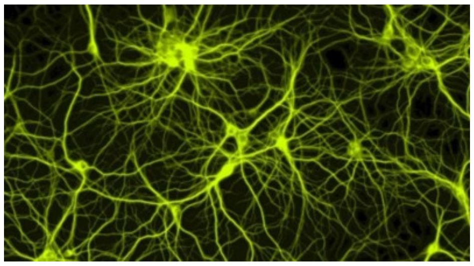
Milo Spencer-Harper Follow
Studied Economics at Oxford University. Founder of www.moju.io and www.magimetrics.com. Interested in Influencers, AI and Python.
Jul 20, 2015 · 6 min read

How to build a simple neural network in 9 lines of Python code



As part of my quest to learn about AI, I set myself the goal of building a simple neural network in Python. To ensure I truly understand it, I had to build it from scratch without using a neural network library. Thanks to an excellent blog post by Andrew Trask I achieved my goal. Here it is in just 9 lines of code:

```
from numpy import exp, array, random, dot
training_set_inputs = array([[0, 0, 1], [1, 1, 1], [1, 0, 1]
training_set_outputs = array([[0, 1, 1, 0]]).T
random.seed(1)
synaptic_weights = 2 * random.random((3, 1)) - 1
for iteration in xrange(10000):
    output = 1 / (1 + exp(-(dot(training_set_inputs, synapti
```

In this blog post, I'll explain how I did it, so you can build your own. I'll also provide a longer, but more beautiful version of the source code.

But first, what is a neural network? The human brain consists of 100 billion cells called neurons, connected together by synapses. If sufficient synaptic inputs to a neuron fire, that neuron will also fire. We call this process "thinking".

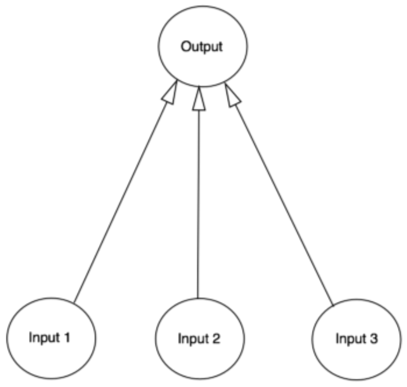
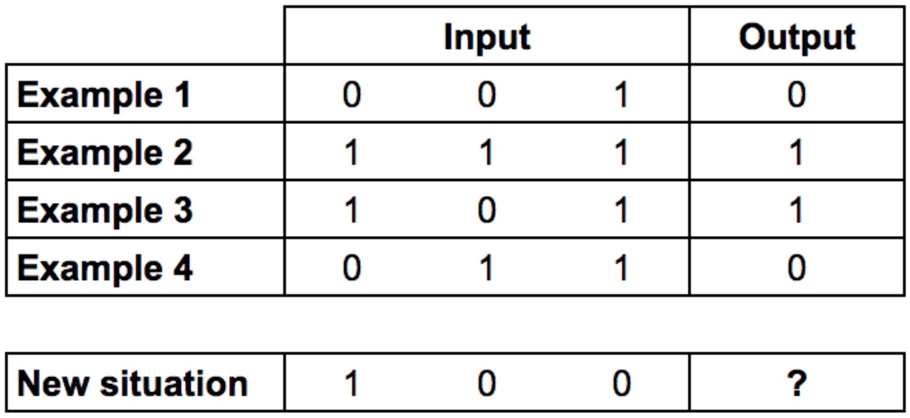


Diagram 1

We can model this process by creating a neural network on a computer. It's not necessary to model the biological complexity of the human brain at a molecular level, just its higher level rules. We use a mathematical technique called matrices, which are grids of numbers. To make it really simple, we will just model a single neuron, with three inputs and one output.

We're going to train the neuron to solve the problem below. The first four examples are called a training set. Can you work out the pattern? Should the '**?**' be 0 or 1?



	Input			Output
Example 1	0	0	1	0
Example 2	1	1	1	1
Example 3	1	0	1	1
Example 4	0	1	1	0
New situation	1	0	0	?

Diagram 2

You might have noticed, that the output is always equal to the value of the leftmost input column. Therefore the answer is the '**?**' should be 1.

