Neural Networks

In this lecture. . .

- Neural networks are meant to reproduce something like what happens in the brain

- They are a 'messy' way of fitting functions or data

- The parameters are often found by backpropagation

Introduction

A neural network is a type of machine learning that is meant to mimic what happens in the brain.

Signals are received by neurons where they are mathematically manipulated and then passed on to more neurons.

The input signal might pass through several layers of neurons before being output, as a prediction, forecast or classification.

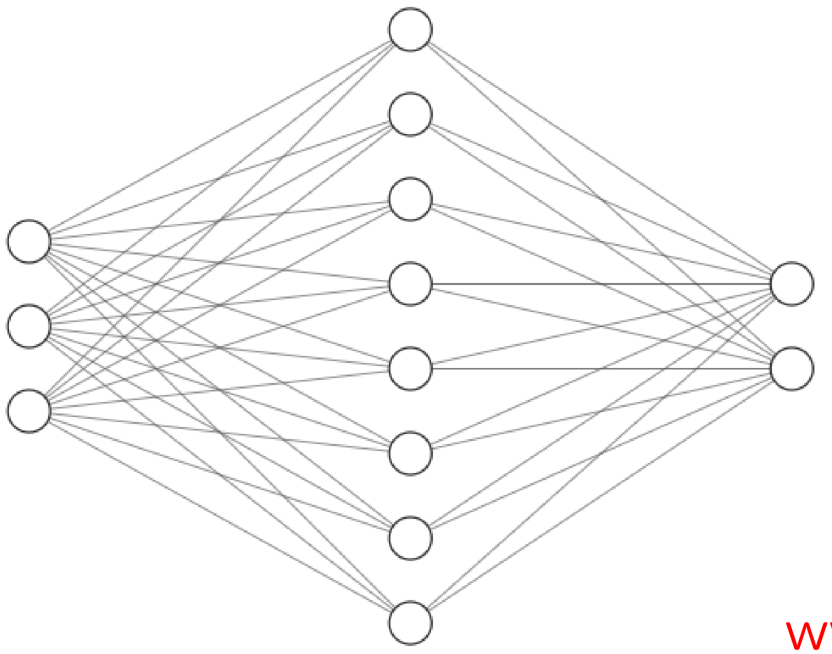
What Is It Used For?

Neural networks are used for

- Modelling complex relationships between inputs and outputs
- Example: Recognising images including handwriting
- Example: Enhancing grainy images
- Example: Translating from one language to another

A Very Simple Network

You will have seen pictures like below, representing a typical neural network. This is a good illustration of the structure of a very simple feedforward network. Inputs go in the left of this picture and outputs come out of the right. In between, at each of the nodes, there will be various mathematical manipulations of the data. It's called **feedforward** because the data and calculations go in one direction, left to right.



In this particular example we have an input being an array or vector with three entries, so three *numerical* quantities.

These are manipulated in the hidden layer of eight nodes before being passed out to an output vector/array with two entries.

This would be an example of trying to predict two quantities from a three-dimensional input.

The outputs will be numerical also but these can still be used for classification.

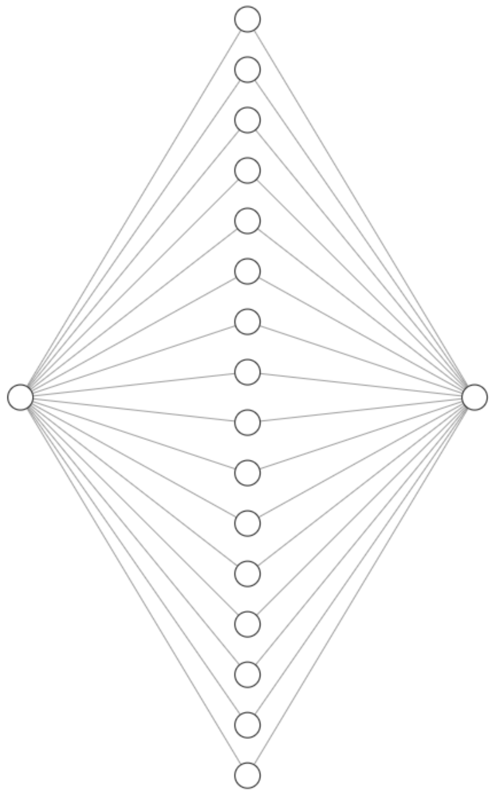
There doesn't have to be a single hidden layer. You can have as many as you want or need, and each with as many nodes as you want or need.

However, the number of input and output nodes will depend on what data you have and what you want to forecast.

The layout of the network is called the **architecture**.

Universal Approximation Theorem

The Universal Approximation Theorem says that, under quite weak conditions, as long as you have enough nodes then you can approximate any continuous function with a single-layer neural network to any degree of accuracy.



This is one of the most important uses for a neural network, function approximation.

But usually our problems are not as straightforward as this implies. Instead of having a single independent variable, the x, we will typically have many.

And instead of a single output, the y, there could be many.

