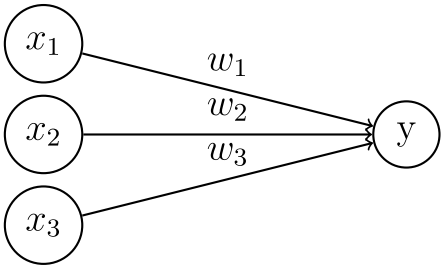
A non-linear classifier.

Obviously one severe limitation of perceptron and other linear classifiers is that it only works for problems that are in some sense linear. There are lots of ways to try to address this, the *kernel trick* for example does a non-linear transformation to the data before running the classifier and relies, very roughly, on the idea that in higher dimensions problems are more likely to be linear. The approach we are going to look at is the neural network[1].

We can start with a simple insight from neuroscience; the perceptron only has two layers, an input and an output, the brain is much more complicated and the visual signal coming in to your eyes goes through a series of successive sets of neurons before you decide you are now reading the word 'lion', or whatever. This makes you think it might be a good idea to add more layers. In the simplest case that leads to a multilayer perceptron or MLP. We will look first at how the MLP might work to solve classification problems and then discuss how it might be trained.

We will start with the simple case of a binary classification, so we imagine the output gives the probability the input belongs to class a, for simplicity the input will have three dimensions in the diagram, of course it is easy to change this. The logistic classifier we looked at before is something like



The idea is that this represents

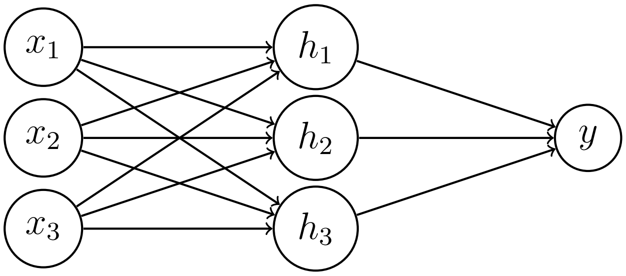
$$y = \phi(\sum_j w_j x_j + b) \tag{1}$$

where ϕ in this case will be the logistic function: $\phi(r) = \sigma(r)$ and the parameters are $\theta = (w_1, w_2, b)$, in this context the ws are called *weights* and the b is a *bias*. Now y is a function of the input $\mathbf{x} = (x_1, x_2)$: you could write $y(\mathbf{x}; \theta)$

[1] It is possible that neural networks are just a complicated way of doing the kernel trick, that's not a discussion for here though!

where the semi-colon is seperating the variables, which are different for each data point and the parameters, which are adjusted during learning. We have seen to make $y(\mathbf{x})$ as close as possible to $p(a)$, but, as we keep saying, this assumes that this mapping, from data values to probability, can be sensibly done using a linear function.

An MLP, inspired by the brain, fixes this by adding more layers, here is what it would look like with one extra *hidden* layer.



Now the function is more complicated, we have

$$h_i = \phi(\sum_j w_{ij}^1 x_j + b_i^1) \tag{2}$$

and, almost as before

$$y = \sigma(\sum_j w_j^2 h_j + b^2) \tag{3}$$

where, since we want y to be a probability, we continue to use a logistic function for the activation. We have a wider choice for $\phi(r)$, a common choice is the ReLU:

$$\phi(r) = \left\{ \begin{array}{ll} r & r \geq 0 \\ 0 & \text{otherwise} \end{array} \right. \tag{4}$$

Why this is a good choice is complicated, it is easy to suggest it has nice properties, but lots of different possibities have other, similar or competing, nice properties and so, for now, the answer is that it seems to work well and robustly.

As a sort of aside, we can write this in matrix form:

$$\mathbf{h} = \phi.(W^1\mathbf{x} + \mathbf{b}) \tag{5}$$

and

$$y = \sigma.(\mathbf{w} \cdot \mathbf{h} + b^2) \tag{6}$$

I have used the ‘broadcast’ notation from Julia, if $f : \mathbf{R} \to \mathbf{R}$ is a function defined on scalars then we lift this to a function on vectors componentwise:

$$f.(\mathbf{x}) = [f(x_1), f(x_2), \ldots, f(x_n) \tag{7}$$

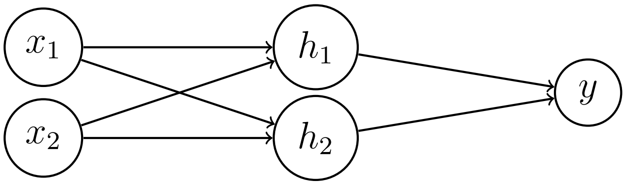
Mostly this is just me being annoying since I spent time with people who worry too much about notation, most people don't bother with the dot; it is important not to get too hung up on notation, but also not to be too lazy about it either!

