layer and the **output (o) delta output sum**.

Calculating the delta output sum and then applying the derivative of the sigmoid function are very important to backpropagation. The derivative of the sigmoid, also known as **sigmoid prime**, will give us the rate of change, or slope, of the activation function at output sum.

Let's continue to code our `Neural_Network` class by adding a `sigmoidPrime` (derivative of sigmoid) function:

```
def sigmoidPrime(self, s):  
    #derivative of sigmoid  
    return s * (1 - s)
```

Then, we'll want to create our `backward` propagation function that does everything specified in the four steps above:

```
def backward(self, X, y, o):  
    # backward propagate through the network  
    self.o_error = y - o # error in output  
    self.o_delta = self.o_error*self.sigmoidPrime(o) # applying derivative of sigmoid to error  
  
    self.z2_error = self.o_delta.dot(self.W2.T) # z2 error: how much our hidden layer weights contributed to error  
    self.z2_delta = self.z2_error*self.sigmoidPrime(self.z2) # applying derivative of sigmoid to z2 error  
  
    self.W1 += X.T.dot(self.z2_delta) # adjusting first set (input --> hidden) weights  
    self.W2 += self.z2.T.dot(self.o_delta) # adjusting second set (hidden --> output) weights
```

We can now define our output through initiating forward propagation and initiate the backward function by calling it in the `train` function:

```
def train (self, X, y):  
    o = self.forward(X)  
    self.backward(X, y, o)
```

To run the network, all we have to do is to run the `train` function. Of course, we'll want to do this multiple, or maybe thousands, of times. So, we'll use a `for` loop.

```

NN = Neural_Network()
for i in xrange(1000): # trains the NN 1,000 times
    print "Input: \n" + str(X)
    print "Actual Output: \n" + str(y)
    print "Predicted Output: \n" + str(NN.forward(X))
    print "Loss: \n" + str(np.mean(np.square(y - NN.forward(X)))) # mean sum squared loss
    print "\n"
    NN.train(X, y)

```

Great, we now have a Neural Network! What about using these trained weights to predict test scores that we don't know?

Predictions

To predict our test score for the input of `[4, 8]`, we'll need to create a new array to store this data, `xPredicted`.

```

xPredicted = np.array([4,8], dtype=float)

```

We'll also need to scale this as we did with our input and output variables:

```

xPredicted = xPredicted/np.amax(xPredicted, axis=0) # maximum of xPredicted (our input data for the predi

```

Then, we'll create a new function that prints our predicted output for `xPredicted`. Believe it or not, all we have to run is `forward(xPredicted)` to return an output!

```
def predict(self):  
    print "Predicted data based on trained weights: ";  
    print "Input (scaled): \n" + str(xPredicted);  
    print "Output: \n" + str(self.forward(xPredicted));
```

To run this function simply call it under the for loop.

```
NN.predict()
```

If you'd like to save your trained weights, you can do so with `np.savetxt`:

```
def saveWeights(self):  
    np.savetxt("w1.txt", self.W1, fmt="%s")  
    np.savetxt("w2.txt", self.W2, fmt="%s")
```

Here's the final result:

```
import numpy as np  
  
# X = (hours studying, hours sleeping), y = score on test, xPredicted = 4 hours studying & 8 hours sleepi  
X = np.array([[2, 9], [1, 5], [3, 6]], dtype=float)
```

```

y = np.array([[92], [86], [89]], dtype=float)
xPredicted = np.array([[4,8]], dtype=float)

# scale units
X = X/np.amax(X, axis=0) # maximum of X array
xPredicted = xPredicted/np.amax(xPredicted, axis=0) # maximum of xPredicted (our input data for the predi
y = y/100 # max test score is 100

class Neural_Network(object):
    def __init__(self):
        #parameters
        self.inputSize = 2
        self.outputSize = 1
        self.hiddenSize = 3

        #weights
        self.W1 = np.random.randn(self.inputSize, self.hiddenSize) # (3x2) weight matrix from input to hidden
        self.W2 = np.random.randn(self.hiddenSize, self.outputSize) # (3x1) weight matrix from hidden to outp

    def forward(self, X):
        #forward propagation through our network
        self.z = np.dot(X, self.W1) # dot product of X (input) and first set of 3x2 weights
        self.z2 = self.sigmoid(self.z) # activation function
        self.z3 = np.dot(self.z2, self.W2) # dot product of hidden layer (z2) and second set of 3x1 weights
        o = self.sigmoid(self.z3) # final activation function
        return o

    def sigmoid(self, s):
        # activation function
        return 1/(1+np.exp(-s))