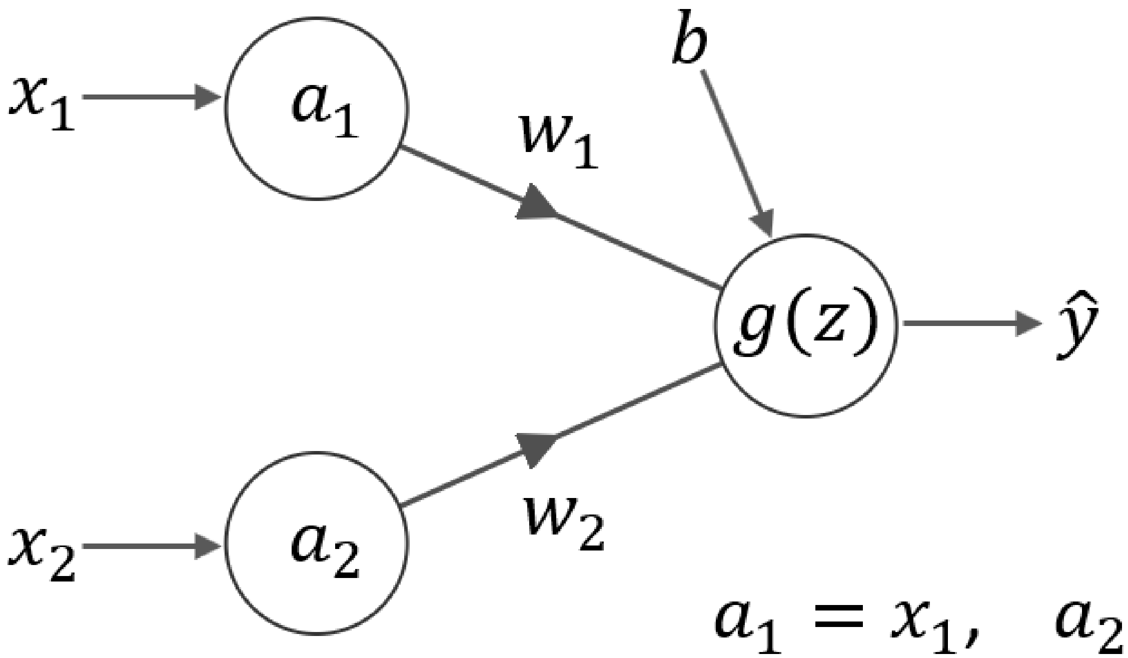
And most importantly instead of having a known function that we want to approximate we have a whole load of data, the inputs and the outputs.

With such complicated problems we'll need the richer structure that you can get by having more than one hidden layer.

Ok, now it's time to tell you what goes on in the hidden layer, what are the mathematical manipulations I've mentioned.

An Even Simpler Network

The figure below shows just about the simplest network, and one that can still be used to describe the manipulations going on.



$$a_1 = x_1, \quad a_2 = x_2$$
$$z = w_1 a_1 + w_2 a_2 + b$$
$$\hat{y} = g(z)$$

Here we have two inputs on the left, x_1 and x_2, and a forecast/output on the right, $\widehat{y}$.

The inputs go into the first nodes: $a_1 = x_1$ and $a_2 = x_2$.

Then each of these node values is multiplied by a **weight**, the ws. And then a **bias**, b, is added:

$$z = w_1 a_1 + w_2 a_2 + b.$$

This result is then acted on by a function $g(z)$ to give the output,

$$\widehat{y} = g(z).$$

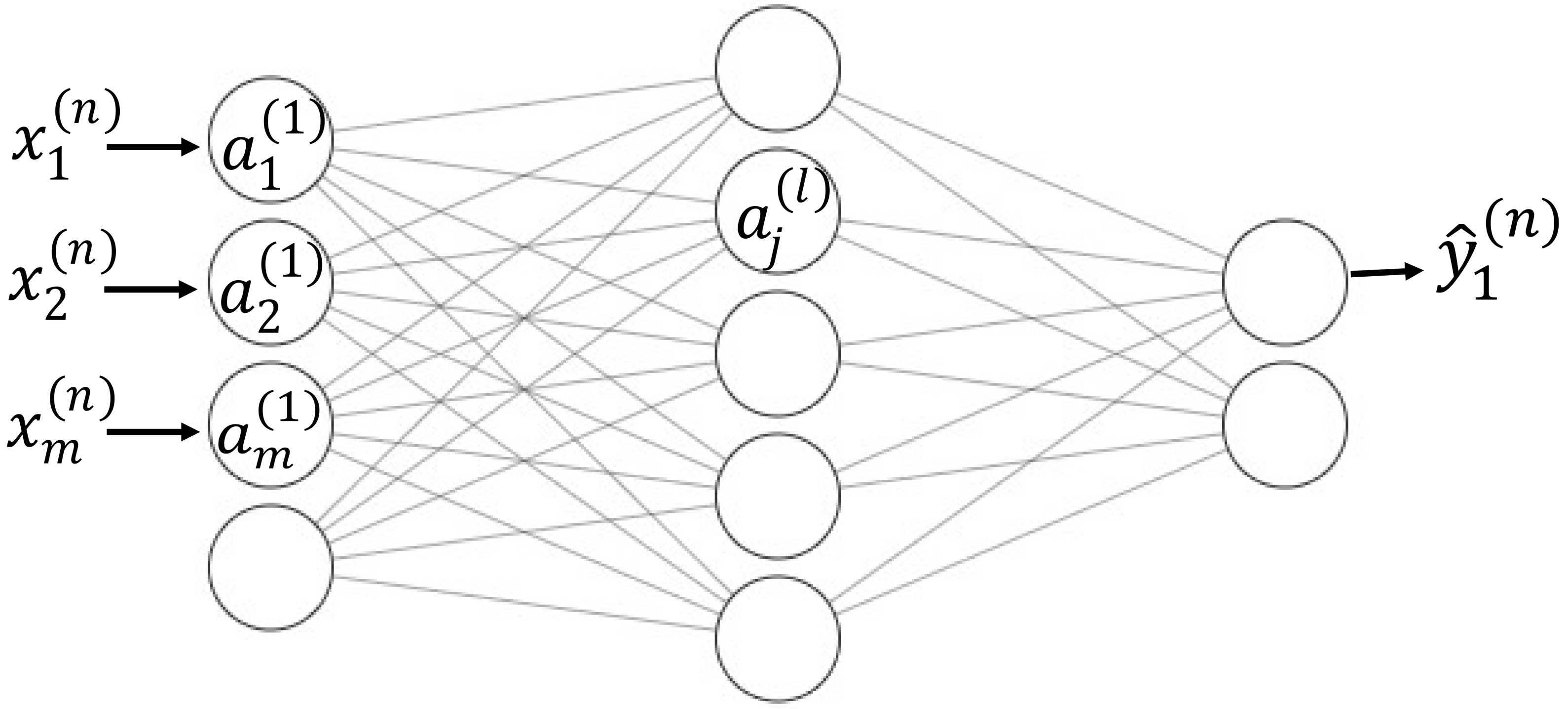
This is a very, very simple transformation of the data.

The function $g(z)$ will be chosen, it is called an **activation** or **transfer function**.

In neural networks we specify the activation function but the weights, the ws, and the bias, b, will be found numerically in order to best fit our forecast output with our given data.

