Chapter 1: What is a neural network?

A neural network is called such because at some point in history, computer scientists were trying to model the brain in computer code.

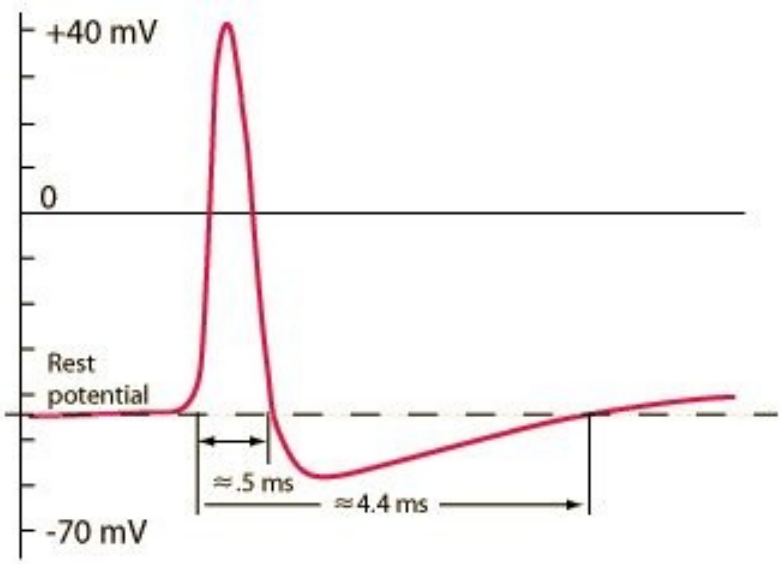
The eventual goal is to create an “artificial general intelligence”, which to me means a program that can learn anything you or I can learn. We are not there yet, so no need to get scared about the machines taking over humanity. Currently neural networks are very good at performing singular tasks, like classifying images and speech.

Unlike the brain, these artificial neural networks have a very strict predefined structure.

The brain is made up of neurons that talk to each other via electrical and chemical signals (hence the term, neural network). We do not differentiate between these 2 types of signals in artificial neural networks, so from now on we will just say “a” signal is being passed from one neuron to another.

Signals are passed from one neuron to another via what is called an “action

potential". It is a spike in electricity along the cell membrane of a neuron. The interesting thing about action potentials is that either they happen, or they don't. There is no "in between". This is called the "all or nothing" principle. Below is a plot of the action potential vs. time, with real, physical units.



These connections between neurons have strengths. You may have heard the phrase, "neurons that fire together, wire together", which is attributed to the Canadian neuropsychologist Donald Hebb.

Neurons with strong connections will be turned "on" by each other. So if one neuron sends a signal (action potential) to another neuron, and their connection is strong, then the next neuron will also have an action potential, would could then be passed on to other neurons, *etc.*

If the connection between 2 neurons is weak, then one neuron sending a signal to

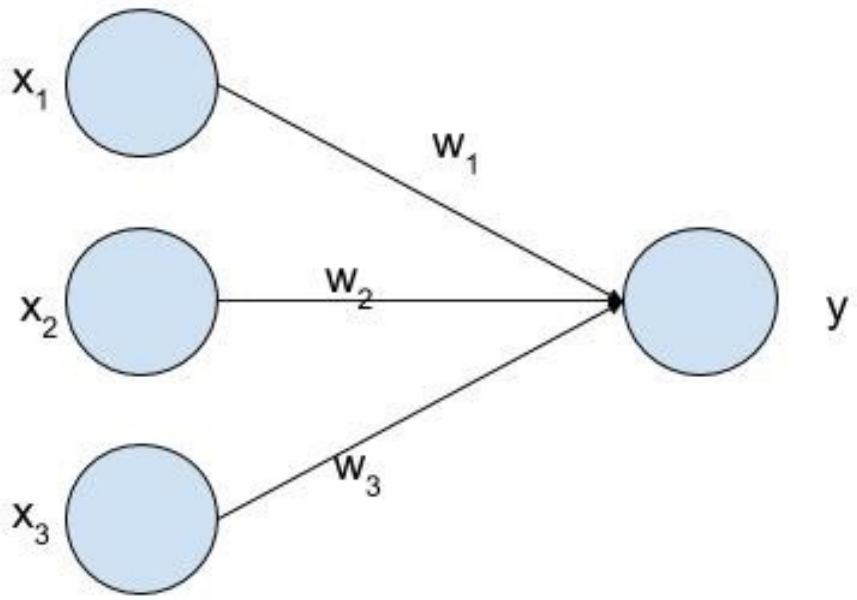
another neuron might cause a small increase in electrical potential at the 2nd neuron, but not enough to cause another action potential.

