Neural networks introduction

Linear models and generalized linear models (logistic regression et al.) are easy to fit. Least squares linear regression has a direct solution (unless the number of parameters is huge). The other variants can be fitted with gradient descent, and logistic/softmax regression has a convex cost function[1], so we can find the global optimum.

I like linear models. If I were consulting, and thought that a linear model would solve a problem, that's the way I'd go. The code would be simple. And the results would be fairly reproducible because I could get the same fit every time if I used a good optimizer. I might have to think about how to transform the inputs and outputs to make the linear model work ("feature engineering"), which may not be cool or glamorous, but is often effective.

Neural networks fit parameters for a series of stages of computation, not just a single weight vector. The result is that we can learn more interesting functions, given enough data. But fitting these parameters is harder; for a start the cost function will not be convex.

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:

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

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

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

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 "neural network". In particular a "feedforward (artificial) neural network", or "multilayer perceptron".

The function has many parameters $\theta = \{\{\mathbf{v}^{(k)}, b^{(k)}\}_{k=1}^K, \mathbf{w}, b\}$. What really makes it a neural network is fitting *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 fitting a cost function.

Why is it called a neural network?

[A quick sloppy interlude — non-examinable]

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.

1. If cost $E(\mathbf{w})$ is convex, then a straight line between two points on the function never goes below the surface: $E(\alpha\mathbf{w} + (1-\alpha)\mathbf{w}') \leq \alpha E(\mathbf{w}) + (1-\alpha)E(\mathbf{w}')$, where $0 \leq \alpha \leq 1$. It's a stronger statement than "unimodal", and makes optimization a lot easier. There are whole books on convex optimization: `http://stanford.edu/~boyd/cvxbook/`

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 $\{x_1, x_2, \ldots x_D\}$ 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 $\{x_1, x_2, \ldots x_D\}$ also form a "visible layer", which is connected to the layer of basis functions.

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

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

where each layer has a matrix of weights $W^{(l)}$, a vector of biases $\mathbf{b}^{(l)}$, and uses some non-linear function g, such as a logistic sigmoid $g(a) = \sigma(a)$, or a Rectified Linear Unit (ReLU) $g(a) = \log(1 + e^a)$. 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 combining the final layer into 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.

Recall why we need non-linearities if we're going to define our function in terms of multiple layers of processing. (See tutorial 1.) Choosing which non-linearities to use, amongst other details, is ultimately somewhat of an empirical science. Different choices can be theoretically motivated, but what cross-validates the best is what ultimately wins in practice.

Fitting and initialization

Neural networks are almost always fitted with gradient based optimizers, such as variants of Stochastic Gradient Descent. I will defer how we compute the gradients to a future note.

How do we set the initial weights before calling an optimizer? *Don't* set all the weights to zero! If different hidden units (adaptable basis functions) start out with the same parameters, they will all compute the same function of the inputs. Each unit will then get the same gradient vector, and be updated in the same way. As each hidden unit remains the same, we can't fit anything much more interesting than logistic regression.

Instead we usually initialize the weights randomly. *Don't* simply set all the weights using `randn()` though! As a concrete example, if all your inputs were $x_d \in \{-1, +1\}$ the activation $(\mathbf{w}^{(k)})^\top \mathbf{x}$ to hidden unit k would have zero mean, but typical size $\sqrt{K}$ if there are K hidden units. (See the review of random walks on the expectations sheet.) If your units saturate, like the logistic sigmoid, most of the gradients will be close to zero, and it will be hard for the gradient optimizer to update the parameters to useful settings.

Summary: initialize the weights to small random values, like `0.1*randn()/sqrt(K)`, assuming your input features are ~ 1.

More advanced neural network architectures have particular tricks for initializing them. Do a literature search if you find yourself trying something ambitious. Or (pragmatically) if you are using a neural network toolbox, try to process your data to have similar properties to the standard datasets that are usually used to demonstrate that software. Then any initialization tricks the software uses are more likely to work.

Further reading

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.

If you want to read more about biological neural networks and theoretical models of how they learn, I recommend Theoretical Neuroscience by Dayan and Abbott.