Chapter 7

Neural Networks

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. Let us summarize those we have seen up to this point:

- The Linear Neural Net: $\boldsymbol{x} \to A\boldsymbol{x} + \boldsymbol{b}$.
- Kohonen's SOM: The graph is the network topology, and the units undergo competition.
- Neural Gas: Same interpretation as Kohonen's SOM.
- Radial Basis Functions: Each "center" represents a computational unit, and the output of unit i was $\boldsymbol{x} \to \phi(\|\boldsymbol{x} - \boldsymbol{c}^{(i)}\|)$

In the case of the Linear Neural Net and the RBF, network training was cast as solving a least squares problem, so that we could use features of linear algebra to solve for the best parameters.

In this chapter, we look at the classic workhorse of the neural network industry: The three-layer, feed-forward neural network.

As with the RBF, we will see that the three-layer, feed-forward network is capable of performing function approximation arbitrarily well when the domain is a compact subset of $\mathbb{R}^n$.

7.1 Biological Motivation

The neuron is the basic building block of the central nervous system. There are many different types of neurons, but each can be anatomically broken down into three parts: the dendrites, the cell body, and the axon. 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.

For the purposes of our basic mathematical model, we will assume the following processing features that a neuron makes to information (we will refer to Neurons as Nodes in what follows):

7.1.1 As information flows across a synapse, information can be amplified, inhibited, or re-polarized. If we let x denote the signal, then synaptic processing is represented as a multiple of x:

$$x \rightarrow wx$$

where w will be referred to as a *weight*. Since there are multiple incoming signals, we denote by w_{ij} the weight connection from Node j to Node i.

7.1.2 At the cell body, information from across the dendrites is collected and processed in the following way.

(a) The incoming signals are summed. Let $j = 1..m$ denote the index of the signal coming to Node i:

$$\sum_{j=1}^{m} w_{ij}x$$

(b) This quantity is added to the resting state of the cell, which is assumed to be constant:

$$\sum_{j=1}^{m} w_{ij}x + b_i$$

and b_i is called the bias of the Node. The summed quantity is referred to as the Prestate of the Node.

(c) Biologically, we assume an "all or nothing" response to the incoming signal. That is, if the signal strength is above a certain threshold, the neuron fires. If not, then the information is not passed along. Mathematically, we use a step function, with the step at 0. This is not a differentiable function, however. We substitute for the step function a "sigmoidal" function, $\sigma(x)$. This is defined to be a monotonically increasing function so that

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

where A and B are constants. Common choices for the sigmoidal function include:

i. The inverse tangent:

$$\sigma(x) = \mathrm{atan}(x)$$

ii. The exponential sigmoidal function:

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

We will use the latter choice, since the computation of its derivative is especially easy to implement on the computer:

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

Other functions that transfer the incoming signal out to the axon are possible. At times, we will use the following, denoted by $t(x)$ as transfer functions:

i. The Linear Node

$$t(x) = x$$

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

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

iii. The Spherical Node (three inputs, three outputs):

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

See [13, 15] for examples of how to implement the last two transfer function types.

7.1.3 The axon sends along the information to the synapse.

In summary, we have the following information processing for the i^{th} single node:

$$x \to \underbrace{\sum_{j=1}^{m} w_{ij}x + b_i}_{\text{Prestate}} \to \underbrace{\sigma\left(\sum_{j=1}^{m} w_{ij}x + b_i\right)}_{\text{State}}$$

In vector form,

$$x \to (\boldsymbol{w}_i)^{\mathrm{T}}x + b_i \to \sigma\left((\boldsymbol{w}_i)^{\mathrm{T}}x + b_i\right)$$

So that, taking $\boldsymbol{x} \in \mathbb{R}^N$, and having k Nodes,

$$\boldsymbol{x} \to \left((\boldsymbol{W}^{(0)})^{\mathrm{T}}\boldsymbol{x} + \boldsymbol{b}\right) \to \sigma\left((\boldsymbol{W}^{(0)})^{\mathrm{T}}\boldsymbol{x} + \boldsymbol{b}\right)$$

where

$$\boldsymbol{W}^{(0)} = (\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_k) \quad \boldsymbol{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{pmatrix}$$

and σ operates on a vector or matrix component-wise. Also note that, if σ were a linear function, then the neural network would be linear.