Thus we can think of a neuron being “on” or “off”. (i.e. it has an action potential, or it doesn’t)

What does this remind you of?

If you said “digital computers”, then you would be right!

Specifically, neurons are the perfect model for a yes *no, true* false, 0 / 1 type of problem. We call this “binary classification” and the machine learning analogy would be the “logistic regression” algorithm.



The above image is a pictorial representation of the logistic regression model. It takes as inputs x_1, x_2, and x_3, which you can imagine as the outputs of other neurons or some other input signal (i.e. the visual receptors in your eyes or the mechanical receptors in your fingertips), and outputs another signal which is a combination of these inputs, weighted by the strength of those input neurons to this output neuron.

Because we're going to have to eventually deal with actual numbers and formulas, let's look at how we can calculate y from x.

$$y = \text{sigmoid}(w_1*x_1 + w_2*x_2 + w_3*x_3)$$

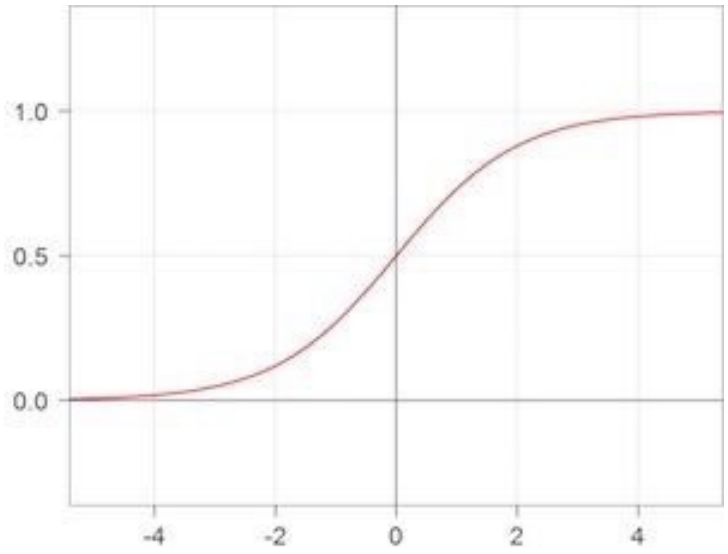
Note that in this book, we will ignore the bias term, since it can easily be included in the given formula by adding an extra dimension x_0 which is always

equal to 1.

So each input neuron gets multiplied by its corresponding weight (synaptic strength) and added to all the others. We then apply a “sigmoid” function on top of that to get the output y. The sigmoid is defined as:

$$\text{sigmoid}(x) = 1 / (1 + \exp(-x))$$

If you were to plot the sigmoid, you would get this:



You can see that the output of a sigmoid is always between 0 and 1. It has 2 asymptotes, so that the output is exactly 1 when the input is + infinity, and the output is exactly 0 when the input is - infinity.