```

```

def sigmoidPrime(self, s):
    #derivative of sigmoid
    return s * (1 - s)

def backward(self, X, y, o):
    # backward propagate through the network
    self.o_error = y - o # error in output
    self.o_delta = self.o_error*self.sigmoidPrime(o) # applying derivative of sigmoid to error

    self.z2_error = self.o_delta.dot(self.W2.T) # z2 error: how much our hidden layer weights contributed
    self.z2_delta = self.z2_error*self.sigmoidPrime(self.z2) # applying derivative of sigmoid to z2 error

    self.W1 += X.T.dot(self.z2_delta) # adjusting first set (input --> hidden) weights
    self.W2 += self.z2.T.dot(self.o_delta) # adjusting second set (hidden --> output) weights

def train(self, X, y):
    o = self.forward(X)
    self.backward(X, y, o)

def saveWeights(self):
    np.savetxt("w1.txt", self.W1, fmt="%s")
    np.savetxt("w2.txt", self.W2, fmt="%s")

def predict(self):
    print "Predicted data based on trained weights: ";
    print "Input (scaled): \n" + str(xPredicted);
    print "Output: \n" + str(self.forward(xPredicted));

NN = Neural_Network()

```

```

for i in xrange(1000): # trains the NN 1,000 times
    print "# " + str(i) + "\n"
    print "Input (scaled): \n" + str(X)
    print "Actual Output: \n" + str(y)
    print "Predicted Output: \n" + str(NN.forward(X))
    print "Loss: \n" + str(np.mean(np.square(y - NN.forward(X)))) # mean sum squared loss
    print "\n"
    NN.train(X, y)

NN.saveWeights()
NN.predict()

```

To see how accurate the network actually is, I ran trained it 100,000 times to see if it would ever get exactly the right output. Here's what I got:

```

# 99999
Input (scaled):
[[ 0.66666667  1.          ]
 [ 0.33333333  0.55555556]
 [ 1.          0.66666667]]
Actual Output:
[[ 0.92]
 [ 0.86]
 [ 0.89]]
Predicted Output:
[[ 0.92]
 [ 0.86]
 [ 0.89]]

```

```
Loss:  
1.94136958194e-18
```

```
Predicted data based on trained weights:  
Input (scaled):  
[ 0.5  1. ]  
Output:  
[ 0.91882413]
```

There you have it! A full-fledged neural network that can learn and adapt to produce accurate outputs. While we thought of our inputs as hours studying and sleeping, and our outputs as test scores, feel free to change these to whatever you like and observe how the network adapts! After all, all the network sees are the numbers. The calculations we made, as complex as they seemed to be, all played a big role in our learning model. If you think about it, it's super impressive that your computer, a physical object, managed to *learn* by itself!

If you want to see how this network can be applied to digit recognition, you can take a look at my implementation of it [here](#).

Make sure to stick around for more machine learning tutorials on other models like Linear Regression and Classification coming soon!

References

[Steven Miller](#)

[Welch Labs](#)

[Kabir Shah](#)

Discuss this project on the community chat.

Visit **enlight** Chat!

17 Comments

Enlight

1 Login ▾

♥ Recommend 2

🔗 Share

Sort by Best ▾



Join the discussion...

LOG IN WITH



OR SIGN UP WITH DISQUS (?)

Name



Mahmood • 3 months ago

Thanks for the code and your effort. However there are some mistakes in the code that can be miss leading.

-> Normalization of the input data (training) and the input data for prediction should have the same Max values. otherwise the results are not correct. In short, you should use the same normalization procedure (and thus max values in your case) should be used for normalizing the input data for regardless of the fact that they are going to be used for training or prediction

1- In this example you should save the max of your input data and then divide your prediction value by that same Max.

2- Its better to use while loop and loop until the error is reduced to a certain value. Essentially looping 1000 time is not a good idea and using this fix value on different data sets could produce networks which are still have a high error margins.

2- Bias is also missing from the calculations.

^ | v • Reply • Share ›



Samay Shamdasani Mod → Mahmood • 3 months ago

Hello,

Thank you so much for your insight. Do you think you can share an updated program so that I may make the changes accordingly?

^ | v • Reply • Share ›



Tamilarasu • 3 months ago

Excellent article for a beginner, but I just noticed Bias is missing your neural network. Isn't it required for simple neural networks?

And also you haven't applied any Learning rate. Will not it make the Gradient descent to miss the minimum?

