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[1]). The other variants can be fitted with gradient descent, and logistic/softmax regression has a convex cost function[2], so we can still find the best possible fit to the training set (a global optimum).

I like linear models. If looking at a new problem, perhaps when consulting, I'd see if I could at least start by setting up a linear model. 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. Even if I needed something more complicated in the end, I'd at least have a baseline that could confirm whether a more advanced system's performance was reasonable.

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*[3] 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 no longer 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 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.

Why is it called a neural network?

[A quick sloppy interlude — non-examinable]

---

1. More detail: I hesitate to say linear regression has a "closed form solution", because a numerical algorithm of some sort is always needed to compute the actual weights. The normal equations approach for D parameters requires $O(D^3)$ time and $O(D^2)$ space to construct and solve the linear system. Numerical linear-algebra approaches using standard QR decompositions (as used by Matlab's \) or dense SVD solvers (as used by NumPy's `lstsq`) cost at least as much. I can't use these "direct" approaches on my laptop if D is 100,000. There are a variety of possible iterative numerical methods for large least-squares problems — including fitting with gradient-based methods. I'd do a quick literature search, or ask someone, if I wanted to know the current state-of-the-art.
2. If cost $C(\mathbf{w})$ is convex, then a straight line between two points on the function never goes below the surface: $C(\alpha\mathbf{w} + (1-\alpha)\mathbf{w}') \leq \alpha C(\mathbf{w}) + (1-\alpha)C(\mathbf{w}')$, 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/`
3. Here I am talking about the simplest "feed-forward" neural networks.

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.

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.

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 $\mathbf{h}^{(l-1)}$:

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

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): $g^{(l)}(a) = \max(0, a)$.[4] If we didn't include non-linearities there wouldn't be any point in using multiple layers of processing (see tutorial 1).

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.

4. 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. Now there are SELUs; I should try them too, but I doubt they will be the final word either. 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.

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 $\sqrt{D}$ if there are D inputs. (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 a weight matrix that transforms K values to small random values, like `0.1*randn()/sqrt(K)`, assuming your input features are $\sim$1.

The MLP course points to Glorot and Bengio's (2010) paper Understanding the difficulty of training deep feedforward networks, which suggests a scaling $\propto 1/\sqrt{K^{(l)} + K^{(l-1)}}$, involving the number of hidden units in the layer after the weights, not just before. The argument involves the gradient computations, which I haven't described in detail for neural networks yet, so I will defer the interested reader to the paper or the MLP slides.

Some specialized neural network architectures have particular tricks for initializing them. Do a literature search if you find yourself trying something other than a standard dense feedforward network: e.g., recurrent/recursive architectures, convolutional architectures, or memory networks. Alternatively, a pragmatic tip: 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. For example, similar dimensionality, means, variances, sparsity (number of non-zero features). Then any initialization tricks that the demonstrations use are more likely to carry over to your setting.

Check your understanding

Have you followed the description of how the layers are composed to form a function?

Write a function in Matlab/Octave or Python+NumPy to evaluate a feedforward neural network function on each of N scalar inputs, using two hidden layers with `H1=100` and `H2=50` hidden units and a logistic sigmoid non-linearity, and a scalar output. Sample all the weights randomly from a standard normal[5]. In the first instance, you could omit extra bias parameters if you like. Plot the resulting function for inputs $x \in [-2, 2]$.[6] Try multiplying the weights by 0.1 and 10, and see how the function changes. Try removing the non-linearity in one or both hidden layers[7], and see how your function changes.

Empirically, what properties make typical functions, specified by random weights, complicated? You could take *complicated* in 1-dimension to mean "have at least several turning points". Are all of these properties strictly necessary for representing a complicated function?

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.

---

5. Here we don't want small weights, as in an initialization, we want to create an example non-linear function.
6. For example: `xx = (-2:0.01:2)'; ff = your_neural_net_function(xx, params); plot(xx, ff);`
7. E.g. in Matlab, set `g1 = @(a) a;` instead of `g1 = @(a) 1./(1+exp(-a));`

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.

Finally, not all neural network layers take the form: $\mathbf{h}^{(l)} = g^{(l)}(W^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)})$. For example, we could construct a function using the radial basis functions (RBFs) we discussed earlier, and then train the centers and bandwidths of these functions. There are also specialized transformations for modelling sequences and images. This course doesn't do a deep dive into the layers for particular applications, and the details of how best to combine them (a changing landscape). The MLP course and Goodfellow et al.'s deep learning textbook are good starting points on the details of recent practice.