Lecture5

September 13, 2017

1 Week 5, Lecture 9, Back-propagation

These notes cover a bit of week 4 as well. It's better to have feed forward, back propagation, and gradient checking all in one place.

2 Further reading

- Elements of Statistical Learning: Chapter 11, Neural Networks (get a copy)
- Computer Age Statistical Inference: Chapter 18, Neural Networks and Deep Learning (p. 351) (get a copy)
- Peter's Notes are a bit mathy and specific, but I've found them helpful when confused

3 Feed forward & back propagation

3.1 Prelude: the perceptron

The following diagram shows a perceptron.

Depending on its setup, it may be known as a neuron, a linear classifier, a linear regression (identity), or a logistic regression (sigmoid).

3.2 Feed forward basics

A neural network is a stack of layers of perceptrons working together, and the term feed forward refers to "feeding" data through this network from start to finish. As you can see with the arbitrary example below, that can be a lot of feeding.

The diagram above is just to scare you. Neural networks typically have *a lot* more connections!

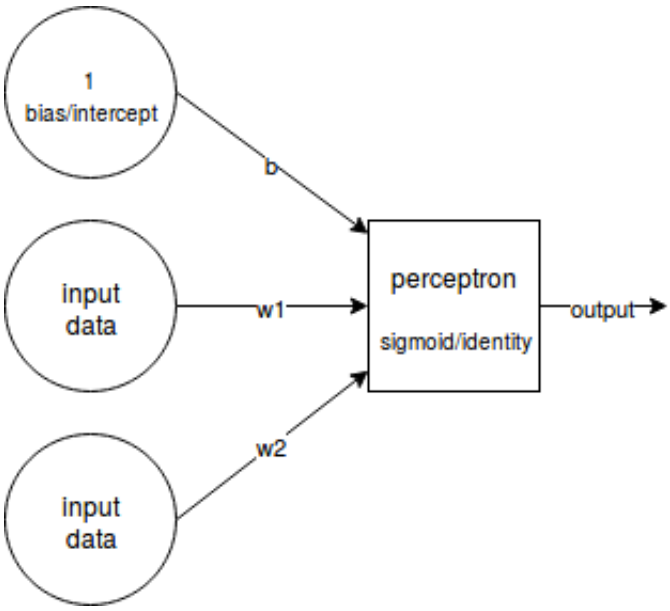
Drawn with this python script

3.2.1 Numpy importing

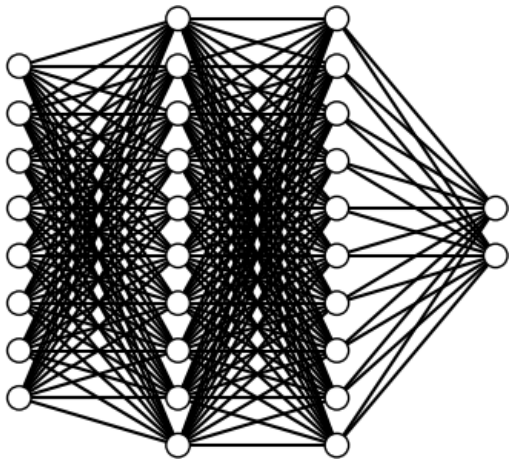
To get started, we'll need to import `numpy` to deal with all the matrices involved.

Each NN library you use will have a way of handling matrices. They tend to be similar and might even just work with `numpy` matrices seemlessly.