^ | v • Reply • Share ›



Aditya Radhakrishnan • 3 months ago

What should you do for backprop with multiple outputs?

^ | v • Reply • Share ›



Balsam • 4 months ago

thank you for this amazing totorial. I'm a beginner in both python and ANN. I'm trying to built my ANN in python which has 6 input layers, 3 hidden layers and one output layers. I have two question:- (1) how can I know the percentage for both training and testing and how can determine that? (2) is how can I know the weights of inputs and hidden layers to see which one od variables is more effective than others?

thanks in advance

^ | v • Reply • Share ›



Sachin Agnihotri • 4 months ago

Well done..It's awesome for beginners...!!

^ | v • Reply • Share ›



Shaiju Janardhanan • 4 months ago

Nice tutorial.The comments in the first code block for X looks incorrect. It should have been hours studying, hours sleeping. Also the dot product values on which the activation function is applied is different from the dot product obtained in the 3rd code block

^ | v • Reply • Share ›



Tim Harris • 5 months ago

This is one of the best tutorials on a basic Neural Network I have ever seen. I feel like I know a lot more after reading this. Thanks for posting this.

^ | v • Reply • Share ›



Shahirasul Rahman Minai • 5 months ago

Thanks for the interesting tutorial.

I would like to ask how do I train a set of photos and then test one photo to see if it successful without using other libraries?

Thank you.

^ | v • Reply • Share ›



Samay Shamdasani Mod → Shahirasul Rahman Minai • 5 months ago

Well, to do this without a library, you will need to build and train your network yourself. In this case, you'll need to analyze all parts of the image. One example I personally found really helpful was the tutorial from the book: "Neural Networks and Deep Learning" (<http://neuralnetworksanddee....> On that site, it walks you through building a NN that can read handwritten digits.

<https://github.com/mnielsen...> Good Luck!

^ | v • Reply • Share ›



Jacek Pawlowski • 6 months ago

small calc error



small calc error.

$(2 * .6) + (9 * .3) = 3.9$ not as you write 7.5 (in The calculations behind our network)

It doesn't change the idea of this great article.

^ | v • Reply • Share ›



Samay Shamdasani Mod → Jacek Pawlowski • 5 months ago

Just fixed!

^ | v • Reply • Share ›



Dinesh • 6 months ago

Thanks for the easy to follow simple neural network. helped me a lot in understanding.

[Suggestion] If you add a screenshot of calculations behind backpropagation (just like the screenshot of forward pass) it will help beginners like me a lot.

^ | v • Reply • Share ›



Lalit Krishna • 6 months ago

I have an image as input with 2500 inputs. What algorithm shall I use to classify them into 3 groups

^ | v • Reply • Share ›



Samay Shamdasani Mod → Lalit Krishna • 6 months ago

So, in this case, you would want to use a classification algorithm. There are many libraries out there like tensorflow, keras, YOLO, sklearn, and more. You can start here:

<https://becominghuman.ai/bu...>

Once you complete your project, feel free to share it with the Enlight community by contributing it! <https://github.com/TryEnlig...>

^ | v • Reply • Share ›



Lalit Krishna → Samay Shamdasani • 6 months ago

Sure. Thank you.

^ | v • Reply • Share ›



Erick Moises Hernandez • 6 months ago



ERICK MOISES HERNANDEZ · 8 months ago

Hi. Thanks a lot for tutorial. However... There is a thing I do not understand:

First you normalize input matrix ($X = X/\text{np.amax}(X, \text{axis}=0)$), but after, you show calculations with original values: $(2 * .2) + (9 * .8) = 7.6$

So... What I missed?

^ | v · Reply · Share ›

ALSO ON ENLIGHT

Build a Stock Prediction Algorithm

13 comments · 8 months ago

Samay Shamdasani — We use the Quandl API, which automatically fetches the latest data to date.

Build a Guess the Number

1 comment · a year ago

angelisacandler — will there be an updated Python 3 version? I see raw_input is being used as opposed to input. As well as the lack of parentheses.

Build a Live HTML/CSS/JS Editor

2 comments · 2 years ago

Abhishek Nanda — Uncaught TypeError: Cannot set property 'onkeyup' of null error raise not running liv coding

Build a Build a Clock

5 comments · 2 years ago

Edgar Gill — Thank you for the tutorial really like it. Will post a link with my results.



Subscribe



Add Disqus to your site



Disqus' Privacy Policy

DISQUS



© Enlight 2018 by shamdasani.org

[Terms of Service](#) | [Privacy Policy](#)