Neural networks introduction

We've seen that we can get a long way with linear models, and generalized linear models (linear models combined with a non-Gaussian observation model).

Linear models are still widely used, and should still be implemented as baselines, even if you're convinced you need something more complicated. However, making a linear model work well might require some insight into how to transform the inputs and outputs ("feature engineering"). You can think of *neural networks*[1] as linear models with additional parts, where at least some of the feature transformations can also be learned.

Parameters are fitted for a series of stages of computation, rather than just the weights for a single linear combination. The benefit of neural networks over linear models is that we can learn more interesting functions. But fitting the parameters of a neural network is harder: we might need more data, and the cost function is not convex.

1 We've already seen a neural net, we just didn't fit it

We've already fitted non-linear functions. We simply transformed our original inputs $\mathbf{x}$ into a vector of basis function values $\boldsymbol{\phi}$ before applying a linear model. For example we could make each basis function a logistic sigmoid:

$$\phi_k(\mathbf{x}) = \sigma((\mathbf{v}^{(k)})^\top \mathbf{x} + b^{(k)}), \tag{1}$$

and then take a linear combination of those to form our final function:

$$f(\mathbf{x}) = \mathbf{w}^\top \boldsymbol{\phi}(\mathbf{x}) + b, \qquad \text{or } f(\mathbf{x}) = \sigma(\mathbf{w}^\top \boldsymbol{\phi}(\mathbf{x}) + b). \tag{2}$$

Here I've chosen to put in bias parameters in the final step, rather than adding a constant basis function. This function is a special case of a "neural network". In particular a "feedforward (artificial) neural network", or "multilayer perceptron" (MLP).

The function has many parameters $\theta = \{\{\mathbf{v}^{(k)}, b^{(k)}\}_{k=1}^K, \mathbf{w}, b\}$. What would make it a neural network is if we fit *all* of these parameters θ to data. Rather than placing basis functions by hand, we pick the family of basis functions, and "learn" the locations and any other parameters from data. A neural network "learning algorithm", is simply an optimization procedure that fits the parameters to data, usually (but not always) a gradient-based optimizer that iteratively updates the parameters to reduce their cost. In practice, optimizers can only find a local optimum, and in practice optimization is usually terminated before convergence to even a local optimum.

2 Some neural network terminology, and standard processing layers

In the language of neural networks, a simple computation that takes a set of inputs and creates an output is called a "unit". The basis functions in our neural network above are "logistic units". The units before the final output of the function are called "hidden units", because they don't correspond to anything we observe in our data. The feature values $\{x_1, x_2, \dots x_D\}$ are sometimes called "visible units".

In the neural network model above, the set of ϕ_k basis functions all use the same inputs $\mathbf{x}$, and all of the basis function values go on together to the next stage of processing. Thus these units are said to form a "layer". The inputs $\{x_1, x_2, \dots x_D\}$ also form a "visible layer", which is connected to the layer of basis functions.

1. Here I am talking about the simplest "feed-forward" neural networks.

The layers in simple feed-forward neural networks apply a linear transformation, and then apply a non-linear function element-wise to the result. To compute a layer of hidden values $\mathbf{h}^{(l)}$ from the previous layer $\mathbf{h}^{(l-1)}$:

$$\mathbf{h}^{(l)} = g^{(l)}(W^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}), \tag{3}$$

where each layer has a matrix of weights $W^{(l)}$, a vector of biases $\mathbf{b}^{(l)}$, and uses some non-linear function $g^{(l)}$, such as a logistic sigmoid: $g^{(l)}(a) = \sigma(a)$; or a Rectified Linear Unit (ReLU): $g^{(l)}(a) = \max(0, a)$.[2] The input to the non-linearity, a, is called an *activation*. If we didn't include non-linearities there wouldn't be any point in using multiple layers of processing (see the week 2 question sheet).

We can define $\mathbf{h}^{(0)} = \mathbf{x}$, so that the first hidden layer takes input from the features of our data. Then we can add as many layers of processing we like before the final layer, which gives the final output of our function.

Implementing the function defined by a standard neural network is very little code! A sequence of linear transformations (matrix multiplies and maybe the addition of a bias vector), and element-wise non-linearities.

3 Why is it called a neural network?

[If you're short of time, skip this section, and go straight to check your understanding]

Why is it called a neural network? The term neural network is rooted in these models' origins as part of *connectionism* — models of intelligent behaviour that are motivated by how processes could be structured, but usually abstracted far from the biological details we know about the brain. An accurate model of neurons in the brain would involve large sets of stochastic differential equations; not smooth, simple, deterministic functions.

There is some basis to the neural analogy. There is electrical activity within a neuron. If a voltage ("membrane potential") crosses a threshold, a large spike in voltage called an action potential occurs. This spike is seen as an input by other neurons. A neuron can be excited or depressed to varying degrees by other neurons (it weights its inputs). Depending on the pattern of inputs to a neuron, it too might fire or might stay silent.

In early neural network models, a unit computed a weighted combination of its input, $\mathbf{w}^\top\mathbf{x}$. The unit was set to one if this weighted combination of input spikes reached a threshold (the unit spikes), and zero otherwise (the unit remains silent). The logistic function $\phi_k(\mathbf{x})$ is a 'soft' version of that original step function. We use a differentiable version of the step function so we can fit the parameters with gradient-based methods.

4 Check your understanding

Before your discussion group: Try writing a Python function that evaluates a random neural network function — a neural network function with randomly sampled weights.

[The website version of this note has a question here.]

With your discussion group: Explore how choices defining the distribution over functions affects the typical functions you see.

Things you could try include:

2. A natural question from keen students at this point is: "what non-linearity should I use?". As with many questions in machine learning, the answer is "it depends" and "we don't know yet". ReLUs (named after Relu Patrascu, a friendly sysadmin at the University of Toronto) replaced logistic sigmoids in generic hidden layers of many neural networks as being easy to fit. However, now I would always use a PReLU instead, which have worked better in cases I've tried. There are several other variants, including GELUs, SELUs. The small differences between these non-linearities don't tend to be where big advances come from. Fully differentiable non-linearities like soft-plus $\log(1 + e^a)$, which looks like a ReLU, will make some optimizers happier. Logistic sigmoids are still useful as switches, used in mixtures of experts, LSTMs, and adapting models. Although some of this work is theoretically motivated, what cross-validates the best is what ultimately wins in practice.

- Sample weights with different standard deviations, e.g., 0.1 and 10.
- Try sampling weights from uniform `np.random.rand`.
- Try removing the non-linearities: setting $g(a) = a$, in every layer.
- Try using ReLUs: $g(a) = \max(a, 0)$.
- Try changing the number of layers.
- Try adding in randomly sampled biases.

Empirically, what choices encourage/discourage typical functions from being *complicated*, that is (in 1-dimension) having at least several turning points? Can you create priors over functions that look different to those when you sample from a Gaussian process? If so, how are they different? Did you notice any interactions between the choices?

5 Further reading

Bishop's introduction to neural networks is Section 5.1. Bishop also wrote another book, published in 1995: *Neural Networks for Pattern Recognition*. Despite being 25 years old, and so missing out on more recent insights, it's still a great introduction!

MacKay's textbook Chapter 39 is on the "single neuron classifier". The classifier described in this chapter is *precisely* logistic regression, but described in neural network language. Maybe this alternative view will help.

Murphy's quick description of Neural Nets is in Section 16.5, which is followed by a literature survey of other variants.

Theoretical Neuroscience (Dayan and Abbott) has more detail about biological neural networks and theoretical models of how they learn.