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        from sklearn.datasets import load_digits
```



Just a lonely neuron



A lot

3.2.2 Values flowing

In a neural network, values flow through layers of synapses and neurons. This is called feed-forward.

To feed-forward data through a neural network is to pass data through the network's weights and activation functions to create an output. The feed-forward receives its data at the input layer, a copy of the input. The activity begins at the first hidden layer, when the input signal is passed through synapses (weights), and adjusted (bias) and transformed (activation function) by the neuron. Here is what it does:

First hidden layer Signal x passing though weights w_1: $x \times w_1$

Signal adjusted by neuron bias b_1: $x \times w_1 + b_1 = z_1$

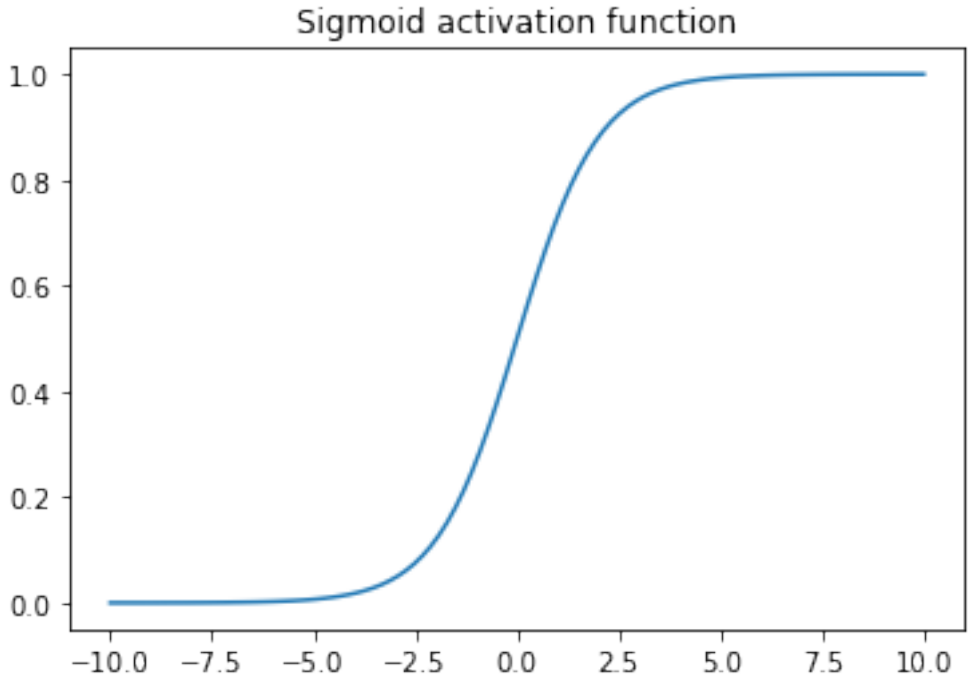
Signal shaped by activation function σ: $a_1 = \sigma(z_1)$

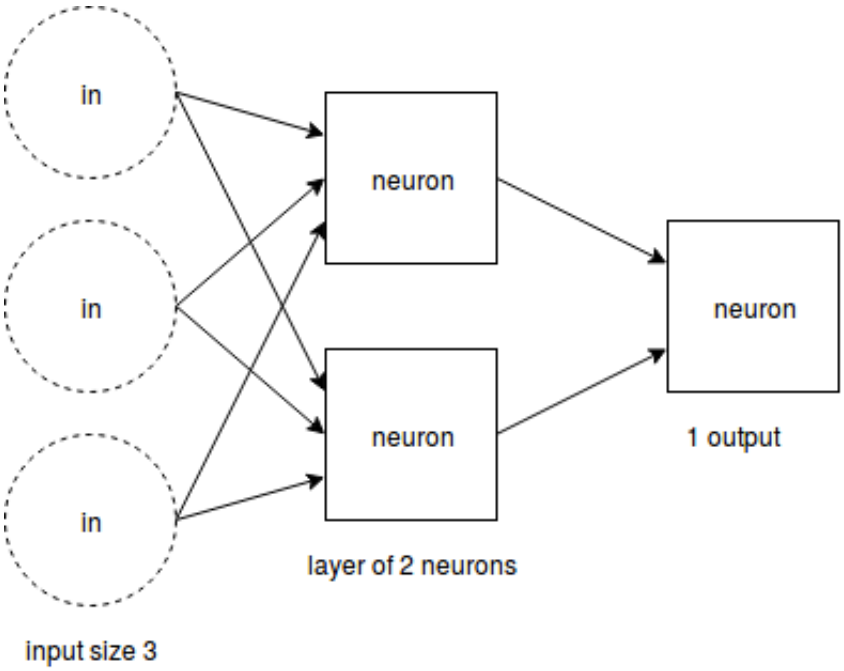
Let's look at that more closely. The w_1 matrix (weights) is the collection of "synapses" of the "neurons": they are the connections the neurons use to pull in data. These synapses can be increased to amplify an incoming variable, set to zero to ignore one, or made negative to invert the inbound signal. During training they are tuned by the neurons to help the NN minimize prediction error.

The biases b_1 are unique to each neuron. They're used by the neuron to adjust what they receive.

The activation function σ, the sigmoid function, causes the neuron to output a binary signal. Unlike a digital computer though, the signal can range *between* 0 and 1 if the neuron is unsure.