The Mathematical Manipulations In Detail

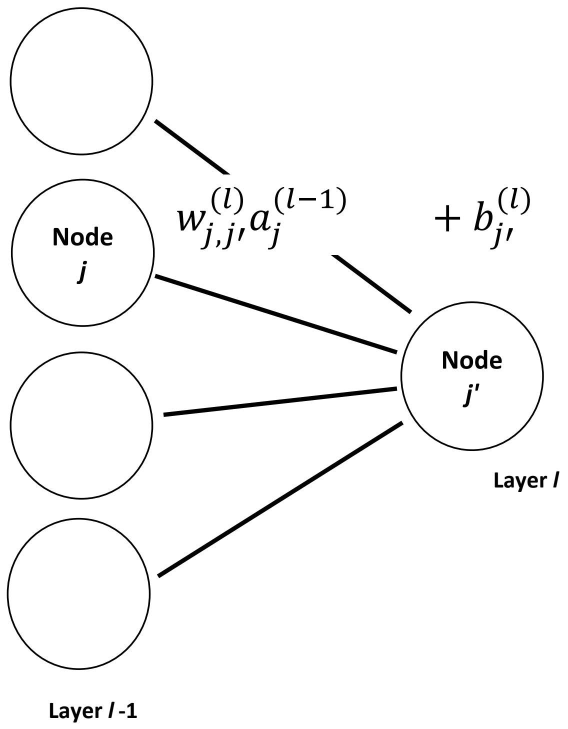
Now let's do that with a larger network, showing all the sub- and superscripts we'll be needing.



Propagation

In going from one layer to the next the first thing that happens is a linear combination of the values in the nodes.

I have labelled the two layers as $l-1$ and l. Also notice that the general node in the left-hand layer is labelled j and one in the right-hand layer, layer l, is labelled j'.



©Paul Wilmott

www.wilmott.com

We want to calculate what value goes into the j' th node of the l^{th} layer.

First multiply the value $a_j^{(l-1)}$ in the j^{th} node of the previous, $(l-1)^{\text{th}}$, layer by the parameter $w_{j,j'}^{(l)}$. Then we add up all these products for every node in layer $l-1$, and finally add another parameter $b_{j'}^{(l)}$.

This is just

$$\sum_{j=1}^{J_{l-1}} w_{j,j'}^{(l)} a_j^{(l-1)} + b_{j'}^{(l)} \tag{1}$$

where J_l means the number of nodes in layer l. I'll call this expression $z_{j'}^{(l)}$.

This is a bit hard to read, such tiny fonts, and anyway it's much easier to write and understand as

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}, \tag{2}$$

with the matrix $\mathbf{W}^{(l)}$ containing all the multiplicative parameters, i.e. the weights $w_{j,j'}^{(l)}$, and $\mathbf{b}^{(l)}$ is called the bias. The bias is just the constant in the linear transformation.

(Sometimes the bias is represented by an extra node at the top of the layer, containing value 1, with lines coming down to the next layer. This way of drawing the structure is exactly equivalent to what we have here.)

The activation function

The activation function gets its name from a similar process in physical, i.e. inside the brain, neurons whereby an electrical signal once it reaches a certain level will 'fire' the neuron so that the signal is passed on. If the signal is too small then the neuron does not fire.

Applying the same idea here we simply apply a function to expressions (1) or (2). Let's call that function $g^{(l)}$. It will be the same for all nodes in a layer but can differ from layer to layer. And we *do* specify this function.

Thus we end up with the following expression for the values in the next layer

$$\mathbf{a}^{(l)} = g^{(l)}(\mathbf{z}^{(l)}) = g^{(l)}\left(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}\right). \tag{3}$$

The function of a vector just means taking the function of each entry.

So a signal is passed from all the nodes in one layer to the next layer where it goes through a function that determines how much of the signal to pass onwards.

And so on, all the way through the network, up to and including the output layer which also has an associated activation function.

This can be interpreted as a regression on top of a regression on top of a . . .

You can see that what comes out of the right is just a messy function of what goes in the left.

I use ‘messy’ in the technical mathematical sense that if you were to write the scalar $\widehat{y}$s explicitly in terms of the scalar xs then it wouldn’t look very pretty.

It would be a very long expression, with lots of sub- and superscripts, and summations. But a function it most definitely is.

Classification problems

Suppose you want to classify fruit. You have peaches, pears, plums, quince, etc. Your raw data for the $\mathbf{x}$s might be numerical quantities representing dimensions, shape, colour, etc. But what will the dependent variable(s) be?

You could have a single dependent y which takes values 1 (for peach), 2 (for pear), etc.

But that wouldn't make any sense when you come to predicting a new out-of-sample fruit.

Suppose your output prediction was $\widehat{y} = 1.5$. What would that mean? Half way between a peach and a pear perhaps? That's fine if there is some logical ordering of the fruit so that in some sense an peach is less than a pear, which is less than a plum, and so on. But that's not the case.

It makes more sense to output a vector $\hat{\mathbf{y}}$ with as many entries as there are fruit.

The input data would have a peach as being $(1, 0, \ldots, 0)^T$, a pear as $(0, 1, \ldots, 0)^T$ and so on. An output of $(0.3, 0.2, \ldots, 0.1)^T$ would then be useful, most likely a peach but with some pear-like features.

Common activation functions

Here are some of the most common activation functions.

Linear function A bit pointless, but

$$g(x) = x.$$

There'll be problems with this activation function when it comes to finding the parameters because its gradient is one everywhere and this can cause problems with gradient descent.

And it also rather misses the essential non-linear transformation nature of neural networks.

Step function/Hard limit

The step function behaves like the biological activation function described above.

$$g(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases} .$$

The signal either gets through as a fixed quantity, or it dies. This might be a little bit too extreme, leaving no room for a probabilistic interpretation of the signal for example. It also suffers from numerical issues to do with having zero gradient everywhere except at a point, where it is infinite. This messes up gradient descent again.

Probably best to avoid.

Positive linear/ReLU function

$$g(x) = \max(0, x).$$

ReLU stands for 'Rectified Linear Units.' It's one of the most commonly used activation functions, being sufficiently non linear to be interesting when there are many interacting nodes. The signal either passes through untouched or dies completely.

