Chapter 13

Neural Networks

To give you an idea of how new this material is, let's do a little history lesson. The origins of neural nets are typically dated back to the early 1940's and work by two physiologists, McCulloch and Pitts. Hebb's work came later, when he formulated his rule for learning. In the 1950's came the *perceptron. The perceptron* is what we called a linear neural network- it became clear that the perceptron could only classify certain types of data (what we now call linearly separable data). This shortcoming led to a stop in neural net research for many years- It was not until the 1980's that neural net research really took off from a coming together of many disparate strands of research- There was a group in Finland led by T. Kohonen that was working with self-organizing maps (SOM); there was the work from the 70s with Stephen Grossberg, and finally several researchers were discovering methods for training new networks that could be put in parallel (back-propagation).

It was in the 80's and 90's that researchers were able to prove that a neural network was a **universal function approximator**- That is, for any function with certain nice properties, we are able to find a neural network that will approximate that function with arbitrarily small error. It is interesting to note that the use of a neural network could be used to solve Hilbert's 13th Problem[1]:

"Every continuous function of two or more real variables can be written as the superposition of continuous functions of one real variable along with addition."

Let's look at some highlights of more recent developments in neural nets:

- 1989: George Cybenko proves that neural nets with certain types of activation functions are universal function approximators.
- 1989: Yann LeCun used convolutional neural networks to read handwritten digits. This led to many rapid developments, and Dr. LeCun shared the Turin prize in 2018 for this work.
- 1989: Q-Learning is something that takes machine learning to a different branching- This greatly improves the area of reinforcement learning.
- 1993: Support Vector Machines (SVM), around for a long time but brought into machine learning, designed by C. Cortes and V. Vapnik.
- 1998: Yann LeCun brings in Stochastic Gradient Descent as a practical solution to large problems.
- 2006 or so: Approximately the beginning of "deep learning", as coined by Geoffrey Hinton (a key player in machine learning for a very long time, also given the Turin award in 2018).
- 2009: The launch of ImageNet (Fei-Fei Li at Stanford). As of 2017, it contains 14 million **labeled** images that are available to researchers. The casual reader may not appreciate how key this is-

[1] In 1900, mathematician David Hilbert listed what he viewed as the 23 greatest unsolved problems. This list has been worked on ever since, and as of 2021, problems 8, 12, 13 and 16 remain unsolved.

"Deep learning" requires vast amounts of data, and this kind of data was not readily available to the community earlier.

- 2011: IBM's Watson computer wins *Jeopardy!*.
- 2011: The use of the Rectified Linear Unit (or ReLU) as activation function. This function is used in deep networks to counter what is called "the vanishing gradient problem".
- 2012: Creation of AlexNet - Its success kicked off a spike in researching convolutional neural nets (that continues to this day).
- 2014:

Generative Adversarial Networks (GAN). Basically, a second neural net is built as an "adversary", where it builds data to try to fool the neural net that is learning a particular task. The basic idea has expanded, and these networks can now generate realistic images of human beings. The person to the right does not exist.
(Photo from Wikipedia Commons)

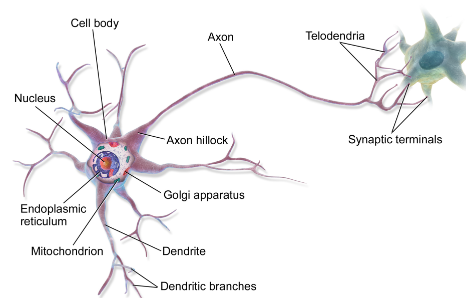
Coming to present day, research is aimed at something called **deep neural nets**- in summary, the difference now is that we have a LOT of training data, and new algorithms and architectures that make it feasible to train complex associations. A nice summary of "The Decade of Deep Learning" is given at the site below (accessed Apr 2021):
`https://bmk.sh/2019/12/31/The-Decade-of-Deep-Learning/`

Back to feed forward nets

The term "neural network" has come to be an umbrella term covering a broad range of different function approximation or pattern classification methods. The classic type of neural network can be defined simply as a connected graph with weighted edges and computational nodes- We've seen a linear neural network (using Widrow-Hoff training rule). We now turn to the workhorse of the neural network community: The feed forward neural network.

13.1 From Biology to Construction

We've seen the basic model before in the linear neural networks. We include some of it again for clarity. Information flows from the dendrites to the cell body through the axon to a synaptic junction connecting to the dendrite to the next neuron.
The synaptic junction is made up of the presynaptic node (the end of an axon), the postsynaptic node (the beginning of a dendrite), and the "empty space", which is the synapse.
A diagram of a neuron is shown to the left.



