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.

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.

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.

2 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.

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

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

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): $g^{(l)}(a) = \max(0, a)$.[4] 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 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.

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. 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.

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 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?

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.

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.

6 For keen students: embedding and pooling

Most neural networks are not applied to arbitrary D-dimensional feature vectors. Instead they take signals such as audio, images, or text, which have structure we know about, but that might not have a fixed size. Audio or text sequences that we want to classify could have different lengths. Images we want to tag could have different sizes.

5. Here we don't want small weights, as in an initialization, we want to create an example non-linear function.
6. In Matlab/Octave your code might look like:
```
xx = (-2:0.01:2)'; ff = your_neural_net_function(xx, params); plot(xx, ff);
```
In Python like:
```
xx = np.linspace(-2, 2, 201); ff = your_neural_net_function(xx, params); plt.plot(xx, ff)
```
7. E.g. in Matlab, set "`g1 = @(a) a;`" instead of "`g1 = @(a) 1./(1+exp(-a));`"
Or in Python, "`def g1(a): return a`" instead of "`def g1(a): return 1/(1+np.exp(-a))`"

This section has some non-examinable detail that shows how neural networks can begin to deal with text and variable-sized inputs. However, this course isn't trying to provide an exhaustive survey of methods. It would also be worth learning about *Convolutional neural Networks* (ConvNets), which are covered in the MLP course.

6.1 Embeddings

Consider a vector $\mathbf{x}$ that is a one-hot encoding of D possible items, such as a choice of one of D English words. We could construct a K-dimensional hidden vector from this input using a standard neural network layer:

$$\mathbf{e} = g(W\mathbf{x} + \mathbf{b}), \tag{4}$$

which could be passed on to further layers of processing in a neural network. When we input the ith choice/word, $x_d = \delta_{id}$, we only use the ith column of the matrix $W_{:,i} = \mathbf{w}^{(i)}$:

$$\mathbf{e}(\text{choice } i) = g(\mathbf{w}^{(i)} + \mathbf{b}). \tag{5}$$

K of the parameters, $W_{:,i} = \mathbf{w}^{(i)}$, are used only when the input contains choice i, and we're setting K hidden vector values. Without reducing the space of functions that we can represent, we could instead just define the hidden values for each choice to be equal to K free parameters:

$$\mathbf{e}(\text{choice } i) = \mathbf{v}^{(i)}, \quad \mathbf{e} = \text{embedding}(i; V). \tag{6}$$

This "embedding" operation replaces a discrete choice (e.g., of a word) with a K-dimensional feature vector, representing or *embedding* our items in a K-dimensional vector space. When we train a whole neural network on some task (such as text classification, or next word prediction), the "embedding vectors" are learned.

Summary: We could learn vector representations of discrete objects by one-hot encoding them, and providing them as inputs to a standard feed-forward neural network. The weights $W_{:,i} = \mathbf{w}^{(i)}$ would "embed"/represent choice i, but the large matrix multiplication $W\mathbf{x}$ would be a wasteful way to implement the layer. Instead, most deep learning frameworks come with *embedding* layers: a look-up table that takes an integer item identifier and returns a vector of learnable parameters for that item.

6.2 Sequences and pooling

It's common to have a variable number of inputs in a prediction problem, often in a sequence. For example, we might have a sentence of text that we want to classify or translate. Or we could have a sequence of events in a log for a particular user, and want to predict something about that user (what they might want to know, or whether they are acting fraudulently). However, standard neural networks require inputs of some fixed dimensionality (usually called D in these notes).

If sentences/sequences were always exactly 25 words long, we could concatenate K-dimensional word embeddings together, and obtain $25K$-dimensional feature vectors. If sentences were always at most 25 words, we could "pad" the sequence with a special token meaning "missing word" and still use a neural network that takes $D = 25K$ inputs. However, to be able to model data where the maximum sequence length is long, we will need to create huge feature vectors. We may not have the computational resources, or enough data, to use these huge vectors.

A simple and cheaper baseline, is to "pool" all of the inputs in a sequence together, for example by just taking an average:

$$\mathbf{x}_{\text{pooled}} = \frac{1}{T}\sum_{t=1}^{T} \text{embedding}(\mathbf{x}^{(t)}; V), \tag{7}$$

where $\mathbf{x}^{(t)}$ is the input at "time" t. No matter how long the sequence is, T, the pooled representation is K-dimensional (where V is a matrix with $K \times D$ free parameters). We could replace the pooling operation with other functions, such as a sum or a max instead of an average.

We could also use a weighted average:

$$\mathbf{x}_{\text{pooled}} = \sum_{t=1}^{T} a(\mathbf{e}^{(t)})\,\mathbf{e}^{(t)}, \quad \mathbf{e}^{(t)} = \text{embedding}(\mathbf{x}^{(t)}; V), \tag{8}$$

where $a(\mathbf{e}^{(t)})$ is a scalar weight, often chosen to be positive and sum to one. If we place all the embeddings in a $T\times K$ matrix E, the simplest way to get such weights is probably:

$$\mathbf{a} = \text{softmax}(E\mathbf{q}), \tag{9}$$

where $\mathbf{q}$ is another K-dimensional vector of free parameters.

The "query" vector $\mathbf{q}$ tells us which embeddings to pay most "attention" to in the average, so $\mathbf{a}$ are sometimes called "attention weights". The weighted average is a simple version of what's called an "attention mechanism". Attention mechanisms are often described as ways of "looking" at parts of a sequence or an image, but in their simplest form, they're just a way of learning how to take a weighted average.

The above pooling operations ignore the times: they would return the same vector if we shuffled the sequence. "Bag of words" models that throw away word order are common for text classification, but would not work for machine translation. We could add the time information to the embedding vectors, as in
`https://papers.nips.cc/paper/7181-attention-is-all-you-need`

Another way to deal with sequences is to build up a representation by absorbing one input at a time. "Recurrent neural networks", such as "LSTMs" do that.