Saturating linear function A gentler version of the step function,

$$g(x) = \begin{cases} 0 & x < 0 \\ x & 0 \le x \le 1 \\ 1 & x > 1 \end{cases}.$$

Sigmoid or logistic function

$$g(x) = \frac{1}{1 + e^{-x}}.$$

This is a gentler version of the step function. And it's a function we have found useful previously. It could be a good choice for an activation function if you have a classification problem.

The tanh function can also be used, but this is just a linear transformation of the logistic function.

Softmax function

The softmax function takes an array of K values $(z_1, z_2, \ldots, z_K)$ and maps them onto K numbers between zero and one, and summing to one. It is thus a function that turns several numbers into quantities that can be perhaps interpreted as probabilities.

$$\frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_k}}.$$

It is often used in the final, output, layer of a neural network.

Hidden layer versus output layer

It's worth mentioning that one can be quite flexible in choosing activation functions for hidden layers but more often than not the activation function in the final, output, layer will be pretty much determined by your problem.

The Goal

Our goal is to ultimately fit a function. But I haven't yet said much about the function we are fitting.

Typically it won't be the sine function we'll be seeing in a moment as our first example. It will come from data.

For each input independent variable/data point of features $\mathbf{x}^{(n)}$ there will be a corresponding dependent vector $\mathbf{y}^{(n)}$. This is our training data.

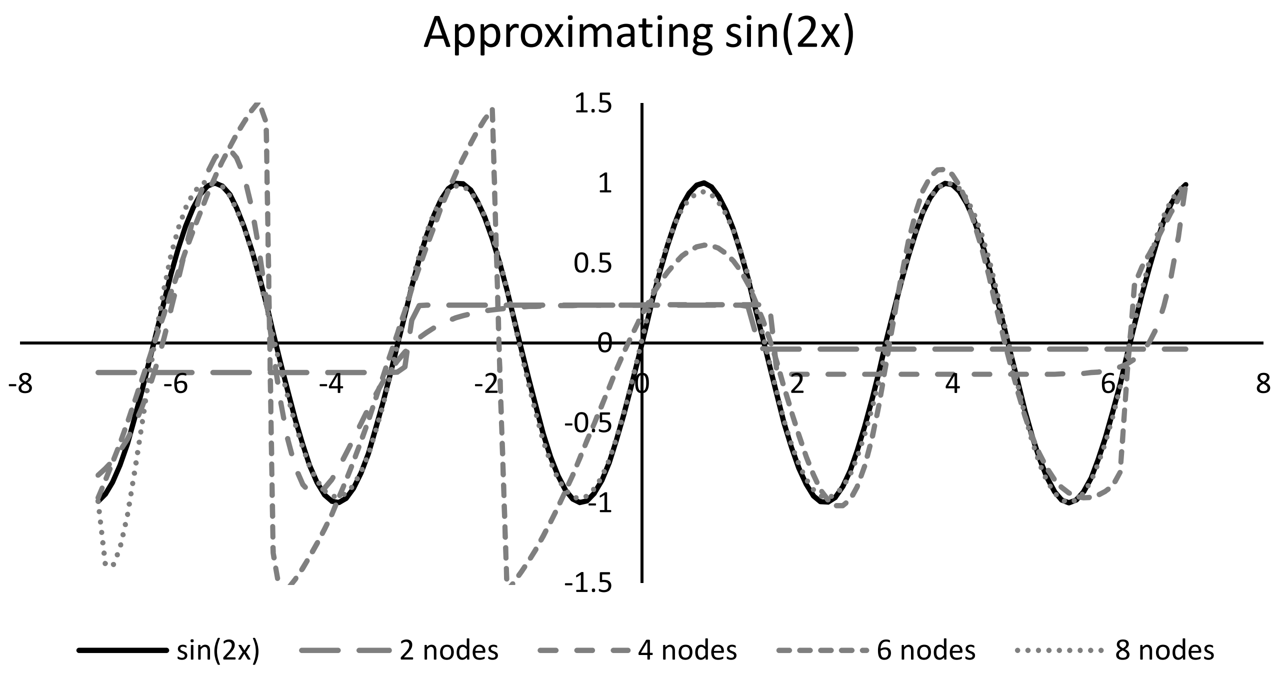
Our neural network on the other hand takes the $\mathbf{x}^{(n)}$ as input, manipulates it a bit, and throws out $\widehat{\mathbf{y}}^{(n)}$. Our goal is to make the $\mathbf{y}^{(n)}$ and $\widehat{\mathbf{y}}^{(n)}$ as close to each other as possible. And we do that by choosing the parameters $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$, for each layer.

This is just a fancy regression.

And so you would rightly expect some discussion of cost functions and numerical methods. And they will indeed be coming up soon.

Example: Approximating a function

Let's look at the Universal Approximation Theorem in action. Take a neural network with one hidden layer and a variety of numbers of nodes in that layer and use it to fit $\sin(2x)$. In the figure we see how well we can fit this function using two, four, six and eight nodes. I used a sigmoidal activation function.



Obviously the more nodes we have the better. With eight nodes the fit is excellent.

How did I derive these results?

How did I find all the weights and biases?.

First we need to decide on a cost function.

Cost Function

As in simpler forms of regression we need to have a measure of how good a job our algorithm is doing at fitting the data. Thus we need a cost function.

A common one, and the one I used in the sine fitting above, is ordinary least squares. The cost function is then

$$J = \frac{1}{2} \sum_{n=1}^{N} \sum_{k=1}^{K} \left(y_k^{(n)} - \widehat{y}_k^{(n)} \right)^2 . \tag{4}$$

The notation is obvious, I hope. $y_k^{(n)}$ is the dependent data for the nth data point, the k representing the kth node in the output vector, and $\widehat{y}_k^{(n)}$ is similar but for the forecast output.

(In the above sine example we only had one output so $K = 1$.)

Classification

In classification problems we still associate our classes with numbers or vectors. If we are labelling emails as spam or not we might label non-spam emails as 0 and spam emails as 1. That would require a single output.