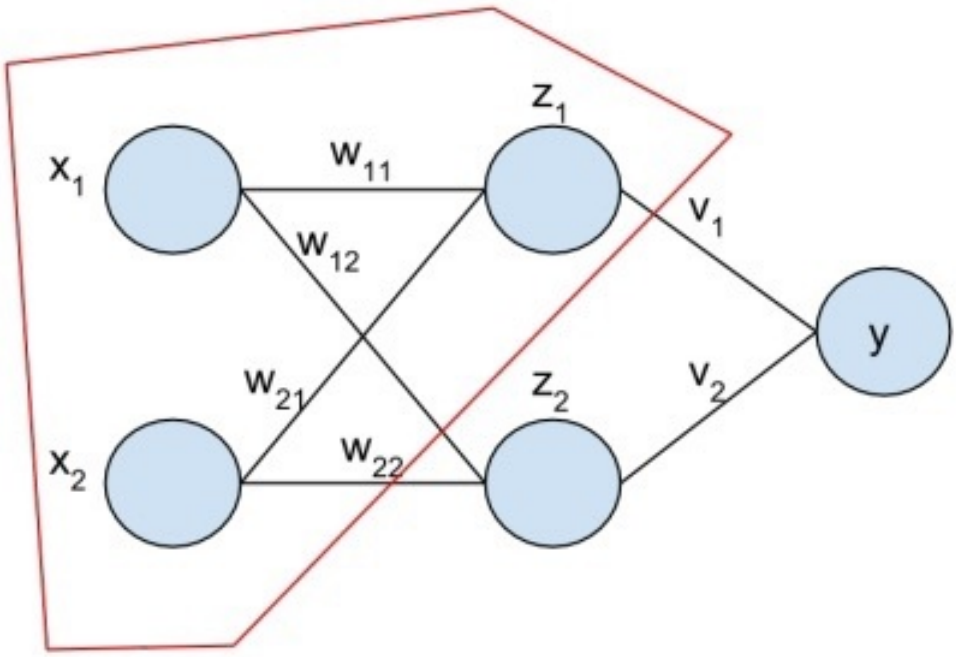
The output is 0.5 when the input is 0.

You can interpret the output as a probability. In particular, we interpret it as the probability:

$$P(Y=1 \mid X)$$

Which can be read as “the probability that Y is equal to 1 given X”. We usually just use this and “y” by itself interchangeably. They are both “the output” of the neuron.

To get a neural network, we simply combine neurons together. The way we do this with artificial neural networks is very specific. We connect them in a feedforward fashion.



I have highlighted in red one logistic unit. Its inputs are (x_1, x_2) and its output is z_1. See if you can find the other 2 logistic units in this picture.

We call the layer of z's the "hidden layer". Neural networks have one or more hidden layers. A neural network with more hidden layers would be called "deeper".

"Deep learning" is somewhat of a buzzword. I have googled around about this topic, and it seems that the general consensus is that any neural network with one or more hidden layers is considered "deep".

Exercise

Using the logistic unit as a building block, how would you calculate the output of a neural network Y? If you can’t get it now, don’t worry, we’ll cover it in Chapter 3.

Chapter 2: Biological analogies

I described in the previous chapter how an artificial neural network is analogous to a brain physically, but what about with respect to learning and other “high level” attributes?

Excitability Threshold

The output of a logistic unit must be between 0 and 1. In a classifier, we must choose which class to predict (say, is this is a picture of a cat or a dog?)

If 1 = cat and 0 = dog, and the output is 0.7, what do we say? Cat!

Why? Because our model is saying, “the probability that this is an image of a cat is 70%”.

The 50% line acts as the “excitability threshold” of a neuron, *i.e.* the threshold at which an action potential would be generated.

Excitatory and Inhibitory Connections

Neurons have the ability when sending signals to other neurons, to send an “excitatory” or “inhibitory” signal. As you might have guessed, excitatory connections produce action potentials, while inhibitory connections inhibit action potentials.

These are like the weights of a logistic regression unit. A very positive weight would be a very excitatory connection. A very negative weight would be a very inhibitory connection.

Repetition and Familiarity

“Practice makes perfect” people often say. When you practice something over and over again, you become better at it.

