Reminder: Method of Gradient Descent

The Derivative of the Sigmoidal

We will have the need to compute the derivative of the sigmoidal function. The function

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

is especially attractive numerically, since the derivative is easy to compute. Show that

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

Backpropagation of Error

We start with a simple example: One input node, one node in the hidden layer, and one output node. In this case, the function is:

$$x \quad \rightarrow [\quad P = w_1 x + b_1 \quad \rightarrow \quad S = \sigma(P) = \sigma(w_1 x + b_1) \quad] \rightarrow \quad y = w_2 S + b_2$$

where the hidden node is inside the brackets, P represents the "prestate", and S represents the state of the node.

Given target t, the error is

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

We want to minimize the error, so we move in the opposite direction of the gradient. That is, we update all parameters u 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, where u is a placeholder for the parameters w_1, w_2, b_1, b_2.

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:

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

We will now proceed by making our one dimension into multidimensional (with one data point). We will assume that $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{y} \in \mathbb{R}^m$, and there are k nodes in the hidden layer. Therefore, the network performs the following sequence to forward propagate a data point:

- $\mathbf{p} = W_1\mathbf{x} + \mathbf{b}_1$

  where W_1 is $k \times n$, and $\mathbf{b}_1$ is a vector of biases in $\mathbb{R}^k$.
- $\mathbf{s} = \sigma(\mathbf{p})$, where σ is applied elementwise to $\mathbf{p}$
- $\mathbf{y} = W_2\mathbf{s} + \mathbf{b}_2$

Now, compute the derivatives from the output to the input (this is back-propagating the error):

1. The error at the output layer: $\vec{\delta_2} = \mathbf{t} - \mathbf{y}$
2. The error at the middle layer:

   $$\vec{\delta_1} = W_2^T\vec{\delta_2} \cdot \sigma'(\mathbf{p}) = W_2^T\vec{\delta_2} \cdot \mathbf{s} \cdot (1 - \mathbf{s})$$

   where multiplication is done componentwise (look at the sizes: $k \times m, m \times 1$ times $k \times 1$)

   Now change the weights:

   $$\Delta W_1 = \vec{\delta_1}\mathbf{x}^T \qquad \Delta W_2 = \delta_2\mathbf{s}^T$$

   $$\Delta b_1 = \vec{\delta_1} \qquad \Delta b_2 = \vec{\delta_2}$$

Now we can minimize the error function (sum of squares error) using an online version of gradient descent.

Numerical Example:

Let us construct a $2-2-1$ network- That means there are 2 nodes at the input layer (where the data comes in- This is the dimension of the input data), there are 2 nodes in the hidden layer, and one output dimension.

We will use the standard sigmoidal function, $f(x) = 1/(1 + \mathrm{e}^{-x})$ so we can take advantage of the derivative. Before going any further, you should think about the dimensions of the parameters W_1, W_2, b_1, b_2.

We will forward propagate a data point $\mathbf{x}$ given the weights and biases below:

$$\mathbf{x} = \begin{bmatrix} 0.5 \\ 1.5 \end{bmatrix}, \quad t = 1.2, \quad W_1 = \begin{bmatrix} 0.5 & -0.1 \\ -0.3 & 0.3 \end{bmatrix}, \quad \mathbf{b}_1 = \begin{bmatrix} 0.2 \\ -0.2 \end{bmatrix}$$

$$W_2 = \begin{bmatrix} 0.3 & 0.3 \end{bmatrix} \qquad b_2 = 0.8$$

Forward propagate the data point $\mathbf{x}$:

$$\sigma(W_1\mathbf{x}+\mathbf{b}_1) = \sigma\left(\begin{bmatrix} 0.5 & -0.1 \\ -0.3 & 0.3 \end{bmatrix}\begin{bmatrix} 0.5 \\ 1.5 \end{bmatrix} + \begin{bmatrix} 0.2 \\ -0.2 \end{bmatrix}\right) =$$

$$\sigma\left(\begin{bmatrix} 0.3 \\ 0.1 \end{bmatrix}\right) = \begin{bmatrix} 0.5744 \\ 0.5250 \end{bmatrix}$$

We think of the vector $[0.3, 0.1]^T$ as the **prestate** P and the last vector above $[0.57, 0.52]^T$ as the **state**, S, of the nodes in the hidden layer.

