Neural network architectures

[If you are short on time this week. This is the note and videos to skip for now.]

The previous note introduced neural networks using a standard layer, an affine transformation, and an element-wise non-linearity:

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

. However, any composition of differentiable functions that could be used, and their free parameters learned (fitted to data) with a gradient-based optimizer. 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.

This course doesn't attempt to do a comprehensive review of the layers for particular applications or use-cases. However, it's worth knowing that there are specialized transformations for representing functions of different types of input, such as sequences or images. The MLP course (2019 archive) and Goodfellow et al.'s deep learning textbook are good starting points on the details of recent practice.

The neural network architectures that are in standard use are still evolving. If in the future you're ever applying neural networks to data with some particular structure, it will be worth reviewing how neural net have been applied to similar data before. Working out what works will then be an empirical exercise, backed up by training, validation, and testing.

1 Skip Connections and Residual Layers

VIDEO 2020-11-04_14-45-24_skip_connections_adaption_and_residual_layers

After seeing a standard feedforward neural network, a common question is whether we could have "skip connections": add the output from a layer, to the input of other layers higher up the neural network, not just the next layer. The answer is yes, but as the search space of architectures is so large, it can be useful to have some reasons for particular choices.

If we provide a one-hot encoded input to every layer of a neural network, then each layer effectively has different biases for each setting of the input variable. Using these connections, the network can learn to turn individual hidden units on or off in different circumstances. There are also more advanced ways of adapting each part of a neural network to behave differently in different contexts[1].

Another special case of a skip connection is the use of *residual layers*. The core idea is to take a standard non-linearity g and transform one hidden layer to another with $r(\mathbf{h}) = \mathbf{h} + g(W\mathbf{h})$. The weights W are used to fit the "residual" difference from a default transformation, that leaves the hidden unit values unchanged. If we initialize a deep stack of residual layers with small weights, we initialize close to the identity function ($f(\mathbf{x}) = \mathbf{x}$).

At the start of training, the weights in the output layer of our neural net then get the same gradients as they would in a (generalized) linear model. In contrast, a randomly-initialized deep stack of standard layers usually produces an output that seems to have nothing to do with the input. This initialization is a difficult starting point for gradient-based fitting to learn anything.

More complicated layers, which also let inputs pass through many stages of processing, were previously developed in the "recurrent neural network" (RNN) literature. Most famously the "LSTM" (long short-term memory) cell, and the simpler GRU (Gated Recurrent Unit). However, we won't cover the details of those methods here.

1. Methods developed in Edinburgh include LHUC, *residual adaptors* and PALs.

2 Embedding vectors

As a side effect of learning a function to minimize some cost function, neural networks often learn representations of our data that could be useful more generally.

Assume our input $\mathbf{x}$ 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:

$$\mathbf{e} = g(W\mathbf{x} + \mathbf{b}), \tag{2}$$

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 $W_{:,i} = \mathbf{w}^{(i)}$:

$$\mathbf{e}(\text{choice } i) = g(\mathbf{w}^{(i)} + \mathbf{b}). \tag{3}$$

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:

$$\mathbf{e}(\text{choice } i) = \mathbf{v}^{(i)}, \quad \mathbf{e} = \text{embedding}(i; V). \tag{4}$$

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.

If you visualize these embedding vectors[2] you'll find that similar words are usually next to each other. When trained on a word prediction task on a large corpus, the vectors could be a useful feature representation for use in tasks with less data. However, the representations may also encode harmful statistical dependencies present in large-scale corpora.

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.

3 Bags and Sequences of items

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 (and many other machine learning methods) 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:

$$\mathbf{x}_{\text{pooled}} = \frac{1}{T}\sum_{t=1}^{T} \text{embedding}(\mathbf{x}^{(t)}; V), \tag{5}$$

2. e.g., using methods such as UMAP or t-SNE.

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:

$$\mathbf{x}_{\text{pooled}} = \sum_{t=1}^{T} a(\mathbf{e}^{(t)})\, \mathbf{e}^{(t)}, \quad \mathbf{e}^{(t)} = \text{embedding}(\mathbf{x}^{(t)}; V), \tag{6}$$

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

$$\mathbf{a} = \text{softmax}(E\mathbf{q}), \tag{7}$$

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
Vaswani et al.'s (2017) "Attention is all you need" paper.

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

4 ConvNets and Transformers

We've focussed on neural networks that take a vector $\mathbf{x}$ or $\mathbf{h}^{(l)}$ and return another vector $\mathbf{h}$ or $\mathbf{h}^{(l+1)}$. However, some neural network layers that take sequences or images as input, return sequences or image-like data as their output. These layers can be composed several times before having to summarize a complicated object into a single vector.

We're just going to mention the existence of these architectures here (we don't really expect you to understand them from these few sentences alone):

The **Transformer** layer (Vaswani et al., 2017) takes a sequence of vectors (such as word embedding vectors) and returns a new sequence of vectors of the same length. These layers can be composed to re-represent a sentence, and are widely used in machine translation and other language and sequence-based tasks. (An impressive rate of adoption, given the paper is from 2017.)

Convolutional Neural Network (or ConvNet) layers take an "image": a spatial array of pixel-values often with multiple "channels" (such as red, blue, green). The output of a layer also has multiple "channels", where each channel is again a spatial array of values.

Each channel has a "linear filter" (a set of weights, which could form an "edge detector") that takes a patch of pixel values centred around a given location, and returns a number. This filter is applied at every location in the original image, to give the grid of values in the corresponding output channel.