The main thing is we still have a function $y(\mathbf{x};\theta)$ but it is a much more complicated function and has lots more parameters

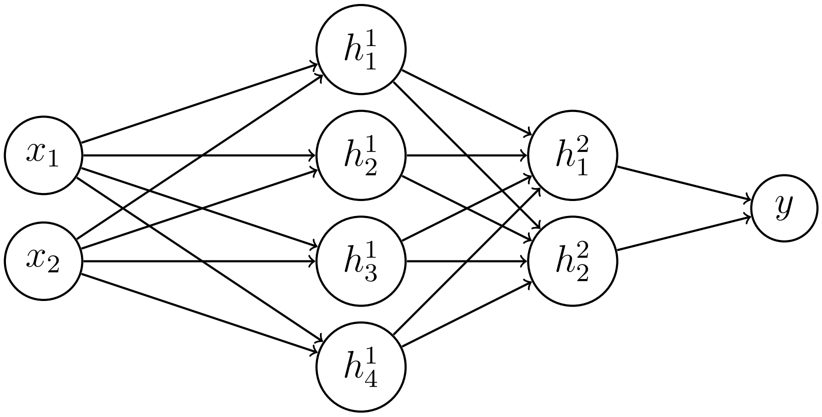
$$\theta = (W^1, \mathbf{b}^2, \mathbf{w}^2, b) \tag{8}$$

In the example here it has, without worrying too much about independence, 9+3+3+1=16 parameters when the perceptron has only four. The hope is that this allows a much richer classification and the straight line or hyperplane is replaced by something more complicated.

As a simple example lets try to classify points in two dimensions, the b points are more than $r = 1$ from the center, the a are less. This would not be possible with a linear classifier and so I tried this



with 400 points, half red and half blue. It didn't seem to work either, the functions it could produce weren't complicated enough, so I expanded the network further to this:



The two hidden layers use a ReLU activation.

With this MLP, with two hidden layers, the classifier works, in Fig. 1 I have plotted y, the estimated probability that a point is from the a class, as a function of position. In other words I randomly produce 200 points whose distance from the center is less than one, the a points, and another 200 points where the distance is greater than one, the b points. I train the neural network to classify these points, with y near one corresponding to an a point, by training the neural network I mean fixing the values of the parameters θ, in this case this is all the weights for the links from the input to the first hidden layer, $2 \times 4 = 8$ of these, the biases for the first hidden layer, the weights between the hidden layers, the biases for the second hidden layer and, finally, the weights from the second hidden layer to the output and the bias for the output. This makes $8 + 4 + 8 + 2 + 2 + 1 = 25$ parameters in all. How I fix these values is the subject of the next note! Finally, to make the picture I plot a heatmap for the map from $\mathbf{x}$, the input, to $\mathbf{y}$, the output. As you can see it is a pretty good approximation of the circle.

It is interesting to experiment with the number and size of the layers; these numbers which aren't fixed by training are often called *meta-parameters*. A $2 \times 8 \times 4 \times 2 \times 1$ MLP, for example, with three hidden layers, gives a rounder decision boundary and the $2 \times 8 \times 4 \times 4 \times 2 \times 1$ MLP gives a very crisp decision boundary. This is shown in Fig. 2.

More than two classes

So far we have looked at examples with just two classes and have used an MLP with a single output neuron, with a high input associated with one class and low with the other. Obviously this is not ideal; we'd like more than one class. The way to do this is to have an output node for each class:

$$y_i = \sum w_{ij} h_j + b_i \tag{9}$$

where the h_j is supposed to represent the preceeding hidden layer, in practice of course there will usually be more than one hidden layer so there might be other indices or labels on the various variable. Now you will notice that this is a linear layer. We do want to use an activation function to convert the y_is into probabilities so that we go from y_i to p_i, the estimated probability that the input corresponds to the ith class. However, these probabilities must add to one:

$$p_1 + p_2 + \ldots + p_c = 1 \tag{10}$$

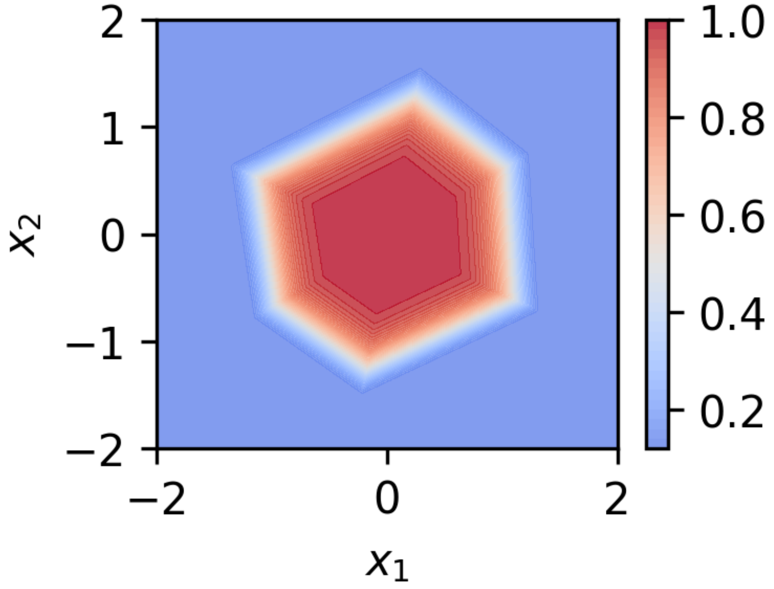


Figure 1: The estimated probability of a point belonging to class a using the MLP with two hidden layers, what could be called a $2 \times 4 \times 2 \times 1$ MLP since the layers have sizes two for the input, four for the first hidden layer, two for the second and one for the output. The a points all lie in the circle of radius one and it is clear the MLP has approximated this decision boundary with a hexagon. The code is available in the github and if you play with it you'll see that increasing the number of points will give a more regular hexagon.

where there are c classes. This means that the non-linearity applied to specific y_i must involve all the y_is at once. The usual solution to this complication is to use the *softmax* function:

$$p_i = \sigma(\mathbf{y})_i = \frac{e^{y_i}}{\sum_j e^{y_j}} \tag{11}$$

It is easy to see that this gives values between zero and one and that the probabilities add to one.

We will look at all this more in the next set of notes when we consider training, but for now lets look at an example, in Fig. 3 we see the p_i in an example with a two-dimensional input and three classes; for it to work, that

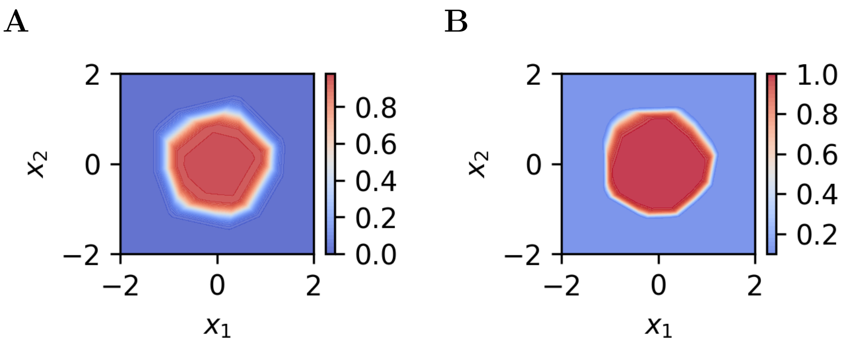


Figure 2: The estimated probability of a point belonging to class a using more complicated MLP, in **A** with three hidden layers giving a $2 \times 8 \times 4 \times 2 \times 1$ and in **B** with four hidden layers giving $2 \times 8 \times 4 \times 4 \times 2 \times 1$.

is for the MLP to be able to learn the decision boundaries, a rather large MLP is used: $2 \times 16 \times 8 \times 4 \times 3$, but with this architecture it succeeds.

Obviously we haven't looked at training yet, that comes next. Another thing to note is that we have only used data sets where there are no 'difficult' points, outliers, or mistakes or just odd twists and turns in the decision boundary; the ability of neural networks to deal reasonably well with this sort of problem is part of what makes them so effective.

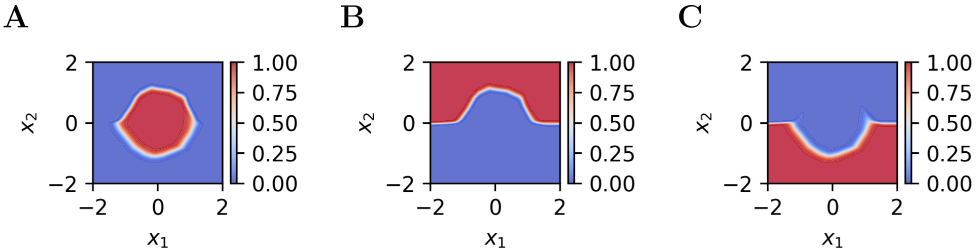


Figure 3: The estimated probability of a point belonging to class each of three classes, with **A**, **B** and **C** each corresponding to one class. Here the input is two-dimensional, points in the first class are all inside a unit circle, the second class has points outside the circle with $x_2 > 0$ and the third class has points outside the circle but with $x_2 < 0$, For training there were 160 of each point type.