```
In [2]: x=np.linspace(-10,10,100)
        plt.plot(x,1/(1 + np.exp(-x)))
        plt.title('Sigmoid activation function')
        plt.show()
```





Dimensions

- Synapses strengthen or weaken incoming variables with their weights
- Biases adjust the sum of these weighted signals
- Neuron does a weighted sum of everything and transforms it with its activation function

The weights are matrices with dimensions $in \times out$, in the size of the data coming in and out the size of the data coming out. out is the number of neurons in the layer, and in is the amount of values each of these neurons is fed during feed-forward. The biases are $1 \times out$, one bias for each neuron.

Above you can see that the input data is 3 values and that each neuron has 3 synapses ($2 \times 3 = 6$ in total). There are 2 neurons in the layer and they produce 2 outputs, one each neuron. There are also 2 biases integrated inside the 2 neurons.

Each layer has only two quantities: how much comes in and how much goes out.

The first aspect of the feed-forward is then the flowing of data through weights, biases, and activation functions.

3.2.3 Neurons working

A neural network has a dual nature: a linear nature at the unit level and a complex non-linear nature at the network level.

The neurons' linear nature helps them perform computations. They each get their own copy of the data to work on. This amazing trick is possible because of matrix multiplication (or dot product). Rows don't mix with other rows, neither columns with other columns.

Matrices are fun Here is a trivial but familiar example. You can see that each neuron (column) does its own thing. Change one of the weight's element to see the effect on the output. The five neurons' outputs are the five elements in the output array.

```
In [3]: x = np.array([[1, 2, 3, 4, 5]])
        weight = np.array([[1, 0, 0, 0, 0],
                           [0, 1, 0, 0, 0],
                           [0, 0, 1, 0, 0],
                           [0, 0, 0, 1, 0],
                           [0, 0, 0, 0, 1]])
        x.dot(weight)

Out[3]: array([[1, 2, 3, 4, 5]])
```

I will give you my own example of using the neurons' synapses to play with the input data. I can perform operations on the inputs separately for each neuron. Look at each weight column vertically.

```
In [4]: x = np.array([[1, 2, 3, 4, 5]])
        weight = np.array([[-1,  0,  0,  0,  0],
                           [ 0,  2,  0,  0,  0],
                           [ 2,  0,  1, -1,  0],
                           [ 0,  0,  0,  0, -1],
                           [ 0,  0,  0,  1,  1]])
        x.dot(weight)