If we have types of animal such as mammal, reptile, amphibian, etc. then we use the vector approach, using a vector with dimension the same as the number of classes, just as described above for fruit, $(1, 0, \ldots, 0)^T$, etc.

The cost function commonly used for such an output is, for a binary, yes/no, classification

$$J = -\sum_{n=1}^{N} \left(y^{(n)} \ln \left(\widehat{y}^{(n)} \right) + (1 - y^{(n)}) \left(1 - \ln \left(\widehat{y}^{(n)} \right) \right) \right).$$

Or if we have three or more classes then we have to sum over all of the K outputs, K being the number of classes:

$$J = -\sum_{n=1}^{N}\sum_{k=1}^{K}\left(y_k^{(n)}\ln\left(\widehat{y}_k^{(n)}\right) + (1 - y_k^{(n)})\left(1 - \ln\left(\widehat{y}_k^{(n)}\right)\right)\right). \quad (5)$$

To these cost functions could be added a regularization term of the form

$$\frac{\lambda}{2}|\mathbf{W}|^2.$$

So we now have something to minimize, but how do we do that numerically?

Backpropagation

We want to minimize the cost function, J, with respect to the parameters, the components of $\mathbf{W}$ and $\mathbf{b}$.

To do that using gradient descent we are going to need the sensitivities of J to each of those parameters. That is we want

$$\frac{\partial J}{\partial w_{j,j'}^{(l)}} \quad \text{and} \quad \frac{\partial J}{\partial b_{j'}^{(l)}}.$$

If we can find those sensitivities then we can use a gradient descent method for finding the minimum.

But this is going to be much harder here than in any other machine-learning technique we have encountered so far. This is because of the way that those parameters are embedded within a function of a function of a. . .

Too messy for an introductory lecture!

Example: Character recognition

Now for a meaty example. I am going to use a neural network to recognise handwritten characters. This is a good, robust test of the technique. It's also relatively easy, and hence commonly found in text books, because of the ease of access to data and there is sample Python code all over the internet.

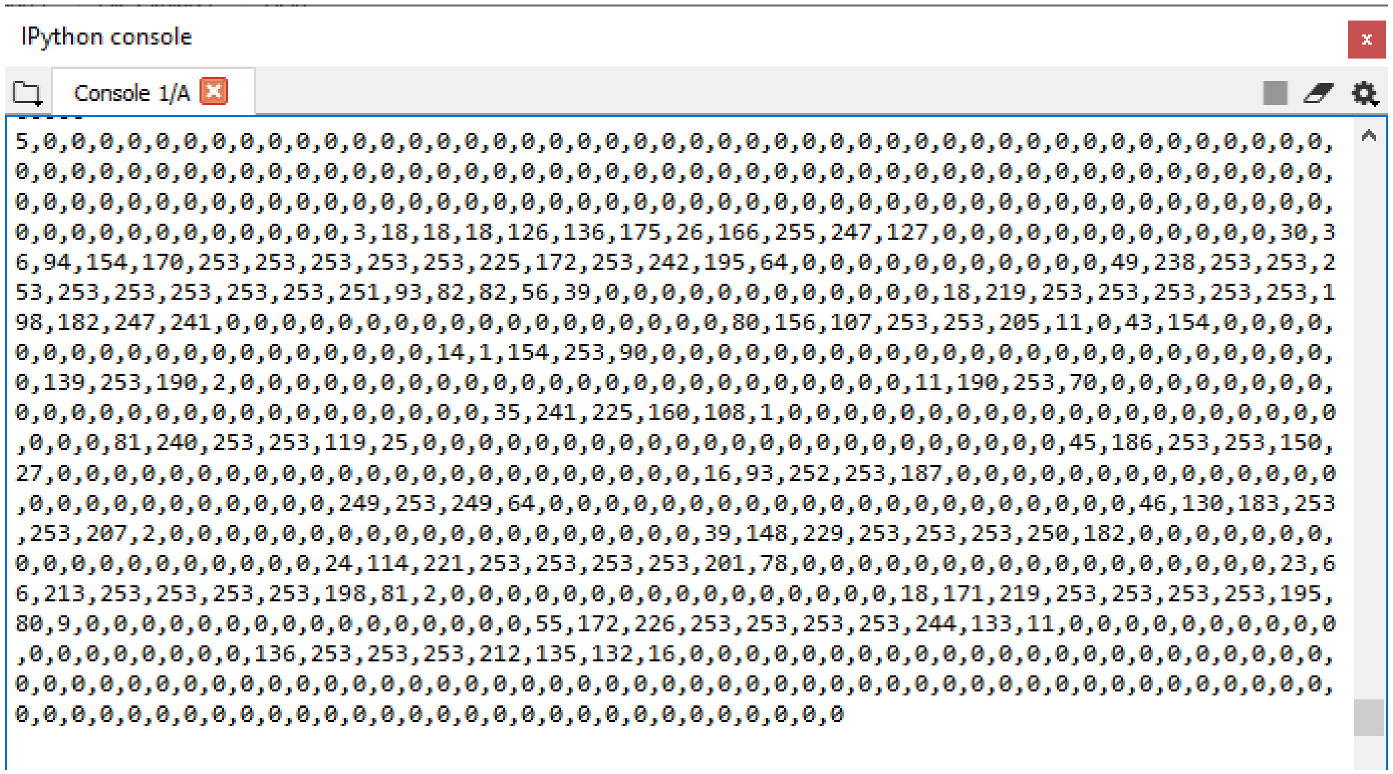
The inputs, the xs, are just many, many examples of handwritten numbers from 0 to 9. Each of these is represented by a vector of dimension 784. That just means that the digits have been pixelated in a square of 28 by 28. Each entry is a number between 0 and 255, representing how 'grey' each pixel is.

And where did I get this data from? There is a very famous data set, the Modified National Institute of Standards and Technology (MNIST) data base of 60,000 training images and 10,000 testing images.

