Neural Nets

To give you an idea of how new this material is, let's do a little history lesson. The origins 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 really 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.

"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."

As we discussed earlier, 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 two types: a linear neural network (using Widrow-Hoff training rule), and the RBF (using Orthogonal Least Squares). We now turn to the workhorse of the neural network community: The feed forward neural network.

General Model Building

As with our other types of neural networks, we assume that we have p data pairs, $(\mathbf{x}_1, \mathbf{t}_1), (\mathbf{x}_2, \mathbf{t}_2), \cdots, (\mathbf{x}_p, \mathbf{t}_p)$ (the letter t is for *target*), and we are looking to build a function F so that ideally,

$$F(\mathbf{x}_i) = \mathbf{t}_i$$

However, we will typically allow for error, and we typically model the error ϵ_i using a normal distribution:

$$\mathbf{t}_i = F(\mathbf{x}_i) + \vec{\epsilon_i}$$

If we assume that the function F depends on parameters (weights and biases, like the RBF), then we might state it as an optimization problem- Find the function F that minimizes the error function:

$$E = \frac{1}{2}\sum_{i=1}^{p} \|\mathbf{t}_i - F(\mathbf{x}_i)\|^2$$

so that E is a function of the parameters of F, and we would go about determining the values of the parameters that minimize the error- And this is what we will do.

Earlier in this class, $F(\mathbf{x}_i = W\mathbf{x}_i + \mathbf{b}$. This was the case of the linear neural network. More recently, we worked with the RBFs, so that

$$F(\mathbf{x}_i) = W\phi(\text{edm}(\mathbf{x}_i, \text{centers})) + \mathbf{b}$$

We will now work through the construction of the multi-layered feed forward network.

The Feed-Forward Neural Network

As in the linear network, we will assume that along the dendrites, our signals can be scaled or re-polarized. If we use k neurons, then this is a linear mapping from $\mathbb{R}^n \to \mathbb{R}^k$, and W_1 is a $k \times n$ matrix. A vector $\mathbf{b}_1$ represents the "standing voltage" of the neuron, and can be added as a bias term:

$$\mathbf{x} \to W_1\mathbf{x} + \mathbf{b}_1$$

Rather than stopping here, we will now call this the *prestate* of the layer of neurons. Next, a nonlinear transfer function is applied, $\sigma(r)$. This is typically called a sigmoidal function because of its shape. The result is a vector in $\mathbb{R}^k$ known as the *state* of the layer of neurons. Adding that to our diagram, we have:

$$\mathbf{x} \to W_1\mathbf{x} + \mathbf{b}_1 = \vec{P} \to \vec{S} = \sigma(\vec{P})$$

Finally, the signal can be recombined in a linear way to produce an output vector $\mathbf{y} \in \mathbb{R}^m$ using W_2 that is $m \times k$ and another bias vector, $\mathbf{b}_2 \in \mathbb{R}^m$:

$$\mathbf{x} \to W_1\mathbf{x} + \mathbf{b}_1 = \vec{P}_i \to \vec{S}_i = \sigma(\vec{P}_i) \to W_2\vec{S}_i + \mathbf{b}_2$$

Putting it all together, we could write the function F explicitly:

$$F(\mathbf{x}_i) = W_2\left(\sigma\left(W_1\mathbf{x}_i + \mathbf{b}_1\right)\right) + \mathbf{b}_2$$

so that, with σ defined, F becomes a function of the *weights* W_1, W_2, and the biases $\mathbf{b}_1, \mathbf{b}_2$.

Defining the Network Architecture

We have constructed what many people call a two layer network (although I typically say it is three layers- Some people don't count the input layer as a real layer):

- The first "layer" is called the *input layer.* If $\mathbf{x}_i \in \mathbb{R}^n$, then the input layer has n "nodes".
- The next layer is called the *hidden layer*, and it consists of k nodes (where k is the number of neurons we're using). The mapping from the input layer to the hidden layer is performed by our first affine map, then σ is applied to that vector.
- The last layer is called the *output layer*, and if $\mathbf{y} \in \mathbb{R}^m$, then the output layer has m nodes.

