Matlab and the RBF

It is convenient to use Matlab's built-in functions where we can, but sometimes the parameters we want to inspect are buried inside data structures. Here we look at where our training parameters are when we use Matlab's routines.

Here is a quick example, where P is the input (for "patterns").

```
P=linspace(-2,2,50);
T=sin(3*P)+0.2*randn(size(P));
eg=0.05;  %eg is error goal
sc=1;       %sc is scaling for the RBF
net=newrb(P,T,eg,sc);
```

When this command is run, we have the data structure called "net". Let's see how Matlab defines the RBF.

Network Structure for the RBF

- The transfer function is the Gaussian (we'll need to discuss the actual width used).

  The scaling of the Gaussian is contained in the vector $\mathbf{b}_1$ =`net.b{1}` (we'll show what we mean by this in the computations below).

- The matrix of centers is located in: `net.IW{1,1}` (note the curly braces).

  In this example, the matrix had dimensions 6×1 (that's 6 centers with dimension 1). If you want to extract the centers as a matrix C, we would write:

  ```
  C=net.IW{1,1};
  ```

- The weights W are in: `net.LW{2,1}`

- What we referred to as the bias is in `net.b{2}`

If we track the sequence of steps we perform to transform a set of vectors in the domain, x, into the function output,

```
% Here is some data:
P=linspace(-2,2,50);
T=sin(3*P)+0.2*randn(size(P));

%Train the RBF
net=newrb(P,T,0.05,1);


xx=linspace(-2,2);  %New data in the domain
NumPts=length(xx);  %Used below in computing A1
```

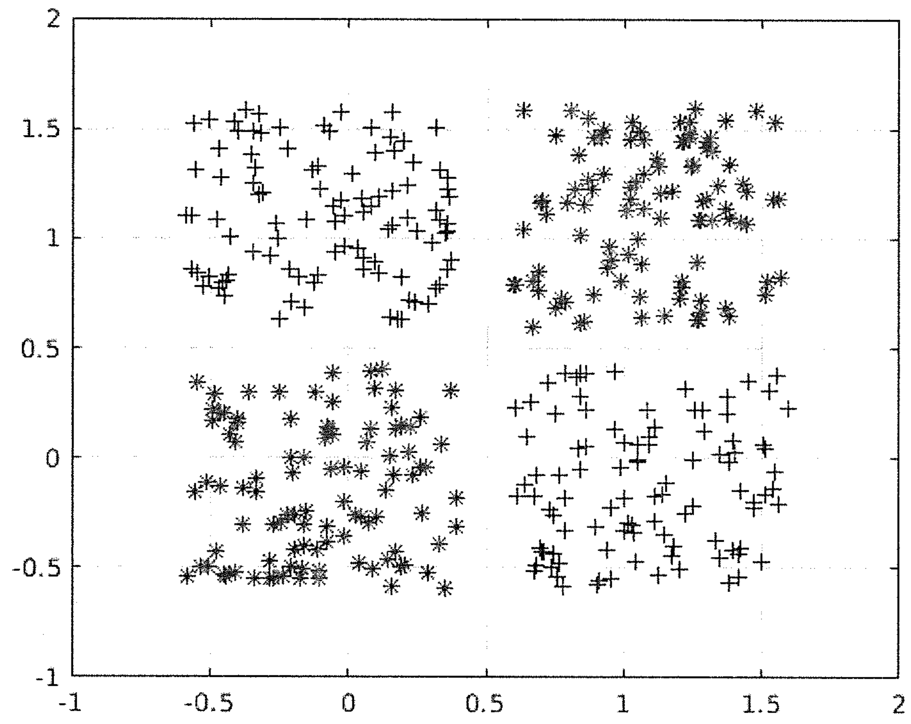
```
%Here are the relevent parameters from the network structure.
Centers=net.IW{1,1};
W=net.LW{2,1};
b1=net.b{1};  %Numcenters x 1- This is the scaling factor for the Gaussian
b2=net.b{2};  %Bias term (See below)

%Now compute the network output "by hand":
A=edm(xx',Centers);           %This is the "edm" function we wrote.
A1=A.*repmat(b1',NumPts,1);  %Multiply by the scaling factor before computing phi

Phi=rbf1(A1,1,1);
Yout=W*Phi'+b2;

%Get the output using Matlab's built in routine
Yout2=sim(net,xx);

% You should see Yout=Yout2:
plot(P,T,'k*',xx,Yout,xx,Yout2);
```

Example 1: Train a RBF using newrbe, plot results

Table of Contents

Step 1: Create the data, plot it:

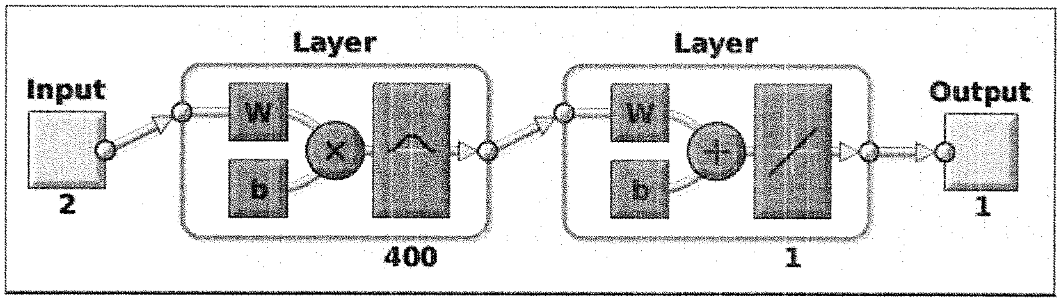
```
close all, clear all, clc, format compact

% Num samples in each cluster
K=100;
% Cluster offset
q=0.6;
% Define two groups- A and B
A= [rand(1,K)-q rand(1,K)+q;
    rand(1,K)+q rand(1,K)-q];
B = [rand(1,K)+q rand(1,K)-q;
rand(1,K)+q rand(1,K)-q];
% plot data
plot(A(1,:),A(2,:),'k+',B(1,:),B(2,:),'b*')
grid on
hold on

% Targets will be +/-1
a=-1; b=1;
P=[A B];
T=[repmat(a,1,length(A)) repmat(b,1,length(B))];
```



Step 2: Construct and train an RBF:

```
% Choose a spread constant:
spread=1;
% Create a neural network (exact RBF)
net=newrbe(P,T,spread);
view(net);
```

Warning: Rank deficient, rank = 107, tol = 1.780798e-12.