Training process

But how do we teach our neuron to answer the question correctly? We will give each input a weight, which can be a positive or negative number. An input with a large positive weight or a large negative weight, will have a strong effect on the neuron's output. Before we start, we set each weight to a random number. Then we begin the training process:

1. Take the inputs from a training set example, adjust them by the weights, and pass them through a special formula to calculate the neuron's output.
2. Calculate the error, which is the difference between the neuron's output and the desired output in the training set example.
3. Depending on the direction of the error, adjust the weights slightly.
4. Repeat this process 10, 000 times.

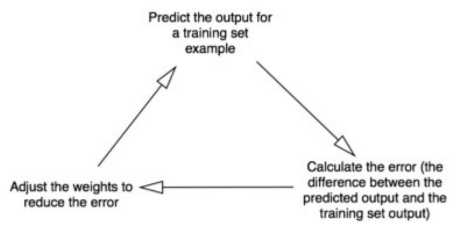

Predict the output for a training set example
Calculate the error (the difference between the predicted output and the training set output)
Adjust the weights to reduce the error

Diagram 3

Eventually the weights of the neuron will reach an optimum for the training set. If we allow the neuron to think about a new situation, that follows the same pattern, it should make a good prediction.

This process is called back propagation.

Formula for calculating the neuron's output

You might be wondering, what is the special formula for calculating the neuron's output? First we take the weighted sum of the neuron's inputs, which is:

$$\sum weight_i \cdot input_i = weight1 \cdot input1 + weight2 \cdot input2 + weight3 \cdot input3$$

Next we normalise this, so the result is between 0 and 1. For this, we use a mathematically convenient function, called the Sigmoid function:

$$\frac{1}{1+e^{-x}}$$

If plotted on a graph, the Sigmoid function draws an S shaped curve.

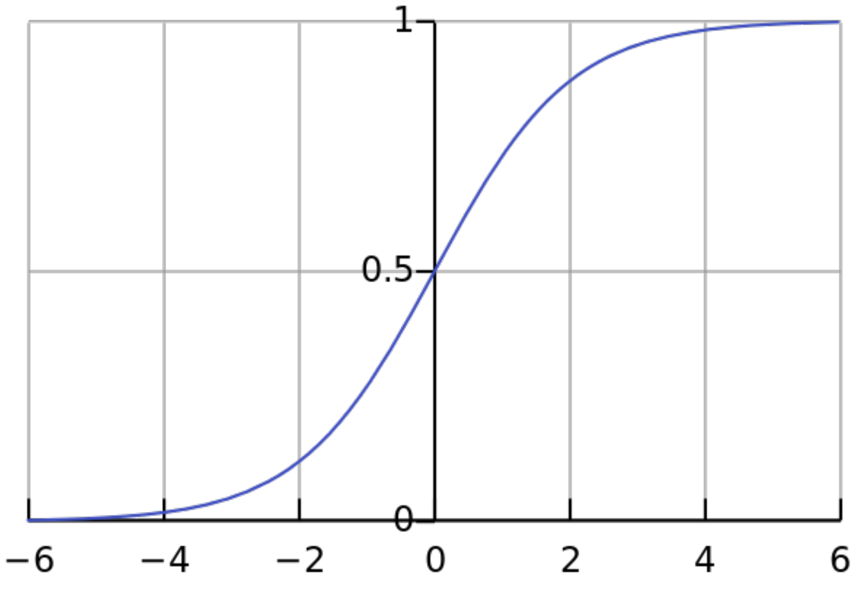


Diagram 4

So by substituting the first equation into the second, the final formula for the output of the neuron is:

$$\text{Output of neuron} = \frac{1}{1+e^{-(\sum weight_i input_i)}}$$

You might have noticed that we’re not using a minimum firing threshold, to keep things simple.