These are samples handwritten by American Census Bureau employees and American high school students. Easy-to-read versions of this data can be found at:

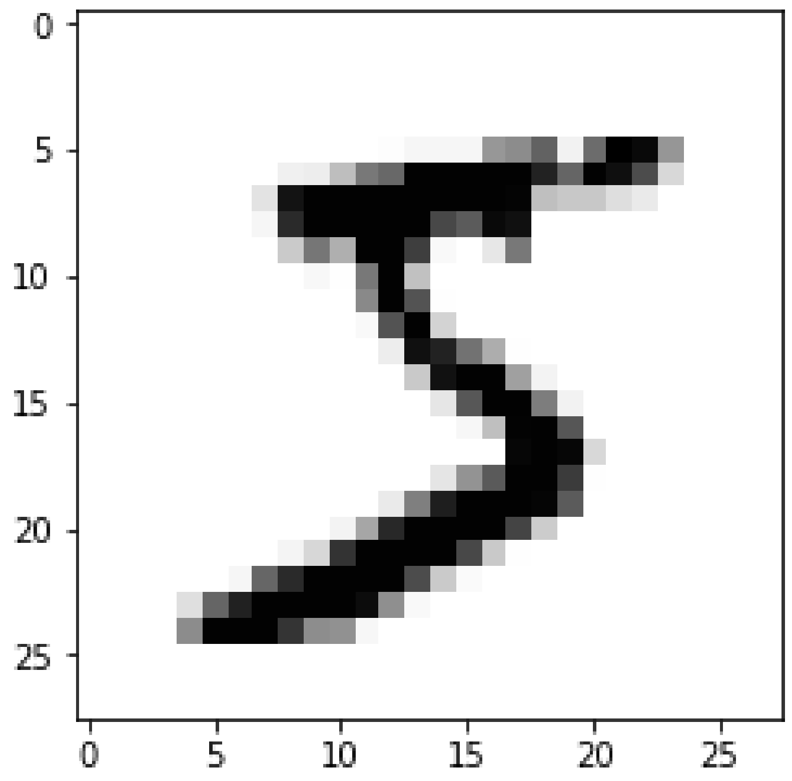
`http://yann.lecun.com/exdb/mnist/`.