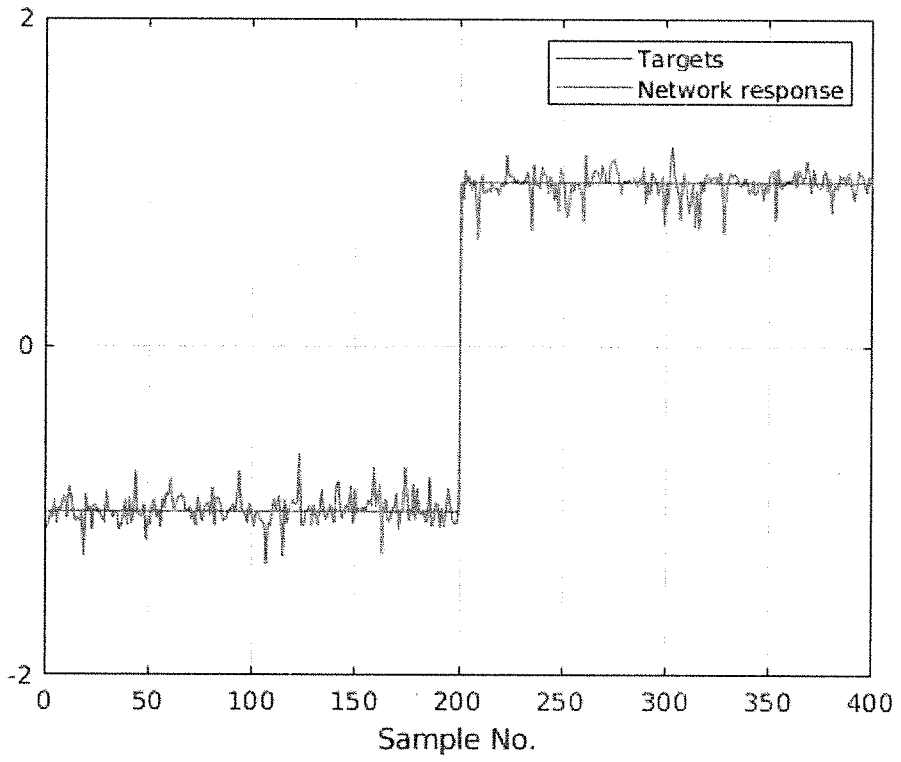


Step 3: Visualize results

```
Y = net(P);
% calculate [%] of correct classifications
correct = 100 * length(find(T.*Y > 0)) / length(T);
```

```
fprintf('\n Spread = %.2f\n',spread)
fprintf('Num of neurons = %d\n',net.layers{1}.size)
fprintf('Correct class = %.2f %%\n',correct)
% plot targets and network response
figure;
plot(T')
hold on
grid on
plot(Y','r')
ylim([-2 2])
set(gca,'ytick',[-2 0 2])
legend('Targets','Network response')
xlabel('Sample No.')
```

```
 Spread = 1.00
Num of neurons = 400
Correct class = 100.00 %
```

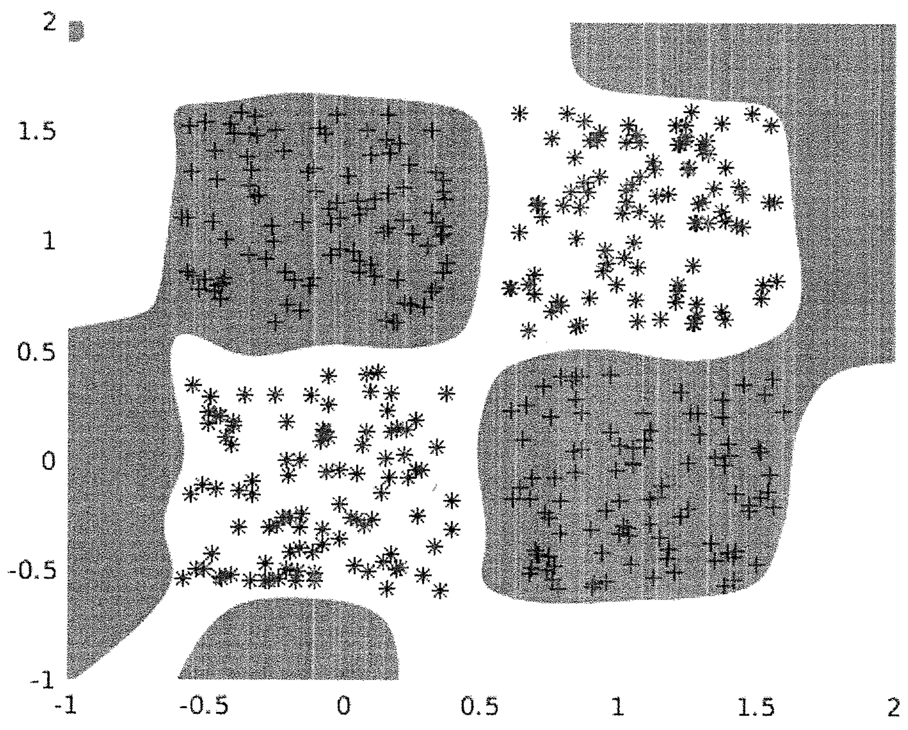


Visualization, continued: Classification regions in the plane.

generate a grid

```
span = -1:.025:2;
```

```
[P1,P2] = meshgrid(span,span);
pp = [P1(:) P2(:)]';
% simualte neural network on a grid
aa = sim(net,pp);
% plot classification regions based on MAX activation
figure(1)
ma = mesh(P1,P2,reshape(-aa,length(span),length(span))-5);
mb = mesh(P1,P2,reshape( aa,length(span),length(span))-5);
set(ma,'facecolor',[1 0.2 .7],'linestyle','none');
set(mb,'facecolor',[1 1.0 .5],'linestyle','none');
view(2)
```



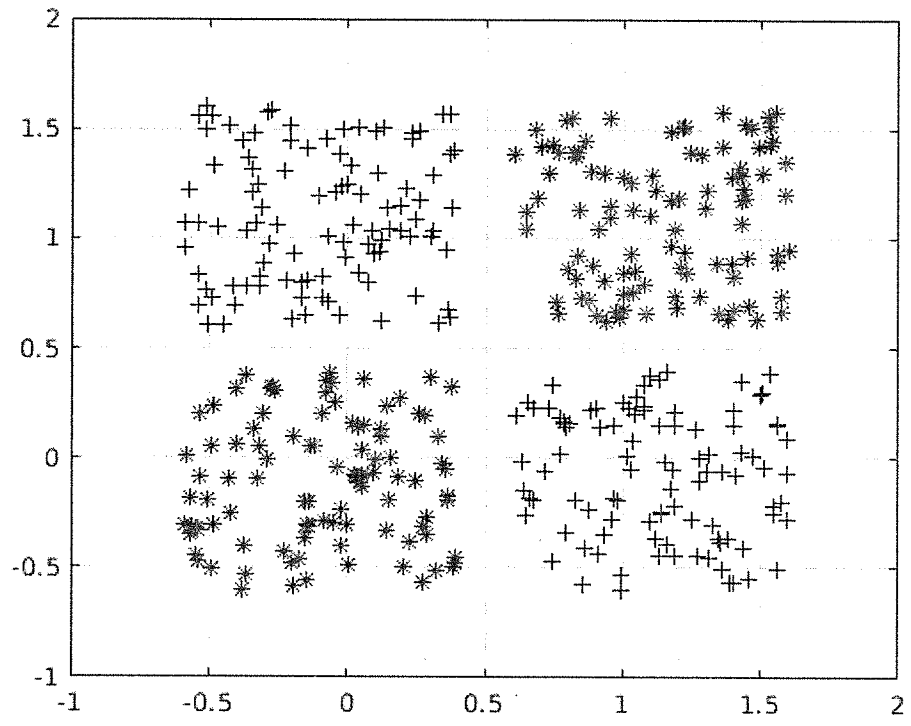
Published with MATLAB® R2018a

Example 2: Train a RBF with newrb, plot the results

Table of Contents

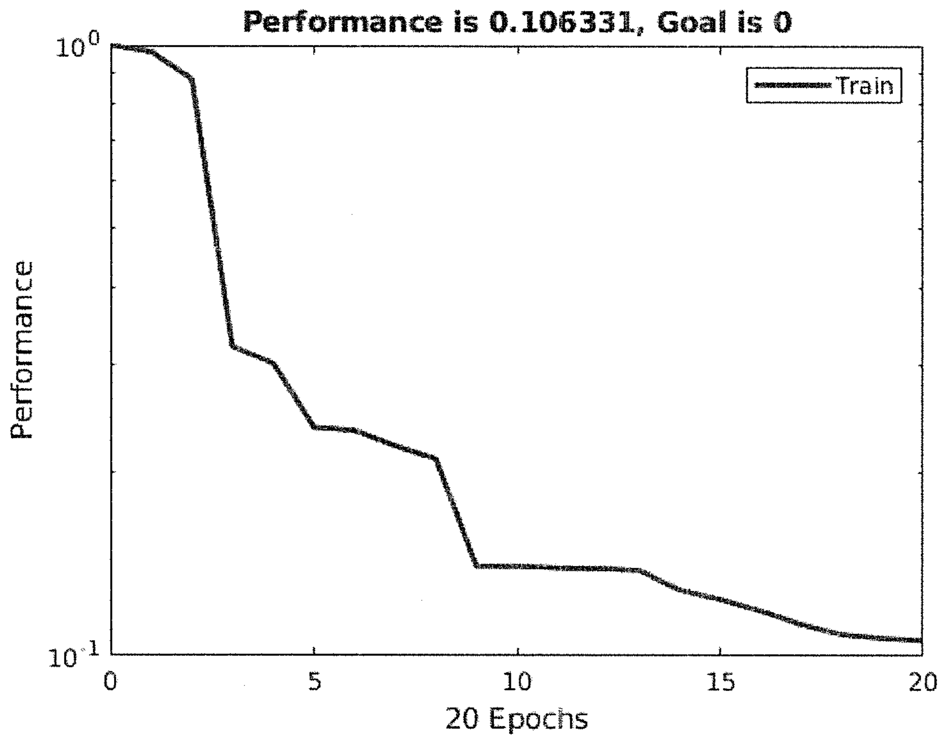
Step 1: Create the data, plot it:

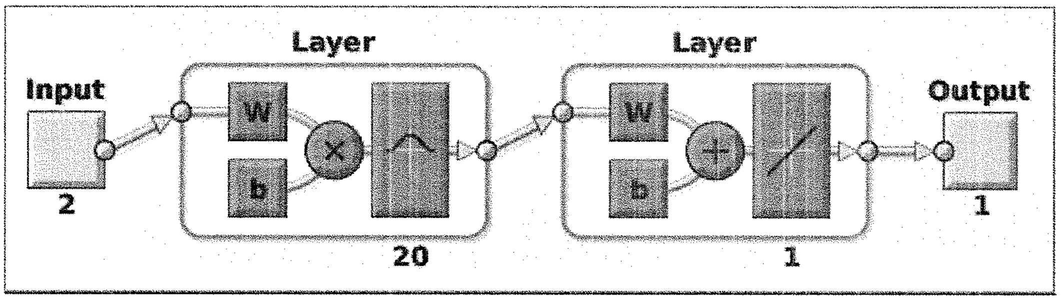
```
close all, clear all, clc, format compact

% Num samples in each cluster
K=100;
% Cluster offset
q=0.6;
% Define two groups- A and B
A= [rand(1,K)-q rand(1,K)+q;
    rand(1,K)+q rand(1,K)-q];
B = [rand(1,K)+q rand(1,K)-q;
rand(1,K)+q rand(1,K)-q];
% plot data
plot(A(1,:),A(2,:),'k+',B(1,:),B(2,:),'b*')
grid on
hold on

% Targets will be +/-1
a=-1; b=1;
P=[A B];
T=[repmat(a,1,length(A)) repmat(b,1,length(B))];
```



Step 2: Construct and train an RBF:

```
% Choose a spread constant:
spread = 2;
% choose max number of neurons
K = 20;
% performance goal (SSE)
goal= 0;
% number of neurons to add between displays
Ki= 4;
% create a neural network
net= newrb(P,T,goal,spread,K,Ki);
% view network
view(net)

NEWRB, neurons = 0, MSE = 1
NEWRB, neurons = 4, MSE = 0.301203
NEWRB, neurons = 8, MSE = 0.210252
NEWRB, neurons = 12, MSE = 0.139079
NEWRB, neurons = 16, MSE = 0.118544
NEWRB, neurons = 20, MSE = 0.106331
```



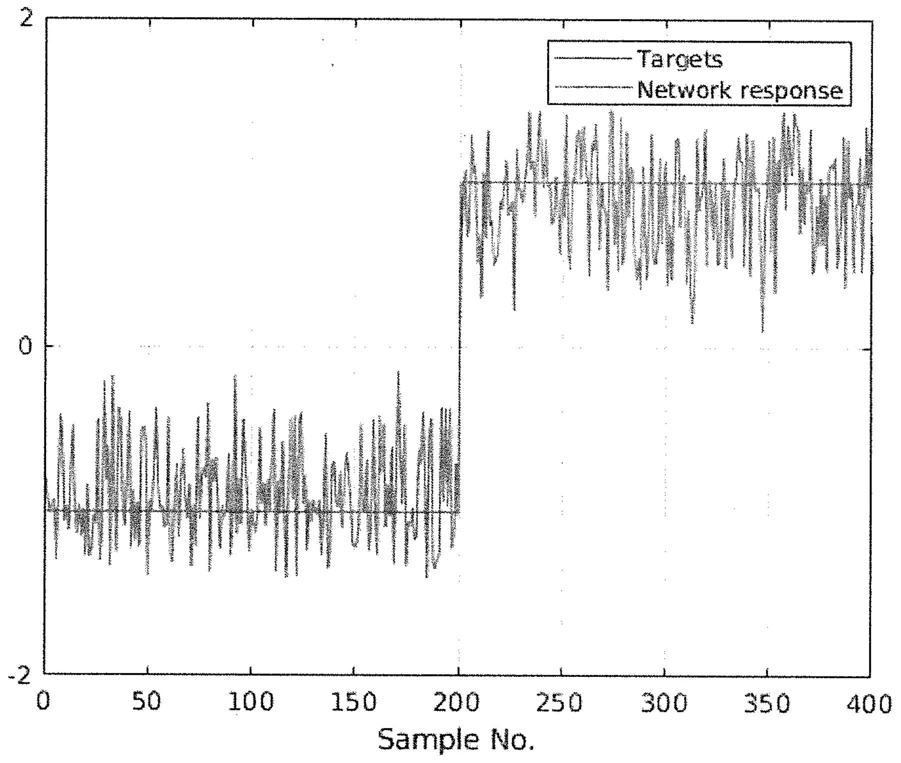


Step 3: Visualize results

```
Y = net(P);
% calculate [%] of correct classifications
correct = 100 * length(find(T.*Y > 0)) / length(T);
fprintf('\n Spread = %.2f\n',spread)
fprintf('Num of neurons = %d\n',net.layers{1}.size)
fprintf('Correct class = %.2f %%\n',correct)
% plot targets and network response
figure;
  plot(T')
  hold on; grid on;
  plot(Y','r')
    ylim([-2 2])
    set(gca,'ytick',[-2 0 2])
    legend('Targets','Network response')
    xlabel('Sample No.')
```

```
hold off
```

```
 Spread = 2.00
Num of neurons = 20
Correct class = 100.00 %
```

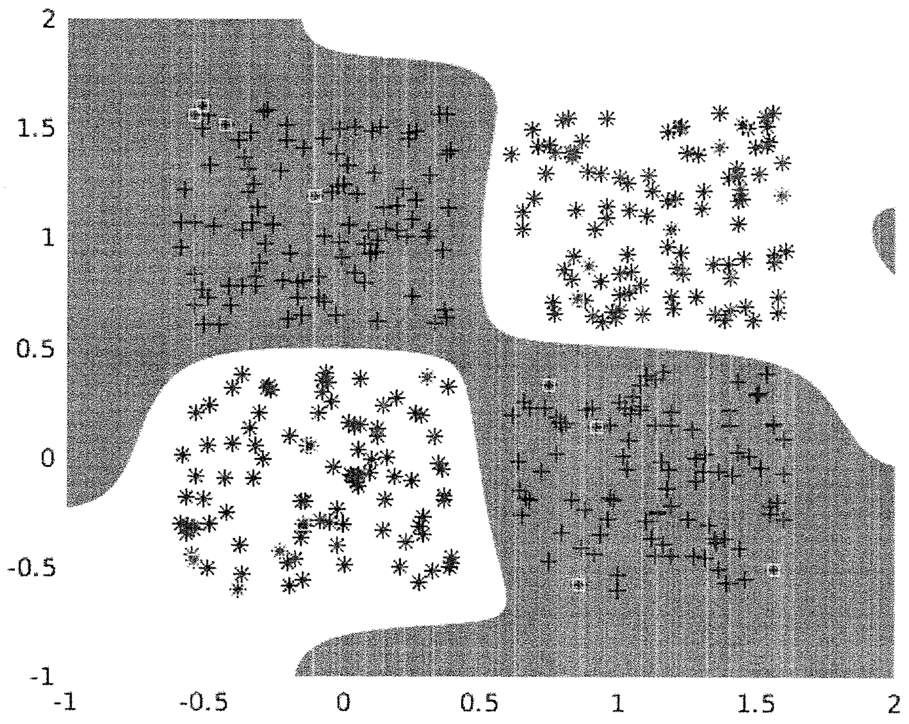


Visualization, continued: Classification regions in the plane.