As information flows across a synapse, information can be amplified, inhibited, or re-polarized. We'll discuss our model of cell body processing and flow over the axon, but this is all the biology we use. It's best to think of neural nets as being inspired by biology, although there are researchers interested specifically in working with biological neural nets.

Before we look at the neural net on the level of a layer of neurons, let's look at how a single neuron (or "node") processes information:

Let $x_1, \ldots, x_n$ denote the signals incoming along the dendrites. We model the processing done here by multiplying the signal by a real number which we call a **weight**.

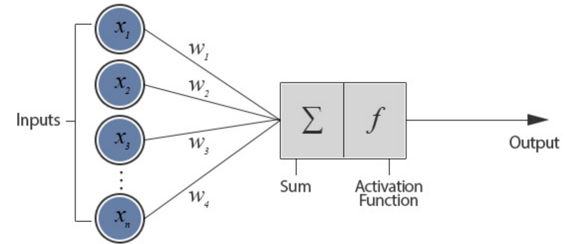
At the cell body, these signals are first collected in what we'll call the **prestate** of the cell. They are added to a "bias term", b, which loosely represents the resting state of the cell.

$$p = \sum_{j=1}^{n} w_j x_j + b = \boldsymbol{w} \cdot \boldsymbol{x} + b$$

An activation function is applied, we'll just call it f for now. Applying f, we get what is called the **state** of the cell, or the cell's **activation**:

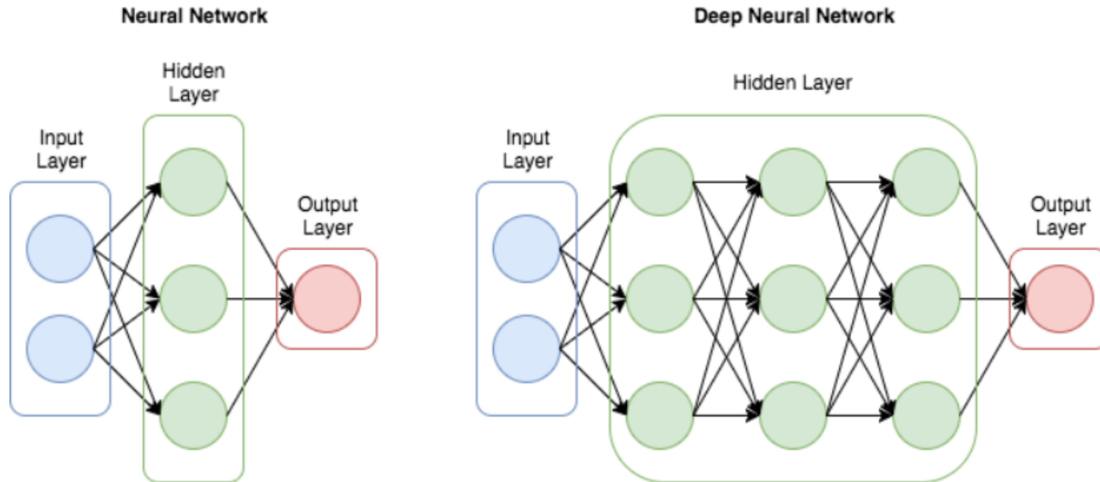
$$s = f(p)$$

And this value is then sent along the axon. Here is a typical way we might make a diagram of a single neuron, where the cell body has been divided into the sum stage, then the activation.



Putting lots of cells together is what a neural network is. Below on the left, we show a three layer network. Data is input at the first layer, the second layer is really the layer of cells, and the last layer is the output layer. In finding a mapping from $\mathbb{R}^n$ to $\mathbb{R}^m$ with k cells, we would construct a $n - k - m$ network.

Below and to the right, we see networks don't have to have a single layer; we can apply lots of layers to get a multilayer network. Training these can be difficult, and is the subject of our next chapter in deep learning. For the time being, we'll keep with a single hidden layer.



Next we'll look at how information flows through the network. It is easiest to describe using layers and linear algebra.

13.2 How Networks Compute

At the input layer, we're presented with a real value at each input node, which we'll denote as a vector $\mathbf{x} \in \mathbb{R}^n$. Next, we move from $\mathbb{R}^n$ to $\mathbb{R}^k$. The affine mapping that takes us from the input layer to the hidden layer is defined as

$$P^{(2)} = W^{(1)}\mathbf{x} + \mathbf{b}^{(i)}$$

where $P^{(2)}$ is the vector in $\mathbb{R}^k$ representing the prestate in each node (or cell). Since the weight matrix $W^{(1)}$ is $k \times n$, we make an important observation:

$$W_{jk}^{(i)} \text{ is the weight connecting cell } k \text{ from layer } i \text{ to cell } j \text{ on layer } i+1.$$

It is tempting to say this connection backwards ("j connected to k"), so please be careful with that- It is the opposite of what you might think it is.

