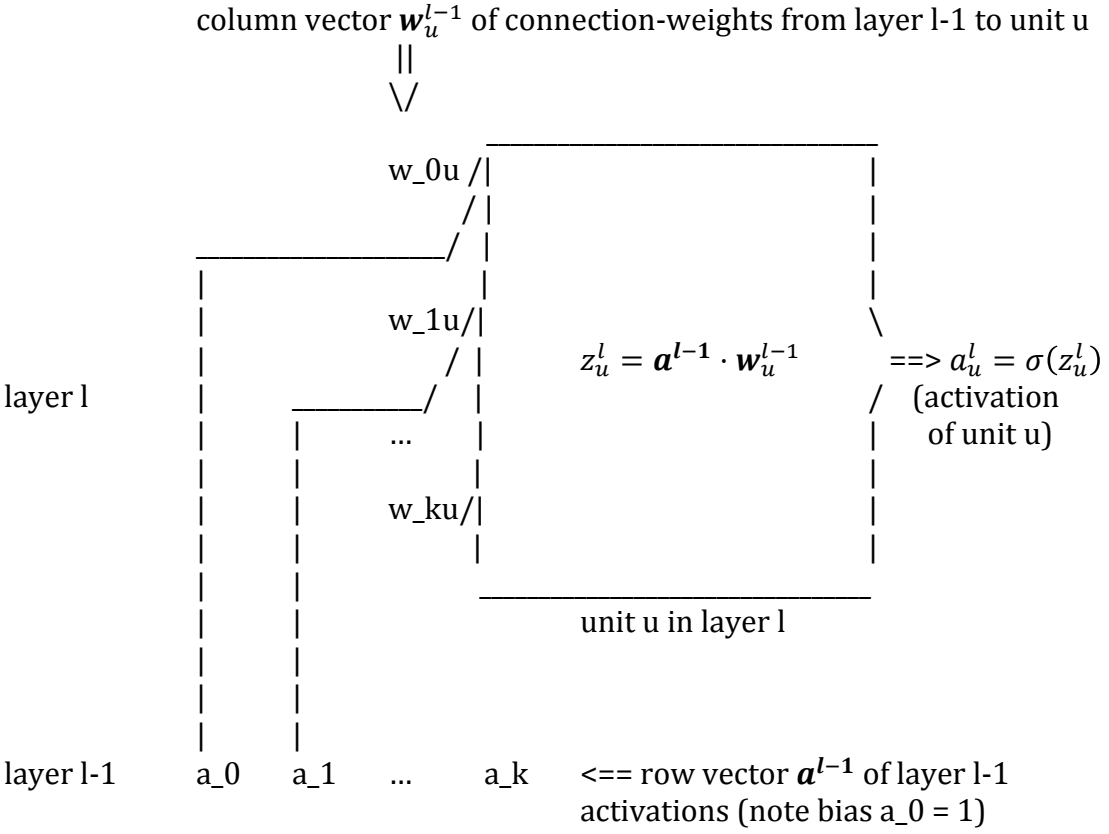
Linear algebra representations for feedforward neural networks

Consider a neural network (a "nnet") N with L layers (numbered 0 through L, hence really L+1 layers, if we count the input layer which does no actual processing). Let u be one of the units at a given layer l. Unit u will have a connection with each unit of the previous layer l-1 (if there are any, i.e., if l>0); there will be a weight w associated with each such connection; and the activation *a* produced by each such layer l-1 unit is sent from that unit to u with resulting "influence" *a*w on u. Since there in general will be many units at layer l-1, then there will be many activation values coming to u via connections from below, and thus many weights (one for each connection). We can think of all these layer l-1 activation values as forming a vector **a**. If we imagine each layer's units to be spread out horizontally from left to right, with layer 0 at the bottom and "later" layers higher up, then the activations of a given layer can be imagined naturally as a row vector. We can also think of the weights for the connections coming to u as forming a vector (of the same length as **a**) but viewed as a column (if we imagine the weights to be stacked up alongside u).

Here is a picture:



The activation (signal value) coming *out* from u is computed, as we have noted earlier, by applying the activation function (sigmoid, for us) to the weighted sum of the incoming activations from below, namely sigma(**a.w**). We can also think of the

argument to sigma as a matrix product, where **a** and **w** are 1xk and kx1 matrices, resp. (In the figure, the superscripts l-1 simply indicate that the entries are associated with the connections coming from layer l-1.)