We did not need to stop with only a single hidden layer- Some researchers like to use two hidden layers as a default neural network. In that case, the mapping (in stages) would look like:

$$\begin{array}{lll}
\mathbf{x} & \to W_1\mathbf{x} + \mathbf{b}_1 & \text{Prestate of layer 1} \\
 & \to \sigma(W_1\mathbf{x} + \mathbf{b}_1) & \text{State of layer 1} \\
 & \to W_2\left(\sigma(W_1\mathbf{x} + \mathbf{b}_1)\right) + \mathbf{b}_2 & \text{Prestate of layer 2} \\
 & \to \sigma(W_2\left(\sigma(W_1\mathbf{x} + \mathbf{b}_1)\right) + \mathbf{b}_2) & \text{State of layer 2} \\
 & \to W_3\sigma(W_2\left(\sigma(W_1\mathbf{x} + \mathbf{b}_1)\right) + \mathbf{b}_2) + \mathbf{b}_3 & \text{State of layer 3 (output)}
\end{array}$$

One can imagine that many layers are possible. However, it has been shown that, at least theoretically, one needs to have only one hidden layer to perform the function approximation to arbitrarily small error. Practically, sometimes it is more efficient to use multiple layers of a small number of neurons than a single layer with a large number of neurons.

Definition: The *architecture* of the neural network is typically defined by stating the number of neurons in each layer. For example, a $2 - 3 - 4$ network has one hidden network of three neurons, and maps $\mathbb{R}^2$ to $\mathbb{R}^4$.

The parameters of a neural network

We have seen that, to define a neural network, we need to define: The weights W_1, W_2 (and more, if we use more layers), the biases $\mathbf{b}_1, \mathbf{b}_2$, and the transfer function ϕ. Although we could define a different ϕ for every neuron, we typically will use the same transfer function for all the neurons in a single layer.

Therefore, when we "train" a neural network, we will usually pre-define ϕ, and then we will find the weights and biases that will (hopefully!) minimize the error function.

$$E(W_1, W_2, \mathbf{b}_1, \mathbf{b}_2) = \frac{1}{2}\sum_{i=1}^{p} \|\mathbf{t}_i - \mathbf{y}_i\|^2$$

where $\mathbf{y}_i$ is the output of the neural net. In the case of a single hidden layer, we have:

$$\mathbf{y}_i = W_2\left(\sigma\left(W_1\mathbf{x}_i + \mathbf{b}_1\right)\right) + \mathbf{b}_2$$

The transfer function

In a neuron, the incoming signals to the cell body must usually surpass some lowest trigger value before the signal is sent out. A graph of this would be a step function, where the step is at trigger.

This is not a good function using notions from Calculus because the voltage function is not continuous and not differentiable at the trigger . We replace the step function by any function that is:

- Increasing.
- Differentiable
- Has finite horizontal asymptotes at $\pm\infty$.

Such a function generally looks like an extended "S"- We call it a *sigmoidal function.*

There are many ways we could define a sigmoidal, but here are some standard choices (going from most to least used):

- $$\sigma(r) = \frac{1}{1 + \mathrm{e}^{-r}}$$
  Matlab calls this the "logsig" function.
- $$\sigma(r) = \arctan(r)$$
  Matlab does not use this one.
- $$\sigma(r) = \tanh(r) = \frac{\mathrm{e}^{2r} - 1}{\mathrm{e}^{2r} + 1}$$
  Matlab calls this one "tansig".

Exercises

1. Compute the limits as $x \to \pm\infty$ for the two types of sigmoidal functions that Matlab uses. Show that they are also monotonically increasing functions.

2. Let $\sigma(x) = \frac{1}{1+\mathrm{e}^{-\beta x}}$. Show that
$$\sigma'(x) = \beta\sigma(x)(1 - \sigma(x))$$

3. 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.

4. 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
$\tanh(x)$						
`logsig`(x)						

   What do you see as x becomes very large? This phenomenon goes by the name of *saturation.*

5. 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).

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. **Extensions of the transfer function**

   Some other interesting transfer 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 [**?**, **?**] for examples of how to implement the last two transfer function types.

Training a Neural Network

As we said previously, training a neural network means to find weights and biases that minimize the error function. There are several techniques available to us for doing this- among them:

- Method of Steepest Descent (or Gradient Descent)
- 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).

For us, we can practice using Gradient Descent, and for the other techniques we'll rely on Matlab.