Now continuing, we apply function f to the prestate vector P (element-wise) to get the state vector S.

$$S^{(2)} = f\left(P^{(2)}\right) = f\left(W^{(1)}\mathbf{x} + \mathbf{b}^{(i)}\right)$$

Now we apply an affine transformation from $\mathbb{R}^k$ to the m dimensional output layer. This means that the second weight matrix has dimensions $m \times k$:

$$P^{(3)} = W^{(2)}S^{(2)} + \mathbf{b}^{(2)}$$

Typically, we don't do any more processing, so we can think of the activation function f on the final layer as the identity function, $f(x) = x$.

$$S^{(3)} = P^{(3)} = W^{(2)}S^{(2)} + \mathbf{b}^{(2)} = W^{(2)}\left(f\left(W^{(1)}\mathbf{x} + \mathbf{b}^{(i)}\right)\right) + \mathbf{b}^{(2)}$$

To be consistent with all layers, we usually define the prestate at the input layer to be just $\boldsymbol{x}$, and the activation function as the identity, so the layer doesn't do anything but output the same as what is input. With that, the computations a neural network takes can be described by the chain:

$$\text{input} = (P^{(1)} \to S^{(1)}) \longrightarrow (P^{(2)} \to S^{(2)}) \longrightarrow (P^3 \to S^{(3)}) = \text{ output}$$

It is simple now to see how to add more layers. We can keep extending this chain almost indefinitely.

The Network Produces Composition(s)

There is one important way to look at this sequence of operations that has some implications when we compute derivatives: This sequence of operations is function composition, which is clear when we write the output layer in terms of the input layer:

$$S^{(3)} = W^{(2)}\left(f\left(W^{(1)}\mathbf{x} + \mathbf{b}^{(i)}\right)\right) + \mathbf{b}^{(2)}$$

If we add more layers, then the state at the third layer should be written again as $f(P^{(3)})$, because presumably, f would not now be the identity. In that case, the output at the added fourth layer would be:

$$S^{(4)} = W^{(3)}\left[f(W^{(2)}\left(f\left(W^{(1)}\mathbf{x} + \mathbf{b}^{(i)}\right)\right) + \mathbf{b}^{(2)})\right] + \boldsymbol{b}^{(3)}$$

and so on.

13.3 The Activation Function

The function f here is key. If f is linear, then the neural net is a linear network (we've already discussed those). Importantly, f on the hidden layer must be nonlinear. And importantly, in 1989, Cybenko proved the result about universal function approximation assuming that f is a **sigmoidal function**.

Definition: Suppose a function $\sigma(x)$ has the following properties:

- It is monotonically increasing.

- $\lim_{x \to -\infty} \sigma(x) = A$
- $\lim_{x \to \infty} \sigma(x) = B$

Then $\sigma(x)$ is called a **sigmoidal function**.

Common choices for the sigmoidal function include:

1. The log sigmoidal function (in Matlab, this is `logsig`)

$$\text{logsig}(x) = \frac{1}{1 + \mathrm{e}^{-x}}$$

2. The hyperbolic tangent (in Matlab, this is `tansig`)

$$\tanh(x) = \frac{\mathrm{e}^{2x} - 1}{\mathrm{e}^{2x} + 1}$$

Typically, we will use the first option, since its derivative is very easy to compute (see the exercises), and typically the so-called "linear activation function", $\sigma(x) = x$ is used at the input and output layer.

13.3.1 Recent Developments for the Activation Function

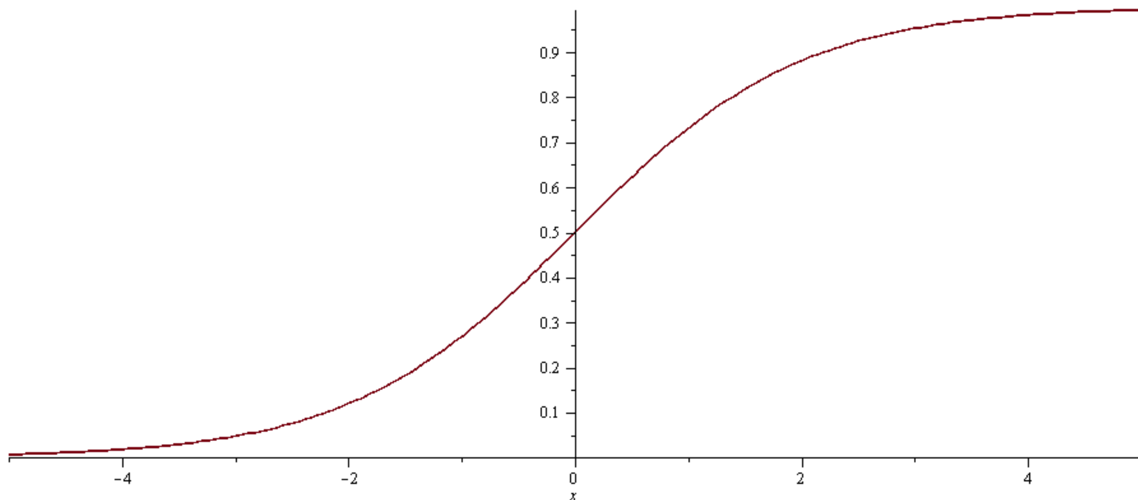
While the sigmoidal function was used by Cybenko in his proof, it has some shortcomings that didn't become clear until very large networks were being trained. The two main shortcomings:

- The "squashing" function becomes saturated.

  The sigmoidal only has a small interval on which it can keep input points apart (see the graph below). If $|x|$ is greater than about 2, then all positive points start to map to 1, and all negative points to 0 (this is explored more in the exercises).

- The vanishing gradient problem.

  From the graph, see if you can esimate the derivative. For most x, the derivative is less than 1. Recall that a neural net is function composition so that the derivative is multiplication. What happens when you multiply a bunch of numbers that are less than 1? The values go to zero- This is the vanishing gradient problem (we'll discuss this more in the section on training).



The Rise of ReLU

The activation function that most use now is the "rectified linear unit", or **ReLU**. Mathematically, this is defined as:

$$\text{ReLU}(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

The perhaps obvious issues with using this function are (1) nondifferentiable at 0, and (2) it becomes unbounded. Further, if a learning rate is set too high, it has been observed that the ReLU's can be "pushed" out so far that they are basically zero- this is the "Dying ReLU problem".

In spite of the difficulties, in 2011 it was found that the ReLU function is a viable alternative to the sigmoidal function, even though it does not have all the properties of a sigmoidal. A few short years later, in 2017 it was found to be the most popular activation function for deep networks.

Exercises

1. Sketch the ReLU transfer function and its derivative.

2. If $\boldsymbol{x} \in \mathbb{R}^n$ and our targets $\boldsymbol{t} \in \mathbb{R}^m$, and we use k nodes in the hidden layer, how many unknown parameters do we have to find?

   As you are constructing your network, keep this number in mind. In particular, you should have at least several data points for each unknown parameter that you are looking for.

3. We have to be somewhat careful when our data is badly scaled. For example, complete this table of values:

x	0	0.5	1	10	40	100
$\tanh(x)$						
$\texttt{logsig}(x)$						

   You should see that, while the change in x between neighboring points is getting very large, that the corresponding changes in y are going to zero. This is problematic- If your data is large, or the weights are large, then the sigmoidal may simply start to output a constant.

   This phenomenon goes by the name of *saturation.*

4. Some people like to scale the sigmoidal function by an extra parameter, β, that is $\sigma(\beta x)$. Show by sketching what happens to the graph of the sigmoidal (either the `tansig` or `logsig`) as you change β.

   It is not necessary to scale the sigmoidal, as this is equivalent to scaling the data instead (via the weights).

5. Show that if $\sigma(x)$ is the `logsig` function, then

   $$\sigma'(\beta x) = \beta\sigma(\beta x)(1 - \sigma(\beta x))$$

   This is the reason this function is so popular- It is very easy to compute its derivative.

6. Show, using the definition of the hyperbolic sine and cosine, that the hyperbolic tangent can be written as:

   $$\tanh(x) = \frac{\sinh(x)}{\cosh(x)}$$

7. Show that the hyperbolic tangent can be computed as:

$$\tanh(x) = \frac{2}{1 + \mathrm{e}^{-2x}} - 1$$

(Matlab claims that this version is faster, but warns about possible numerical error)

8. **Other Extensions of the activation function**

   Some other interesting activation functions can be used at the nodes. Here are a couple of unique ones- They are used to encode circular or spherical information:

   (a) The Circular Node (two inputs, two outputs per node):

   $$\sigma(x, y) = \left(\frac{x}{\sqrt{x^2 + y^2}}, \frac{y}{\sqrt{x^2 + y^2}} \right)$$

   (b) The Spherical Node (three inputs, three outputs):

   $$\sigma(x, y, z) = \left(\frac{x}{\sqrt{x^2 + y^2 + z^2}}, \frac{y}{\sqrt{x^2 + y^2 + z^2}}, \frac{z}{\sqrt{x^2 + y^2 + z^2}} \right)$$

   See [19, 20] for examples of how to implement the last two transfer function types.