Here's one example below, this is the first line of the raw MNIST training data. It is interpreted as follows. The first number in the top left corner, here '5,' is the number represented. The rest of the numbers are the greyness of each pixel.



```
5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,3,18,18,18,126,136,175,26,166,255,247,127,0,0,0,0,0,0,0,0,0,0,0,0,30,3
6,94,154,170,253,253,253,253,253,225,172,253,242,195,64,0,0,0,0,0,0,0,0,0,0,0,49,238,253,253,2
53,253,253,253,253,253,251,93,82,82,56,39,0,0,0,0,0,0,0,0,0,0,0,0,18,219,253,253,253,253,253,1
98,182,247,241,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,80,156,107,253,253,205,11,0,43,154,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,14,1,154,253,90,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,139,253,190,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,11,190,253,70,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,35,241,225,160,108,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,81,240,253,253,119,25,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,45,186,253,253,150,
27,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,16,93,252,253,187,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,249,253,249,64,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,46,130,183,253
,253,207,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,39,148,229,253,253,253,250,182,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,24,114,221,253,253,253,253,201,78,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,23,6
6,213,253,253,253,253,198,81,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,18,171,219,253,253,253,253,195,
80,9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,55,172,226,253,253,253,253,244,133,11,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,136,253,253,253,212,135,132,16,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

And this is what that ‘5’ looks like:



The network that I use has 784 inputs, just one hidden layer with a sigmoidal activation function and 50 nodes, and 10 outputs. Why 10 outputs and not just the one? After all we are trying to predict a number.

The reason is obvious, as a ‘drawing’ it is not possible to say that, say, a 7 is mid way between a 6 and an 8. So this is a classification problem rather than a regression.

Training And Testing

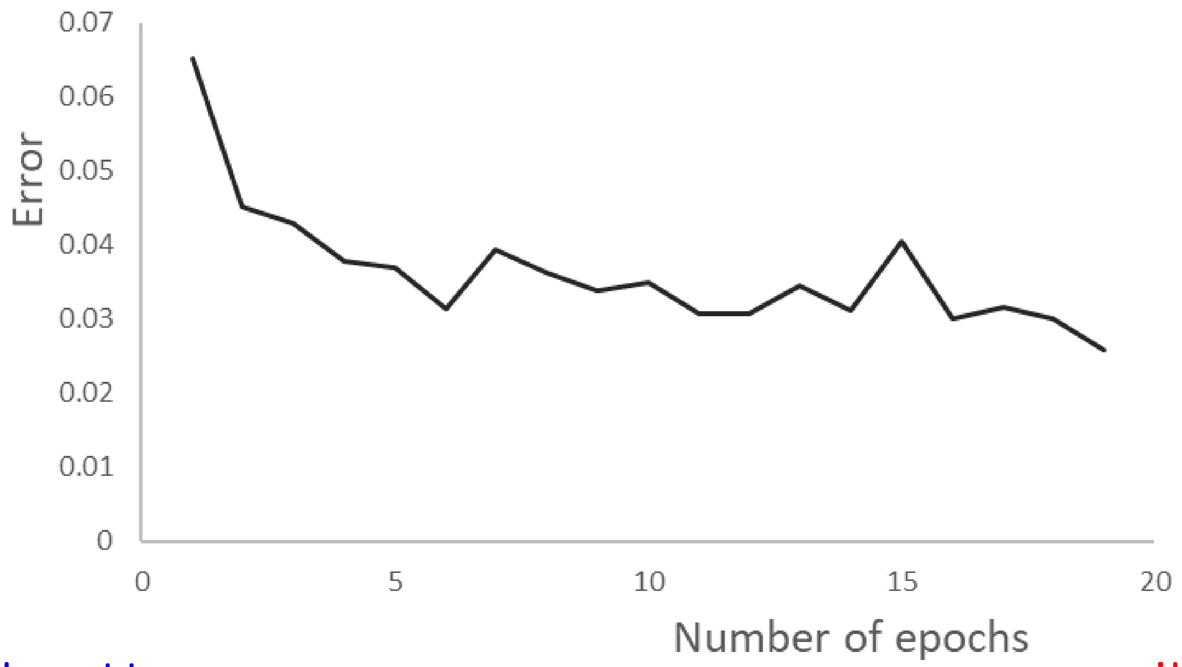
The neural network is trained on the data by running each data point through the network and then updating the weights and biases using the backpropagation algorithm and stochastic gradient descent.

If we run all the data through once that is called an epoch. It will give us values for the weights and biases. Although the network has been trained on all of the data the stochastic gradient descent method will have only seen each data point once. And because of the usually small learning rate the network will not have had time to converge.

So what we do is give the network another look at the data. That would then be two epochs. Gradually the weights and biases move in the right direction, the network is learning. And so to three, four, and more epochs.

The error (measured by the fraction of digits that are misclassified) decreases as the number of epochs increases.

It will typically reach some limit beyond which there is no more improvement. This convergence won't be monotonic because there will be randomness in the ordering of the samples in the stochastic gradient descent.



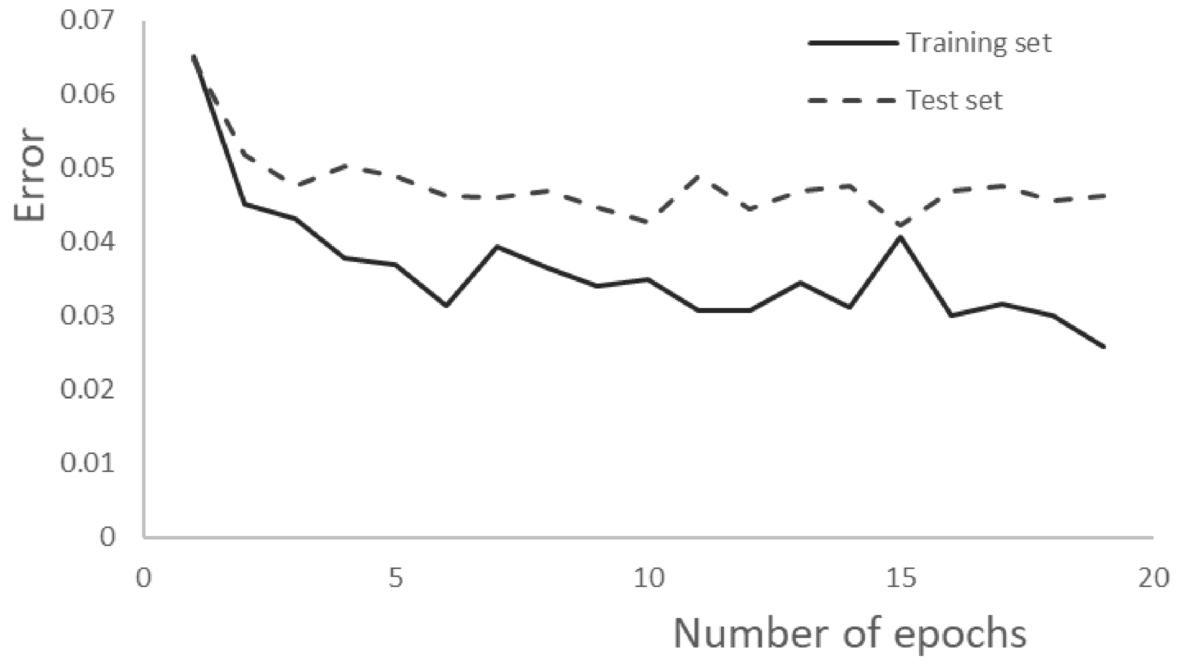
After 19 epochs we are getting a 97.4% accuracy.

So the network has learned, the error is decreasing. But what has it learned? Has it learned the right thing?

We want the network to learn to recognise handwritten digits, but we don't want it to memorize the data we have given it. We don't want to overfit. That's why we hold back some data for testing purposes. And obviously the data we hold back will be a random subset of the entire data set.

How well does the trained network cope with the test data?