generate a grid

```
span = -1:.025:2;
[P1,P2] = meshgrid(span,span);
pp = [P1(:) P2(:)]';

% simualte neural network on a grid
aa = sim(net,pp);

% plot classification regions based on MAX activation
figure(1)
ma = mesh(P1,P2,reshape(-aa,length(span),length(span))-5);
mb = mesh(P1,P2,reshape( aa,length(span),length(span))-5);
set(ma,'facecolor',[1 0.2 .7],'linestyle','none');
set(mb,'facecolor',[1 1.0 .5],'linestyle','none');
```

```
hold on
plot(net.iw{1}(:,1),net.iw{1}(:,2),'gs')
hold off
view(2)
```



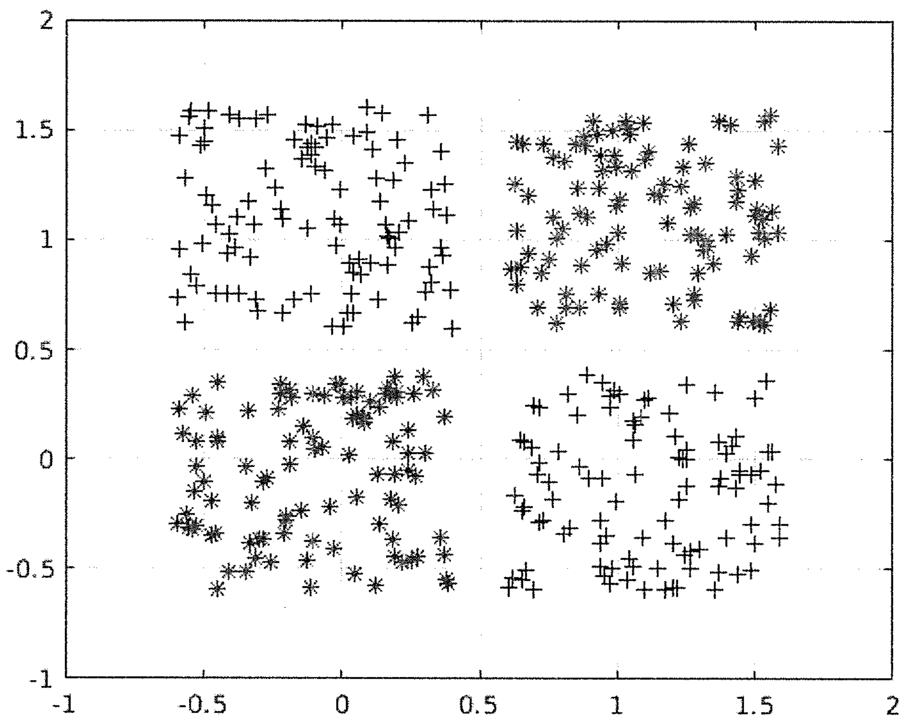
Published with MATLAB® R2018a

Example 3: Train a RBF with newrb, force too small radius

Table of Contents

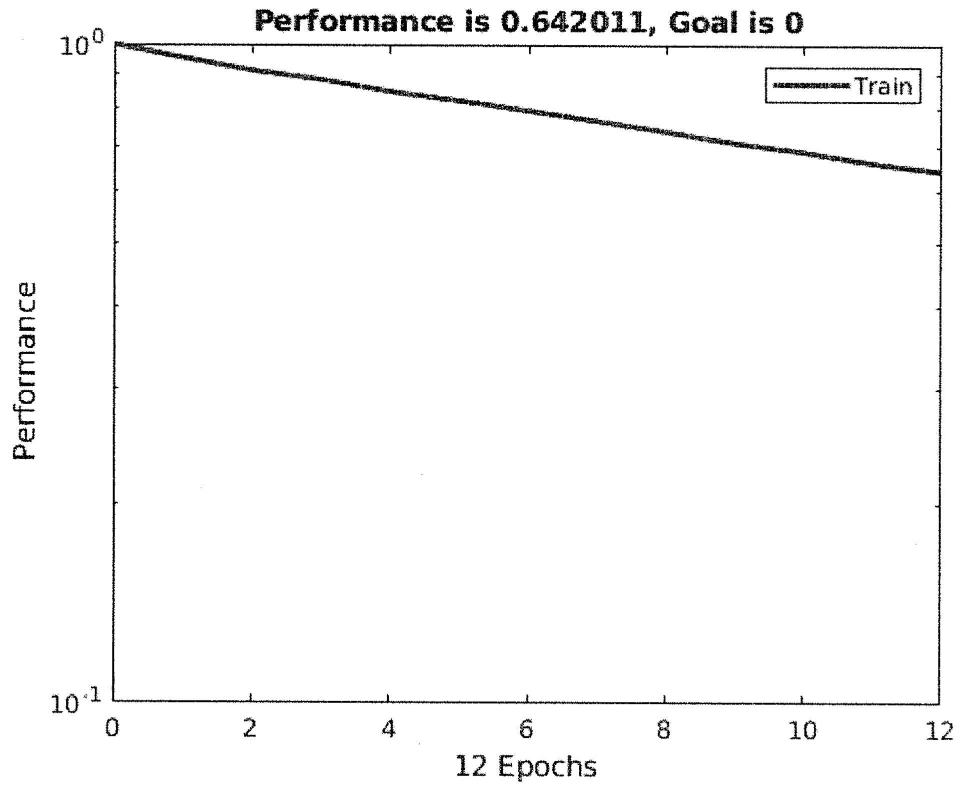
Step 1: Create the data, plot it:

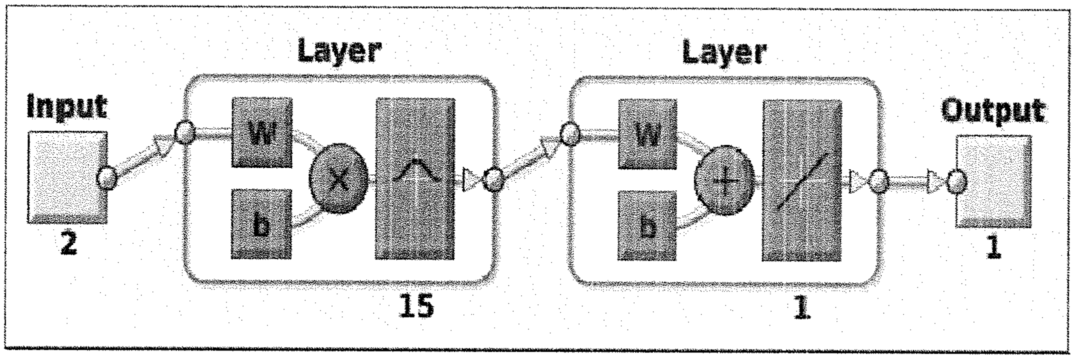
```
close all, clear all, clc, format compact

% Num samples in each cluster
K=100;
% Cluster offset
q=0.6;
% Define two groups- A and B
A= [rand(1,K)-q rand(1,K)+q;
    rand(1,K)+q rand(1,K)-q];
B = [rand(1,K)+q rand(1,K)-q;
rand(1,K)+q rand(1,K)-q];
% plot data
plot(A(1,:),A(2,:),'k+',B(1,:),B(2,:),'b*')
grid on
hold on

% Targets will be +/-1
a=-1; b=1;
P=[A B];
T=[repmat(a,1,length(A)) repmat(b,1,length(B))];
```



Step 2: Construct and train an RBF:

```
% Choose a spread constant:
spread = 0.1;
% choose max number of neurons
K = 15;  % Bring this down for the spread example
% performance goal (SSE)
goal= 0;
% number of neurons to add between displays
Ki= 4;
% create a neural network
net= newrb(P,T,goal,spread,K,Ki);
% view network
view(net)

NEWRB, neurons = 0, MSE = 1
NEWRB, neurons = 4, MSE = 0.847426
NEWRB, neurons = 8, MSE = 0.736946
NEWRB, neurons = 12, MSE = 0.642011
```