13.4 Training and Error

To define a three layer neural network in the form $n - k - m$, we should first define the activation function f. Although we could define a different function for every neuron, we typically will use the same transfer function for all the neurons in a single layer.

Once that is done, then we have to find matrices $W^{(1)}, W^{(2)}$ and the bias vectors $\mathbf{b}^{(1)}, \mathbf{b}^{(2)}$. Altogether, this makes $(nk + k) + (mk + m)$ parameters. Ideally, we would have much more data than that in order to get good estimates. In any case, we want to minimize the usual sum of squared error:

$$E(W^{(1)}, W^{(2)}, \mathbf{b}^{(1)}, \mathbf{b}^{(2)}) = \frac{1}{2} \sum_{i=1}^{p} \|\mathbf{t}^{(i)} - \mathbf{y}^{(i)}\|^2$$

where $\mathbf{y}^{(i)}$ is the output of the neural net using the ith input. In the case of a single hidden layer, we have:

$$\mathbf{y}^{(i)} = W^{(2)} \left(\sigma \left(W^{(1)} \mathbf{x}^{(i)} + \mathbf{b}^{(1)} \right) \right) + \mathbf{b}^{(2)}$$

As usual, "training" means finding the weights and biases that minimize the error function, and now we're back at function optimization. We've discussed these methods before, but here is a short list:

- Method of Steepest Descent (or Gradient Descent).
- Stochastic Gradient Descent is an excellent one to use when we're programming it ourselves, since we only have to deal with the derivative one data point at a time.
- Newton's Method (an indirect method, solving for where the derivative of the error is 0).
- Conjuage Gradient (Search along the eigenvectors of the Hessian of the error)
- Levenburg-Marquardt (A combination of the techniques above).

These techniques are typically implemented in any good system- one can find these solvers in Matlab and in Python, look at `scipy.optimize.least_squares`. Before we go too much farther into the general case, let's take a look at a concrete example.

Most optimization algorithms will require that we obtain expressions to compute the partial derivatives of E with respect to the weights and biases. It is fortunate that, for large networks, there is a recursive algorithm to accomplish this called the **backpropagation of error**. In the next section, we're going to look at the computational steps of this algorithm, then after that, we'll prove that we have actually computed the desired partial derivatives.

13.5 Backpropagation of Error

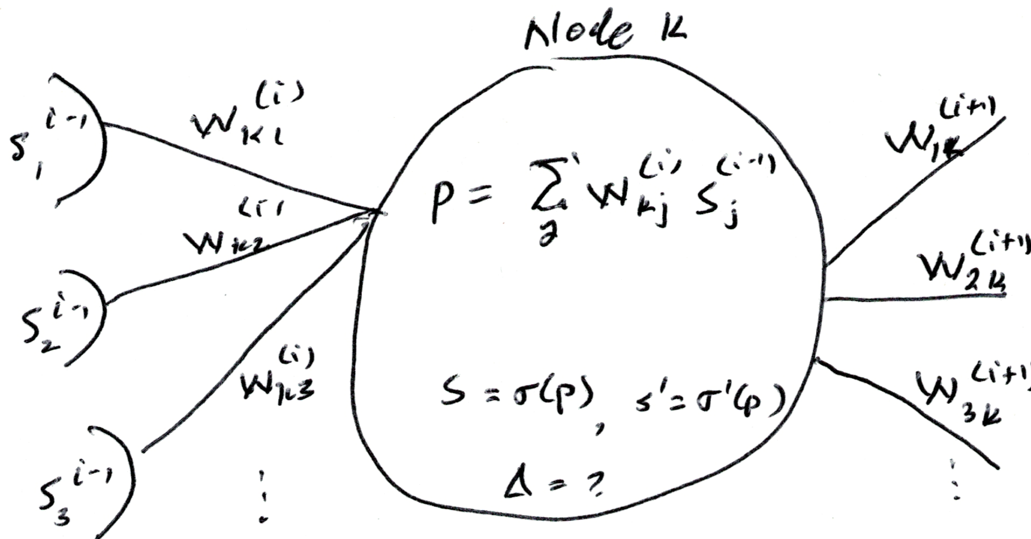
First, let's reconsider the single node of a neural network, and consider the computations it makes. Since we're looking to program this soon, think of each node as storing these values somewhere for future access.