Formula for adjusting the weights

During the training cycle (Diagram 3), we adjust the weights. But how much do we adjust the weights by? We can use the “Error Weighted Derivative” formula:

$$\text{Adjust weights by} = error \cdot input \cdot SigmoidCurveGradient(output)$$

Why this formula? First we want to make the adjustment proportional to the size of the error. Secondly, we multiply by the input, which is either a 0 or a 1. If the input is 0, the weight isn't adjusted. Finally, we multiply by the gradient of the Sigmoid curve (Diagram 4). To understand this last one, consider that:

1. We used the Sigmoid curve to calculate the output of the neuron.
2. If the output is a large positive or negative number, it signifies the neuron was quite confident one way or another.
3. From Diagram 4, we can see that at large numbers, the Sigmoid curve has a shallow gradient.
4. If the neuron is confident that the existing weight is correct, it doesn't want to adjust it very much. Multiplying by the Sigmoid curve gradient achieves this.

The gradient of the Sigmoid curve, can be found by taking the derivative:

$$\text{SigmoidCurveGradient(output)} = output \cdot (1 - output)$$

So by substituting the second equation into the first equation, the final formula for adjusting the weights is:

$$\text{Adjust weights by} = error \cdot input \cdot output \cdot (1 - output)$$

There are alternative formulae, which would allow the neuron to learn more quickly, but this one has the advantage of being fairly simple.

Constructing the Python code

Although we won't use a neural network library, we will import four methods from a Python mathematics library called numpy. These are:

- exp—the natural exponential
- array—creates a matrix
- dot—multiplies matrices
- random—gives us random numbers

For example we can use the array() method to represent the training set shown earlier:

```
training_set_inputs = array([[0, 0, 1], [1, 1, 1], [1, 0, 1]
training_set_outputs = array([[0, 1, 1, 0]]).T
```

The '.T' function, transposes the matrix from horizontal to vertical. So the computer is storing the numbers like this.

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Ok. I think we're ready for the more beautiful version of the source code. Once I've given it to you, I'll conclude with some final thoughts.

I have added comments to my source code to explain everything, line by line. Note that in each iteration we process the entire training set simultaneously. Therefore our variables are matrices, which are grids of numbers. Here is a complete working example written in Python:

```
from numpy import exp, array, random, dot

```

```
class NeuralNetwork():
    def __init__(self):
        # Seed the random number generator, so it generates
        # every time the program runs.
        random.seed(1)

        # We model a single neuron, with 3 input connection
        # We assign random weights to a 3 x 1 matrix, with
        # and mean 0.
        self.synaptic_weights = 2 * random.random((3, 1)) -

    # The Sigmoid function, which describes an S shaped cur
    # We pass the weighted sum of the inputs through this f
    # normalise them between 0 and 1.
    def __sigmoid(self, x):
        return 1 / (1 + exp(-x))

    # The derivative of the Sigmoid function.
    # This is the gradient of the Sigmoid curve.
    # It indicates how confident we are about the existing
    def __sigmoid_derivative(self, x):
        return x * (1 - x)

    # We train the neural network through a process of tria
    # Adjusting the synaptic weights each time.
    def train(self, training_set_inputs, training_set_outpu
        for iteration in xrange(number_of_training_iteratio
            # Pass the training set through our neural netw
            output = self.think(training_set_inputs)

            # Calculate the error (The difference between t
            # and the predicted output).
            error = training_set_outputs - output

            # Multiply the error by the input and again by
            # This means less confident weights are adjuste
            # This means inputs, which are zero, do not cau
            adjustment = dot(training_set_inputs.T, error *

            # Adjust the weights.
            self.synaptic_weights += adjustment

    # The neural network thinks.
    def think(self, inputs):
```

Also available here: https://github.com/miloharper/simple-neural-network

Final thoughts

Try running the neural network using this Terminal command:

python main.py

You should get a result that looks like:

```
Random starting synaptic weights:
[[-0.16595599]
[ 0.44064899]
[-0.99977125]]

New synaptic weights after training:
[[ 9.67299303]
[-0.2078435 ]
[-4.62963669]]
```

We did it! We built a simple neural network using Python!

First the neural network assigned itself random weights, then trained itself using the training set. Then it considered a new situation [1, 0, 0] and predicted 0.99993704. The correct answer was 1. So very close!

Traditional computer programs normally can't learn. What's amazing about neural networks is that they can learn, adapt and respond to new situations. Just like the human mind.

Of course that was just 1 neuron performing a very simple task. But what if we hooked millions of these neurons together? Could we one day create something conscious?

In my next blog post, I expand our neural network's capabilities by adding a second layer of neurons.