SEMT20003, Part 4: Neural Networks

Laurence Aitchison

Most of the content this week is in the Colab notebooks. But there's a few pages here introducing neural networks.

So far, we have worked with linear functions, of the form,

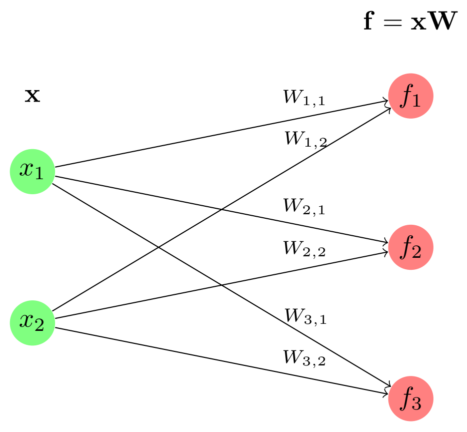
$$\mathbf{f}(\mathbf{x}) = \mathbf{x}\mathbf{W}. \tag{1}$$

And we used this function as the prediction in linear regression (with a squared error loss function), or as the logits in classification (with a maximum-likelihood objective / cross-entropy loss function).

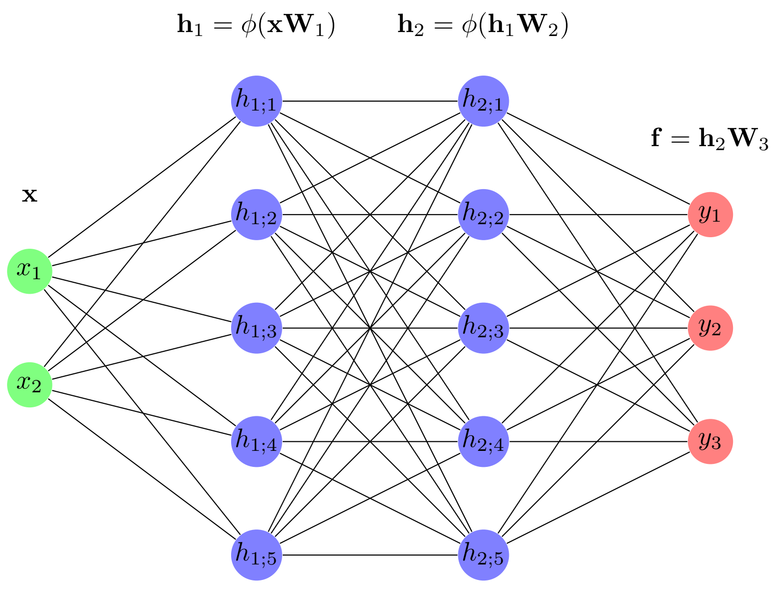
But we can use *any* function here. So what function should we use? Well, there's a few requirements.

- lots of parameters (so gradient descent has lots of knobs to twiddle to improve the function).
- doesn't reduce to something simpler.
- fast (e.g. uses matmuls).

We can draw the current setup as,



Now, one way to get a more interesting function is to stack multiple layers (i.e. connect the outputs of one linear layer to the inputs of another layer):


$\mathbf{h}_1 = \phi(\mathbf{x}\mathbf{W}_1)$
$\mathbf{h}_2 = \phi(\mathbf{h}_1\mathbf{W}_2)$
$\mathbf{f} = \mathbf{h}_2\mathbf{W}_3$
$\mathbf{x}$
x_1
x_2
$h_{1;1}$
$h_{1;2}$
$h_{1;3}$
$h_{1;4}$
$h_{1;5}$
$h_{2;1}$
$h_{2;2}$
$h_{2;3}$
$h_{2;4}$
$h_{2;5}$
y_1
y_2
y_3

Note that we have included an "nonlinearity", ϕ. This nonlinearity is necessary, because without it (or if we set $\phi(\mathbf{h}) = \mathbf{h}$), we end up with,

$$\mathbf{h}_1 = \mathbf{x}\mathbf{W}_1 \tag{2a}$$
$$\mathbf{h}_2 = \mathbf{h}_1\mathbf{W}_2 \tag{2b}$$
$$\mathbf{f} = \mathbf{h}_2\mathbf{W}_3 \tag{2c}$$

If we substitute $\mathbf{h}_1$ into the expression for $\mathbf{h}_2$, and $\mathbf{h}_2$ into the expression for $\mathbf{f}$, we end up with,

$$\mathbf{f} = \mathbf{x}\underbrace{\mathbf{W}_1\mathbf{W}_2\mathbf{W}_3}_{=\mathbf{W}} = \mathbf{x}\mathbf{W}. \tag{3}$$

This is again just a linear combination of the inputs, $\mathbf{x}$, so offers no advantages over the original linear model.

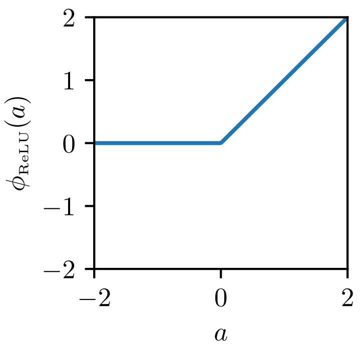
Including a nonlinearity prevents the collapse back to a linear model, giving us a strictly more powerful class of models.

$$\mathbf{h}_1 = \phi(\mathbf{x}\mathbf{W}_1) \tag{4a}$$
$$\mathbf{h}_2 = \phi(\mathbf{h}_1\mathbf{W}_2) \tag{4b}$$
$$\mathbf{f} = \mathbf{h}_2\mathbf{W}_3 \tag{4c}$$

There are lots of different choices for the nonlinearity. Perhaps the most common is the "rectified linear unit" or ReLU,

$$\phi_{\text{ReLU}}(a) = \begin{cases} a & \text{if } a > 0 \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

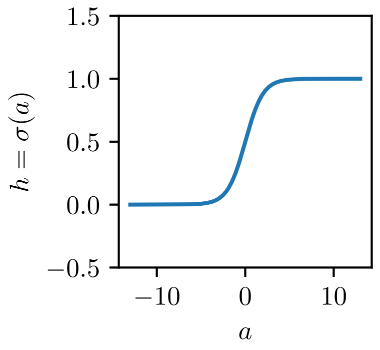
This is commonly known as a 3-layer network, as there are three weight-matrices.



Q: Why don't we have a nonlinearity in the final layer (i.e. for producing the output, f)? Nonlinearities can impose constraints on their outputs. For instance, ReLU can't return negative numbers. Alternatively, in the 1990's people sometimes used a sigmoid as a nonlinearity, but that can only output numbers between 0 and 1. We don't want to impose such constraints on our outputs, which is why we don't apply the output at the final layer.

Q: Is there a better way of understanding what nonlinearities are doing than just "not being linear"? Yes ... but you need quite a bit of theory to get there.

Q: Why is relu far more commonly used than e.g. sigmoid? This is to do with propagation of the gradients. Specifically, the gradients of the sigmoid are often very small: if $h = \phi_{\text{sigmoid}}(a) = \sigma(a)$,



then the gradient, $\partial h/\partial a$, is small for large positive or large negative a. That makes gradient descent difficult, as lots of the gradients get too small. In contrast, the gradients for relu don't explode/vanish, as the gradient, $\partial h/\partial a = 1$ for all $a > 0$.

1 Exercises

There are no exercises this week! (See CoLab notebooks).