When we go through what's called a **forward phase** for a network, we move from left to right computing the given values, where P is the prestate, S is the state (so S' is its derivative evaluated at P), and Δ is a placeholder for the **backwards phase**

In particular, looking at node k in layer i as in the diagram below, the computations are (layer-wise):

$$P^{(i)} = W^{(i)}S^{(i-1)} + \boldsymbol{b}^{(i)}, \qquad S^{(i)} = \sigma(P^{(i)}) = \sigma\left(W^{(i)}S^{(i-1)} + \boldsymbol{b}^{(i)}\right) \qquad {S^{(i)}}' = \sigma'(P^{(i)})$$

If N_{i-1}, N_i, N_{i+1} are the number of nodes in layers $i-1$, i, and $i+1$ respectively. The first operation through this layer represents an affine map from $\mathbb{R}^{N_{i-1}}$ to $\mathbb{R}^{N_i}$ that is the layer's prestate. So $W^{(i)}$ is an $N_i \times N_{i-1}$ matrix, and $P^{(i)}$ is a vector in $\mathbb{R}^{N_i}$, and so is $S^{(i)}$. The derivative is the same size, and the derivatives (evaluated at the current prestate) are stored in ${S^{(i)}}'$. In the picture below, P, S, and S' are all scalar values being computed specifically at node k.



There are two special cases:

- The input layer:
$$P^{(1)} = \boldsymbol{x}, \qquad S^{(1)} = \boldsymbol{x} \qquad S(1)' = \texttt{ones}$$

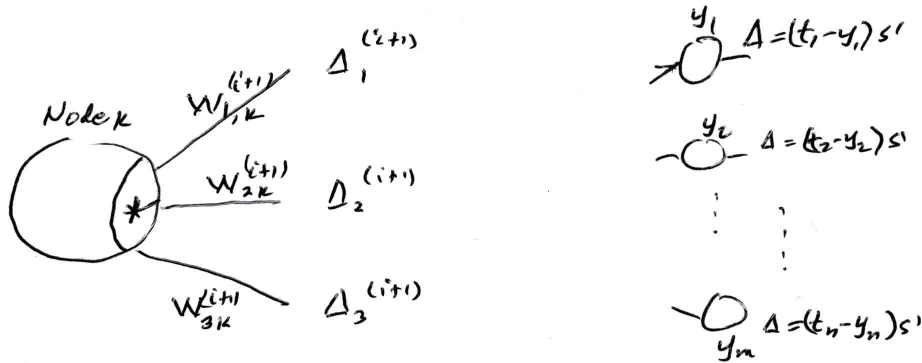
- At the output layer, we will make the assumption that the activation function is the identity. If we have L layers, then:

$$P^{(L)} = W^{(L)}S^{(L-1)} + \boldsymbol{b}^{(L)}, \qquad S^{(L)} = P^{(L-1)}, \qquad {S^{(L)}}' = \texttt{ones}$$

And the computation at the output layer finishes the forward phase.

The "backwards phase" for the network is to move information from right to left, where our goal is now to compute the values of Δ that were left blank.

Starting at the output layer, $\Delta^{(L)} = \boldsymbol{t} - \boldsymbol{y}$, which is the error between the target value and the output of the net. Notice that in that equation, Δ is a vector of values, one value for each output node, and in the picture to the right below, we included S', but if the activation function is the identity, this is just equal to 1.



The asterisk in the image above and to the left represents the computation we make for that node's Δ value. In this particular case, think of the $\Delta's$ in layer $i+1$ moving to the left so we have to multiply by the corresponding weight. We then sum those up and multiply by the derivative there. That is:

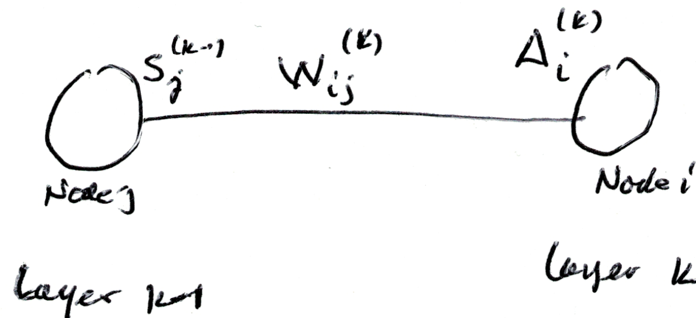
$$\Delta_k^{(i)} = S_k^{(i)'} \sum_{j=1}^{N_{i+1}} \Delta_j^{(i+1)} W_{jk}^{(i+1)}$$

This may look complicated, but think of it in terms of the reverse affine map from $\mathbb{R}^{N_{i+1}}$ to $\mathbb{R}^{N_i}$. Then we can compute all of the $\Delta's$ in layer i all at once:

$$\Delta^{(i)} = ((W^{(i+1)})^T \Delta^{(i+1)}) .* S^{(i)'}$$

Somewhere a linear algebra teacher is crying. OK, so just to be crystal clear, our multiplication is not a dot product, we are performing the multiplication (using the Matlab symbol `.*`) between two vectors *element-wise* so the result is also a vector of the same size. Also to be very clear, let's figure out the dimensions of everything there: $W^{(i+1)}$ is $N_{i+1} \times N_i$ (so the transpose reverses those), $\Delta^{(i)}$ has N_{i+1} elements, and $S^{(i)'}$ has N_i elements.