It will convenient later to have a shorter name for this product, which represents the total influence of all the units at layer l-1 on the unit u at layer l. We will write it simply as

$$z_u^l = \boldsymbol{a^{l-1}} \cdot \boldsymbol{w}_u^{l-1} \text{(as seen in figure above)}$$

so that the activation of u is $\sigma(z_u^l)$, which we also write as a_u^l — the activation of the u-the unit of layer l:

$$a_u^l = \sigma(z_u^l) = \sigma\left(\boldsymbol{a^{l-1}} \cdot \boldsymbol{w}_u^{l-1}\right) = 1 / (1 + e^{\{-z_u^l\}})$$

Now, u is just one of the (possibly many) units of layer l. So in fact, there will be equally many column vectors of connection-weights between layers l-1 and l for *each* unit of layer l. And they all have the same length, namely the number k+1 of units at layer l-1 (counting the bias unit). So we can group them all together into one $(k+1)\times m$ matrix W^{l-1} where the units at layer l can be numbered 0,…,m (of which u is one); recall that bias unit 0 has no input connections, hence no weights.

If we then collect all the total influences of all the units at layer l, into one row vector $\boldsymbol{z}^l$, then the activations of all those units 1,…,m are nicely expressed in a single vector as follows:

$$\boldsymbol{a}^l = \sigma(\boldsymbol{z}^l) = \sigma(\boldsymbol{a^{l-1}} W^{l-1)})$$

where it is understood that σ is applied separately, one by one, to all the elements of its vector argument.

There is nothing particularly subtle going on here, just a very compact notation to represent a potentially huge number of values connected in an equally huge number of ways. If we did not do this, we'd need more subscripts and superscripts, not to mention summations, and the result would be somewhat hard to read. In addition, we are writing down above just the dependence of one layer's activation on the activations and weights coming from the connections one level below. But ultimately, the final layer-L outputs are a function of all the weights of all the connections at all layers together with the input values at layer 0. Not only that: many programming languages supply libraries that perform linear algebra computations such as matrix multiplication with no extra effort by the programmer.

Let's look at the ugly details when we try to write out this dependence all the way, even at a very simple example, with just three layers, and only two or three units per layer. (Note that output layers never have need for a bias unit.)

Layer#	bias units	other units	
2	(none)	1	2
1	0	1	
0	0	1	2

The weight matrix W^0 for connections from layer 0 to layer 1 will be 3x1, and that from layer 1 to layer 2, 2x2. (The unit numbers above are simply their left-to-right positions within each layer.) Let's try to write down the full expressions for the activation values at the output layer (layer 2). There are just two of them, a_1^2 and a_2^2.

$$(*) \quad a_1^2 = \sigma(z_1^2) = \sigma\left(\boldsymbol{a}^1 \cdot \boldsymbol{w_1^1}\right) = \sigma(< a_0^1, a_1^1 > \cdot < w_{01}^1, w_{11}^1 >)$$

$$= \sigma(a_0^1 w_{01}^1 + a_1^1 w_{11}^1)$$

$$= \sigma(\ 1w_{01}^1 + \ \sigma(z_1^1) w_{11}^1\ )$$

$$= \sigma(\ \ w_{01}^1 + \ \sigma\left(\boldsymbol{a^0} \cdot \boldsymbol{w_1^0}\right) w_{11}^1\ \ )$$

$$= \sigma(\ \ w_{01}^1 + \ \sigma(a_0^0 w_{01}^0 + a_1^0 w_{11}^0 + a_2^0 w_{21}^0)\ w_{11}^1\ \ )$$

This is the full expression for the first output in terms of the input data at layer 0; the second output is essentially the same, but with some subscripts on the w's changing from 1 to 2. The lesson is that this is a mess to write, read, and understand. Far better to keep it simple and abstract and focused just on connections between two layers at a time. In addition, this *local focus* will turn out to be crucial for using gradient descent efficiently.

[A note on bias units: Without them, especially in a small network, poor performance can occur. For instance, if all the data inputs are zero, the outputs will be zero if no bias units are present, no matter what the weights are. So a no-bias network cannot learn to output a nonzero result from such data. Thus even simple linear functions like y = x+1 would not be learnable! It is true that such cases are extremely rare for most large-network applications, but developers generally build the bias units in, so we might as well let them stay.]