7.2 The Three Layer Feedforward Neural Network

The usual method of describing a neural network's architecture is to use a connected graph with computational nodes. The basic feedforward neural network uses three layers of "nodes". The first layer is the input layer, so the number of nodes is equal to the dimension of $\boldsymbol{x}$, and the transfer function is linear.

The hidden layer contains the nonlinear transfer function, and there is a choice as to how many nodes one includes in the hidden layer. We assume there are k nodes in the hidden layer.

The output layer again has a linear transfer function, and there are as many nodes here as there are output variables. Each edge of the graph is weighted, and we refer to edges from the input layer to the hidden layer as the matrix

$$\boldsymbol{W}^{(0)} = \left[W_{ij}^{(0)}\right]_{i=1:n(1),\, j=1:n(0)}$$

where $n(i)$ is the number of nodes in Layer i, and

$$\boldsymbol{W}^{(1)} = \left[W_{ij}^{(0)}\right]_{i=1:n(2),\, j=1:n(1)}$$

so that W_{ij}^{k} refers to the weight in the k^{th} layer, going from Node j in the k^{th} layer to Node i in the $k+1^{\text{st}}$ layer.

There are also vectors of biases associated with the hidden layer, and output layer, respectively:

$$\boldsymbol{b}^{(0)} \text{ and } \boldsymbol{b}^{(1)}$$

For the purposes of the backpropagation algorithm given later, note that the biases can also be represented as one additional node in the input and hidden layers, with constant transfer function, $\sigma(x) = 1$. We will adopt this notation, and suppress the bias specification in Section 4.2.1. Figure 7.1 shows the graphic layout of the three layer, feedforward neural network.

The purpose of this network is to provide an approximation to an arbitrary continuous function, $\boldsymbol{F}$, defined on a compact domain X, and we will use the notation $\mathbf{y} = \boldsymbol{F}(\mathbf{x})$, where $\mathbf{x} \in \mathbb{R}^N$ and $\mathbf{y} \in \mathbb{R}^M$. The network will render the approximation in terms of a sigmoidal basis so that the approximation to $\boldsymbol{F}$ is given pointwise by:

$$\boldsymbol{y}^{(i)} = \boldsymbol{F}(\boldsymbol{x}^{(i)}) \approx \left(\mathbf{W}^{(1)}\sigma\left(\mathbf{W}^{(0)} \cdot \mathbf{x}^{(\mathbf{i})} + \mathbf{b}^{(0)}\right) + \mathbf{b}^{(1)}\right) \tag{7.1}$$

In a paper of fundamental importance, Cybenko [6] shows that sums of the form in equation (7.1) are dense in the space of continuous functions defined on the unit cube. In other words, a neural network is capable of approximating any continuous function arbitrarily well by increasing the number of nodes in the hidden layer. As the number of nodes in the hidden layer increases, the error between the network approximation and the continuous function goes to zero. This is very significant for several reasons:

- It gives us a theoretical framework from which to work.