        # I could also just flip the identity matrix horizontally to do this

Out[4]: array([[5, 4, 3, 2, 1]])
```

Matrix algebra allows the NN to perform arithmetic.

The non-linear mixing Data is not mixed within a layer, but it is mixed between them. Neural networks get their power from the interactions of their hidden layers.

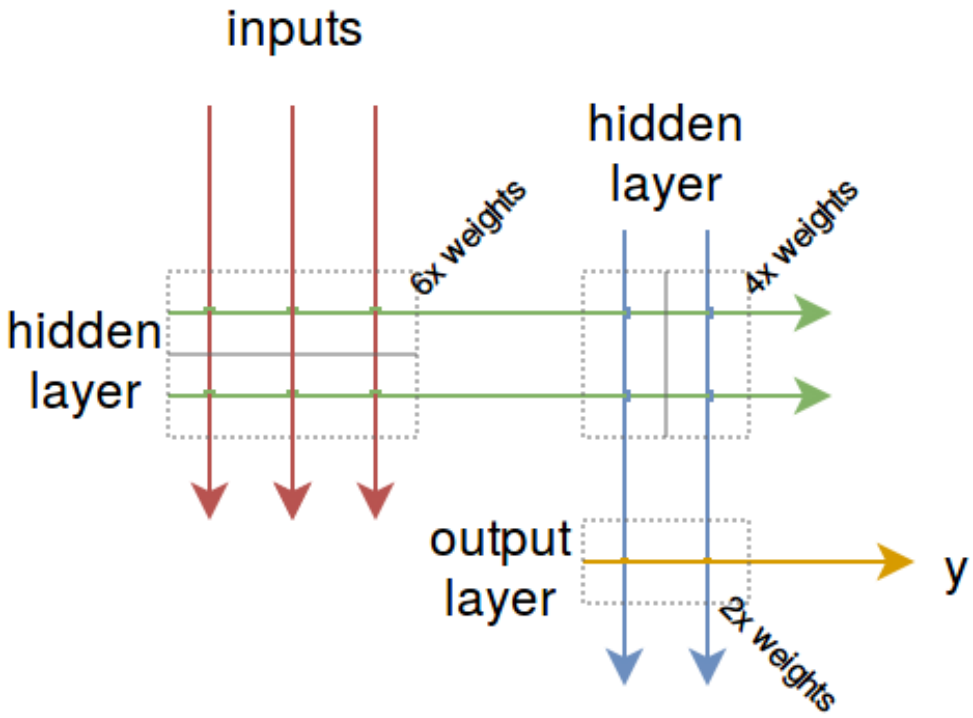
When you take a single-layer model like linear or logistic regression and add layers to it, you get the extra power.

The feed-forward is a mixing of data over many layers of neurons. Each layer expands data into multiple copies and its neurons compress it back into a few outputs. In the diagram above you see 3 units of inputs expanded into 6 synapse signals and then collapsed into 2 output signals. The power of the neural network comes from the fact that the next layer *then copies* these 2 output signals to each of its own neurons, so everything affects everything.

As you can see in the example above, data expands and contracts. It is also generously connected.

Or in other words, feed-forward is like a decision reached by successive committees. The neurons in a layer form a committee that looks at data together, performs analysis, and then summarizes its findings into a small report. Higher committees then analyze this report at a higher level, and so on. The final output layer makes a decision based on the accumulated wisdom of the executive summary it receives: it outputs a single value between 0 and 1.

In the committee example, the office workers use



Colorful

- weights to increase, decrease or invert the importance of data
- biases to make their voices louder or weaker
- activation functions to simply their reports into a range [0,1]

The expansion and contraction of information is repeated multiple times in the neural network. All this mixing allows the neural network to work with very complicated data.

3.2.4 Hiddens layers feed-forwarding

With all that in mind, this is the feed-forward:

1. $z_1 = X \times W_1 + B_1$
2. $a_1 = \sigma(z_1)$
3. $z_2 = a_1 \times W_2 + B_2$
4. $a_2 = \sigma(z_2)$
5. $z_{output} = a_2 \times W_{output} + B_{output}$
6. $a_{output} = \sigma(z_{output})$

By way of comparison, here is Andrew Ng's notation.

1. $a^{(1)} = x$
2. $z^{(2)} = \theta^{(1)} a^{(1)})$
3. $a^{(2)} = g(z^{(2)})$
4. $z^{(3)} = \theta^{(2)} a^{(2)})$

5. $a^{(3)} = g(z^{(3)})$
6. $z^{(4)} = \theta^{(3)} a^{(3)})$
7. $a_{(4)} = h_\theta(x) = g(z^{(4)})$

Let's generate some data. Thanks to the properties of matrix multiplication, I can have 4 rows of input data and these will be processed fully separately, yielding 4 rows of output data.

```
In [5]: x = np.random.random((4,5)) # Four records of 5 variables
        b1 = np.random.random((1,3)) # Bias: 1 x layer_1_size
        w1 = np.random.random((5,3)) # Weight: input_vars x layer_1_size
        b2 = np.random.random((1,2)) # Bias: 1 x layer_2_size
        w2 = np.random.random((3,2)) # Weight: layer_1_size x layer_2_size
        b_out = np.random.random((1,1)) # Bias: 1 x output_size
        w_out = np.random.random((2,1)) # Weight: layer_2_size x output_size
```

Here are the activations of all the layers participating in the feed forward.

```
In [6]: def sigmoid(z):
            return 1/(1+np.exp(-z))

        # First hidden layer, three neurons each give an output
        z1 = x.dot(w1) + b1
        a1 = sigmoid(z1)
        print(a1)
```

```
[[ 0.87557647  0.87428158  0.91214843]
 [ 0.82176249  0.88588182  0.90908784]
 [ 0.83893424  0.89048593  0.90075259]
 [ 0.86566837  0.92157027  0.9329293 ]]
```

```
In [7]: # Second hidden layer, two neurons each give an ouput
        z2 = b2 + a1.dot(w2)
        a2 = sigmoid(z2)
        print(a2)
```

```
[[ 0.85003869  0.7853562 ]
 [ 0.84989888  0.77947092]
 [ 0.85081073  0.7813248 ]
 [ 0.8556034   0.78787513]]
```

```
In [8]: # Output layer: one output for each input record
        z_out = b_out + a2.dot(w_out)
        a_out = sigmoid(z_out)
        print(a_out)
```

```
[[ 0.82023352]
 [ 0.81985266]
 [ 0.8200697 ]
 [ 0.82100979]]
```