Yet despite the ugliness of the all-written-out version above, note again that it does illustrate something of value: the output activations ultimately are functions of the layer-0 inputs and of all the weights. The inputs are part of the problem definition (training data x-values, for instance); but the weights are what we can fiddle with to

get outputs closer to what we want (i.e., to match – or at least closely approximate – the training data y-values). If we now pick a suitable cost function C to measure closeness of nnet outputs to the desired y values, we can employ gradient descent on it to approach its minimum, i.e., to find weights for which the nnet performs very much as we want. This will depend on the C function and the activation function being "nice". We have chosen the sigmoid for the latter, and will choose MSE – mean-squared error – as our C. Note that while C is defined as a function of the outputs (compared to y values), it ultimately depends on the weights because that is what the outputs depend on, as we just noted.

The grand plan then is to use gradient descent on C as a function of the all the weights w in the nnet. This means finding the partial derivatives of C with respect to all the w's.

One way to do this is by direct estimation from the definition of derivative, i.e., by computing the value of C for a given w and also for a slightly changed w, finding subtracting the latter from the former, and dividing by the change in w. Doing that would involve "running" the nnet all the way through, for *each* weight separately, to find the output values and then the C-values for w and its changed version. Since nnet applications often involve millions of weights, this is hugely impractical. More specifically: if there are N weights in all, then it takes time proportional to N to pass through the nnet computing activations from the input layer all the way to the output layer. To do this twice (once for a given w and once for its modified value) takes 2N time; and to repeat that for all the w's will take time proportional to 2N^2, i.e., quadratic in N. Doing one million (or two million) computations is one thing – doing a million-squared is altogether another.

Another way is to use the chain-rule for derivatives. This leads to so-called *back-propagation*, which is the basic technique that makes gradient descent an effective tool for training feedforward neural nets. Even so, it was not practical until computer speeds got faster and memories larger. The basic idea is to run the nnet through once, finding all the activations; then find the effect (i.e., partial derivatives) on the final activation-outputs and hence on C of *just* the final weights (from layer L-1 to L); and then use those to find the effect of the weights from the previous layer; and so on back to the input layer.

This can already be intuited from equation (*) where an a-value at the output layer (l=2) is expressed in terms of the previous layer's weights and activations, and in fact for example the partial derivative of C wrt the connection-weight from the bias unit in layer 1 to layer 2 is just the derivative of sigma times the a-value for that same connection. That is, since C is determined at the output layer, its partial wrt weights directly involved there are easy to calculate; and from those we work all the way back to earlier and earlier layers. Hence the name backpropagation. So the network is traversed just twice (for a given training pair x,y): forward to get the activations, and backward to get the derivatives.

Let's step back for a minute and survey where we are. To train (i.e., use supervised learning on) a (feedforward) neural network, one first needs to make some *choices*:

1. training pairs (x,y) – usually lots of them
2. nnet architecture (how many layers, units in each layer)
3. initial weight values
4. learning rate
5. activation function (e.g, sigmoid)
6. batch size (entire training set or a subset)
7. cost function C

Next one does the following *procedures*:

1. shuffle the training pairs
2. run the nnet on the first batch
3. use the results to compute C
4. use gradient descent to find Grad C (wrt the w's)
5. adjust the w's one step
6. repeat 2-5 on the next batch over and over, until all the way through the training set
7. repeat 1-6 (each pass through 1-6 is called an epoch); do as many epochs as desired

There are various trade-offs on batch size. A size less then the entire training data is called a mini-batch. A mini-batch can be as small as a single training pair, but more commonly is larger than this. The idea is part is to pick (at random) a (hopefully) representative sample. For instance, the *chainer* software package – when run on the MNIST training set of 60,000 pairs – by default uses a mini-batch of 100 training pairs, then another 100, and so on, 600 times until all the pairs have been used (in the first epoch). Usually the first epoch is not nearly good enough; after all, by then the weights have been adjusted only *once* for each pair, so one would not expect very good performance. Chainer (again by default) performs 20 epochs on MNIST.

After training (or even after each epoch) one sees how the nnet performs on another set of pairs – the test set. This can be used to help decide whether to perform more training epochs or stop.