Neural networks are the same way. If you train a neural network on the same or similar examples again and again, it gets better at classifying those examples.

Your mind, by practicing a task, is lowering its internal error curve for that particular task.

You will see how this is implemented in code when we talk about backpropagation, the training algorithm for a neural network.

Essentially what we are going to do is do a for-loop a number of times, looking at the same samples again and again, doing backpropagation on them each time.

Exercise

In preparation for the next chapter, you'll need to make sure you have the following installed on your machine: Python, Numpy, and optionally Pandas.

Chapter 3: Getting output from a neural network

Get some data to work with

Assuming you don't yet have any data to work with, you'll need some to do the examples in this book. https://kaggle.com is a great resource for this. I would recommend the MNIST dataset. If you want to do binary classification you'll have to choose another dataset.

The data you'll use for any machine learning problem often has the same format.

We have some inputs X and some labels or targets Y.

Each sample (pair of x and y) is represented as a vector of real numbers for x and a categorical variable (often just 0, 1, 2, …) for y.

You put all the sample inputs together to form a matrix X. Each input vector is a row. So that means each column is a different input feature.

Thus X is an N x D matrix, where N = number of samples and D = the dimensionality of each input. For MNIST, D = 784 = 28 x 28, because the original images, which are 28 x 28 matrices, are “flattened” into 1 x 784 vectors.

If y is not a binary variable (0 or 1), you can turn it into a matrix of indicator variables, which will be needed later when we are doing softmax.

So for the MNIST example you would transform Y into an indicator matrix (a matrix of 0s and 1s) where Y_indicator is an N x K matrix, where again N = number of samples and K = number of classes in the output. For MNIST of course K = 10.

Here is an example of how you could do this in Numpy:

```
def y2indicator(y):

N = len(y)

ind = np.zeros((N, 10))
```

```
for i in xrange(N):

ind[i, y[i]] = 1
```

```
return ind
```

In this book, I will assume you already know how to load a CSV into a Numpy array or Pandas dataframe and do basic operations like multiplying and adding Numpy arrays.

Architecture of an artificial neural network

Unlike biological neural networks, where any one neuron can be connected to any other neuron, artificial neural networks have a very specific structure. In particular, they are composed of layers.

Each layer feeds into the next layer. There are no “feedback” connections. (Actually there can be, and these are called recurrent neural networks, but they are outside the scope of this book.)

You already saw what a neural network looks like in Chapter 1, and how to calculate the output of a logistic unit.

Suppose we have a 1-hidden layer neural network, where x is the input, z is the hidden layer, and y is the output layer (as in the diagram from Chapter 1).

Feedforward action

Let us complete the formula for y. First, we have to compute z_1 and z_2.

$$z1 = s(w11*x1 + w12*x2)$$

$$z2 = s(w21*x1 + w22*x2)$$

s() can be any non-linear function (if it were linear, you'd just be doing logistic regression). The most common 3 choices are, 1:

```
def sigmoid(x):

return 1 / (1 + np.exp(-x))
```

Which we saw earlier.

2, the hyperbolic tangent: np.tanh()

And 3, the rectifier linear unit, or ReLU:

```
def relu(x):

if x < 0:

return 0

else:
```

return x

Prove to yourself that this alternative way of writing relu is correct:

```
def relu(x):
return x * (x > 0)
```

This latter form is needed in libraries like Theano which will automatically calculate the gradient of the objective function.

And then y can be computed as:

$$y = s'(v1*z1 + v2*z2)$$

Where s'() can be a sigmoid or softmax, as we discuss in the next sections.

Note that inside the sigmoid functions we simply have the “dot product” between the input and weights. It is more computationally efficient to use vector and matrix operations in Numpy instead of for-loops, so we will try to do so where possible.

This is an example of a neural network using ReLU and softmax in vectorized form:

```
def forward(X, W, V):

Z = relu(X.dot(W))

Y = softmax(Z.dot(V))
```