Once we reach the input layer, we're done. Now what do we do with these Δ values? I like to use the following memory device to help me remember how this goes. Below we see an edge, and we want to use the **state on the left** and the **Δ on the right** with the edge and weight connecting the two.



Then here is the big conclusion:

Backpropagation of Error

$$\Delta W_{ij}^{(k)} = S_j^{(k-1)} \Delta_i^{(k)} \tag{13.1}$$

The update rule for the weights is now (using gradient descent):

$$\text{new } W_{ij}^{(k)} = \text{old } W_{ij}^{(k)} + \alpha \Delta W_{ij}^{(k)}$$

where the plus sign is not a typo, and α is the learning rate.

Thinking back, this kind of update rule was a lot like Widrow-Hoff, and in that case, we were able to write the update using linear algebra so we could update weights all at once.

Update Rule Using Linear Algebra

Remember that matrix $W^{(i)}$ connects layer $i-1$ to layer i, so that $W^{(i)}$ is $N_i \times N_{i-1}$. In our update rule, we are taking states from layer $i-1$ (so we have a vector with N_{i-1} entries) and the deltas from layer i (so that is a vector with N_i entries. To give away the punch line, if we take the outer product between these two vectors (in the right order), we'll have a matrix the same size as $W^{(i)}$. You should verify this works by looking at the dimensions of the three objects below.

$$\Delta W^{(i)} = \Delta^{(i)} (S^{(i-1)})^T$$

And finally,

$$W_{\text{new}}^{(i)} = W^{(i)} + \alpha \Delta W^{(i)}$$

You might think we're done, but there are still three big questions we have to answer:

1. These computations were for a single data point. What do I do with p data points?
2. Is this really gradient descent? Prove it!
3. Where did the bias terms go??

These questions will be answered in the next section, where we do actually show that our computations do result in gradient descent.

13.6 Backprop Proved

The answer to the **first question** goes pretty quickly. With p data points, if we use the sum of squares as the error, then each of the 4 numbers we computed per node should be summed over all the input (that's before the backward phase). Some people use the average error, and in that case, you would average each of the 4 numbers over all the input.

For the **second question**, we'll need to break out our Calculus. The function we want to minimize is the error function. We'll put a 1/2 in the front so we don't have to deal with putting 2 in front of everything when we differentiate.

$$E = \frac{1}{2} \sum_{i=1}^{p} \|\boldsymbol{t}^{(i)} - \boldsymbol{y}^{(i)}\|^2$$

We'll show that our rules do indeed produce the gradient descent. Recall that W^l_{mn} connects node n in the layer to the left to node m in the layer to the right. Therefore, S^l_m is the state of node m in layer l (to the right of the edge labeled W^l_{mn}). Using these relationships, we can write:

$$\frac{\partial E}{\partial W^l_{mn}} = \frac{\partial E}{\partial S^l_m}\,\frac{\partial S^l_m}{\partial P^l_m}\,\frac{\partial P^l_m}{\partial W^l_{mn}} \tag{13.2}$$

The two values on the right can readily be computed:

$$\frac{\partial S^l_m}{\partial P^l_m} = \sigma'\left(P^l_m\right) \qquad \frac{\partial P^l_m}{\partial W^l_{mn}} = S^{l-1}_n \tag{13.3}$$

This leaves the first term which can be evaluated on the output layer L:

$$\frac{\partial E}{\partial S^L_m} = \frac{\partial E}{\partial y_m} = (t_m - y_m)(-1)$$

On the rest of the layers, the term can be defined recursively,

$$\frac{\partial E}{\partial S^l_m} = \sum_{j\in \text{nextlayer}} \frac{\partial E}{\partial S^{l+1}_j}\,\frac{\partial S^{l+1}_j}{\partial S^l_m} \tag{13.4}$$

Since $S^{l+1}_j = \sigma(P^{l+1}_j) = \sigma(W^{l+1}_{jm}S^l_m + \text{ other terms})$, the derivative will be

$$\frac{\partial S^{l+1}_j}{\partial S^l_m} = \sigma'(P^{l+1}_m)W^{l+1}_{jm}$$

Now we'll connect up the two sets of notation:

Definition: We'll define Δ as the product of the first two terms of Equation (13.2):

$$\Delta^l_m = -\frac{\partial E}{\partial S^l_m}\,\frac{\partial S^l_m}{\partial P^l_m} = -\frac{\partial E}{\partial S^l_m}\sigma'(P^l_m)$$

Therefore, on the output layer,

$$\Delta^L_m = \frac{\partial E}{\partial S^L_m}\,\frac{\partial S^L_m}{\partial P^L_m} = -(t_m - y_m)(-1)\sigma'(P^L_m)$$

which matches Equation (**??**). Now, using Equation (13.4), the nodes on the previous layer are computed:

$$\Delta^l_m = -\frac{\partial E}{\partial S^l_m}\,\frac{\partial S^l_m}{\partial P^l_m} = \left(\sum_{j\in \text{layer}l+1} -\frac{\partial E}{\partial S^{l+1}_j}\,\frac{\partial S^{l+1}_j}{\partial S^l_m}\right)\sigma'(P^l_m) =$$

$$\sigma'(P^l_m)\left(\sum_{j\in \text{layer}l+1} -\frac{\partial E}{\partial S^{l+1}_j}\,\sigma'(P^{l+1}_m)W^{l+1}_{jm}\right) =$$

$$\sigma'(P^l_m)\left(\sum_{j\in\text{ layer}l+1} \Delta^{l+1}_j W^{l+1}_{jm}\right)$$

And this is Equation (**??**). Finally, by Equation (13.3), we can write:

$$-\frac{\partial E}{\partial W^l_{mn}} = \left(-\frac{\partial E}{\partial S^l_m}\frac{\partial S^l_m}{\partial P^l_m}\right)\frac{\partial P^l_m}{\partial W^l_{mn}} = \Delta^l_m S^{l-1}_n$$

which proves that the update rule in Equation (13.1) is indeed gradient descent. Now for the second question: How should we deal with bias terms?

For the bias term, we will add a node to each layer, but for these nodes, the state is the constant function, $S = \sigma(x) = 1$, and the weight connecting this node to node k of the next layer could be labeled b_k^l. That also means that Δ for a bias node is 0, but now Equation (**??**) becomes:

$$b_k^l = b_k^l + \epsilon \Delta_k^l$$

where the Δ_k^l is the value of Δ to the node to which b_k^l is connected.

13.7 Simple Construction of a Feed Forward Neural Net

The neural net is simple to construct if we view it in terms of its layers.

- Initialize the $n - k - m$ network:

  Build the weights and biases with random numbers. Be sure to pay attention to dimensions.

$$W^{(1)} \text{ is } k \times n \qquad \boldsymbol{b}^{(1)} \text{ is } k \times 1$$

$$W^{(2)} \text{ is } m \times k \qquad \boldsymbol{b}^{(2)} \text{ is } m \times 1$$

- Compute the output of a neural net given the weights, biases. Here we'll assume that the activation function is $\sigma(x)$, and has been determined. The computation proceeds in layers. For the three layer network (layers 0, 1, and 2), we'll have the series of computations. This is a forward pass:
  - $P^{(1)} = W^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}$
  - $S^{(1)} = \sigma(P^{(1)})$. Compute $S(1)'$ as well.
  - $P^{(2)} = W^{(2)}S^{(1)} + \boldsymbol{b}^{(2)}$
  - $S^{(2)} = P^{(1)}$. Then $S(1)'$ is a vector of ones. The output is $S^{(2)}$.
- A backwards pass for backpropagation of error:
  - $\Delta^{(2)} = (\mathbf{t} - \mathbf{y})$ (This is a vector in $\mathbb{R}^m$).
  - $\Delta^{(1)} = (W^{(2)})^T \Delta^{(2)}$ `.*` $S^{(i)'}$ (We use Matlab's `.*` to denote elementwise mutliplication)
- Compute the changes to the weights and biases:
  - $\Delta W^{(1)} = \Delta^{(1)} \mathbf{x}^T$
  - $\Delta W^{(2)} = \Delta^{(2)} (S^{(1)})^T$
  - $\Delta \mathbf{b}^{(1)} = \Delta^{(1)}$
  - $\Delta \mathbf{b}^{(2)} = \Delta^{(2)}$
- Apply the changes:

$$W^{(1)} = W^{(1)} + \alpha \Delta W^{(1)}, \quad W^{(2)} = W^{(2)} + \alpha \Delta W^{(2)}, \quad \mathbf{b}^{(1)} = \mathbf{b}^{(1)} + \alpha \Delta \mathbf{b}^{(1)} \quad \mathbf{b}^{(2)} = \mathbf{b}^{(2)} + \alpha \Delta \mathbf{b}^{(2)}$$