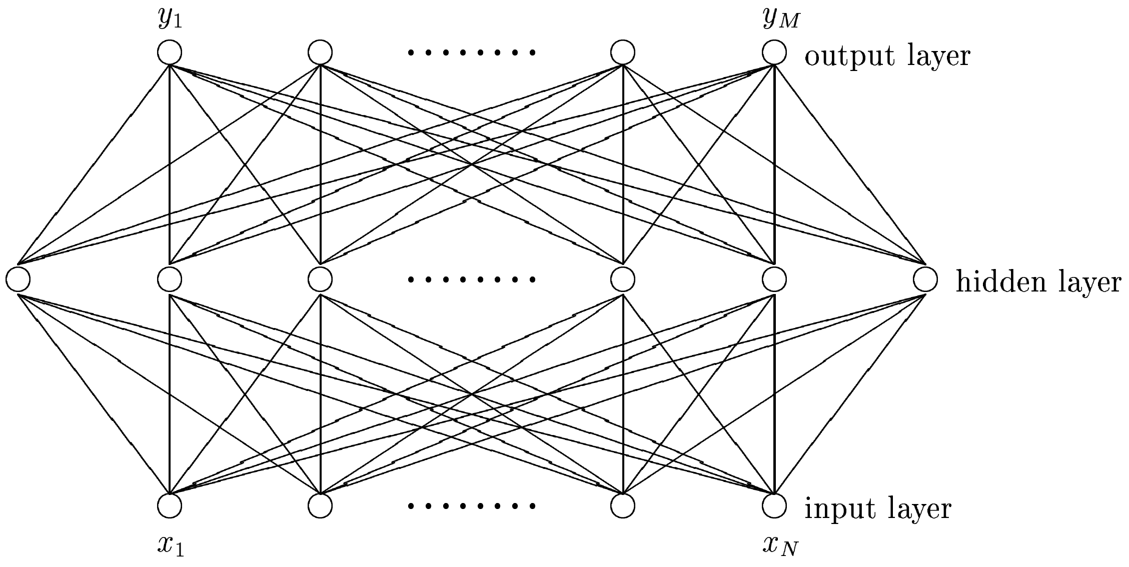

Figure 7.1: The neural network architecture.

- The proof of this theorem ties neural networks to the classical approaches to function approximation in mathematics.
- It solves one of Hilbert's problems from the turn of the last century. Kolmogorov had given the first proof in the 1950's, but it is still not clear that the functions he used can be implemented in a computer algorithm.
- The proof revitalized neural network research. Prior to this, research had been stagnant after the discovery that the first neural networks (in the 1960's) were incapable of modeling certain standard functions (the XOR function). We know now that this was because the first networks only worked on linearly seperable data.
- Although we have taken outrageous liberties (biologically speaking) in our model, the model shows that a very simple structure is capable of very complex behavior.

In fact, there are more general results that can be found in [11, 12, 9, 10, 7, 8] and more specifically, they show that a neural approximation not only approximates the function, but can also simulatenously approximate the first m derivatives using the same functional parameters.

7.3 Training and Error

Theoretically, we have now seen the neural network as a "Universal Function Approximator". The real question now is, given an input and output set, how does one build the neural net? This question can be reframed as: Find the weights w^i_{jk} and biases b^i_j that perform a desired input-output. This process is also known as network training.

For a fixed value of the parameters in the matrices $\boldsymbol{W}^{(i)}$ and biases $\boldsymbol{b}^{(i)}$, the network outputs a smooth function in $\boldsymbol{x}$. As mentioned previously, we adopt the notation of including the biases in the parameters $\boldsymbol{W}^{(i)}$, and add an extra node on the hidden and input layers.

Define the mean square error of the approximation, given a set of parameters $\{\boldsymbol{W}^{(0)}, \boldsymbol{W}^{(1)}\}$, as:

$$E(\boldsymbol{W}^{(0)}, \boldsymbol{W}^{(1)}) = \frac{1}{2|X|} \sum_{i=1}^{|X|} \|F(\mathbf{x}^{(\mathbf{i})}) - \left(\mathbf{W}^{(1)}\, \sigma\left(\mathbf{W}^{(0)} \cdot \mathbf{x}^{(\mathbf{i})}\right)\right)\|_2^2 \tag{7.2}$$

where $|X|$ denotes the number of points in the data set X. Then E is a smooth function of the weights and biases, so that it can be minimized. From the method of steepest descent, we update the weights:

$$W^{(i)}_{jk}(t+1) = W^{(i)}_{jk}(t) + \alpha(t)d(t) \tag{7.3}$$

until the error has become smaller than a prescribed value. For the standard gradient descent, for example, we have

$$d(t) = \frac{\partial E}{\partial W^{(i)}_{jk}}(t)$$

How we find the value of $\alpha(t)$ and the definition of $d(t)$ is where the divergence in training algorithms occurs. There are many algorithms built especially for this purpose, such as Line Minimization. Other update methods can be applied, since training is a general, unconstrained optimization problem:

$$\min_{\boldsymbol{W}^{(0),(1)}} \quad E$$

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. This will be discussed in the next section. First, we introduce some notation that will make the algorithm more clear.