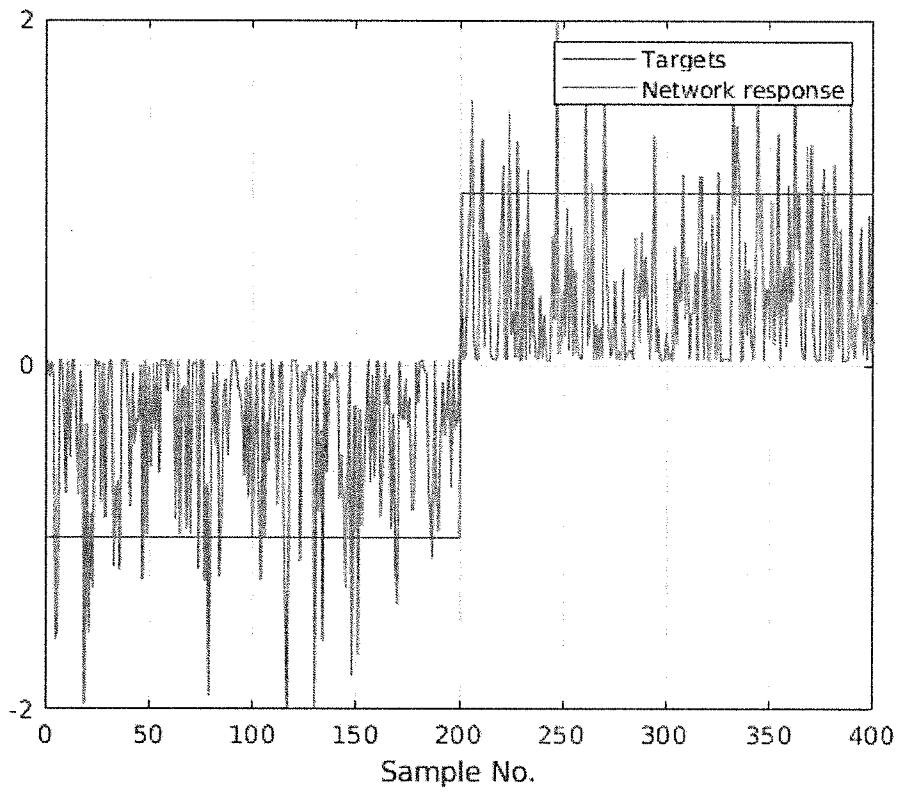




Step 3: Visualize results

```
Y = net(P);
% calculate [%] of correct classifications
correct = 100 * length(find(T.*Y > 0)) / length(T);
fprintf('\n Spread = %.2f\n',spread)
fprintf('Num of neurons = %d\n',net.layers{1}.size)
fprintf('Correct class = %.2f %%\n',correct)
% plot targets and network response
figure;
  plot(T')
  hold on; grid on;
  plot(Y','r')
    ylim([-2 2])
    set(gca,'ytick',[-2 0 2])
    legend('Targets','Network response')
    xlabel('Sample No.')
hold off
```

```
 Spread = 0.10
Num of neurons = 15
Correct class = 85.50 %
```

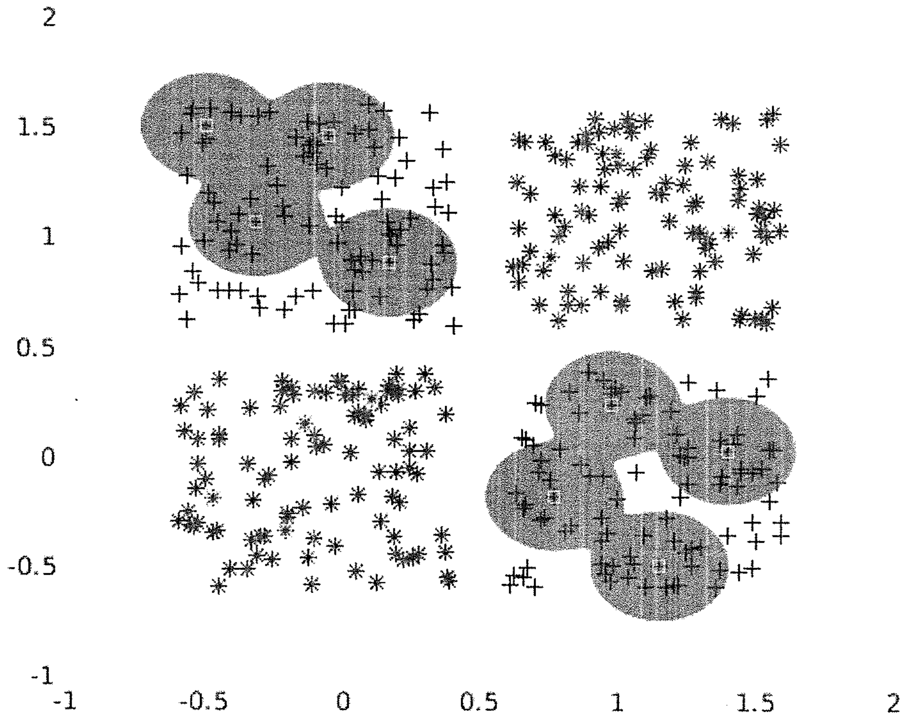


Visualization, continued: Classification regions in the plane.

generate a grid

```
span = -1:.025:2;
[P1,P2] = meshgrid(span,span);
pp = [P1(:) P2(:)]';

% simualte neural network on a grid
aa = sim(net,pp);

% plot classification regions based on MAX activation
figure(1)
```

```
ma = mesh(P1,P2,reshape(-aa,length(span),length(span))-5);
mb = mesh(P1,P2,reshape( aa,length(span),length(span))-5);
set(ma,'facecolor',[1 0.2 .7],'linestyle','none');
set(mb,'facecolor',[1 1.0 .5],'linestyle','none');
hold on
plot(net.iw{1}(:,1),net.iw{1}(:,2),'gs')
hold off
view(2)
```



Published with MATLAB® R2018a

Neural Nets

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.

"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."

Coming to present day, research is aimed at something called **deep neural nets** that perform **automatic feature extraction** and we'll discuss those once we've looked at the typical neural net.

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.

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,

$$F(\mathbf{x}^{(i)}) = \mathbf{t}^{(i)} \qquad \text{for } i = 1, 2, \ldots, p$$

However, we will typically allow for error, and we typically model the error ϵ_i using a normal distribution. Let $\mathbf{y}^{(i)}$ denote the output of the model so that now

$$\mathbf{y}^{(i)} = F(\mathbf{x}^{(i)}) \qquad \text{and} \qquad \mathbf{t}^{(i)} = \mathbf{y}^{(i)} + \vec{\epsilon_i}$$

If we assume that $\mathbf{y}^{(i)}$ 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:

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

so that E is a function of the parameters in F, and we would go about determining the values of the parameters that minimize the error, and that will include differentiating E.

Looking at the Derivative

If we focus on only one term of the sum, then:

$$\|\mathbf{t}-\mathbf{y}\|^2 = (t_1 - y_1)^2 + (t_2 - y_2)^2 + \cdots + (t_m - y_m)^2$$

Therefore, if $\mathbf{y}$ depends on some parameter α, we can differentiate both sides by α to get:

$$\frac{\partial}{\partial \alpha}(\|\mathbf{t}-\mathbf{y}\|^2) = -2(t_1 - y_1)\frac{\partial y_1}{\partial \alpha} - 2(t_2 - y_2)\frac{\partial y_2}{\partial \alpha} - \cdots - 2(t_m - y_m)\frac{\partial y_m}{\partial \alpha}$$

Another way to write this may be:

$$\frac{\partial}{\partial \alpha}(\|\mathbf{t}-\mathbf{y}\|^2) = -2(\mathbf{t}-\mathbf{y}) \cdot \frac{\partial \mathbf{y}}{\partial \alpha}$$

Notice we have a new definition there about differentiating a vector. Now we'll be more specific. Earlier in this class we looked at a linear neural net, $\mathbf{y}^{(i)} = W\mathbf{x}^{(i)} + \mathbf{b}$, so let's look at this derivative in this particular case. It is convenient to write this in terms of the rows of W using Matlab notation:

$$\mathbf{y} = W\mathbf{x} + \mathbf{b} = \begin{bmatrix} W(1,:)\mathbf{x} + b_1 \\ W(2,:)\mathbf{x} + b_2 \\ \vdots \\ W(m,:)\mathbf{x} + b_m \end{bmatrix} \quad \Rightarrow \quad \frac{\partial \mathbf{y}}{\partial W_{ij}} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ x_j \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

where x_j is in the i^{th} coordinate position. Altogether, for the linear network, we see that

$$\frac{\partial}{\partial W_{ij}}(\|\mathbf{t}-\mathbf{y}\|^2) = -2(t_i - y_i)x_j$$

which is a scalar multiple of our Widrow-Hoff rule.