To finish,

$$W_2S + b_2 = 1.1298$$

Now we backpropagate the error. We wanted $t = 1.2$, so

- $\delta_2 = t - y = 0.0702$
- In Matlab notation for the element-wise multiplication,

$$\delta_1 = W_2^T\delta_2. * \sigma'(P) = W_2^T\delta_2. * S. * (1-S) =$$

$$\begin{bmatrix} 0.3 \\ 0.3 \end{bmatrix}(0.0702)\,.*\begin{bmatrix} 0.5744 \\ 0.5250 \end{bmatrix}.*\left(\begin{bmatrix} 1 \\ 1 \end{bmatrix} - \begin{bmatrix} 0.5744 \\ 0.5250 \end{bmatrix}\right) = \begin{bmatrix} 0.0051 \\ 0.0052 \end{bmatrix}$$

And we update the weights and biases by the following changes. Note in particular the sizes (dimensions) of the weights and biases:

$$\Delta W_2 = \delta_2 S^T = (0.0702)\begin{bmatrix} 0.5744 & 0.5250 \end{bmatrix} = \begin{bmatrix} 0.0403 & 0.0368 \end{bmatrix}$$

$$\Delta W_1 = \delta_1\mathbf{x}^T = \begin{bmatrix} 0.0051 \\ 0.0052 \end{bmatrix}\begin{bmatrix} 0.5 & 1.5 \end{bmatrix} = \begin{bmatrix} 0.0026 & 0.0077 \\ 0.0026 & 0.0079 \end{bmatrix}$$

and the changes in the biases:

$$\Delta\mathbf{b}_1 = \begin{bmatrix} 0.0051 \\ 0.0052 \end{bmatrix} \qquad \Delta b_2 = 0.0702$$

Summary so far: We can construct a simple feedforward neural network and minimize the error by updating the weights and biases by using the backpropagation algorithm.

We see that a neural network is defined by its architecture. That means defining the number of nodes in each layer, and the transfer function σ of eacy node/layer.

Many people have the sigmoidal function on every node in the hidden and output layers, but that is not necessary (that was part of Cybenko's theorem we discussed in class).

The following is the script file: `backpropexamp1.m`

```
%% Numerical example by hand- Neural net 2-2-1

lr=0.1;  %learning rate, or alpha

W1=[0.5 -0.1;-0.3 0.3];
b1=[0.2;-0.2];
W2=[0.3 0.3];
b2=0.8;

x=[0.5;1.5];
t=1.2;

for j=1:10
    %Forward propagation:
    P=W1*x+b1;
    S=sigma(P);
    y=W2*x+b2;
    err(j)=0.5*(t-y)^2;

    %Backprop
    delta2=t-y;
    delta1=W2'*delta2.*S.*(ones(size(S))-S);

    DeltaW2=delta2*S';
    DeltaW1=delta1*x';
    Deltab1=delta1;
    Deltab2=delta2;

    %Update parameters
    W1=W1+lr*DeltaW1;
    W2=W2+lr*DeltaW2;
    b1=b1+lr*Deltab1;
    b2=b2+lr*Deltab2;
end

plot(err)
```

Using Matlab and the feedforward neural network. Before we formally get started, here's a quick template:

```
P = [0 1 2 3 4 5 6 7 8 9 10]; %Domain is dim x numpts
T = [0 1 2 3 4 3 2 1 2 3 4];  %Range is dim x numpts
net = newff(P,T,5);  %5 nodes in the hidden layer

%Set training parameters:
net.trainParam.epochs = 100;  %How many times should we train?
net = train(net,P,T);

P1=linspace(0,10);  %Get new domain data
Y = sim(net,P1);     %Get new outputs
plot(P,T,'ro',P1,Y)
```

The `newff` command

The `newff` command has many options (See the help file, `doc newff`). In its basic form, it creates the network structure- It has a lot of sub-parts, but only a few are of interest. Keeping with our usual notation, we have $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{y} \in \mathbb{R}^m$, and k nodes in the hidden layer.

- Our W_1 is $k \times n$ and is `net.IW{1,1}`
- Our W_2 is $m \times k$ and is `net.LW{2,1}`
- Our $\mathbf{b}_1$ is $k \times 1$ and is `net.b{1}`
- Our $\mathbf{b}_2$ is $m \times 1$ and is `net.b{2}`

Additionally, sometimes it is useful to change some of the parameters. Here are a few popular choices:

- `net.trainParam.epochs` This is the number of times to go through the training algorithm (default is 100).
- `net.trainParam.goal` Set the error goal- The default is 0.
- `net.trainParam.show` How often should the graph be updated? Default: Every 25 iterations.

The current version of the Neural Network Toolbox also contains a method for automatically sorting the data into training, validation (which we seldom use), and testing. The parameters are shown below, with their default values:

- `net.divideFcn = 'dividerand'` (Default- We won't change it)
- `net.divideParam.trainRatio = 0.6` (60% is the default)
- `net.divideParam.valRatio   = 0.2` (Uses a validation set)
- `net.divideParam.testRatio  = 0.2` (Uses a test set)

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.