Define the prestate for layer i, Node j at time t:

$$P_j^i = \sum_{k=1}^{n(i-1)} W_{jk}^{i-1} S_k^{i-1}, \text{ for } i = 1 \dots L$$

where S_j^i is the State of Node j on layer i, which is

$$S_j^i = \begin{cases} \sigma\left(P_j^i\right) & i > 0 \\ x & i = 0 \end{cases}$$

7.3.1 Backpropagation of Error

We now detail a recursive algorithm, known as the backpropagation of error [1], which gives an efficient method for computing the partial derivatives for training (see Equation (7.3)).

The goal is to compute

$$\frac{\partial E}{\partial W_{jk}^{i-1}}$$

for each i, j, k.

From the chain rule, we can write

$$\frac{\partial E}{\partial W_{jk}^{i-1}} = \frac{\partial E}{\partial P_j^i} \cdot \frac{\partial P_j^i}{\partial W_{jk}^{i-1}}$$

which we can re-express, using:

$$\frac{\partial P_j^i}{\partial W_{jk}^{i-1}} = S_k^{i-1} \quad \frac{\partial P_j^{i+1}}{\partial P_k^i} = W_{jk}^i \sigma'(P_k^i)$$

and defining:

$$\delta_j^i = \frac{\partial E}{\partial P_j^i}$$

Then we have:

$$\frac{\partial E}{\partial W_{jk}^{i-1}} = \delta_j^i \cdot S_k^{i-1}$$

So, we need an algorithm to compute δ_j^i.

For the output layer (L), with aribtrary transfer function σ,

$$\delta_j^L = \frac{\partial E}{\partial P_j^L} = \frac{\partial E}{\partial S_j^L}\frac{\partial S_j^L}{\partial P_j^L} = \sigma'(P_j^L)\frac{\partial E}{\partial S_j^L}$$

where $\frac{\partial E}{\partial S_j^L}$ is computed from the error by substituting S_j^L into Equation (7.2):

$$E = \frac{1}{|X|}\sum_{i=1}^{2|X|} \|\boldsymbol{y}^{(i)} - \begin{pmatrix} S_1^{(L)} \\ \vdots \\ S_m^{(L)} \end{pmatrix} \|_2^2$$

which implies that:

$$\frac{\partial E}{\partial S_j^L} = \frac{-1}{|X|}\sum_{i=1}^{|X|} |y_j^{(i)} - S_j^L| \tag{7.4}$$

For the hidden nodes,

$$\delta_j^i = \frac{\partial E}{\partial P_j^i} = \sum_{k=1}^{n(i+1)} \frac{\partial E}{\partial P_k^{i+1}}\frac{\partial P_k^{i+1}}{\partial P_j^i}$$

Noting $\frac{\partial P_k^{i+1}}{\partial P_j^i} = W_{kj}^i \sigma'(P_j^i)$, we obtain:

$$\delta_j^i = \sigma'(P_j^i)\sum_{k=1}^{n(i+1)} \frac{\partial E}{\partial P_k^{i+1}} W_{kj}^i$$

Which, from our definition of δ_j^i, is recursively defined as:

$$\delta_j^i = \sigma'(P_j^i)\sum_{k=0}^{n(i+1)} \delta_k^{i+1} W_{kj}^i \tag{7.5}$$

This can be interpreted as a backpropagating the δ_j^i backwards through the network. The algorithm is summarized as:

Algorithm 7.3.1 *The Backpropagation Algorithm*

7.3.1 Forward propagate all the x's, keeping track of the error.

7.3.2 Compute δ_j^L, for $j = 0 \ldots M-1$ using:

$$\delta_j^L = \sigma'(P_j^L)\frac{\partial E}{\partial S_j^L}$$

7.3.3 Compute all other δ's:

$$\delta_j^i = \sigma'(P_j^i)\sum_{k=0}^{n(i+1)} \delta_k^{i+1} W_{kj}^i$$

7.3.4 Compute:

$$\frac{\partial E}{\partial W_{jk}^{i-1}} = \delta_j^i \cdot S_k^{i-1}$$

for $i = 1 \ldots L$, $j = 1 \ldots n(i+1)$, and $k = 1 \ldots n(i)$.

7.3.2 Nonlinear Optimization Techniques

There are many methods to choose from, and some are included below:

- Steepest Descent (with variations on how to compute α)
- Newton's Method (This is indirect - we solve for where $f'(x) = 0$).
- Conjugate Gradient (Search along the eigenvectors of the Hessian).
- Levenburg-Marquardt (A combination of the methods above).

To really study all of these techniques would take us too far afield in this course. Any text on nonlinear optimization (as well as "Numerical Recipes for C") will cover the basic ideas.

The introduction of a nonlinear optimization technique is the primary reason we will use Matlab to perform our neural network training. Programming these techniques can be a delicate task! For our purposes, Levenburg-Marquardt is more than adequate.

7.4 Neural Networks and Matlab

Matlab has set up a very general data structure by which to construct and train almost any type of neural network. Here we focus on the construction and operation of the three-layer feed-forward network.

Because the data structure is so general, building and training a neural network is broken into multiple stages.

- Identify the network architecture. This includes constructing neural layers, connections, identifying the transfer functions for each node. Furthermore, one must define the error measure (or performance index), the training function, and how one wishes to compute a Jacobian matrix. This step would be extremely tedious if we had to do it every time! Matlab has several built in architectures, and we've dealt with a few. For example:
  - `newsom` initializes Kohonen's SOM.
  - `newlin` initializes a linear neural network.
  - `newrb` initializes a radial basis function network.
  - `newrbe` initializes (and trains) an exact RBF.

  The new function here is `newff`.
- Train the network. This is performed (for all network types) by the command: `net=train(net,X)`. The specific functions and training parameters have already been set up by the initialization process.
- Simulate the network with new data. Again, this is performed generically by: `Y=sim(net,X)`. The specific functions and parameters have been set in the network data structure.

Before continuing, lets look at an example:

Sample Training Session 1:

```
P = [0 1 2 3 4 5 6 7 8 9 10];
T = [0 1 2 3 4 3 2 1 2 3 4];
net = newff([0 10],[5 1],{'tansig' 'purelin'});
Y = sim(net,P);
```

```
5  plot(P,T,P,Y,'o')
6  net.trainParam.epochs = 50;
7  net = train(net,P,T);
8  X = linspace(0,10);
8  Y = sim(net,X);
9  plot(P,T,'ko',X,Y)
```

Training Session 1 explanation:

- Lines 1 and 2 set up the training patterns and targets. Notice that the domain runs between a minimum of 0 and a maximum of 10, which are the numbers appearing in the first argument of `newff`. Notice also that the sizes of the domain and range sets are given as $m \times n$, where m =dimension and n =number of data points.

- Line 3 initializes a feed forward neural network. The arguments shown are required, and later we will see what other optional arguments are allowed. For now, notice that the first argument gives the minimum and maximum values in the domain set. Normally, one uses `minmax(P)` in place of actually typing these values in. This also implicitly defines the number of nodes in the input layer (which is the domain dimension).

  The second argument in Line 3 defines the number of nodes in the hidden layer and in the output layer. This vector also implicity defines how many layers you have by counting the size of this vector. Thus, we can have as many layers as we wish.[1] Later, we will define a network with multiple hidden layers.

  The last required argument in Line 3 is the type of transfer function that will be used. The `tansig` function is the inverse tangent function, which gives the same performance as our standard sigmoidal function. We will always use a linear layer in the output, so this argument is always `purelin`.