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* (denoted by $\mathbf{P}_1$) 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* (denoted by $\mathbf{S}_1$) of the layer of neurons. Adding that to our diagram, we have:

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

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, $\mathbf{b}_2 \in \mathbb{R}^m$. Starting from the input layer, here are the operations performed by our network:

$$\begin{array}{rcll}
\mathbf{P}_0 = \mathbf{x} & \to & \mathbf{S}_0 = \mathbf{P}_0 & \text{Input Layer} \\
 & \downarrow & & \\
\mathbf{P}_1 = W_1\mathbf{S}_0 + \mathbf{b}_1 & \to & \mathbf{S}_1 = \sigma(\mathbf{P}_1) & \text{Hidden Layer} \\
 & \downarrow & & \\
\mathbf{P}_2 = W_2\mathbf{S}_1 + \mathbf{b}_2 & \to & \mathbf{S}_2 = \mathbf{P}_2 & \text{Output Layer}
\end{array}$$

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 multiple hidden layers as a default neural network. In that case, the mapping (in stages) would look like:

$$\begin{array}{rll}
\mathbf{P}_0 = \mathbf{x} & \rightarrow \mathbf{S}_0 = \sigma(\mathbf{P}_0) & \text{Input Layer 0} \\
 & \downarrow & \\
\mathbf{P}_1 = W_1\mathbf{S}_0 + \mathbf{b}_1 & \rightarrow \mathbf{S}_1 = \sigma(\mathbf{P}_1) & \text{Layer 1} \\
 & \downarrow & \\
\mathbf{P}_2 = W_2\mathbf{S}_1 + \mathbf{b}_2 & \rightarrow \mathbf{S}_2 = \sigma(\mathbf{P}_2) & \text{Layer 2} \\
 & \downarrow & \\
\mathbf{P}_3 = W_3\mathbf{S}_2 + \mathbf{b}_3 & \rightarrow \mathbf{S}_3 = \sigma(\mathbf{P}_3) & \text{Layer 3} \\
 & \downarrow & \\
 & \vdots &
\end{array}$$

One can imagine that many layers are possible, but 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. In practice, the computations involved are often faster with multiple layers (each with a small number of nodes) than a very large single hidden layer.

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

To define a three layer neural network in the form $n-k-m$, we should first set up the transfer function σ. Although we could define a different σ 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 more, if we use more layers) 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. 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:

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

Backpropagation of Error

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

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

Given a 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 for all the different parameters:

$$y = w_2 S + b_2$$

$$\frac{\partial y}{\partial w_2} = S \qquad \frac{\partial y}{\partial b_2} = 1$$

$$\Delta w_2 = (t-y)S \qquad \Delta b_2 = (t-y)$$

And for the other parameters,

$$y = w_2\sigma(P) + b_2$$

$$\frac{\partial y}{\partial w_1} = w_2\sigma'(P)\cdot x \qquad \frac{\partial y}{\partial b_1} = w_2\sigma'(P)$$

$$\Delta w_1 = (t-y)w_2\sigma'(P)\cdot x \qquad \Delta b_1 = (t-y)w_2\sigma'(P)$$

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=feedforwardnet(10);  %10 nodes in hidden layer
net=train(net,P,T);      %Train the network
y=sim(net,P);            %Get the output of the net
plot(P,T,P,y,'o');       %Plot the data and the net output