Clearly the network is not doing so well on the test data, with only a 95% accuracy. That's almost double the error on the training data. But it's still pretty good for such a simple network.



In practice one would conduct further experiments, varying the number of nodes in the hidden layer and also varying the number of hidden layers.

As a rule the more neurons you have then the more epochs you'll need for training.

Also with more neurons you often find that for the test data the error gets worse with increasing number of neurons. This is a sign that you have definitely overfitted the data. And it's really easy to overfit if you have a lot of neurons.

Once one has an idea of accuracy versus architecture one can then look at speed of prediction for new samples. The time of training is not necessarily important but if speed of prediction is important then one will need an architecture that is fast. It's very easy to estimate the time taken for an architecture. For example suppose one has 784 inputs, 50 nodes in one hidden layer and ten outputs then the time taken will be proportional to

$$784 \times 50 + 50 \times 10 = 39,700.$$

But if we had, say, 784 inputs, one hidden layer of 30, another of 20, and ten outputs then the time would be proportional to

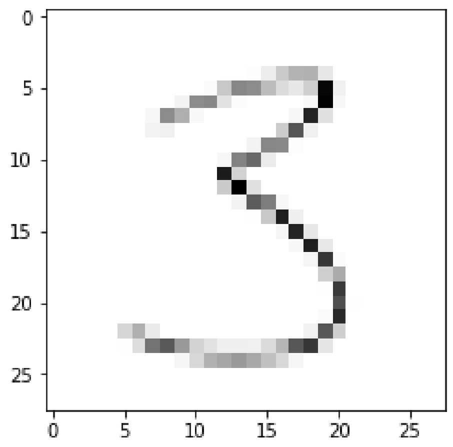
$$784 \times 30 + 30 \times 20 + 20 \times 10 = 24,320.$$

The same number of nodes, but only two thirds of the time.

(This is assuming that different activation functions all take roughly the same time to compute.)

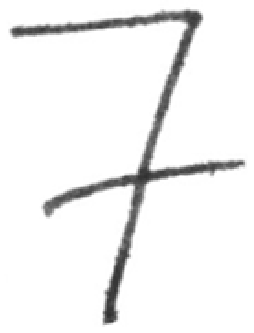
My digits

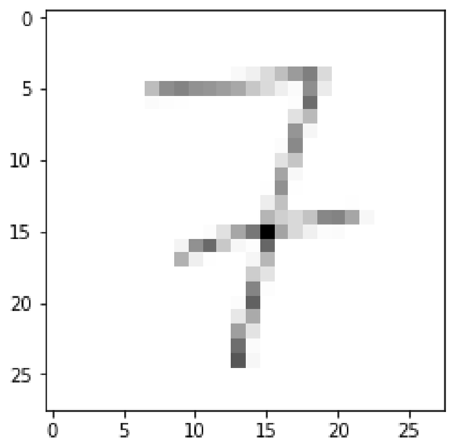
I showed a few of my handwritten digits to the trained network. Below is my number 3 together with the digitised version. It got this one right.



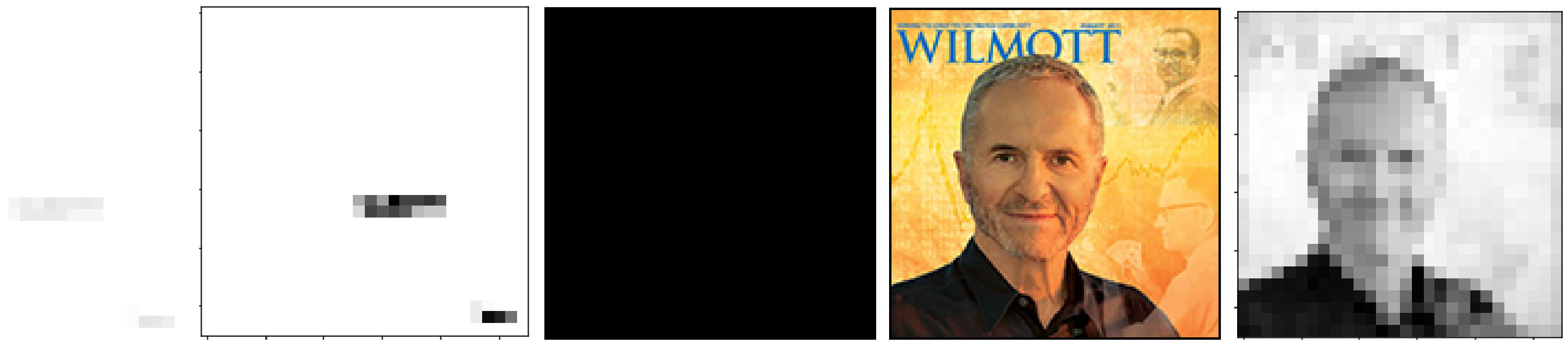
Below is my seven, it got this one wrong. Perhaps because it's European style, with the bar.

The output from the network is a ten-dimensional vector, and by looking at the entries we can see what the network thinks of my handwriting. This output suggests that the network sees my number 7 as 84.2% number 2, 6.8% number 7 and 5.7% number 4. You can see where it's coming from.





Often one sees tests of character-recognition networks using input noise to see which number is forecast. I thought I'd try something similar but different.



Seen as 5, 0 and 3!

More Architectures

Autoencoder

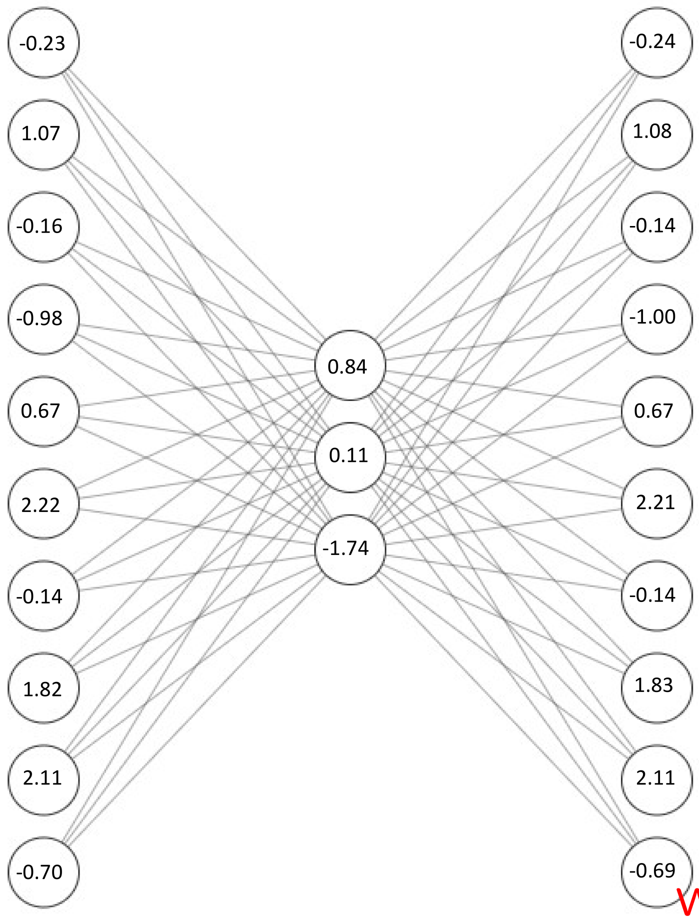
The autoencoder is a very clever idea that has outputs the same as the inputs. Whaaat?

The idea is to pass the inputs through a hidden layer (or more than one) having significantly lower dimension than the input layer and then output a good approximation of the inputs. It's like a neural network version of Principal Components Analysis in which the original dimension of the data is reduced (to the number of nodes in the hidden layer).

This is clearly an unsupervised technique because we don't tell the network what makes up the reduced dimensions.

Training is done exactly as described above, just that now the ys are the same as the xs.

You can see the idea illustrated in this figure.



The data goes in the left-hand side (the figure just shows one of the samples). It goes through the network, passing through a bottleneck, and then comes out the other side virtually unchanged. You'll see in the figure that the output data is slightly different from the input since some information has been lost.

But we have reduced the dimension from ten to three in this example.

Now if you want to compactly represent a sample you just run it through the autoencoder to the bottleneck and keep the numbers in the bottleneck layer (here the 0.84, 0.11, -1.74). Or if you want to generate samples then just put some numbers into the network at the bottleneck layer and run to the output layer.

Summary

Please take away the following important ideas

- Neural networks were originally designed to represent the firing of neurons in the brain

- They can be used as complicated regression tools

- Finding the parameters is harder than in many other forms of machine learning