- Lines 4 and 5 show you what the neural network output looks like before network training (the weights were initialized using `initwb`).

[1]Theoretically, we require only one hidden layer. However, many researchers find it easier to train a network with two small hidden layers versus one large hidden layer. This is mainly a matter of taste.

- Line 6 sets up how many training iterations (or epochs) will be performed. This means that the network will run through all of the data 50 times.
- Line 7 trains the network using Levenburg-Marquardt (which is the default).
- Lines 8 and 9 simulate the network and plots the result. Note that we plot more than just the trained points to see if the network generalizes well.

Exercise:

7.4.1 Run the sample exercise as written and plot the output.

7.4.2 Increase the number of epochs to 1000 and retrain and replot.

7.4.3 Create a new network with 10 nodes in the hidden layer, train using 600 epochs. Plot the result.

7.4.4 Compare the plots and discuss the results.

The NEWFF command

The full `newff` command looks like:

```
net = newff(M,[S1 S2...SN],{T1 T2...TN},F1,F2,F3)
```

Description of the input parameters:

- M is a vector of minimum and maximum values of the data. The size of M will be $n \times 2$, where n is the dimension of the input data (and so will define the number of nodes in the input layer). We will almost always use the function `minmax(P)` for this argument.
- The next vector lists the number of nodes in the next N layers (not the input layer). Thus, for a three layer network, we will always use $[a \;\; b]$ where a is the number of nodes in the hidden layer, and b is the output dimension.

- The next list defines the transfer function for each layer. We will almost always use `{'tansig','purelin'}`, although other options are available. In fact, below we will write our own transfer function.
- The next three functions define (in order):
  - `F1` is the nonlinear optimization function to use. The default is Levenburg-Marquardt (`trainlm`). Others are possible. Type `help newff` to read the WARNING message about memory requirements and different possible options.
  - `F2` defines how the Jacobian matrix will be computed. The default function is `learngdm` which computes the Jacobian via backpropagation.
  - `F3` is the performance function, whose default uses the mean squared error (`'mse'`), which we will use.

7.4.1 Training Notes

In this section, we look at how to train a neural network, and what to do if the training is not progressing well. What constitutes a good network? How do we choose the number of nodes in the hidden layer?

Parameters and Preprocessing

What may be striking in the `newff` command is the absence of any training parameters. Recall that in the section on Radial Basis functions, there was a parameter that controlled the width of the Gaussians. However, also recall that changing the widths of the Gaussians had the same effect as rescaling the data. That is what we may have to do when using a feed forward network. Let's see an example.

In Figure 7.2, we see a standard sigmoidal function. For small values of x, the corresponding images, $\sigma(x)$ are quite different. We say that σ is able to differentiate between these data points. On the other hand, suppose that most of your data is out at the "tails" of the sigmoidal. Then the corresponding images begin to pile up at the horizontal asymptotes. That is, for example, that $\sigma(19)$ becomes indistinguishable from $\sigma(50)$. For example, if $\sigma(x) = \frac{1}{1+e^{-x}}$, then a calculator might say that $\sigma(19) = 0.9999999$, and

$\sigma(50) = 1$. Therefore, if all of the data is between 19 and 50, the output will almost always be 1.

This effect is called **saturation-** we say that the sigmoidal function has been saturated if the data has not been scaled appropriately.

This is an effect to keep in mind if the neural network is not training very well. Rather than having a lot of training parameters in the function call, we will assume that the data has been scaled "appropriately". In other words, we will always look at preprocessing the data before giving it to the neural network to train. Let's look at some options in preprocessing that Matlab gives us. Each method has three associated commands. One command is to calculate the processing parameters from the given data and to perform that method on the data (the method begins with `pre`), another command is for postprocessing the data (the inverse of the preprocessing function, and begins with `post`), and a third command that will process new data using parameters that have already been computed (for sending new data to the network, and begins with `tra`).

prestd	Normalize data for zero mean, stand. dev. 1
poststd	Undo the PRESTD command
trastd	Process new data using the precalculated parameters.
premnmx	Normalize data for maximum of 1 and minimum of -1.
postmnmx	Undo the PREMNMX command.
tramnmx	Transform new data with precalculated parameters.
prepca	Principal component analysis on input data with zero mean.
	Matlab doesn't provide a post processor.
trapca	Transform data with PCA matrix computed by PREPCA.