tt=linspace(-1,1);       %New domain for the plot
yy=sim(net,tt);          %Get the output from the net
plot(P,T,tt,yy,'k-');    %Plot together...
```

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 a relatively good number- It is easy to get locked into a local minimum!

   Matlab has some methods for assigning weights that tries to give you a good start, but it is always possible to get a bad set.

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).

   Matlab has some data preprocessing built-in when we configure the network. Keep in mind that the default behavior is to perform scaling, so if you don't want scaling, check the documentation.

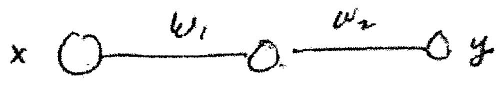
4. Too many nodes in the hidden layer: Use the test set/validation set to be sure you're not memorizing the data (the default settings work pretty well).

Next: Details of Matlab Implementation of Neural Nets

Backpropagation Notes

Before getting into the full theory, we'll first look at two small computational examples below to get you started.

As a very simple starting point, consider a $1-1-1$ neural network as shown below.



The neural network output is then very simple:

$$y = w_2(\sigma(w_1(x)))$$

If the error function is approximated at a single point by $E = (t-y)^2$, then the weights are updated using gradient descent on the error:

$$w_i \leftarrow w_i - \epsilon \frac{\partial E}{\partial w_i}$$

where

$$\frac{\partial E}{\partial w_2} = 2(t-y)\left(-\frac{\partial y}{\partial w_2}\right) = -2(t-y)\sigma(w_1 x)$$

Similarly,

$$\frac{\partial E}{\partial w_1} = 2(t-y)\left(-\frac{\partial y}{\partial w_1}\right) = -2(t-y)w_2\sigma'(w_1 x)(x)$$

We would then plug these into the gradient descent formula, and we're done.

However, in thinking that we're going to want to extend the formulas to an arbitrarily large neural network, let's try the following.

- At the output layer, define $\Delta_2 = (t-y)$.
- At the middle layer, define $\Delta_1 = w_2\sigma'(w_1 x)\Delta_2$
- At the input layer, we don't need to define Δ.

The change in the weight is then the state value to the left times the Delta to the right:

$$\Delta w_1 = (\text{state } 0) \times \Delta_1 = x\Delta_1 = x\, w_2\sigma'(w_1 x)\Delta_2 = x w_2 \sigma'(w_1\, x)(t-y)$$

$$\Delta w_2 = (\text{state } 1) \times \Delta_2 = \sigma(w_1 x)\Delta_2 = \sigma(w_1 x)(t-y)$$

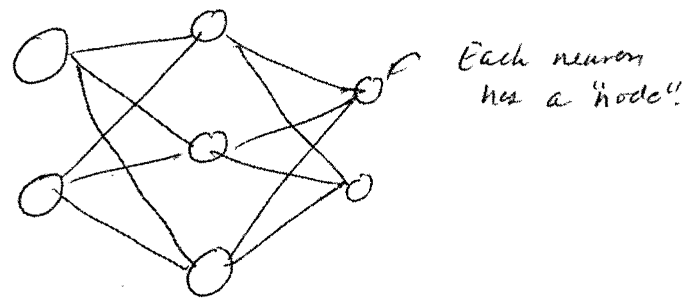
Then the weight updates are (note the change in sign from gradient descent):

$$w_i + \epsilon \Delta w_i$$

Some people will include the "2", but it can simply be incorporated into the ϵ.

A General Network

Think of a network as comprised of computational nodes, as shown below.



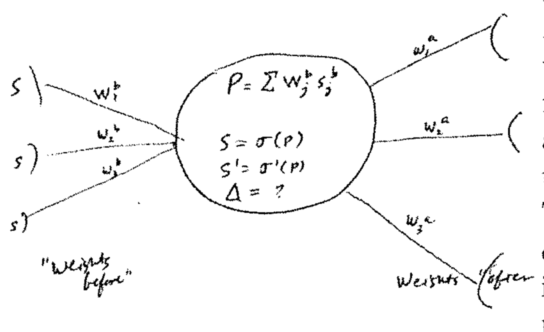
Each node performs computations. When we're programming the network, we'll keep track of 4 numerical values:

- The prestate P. This is the incoming numerical value, and the way it is computed may depend on the layer.
- The state, S. Given a real valued sigmoidal, $\sigma(x)$, then

$$S = \sigma(P)$$

- The derivative, $S' = \sigma'(P)$. With some functions σ this will be very easy to compute from S.
- Value Δ. This value is used to backpropagate the error backwords through the network.

Each node is connect to other nodes by edges, where we think of a **weight** as being attached to an edge.



In the "forward phase", we fill in the values of P, S, S' for each node as indicated. Think of the flow of information as coming from left to right. The state values of the previous nodes are incoming, multiplied by the weights, and those values are collected and summed in P. The result is put through the transfer function σ to get the state value S, and the derivative is computed (for later). The value of S is then passed along to the next layer.

We should note some special cases: In the first layer, the prestate and state values are the same, so the derivative is one (thinking of σ as the identity map). In the last layer, the states coming out may be multiplied by weights, but often are not.

So now, given a set of weights, we have forward propagated our information. We can now compute the values of the output of the network as $\mathbf{y}$.

We will now backpropagate the error from the last layer forward, and as we go, we will update the weights so that, hopefully, the error will decrease.

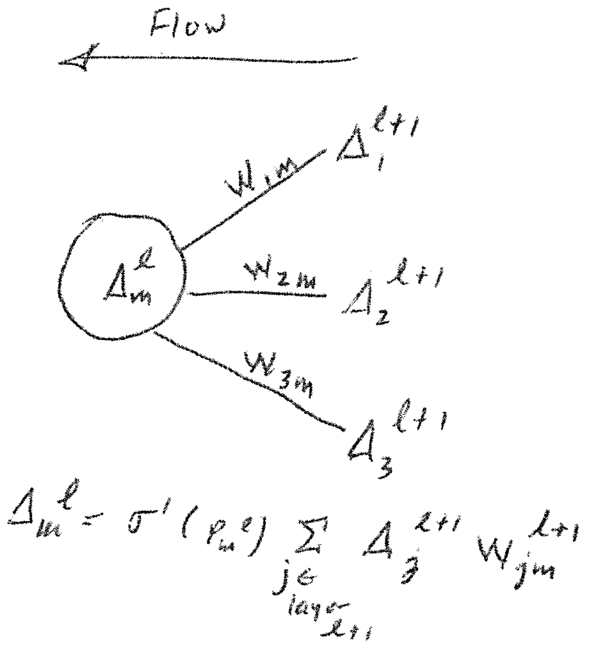
To start the backpropagation, we compute the error at the output nodes. More specifically, for the j^{th} output node (and given target t_j), the error is $t_j - y_j$. We then define the value of Δ for the j^{th} node on the output layer: to be:

$$\Delta_j = (t_j - y_j)S'_j$$

Now for every other layer ℓ, on the m^{th} node, the value of Δ^ℓ_m is given by the following sum, where j is indexing the cells in layer $\ell + 1$.

$$\Delta^\ell_m = \sigma'(P^\ell_m) \sum_j \Delta^{\ell+1}_j W^{\ell+1}_{jm}$$

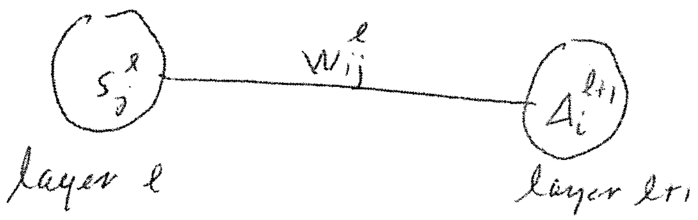
This can be hard to visualize, so below we've provided a diagram for what it is you're computing.


Flow
Δ^ℓ_m
W_{1m}
W_{2m}
W_{3m}
$\Delta_1^{\ell+1}$
$\Delta_2^{\ell+1}$
$\Delta_3^{\ell+1}$
$\Delta^\ell_m = \sigma'(P^\ell_m) \sum_{j \in \text{layer } \ell+1} \Delta_j^{\ell+1} W_{jm}^{\ell+1}$

Now we make the key computation: We use Δ to update the weights. Recall that W^ℓ_{ij} is the edge between cell j in layer ℓ and cell i in layer $\ell + 1$.

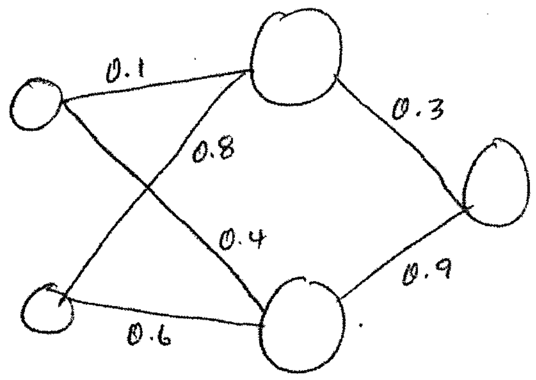
$$W^\ell_{ij} \leftarrow W^\ell_{ij} + \epsilon S^\ell_j \Delta^{\ell+1}_i$$

"The change in the weight is the product of the state on the left and the Δ on the right"...



This process is called the **backpropagation of error**, and in the situation shown above, we backpropagated the error we got from a single data point (online training). It is possible to "batch" that by averaging all values over multiple data points.

In the following, we'll manually build a small neural network so that we can program the backprop of error ourselves and see the results. In this case, we'll have a $2 - 2 - 1$ network as shown below, where we show the initial weights.



We'll train this net on a single training point:

$$(0.35, 0.9) \to 0.5$$

Our code below will step through the process.

- Forward Pass. Compute P, S, S'.
- Backwards Pass.

  Compute Δ for each node.
- Update the weights.

Here's some sample code that implements the equations. The first part is to initialize all the parameters that we'll use.

```
% Initialize:
s=@(x) 1./(1+exp(-x)); %Transfer function
STEP=0.1;              %Stepsize (epsilon in the notes)
NumIts=100;            %Number of iterations.
Out=zeros(1,NumIts);   %Vector to hold the output of the net
Err=zeros(1,NumIts);   %Vector to hold the errors.

x=[0.35;0.9]; t=0.5;
w11=[0.1;0.8];
w12=[0.4;0.6];
w2=[0.3;0.9];
```

Next we'll initialize the main loop. Inside the loop, we'll compute the forward pass, then backpropagate the error.

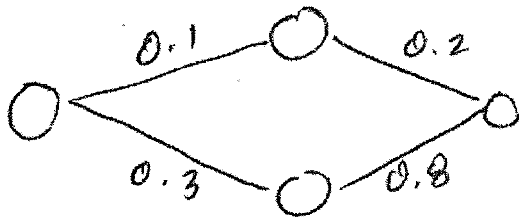
```
for j=1:NumIts

  % Forward pass:

      % Layer 0:
      P01=x(1);
      S01=P01;
      dS01=1;

      P02=x(2);
      S02=P02;
      dS02=1;
```

```
    % Hidden Layer:
    P11=w11'*[S01;S02];
    S11=s(P11);
    dS11=S11*(1-S11);

    P12=w12'*[S01;S02];
    S12=s(P12);
    dS12=S12*(1-S12);

    % Output Layer:
    P2=w2'*[S11;S12];
    S2=P2;
    dS2=1;

    Out(j)=S2;  Err(j)=t-S2;

  % Backprop- Compute the Deltas

    % Last layer:
    Delta2=Err(j)*dS2;

    Delta11=dS11*w2(1)*Delta2;
    Delta12=dS12*w2(2)*Delta2;

    Delta01=dS01*(w11(1)*Delta11+w12(1)*Delta12);
    Delta02=dS02*(w11(2)*Delta11+w12(2)*Delta12);