Going into the next section, recall that in the exercises, we showed that, if $\beta = 1$, then:

$$\sigma'(x) = \sigma(x)\left(1 - \sigma(x)\right)$$

Backpropagation of Error

We start with a simple example: A 1-1-1 network:

$$x \to w_1x + b_1 = P \to \sigma(w_1x + b_1) = S \to w_2\sigma(w_1x + b_1) + b_2$$

where P represents the "prestate", and S represents the state of the node.

Given target t, the error is

$$E(w_1, w_2, b_1, b_2) = \frac{1}{2}(t - y)^2 = \frac{1}{2}(t - (w_2\sigma(w_1x + b_1) + b_2))^2$$

We want to minimize the error, so we move in the opposite direction of the gradient. Suppose we let the symbol u denote a generic parameter (either a weight or a bias). Then given a particular value of the parameter, it is updated to (hopefully) get a better error. Using gradient descent, u is updated by:

$$u_{\text{new}} = u_{\text{old}} - \alpha\frac{\partial E}{\partial u} = u_{\text{old}} + \alpha\Delta u$$

where α is called the **learning rate**, and the change in u is computed via the chain rule on the error.

Notice that we incorporated the negative sign into Δu- It will become clear why we did that (its because of the $(t - y)$ term- the derivative will always be negative $t - y$). In particular,

$$\Delta u = -\frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial u} = -(t - y) \cdot -\frac{\partial y}{\partial u} = (t - y)\frac{\partial y}{\partial u}$$

Now let us compute these partial derivatives:

$$\begin{array}{llll}
\frac{\partial y}{\partial w_2} = S \Rightarrow \Delta w_2 = (t-y)S & & \frac{\partial y}{\partial b_2} = 1 \Rightarrow \Delta b_2 = (t-y) & \\
\frac{\partial y}{\partial w_1} & = w_2\sigma'(P) \cdot x & \frac{\partial y}{\partial b_1} & = w_2\sigma'(P) \\
\Delta w_1 & = (t-y)w_2\sigma'(P) \cdot x & \Delta b_1 & = (t-y)w_2\sigma'(P)
\end{array}$$

Matlab and the Feedforward Network

Here is an example training session:

```
P=-1:0.1:1;
T=sin(pi*P)+0.1*randn(size(P));
net = newff(P,T,10,{'tansig','logsig'});
Y = sim(net,P);
plot(P,T,P,Y,'o');

title('Before Training');
net = train(net,P,T);
```

```
X = linspace(-1,1,200);
Y = sim(net,X);

figure(2)
plot(P,T,'ko',X,Y)
title('After Training');
```

Bad things that might happen...

1. A particularly bad random set of initial weights and biases might be assigned. You should always try training several times to make sure that your error is as small as it should be- It is easy to get locked into a local minimum!

2. Saturation. This is when the data is badly scaled. The problem has to do with our sigmoidal function. An example might be in order: Consider the table of values

x	-1	1	5	8	10	50	5000
$\sigma(x)$	0.269	0.731	0.993	0.9997	1.00	1.00	1.00

   We see that the transfer function begins to output 1 for ANY large number, so we say that it has lost its ability to distinguish between input patters (the function has become saturated). The same behavior happens for very negative input values as well.

   If your network begins to output the same numbers for wildly different inputs, then this is probably the reason (the weights could be large- See below).

3. Your data may be badly scaled. For example, suppose you have 4 dimensional input, and it represents temperatures from 200 degrees to 300 degrees in the first dimension, error values from 0.0005 to 0.001 in the second dimension, altitudes like 2000 to 5000 feet in the third dimension, and integers from 1 to 10 in the fourth. Here are some samples:

   $$(250, 0.0001, 2450, 6)$$

   The second column will disappear in terms of the network- the error minimization will end up focusing almost entirely on the third column.

   If possible, try to keep all of the scalings similar. For example, scale so that each dimension has mean zero and unit standard deviation (mean subtract and scale by the inverse of the standard deviation).

4. Too many nodes in the hidden layer: Use the test set/validation set to be sure you're not memorizing the data (the default settings work pretty well).

5. You should take a quick look at the magnitude of the numbers in the weights and biases- They should all be "reasonable"- Numbers that are out of scale with respect to the others should be suspect (and can lead to saturation). Recommended: `hintonw(net.IW{1,1})` and `hintonw(net.LW{2,1})`. The colors are negative (versus positive), and the sizes of the squares are related to the magnitude of the numbers.