Command Input/Output Arguments:

```
[pn,meanp,stdp,tn,meant,stdt] = prestd(p,t)
[p,t] = poststd(pn,meanp,stdp,tn,meant,stdt)
[pn] = trastd(p,meanp,stdp)

[pn,minp,maxp,tn,mint,maxt] = premnmx(p,t)
[p,t] = postmnmx(pn,minp,maxp,tn,mint,maxt)
[pn] = trastd(p,minp,maxp)
```

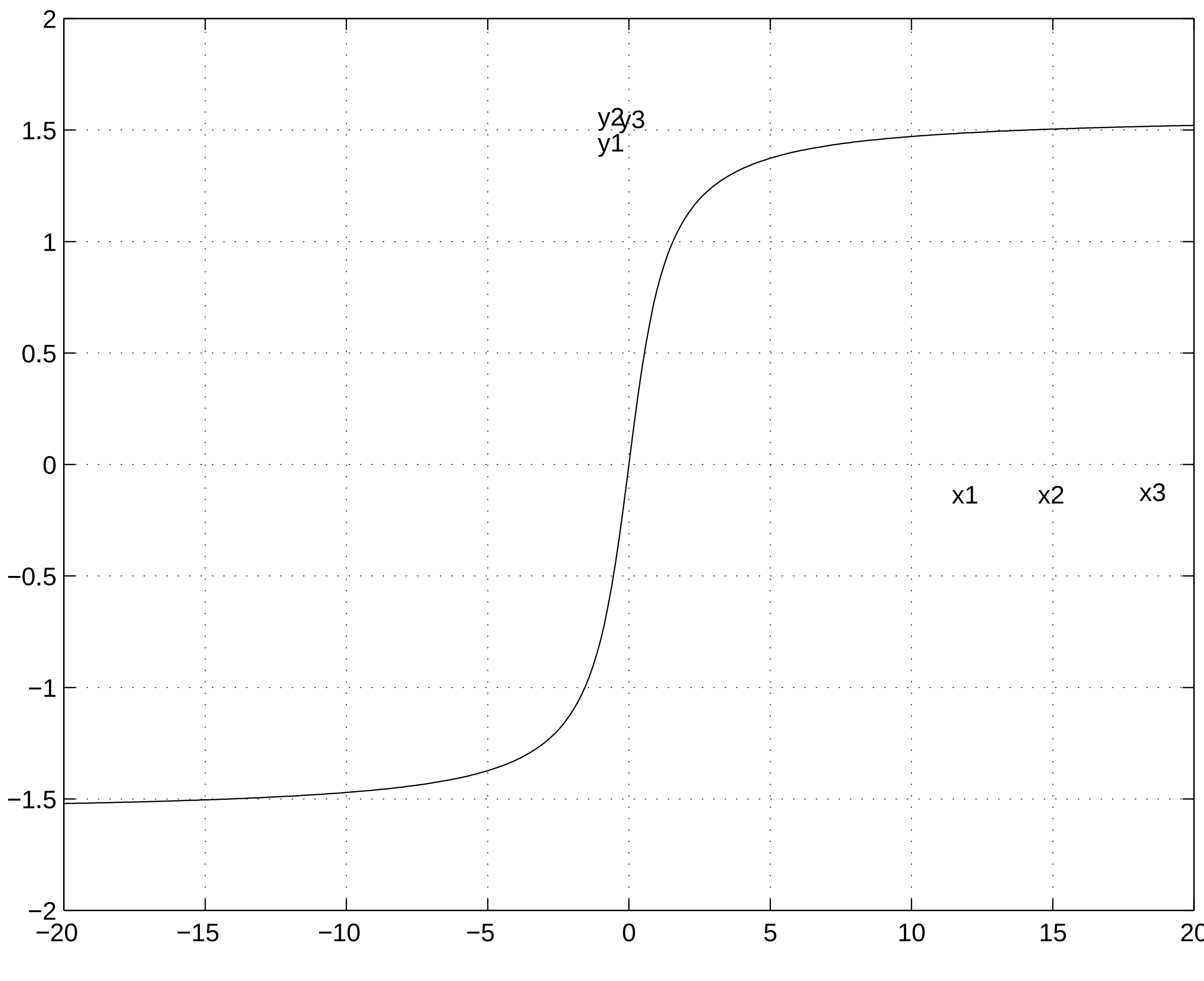

Figure 7.2: The Sigmoidal Function, and an example of "Saturation": As the size of the input values gets large, σ loses its ability to distinguish between data points.

```
[ptrans,transMat] = prepca(P,min_frac)
[Ptrans] = trapca(P,TransMat)
```

Exercise: Write commands prekl, postkl and trakl that are similar to the pca commands, but also performs mean subtraction. You'll need to decide on input/output arguments.

Exercise: Write commands premax, postmax, and tramax that transform a data set so that it has zero mean and a maximum absolute value (in each dimension) of 1. Again, you'll need to decide on input/output arguments.

Network Complexity and Generalization

By the theorems we stated earlier, it is possible for a three layer neural net to approximate an arbitrary continuous function over a compact domain to an arbitrary precision. That is, if we added enough nodes in the hidden layer, we could get arbitrarily close to the function. By the way, in the proof of such statements, it is assumed that there is enough data to accurately reproduce the function to arbitrary accuracy.

So adding new hidden nodes in a three layer network is akin to adding new centers to an RBF. Thus, we begin to run into the same old problem in the approximation of a function: The tradeoff between accuracy on the training set, and generalization off the training set.

The rule of thumb in getting a good neural net is to use the least number of hidden nodes that you can to still get the accuracy that is desired.

Nice Demonstration: In Matlab, run the demonstration `nnd11gn`. This will animate the network response as the training algorithm proceeds.