  % Weight update:
    w11(1)=w11(1)+STEP*Delta11*S01;
    w11(2)=w11(2)+STEP*Delta11*S02;
    w12(1)=w12(1)+STEP*Delta12*S01;
    w12(2)=w12(2)+STEP*Delta12*S02;

    w2(1)=w2(1)+STEP*Delta2*S11;
    w2(2)=w2(2)+STEP*Delta2*S12;

end
```

(Homeowork on next page)

Homework Set 1, Backprop

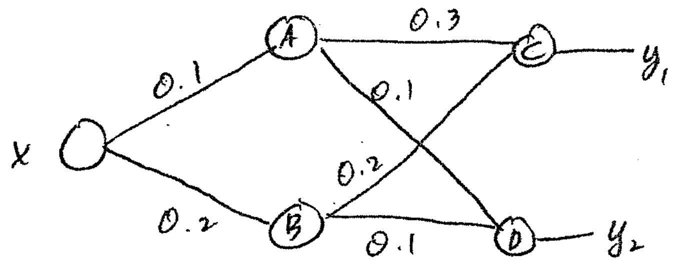
Consider the 1-2-1 network below.



1. If the desired pairing is $x = 1/2$, $t = 2/3$, then forward propagate to get $P, S, and S'$ at each node.
2. Backpropagate to get Δ for each node.
3. Find the change in weight for each edge.
4. Modify the Matlab program to double check your work.

In-Class Example:

Suppose we have a 1-2-2 network as shown below, with the transfer function at the hidden nodes: $\sigma(x) = 1/(1 + e^{-x})$. Recall that $\sigma'(x) = \sigma(x)(1 - \sigma(x))$. The other transfer functions are the identity map, $\sigma(x) = x$.



1. Numerically compute each value, given that $x = 1$:
   (a) Node A:
   - Prestate:
   - State:
   - Derivative, $\sigma'(P)$:

   (b) Node B:
   - Prestate:
   - State:
   - Derivative, $\sigma'(P)$:

   (c) Node C:
   - Prestate:
   - State:
   - Derivative of transfer:

   (d) Node D:
   - Prestate:
   - State:
   - Derivative of transfer:

Backpropagation

Summary so far:

A feed forward neural network is like a directed graph of computational nodes. The one we've been looking at is a three-layered network, but we could have multiple layers. The neural net is the model of a function mapping $\mathbb{R}^n$ to $\mathbb{R}^m$ (so the network would have n nodes in the input layer and m nodes in the output layer). The adjustable parameters of a neural network are: (a) the number of nodes in the hidden layer (or we might have multiple layers), (b) the weights and biases, and (c) the definition of $\sigma(x)$ for each node. Typically, we will choose the identity map for the input and output layers, and $\sigma(x) = 1/(1+\mathrm{e}^{-x})$ for the hidden layer.

Once the parameters are fixed, we can compute the output for the neural network (given input values), and we can compare the network output, denoted by $\mathbf{y}$, to the desired target, $\mathbf{t}$. If they are not equal, then we consider the error function. The full error function on p points is given by:

$$E = \frac{1}{2}\sum_{i=1}^{p} \|\mathbf{t}_i - \mathbf{y}_i\|^2$$

However, to describe the backpropagation of error algorithm, we will assume that we only have a single point, $(\mathbf{x}, \mathbf{t})$, so the error becomes:

$$E \approx \frac{1}{2}\|\mathbf{t}-\mathbf{y}\|^2 = \frac{1}{2}\left((t_1-y_1)^2 + (t_2-y_2)^2 + \cdots + (t_m-y_m)^2\right) \tag{1}$$

Using the current set of parameters (two layers of weights and biases), we want to adjust the parameters to make the error decrease. From calculus, we know that we need to move in the negative direction of the gradient of E (with respect to the weights and biases). If w represents a general weight or bias term, that means:

$$w^{\text{new}} = w^{\text{old}} - \epsilon \frac{\partial E}{\partial w}$$

where ϵ is the step size (the amount we move in the direction of the negative gradient).

Using Equation (1), we see that

$$\frac{\partial E}{\partial w} = \sum_{j=1}^{m} (t_j - y_j)\left(-\frac{\partial y_j}{\partial w}\right)$$

We need an efficient way of computing these values for each weight (and bias) term in the network.

In the handout last time, the main idea was to compute a value we called Δ for each node. Let Δ_m^l denote the value for the mth node in the lth layer. This node has *output* edges we can denote by $W_{\cdot,m}^{l+1}$

- For nodes on the output layer L,

$$\Delta_j^L = (t_j - y_j)\sigma'(P_j^L) \tag{2}$$

Normally, this derivative is 1, but we include it for completeness.

- For the other nodes in layers $l = 1, 2, \cdots, L-1$, the Δ are defined recursively (from the layer with the highest index to the lowest):

$$\Delta_m^l = \sigma'(P_m^l) \sum_{j \in \text{nextlayer}} \Delta_j^{l+1} W_{jm}^{l+1} \tag{3}$$

- Now, if W_{mn}^l connects node n in layer $l-1$ to node m in layer l, then it is updated as follows:

$$W_{mn}^l = W_{mn}^l + \epsilon \Delta_m^l S_n^{l-1} \tag{4}$$

(that is, the State of the node to the left times the Delta of the node to the right).

There are still a few things for us to work out:

1. Is this update rule the same as gradient descent?

2. What about biases?

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

$$\frac{\partial E}{\partial W_{mn}^l} = \frac{\partial E}{\partial S_m^l} \frac{\partial S_m^l}{\partial P_m^l} \frac{\partial P_m^l}{\partial W_{mn}^l} \tag{5}$$

The two values on the right can readily be computed:

$$\frac{\partial S_m^l}{\partial P_m^l} = \sigma'\left(P_m^l\right) \qquad \frac{\partial P_m^l}{\partial W_{mn}^l} = S_n^{l-1} \tag{6}$$

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

$$\frac{\partial E}{\partial S_m^L} = \frac{\partial E}{\partial y_m} = (t_m - y_m)(-1)$$

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

$$\frac{\partial E}{\partial S_m^l} = \sum_{j \in \text{nextlayer}} \frac{\partial E}{\partial S_j^{l+1}} \frac{\partial S_j^{l+1}}{\partial S_m^l} \tag{7}$$

Since $S_j^{l+1} = \sigma(P_j^{l+1}) = \sigma(W_{jm}^{l+1} S_m^l + \text{ other terms})$, the derivative will be

$$\frac{\partial S_j^{l+1}}{\partial S_m^l} = \sigma'(P_m^{l+1}) W_{jm}^{l+1}$$

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 (5):

$$\Delta_m^l = -\frac{\partial E}{\partial S_m^l} \frac{\partial S_m^l}{\partial P_m^l} = -\frac{\partial E}{\partial S_m^l} \sigma'(P_m^l)$$

Therefore, on the output layer,

$$\Delta_m^L = \frac{\partial E}{\partial S_m^L} \frac{\partial S_m^L}{\partial P_m^L} = -(t_m - y_m)(-1)\sigma'(P_m^L)$$

which matches Equation (2). Now, using Equation (7), the nodes on the previous layer are computed:

$$\Delta_m^l = -\frac{\partial E}{\partial S_m^l} \frac{\partial S_m^l}{\partial P_m^l} = \left(\sum_{j \in \text{layer}_{l+1}} -\frac{\partial E}{\partial S_j^{l+1}} \frac{\partial S_j^{l+1}}{\partial S_m^l} \right) \sigma'(P_m^l) =$$

$$\sigma'(P_m^l) \left(\sum_{j \in \text{layer}_{l+1}} -\frac{\partial E}{\partial S_j^{l+1}} \sigma'(P_m^{l+1}) W_{jm}^{l+1} \right) =$$

$$\sigma'(P_m^l) \left(\sum_{j \in \text{ layer}_{l+1}} \Delta_j^{l+1} W_{jm}^{l+1} \right)$$

And this is Equation (3). Finally, by Equation (6), we can write:

$$-\frac{\partial E}{\partial W_{mn}^l} = \left(-\frac{\partial E}{\partial S_m^l} \frac{\partial S_m^l}{\partial P_m^l} \right) \frac{\partial P_m^l}{\partial W_{mn}^l} = \Delta_m^l S_n^{l-1}$$

which proves that the update rule in Equation (4) 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 (4) 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.