An alternative procedure that I would not recommend, but other researchers do is to use plenty of hidden nodes, but to stop training early. The reason I don't like this option is that the more hidden nodes one uses, the higher the dimension of the error space. For example, if the input dimension is n and the output dimension is m, then adding another hidden node adds $n + m + 1$ extra training parameters, and so increases the dimension of the error space by $n + m + 1$. It is common in network training to get stuck in local minima. By increasing the dimension of the error space, it is my opinion that you increase the chances of getting stuck (not to mention that you increase the amount of computation and memory required to train).

One way that we might think about the number of nodes in the hidden layer is to consider that the first part of the mapping (before the sigmoidals are applied) is a linear transformation from $\mathbb{R}^n$ to $\mathbb{R}^k$, where the input dimension is n and we have k hidden nodes. Heuristically, we need enough nodes in the hidden layer to unfold the data in such a way as to allow the sigmoidals to differentiate between data points (both in terms of saturation and in keeping the projection $1-1$).

7.5 Post Training Analysis

How do we determine whether or not our network is doing a good job? Below we consider some options for testing the neural network.

7.5.1 Test how the network works on the Validation Set. The validation set is there as a check to make sure we're not overtraining the neural network. One way to use the validation set is to stop training and check the error on the validation set. The idea is that initially the error on both the training and validation sets will decrease, but as the network begins to overtrain (if it does), then the error on the validation set will increase. If this occurs, we stop training.

Matlab has this utility built into its `train` function.

An example:

```
%Create training set
p=[-1:0.05:1];
t=sin(2*pi*p)+0.1*randn(size(p));

%Create a validation set
v.P=[-0.975:0.05:0.975];
v.T=[sin(2*pi*v.P)+0.1*randn(size(v.P))];

%Create a network, and train using the validation
net=newff(minmax(p),[20,1]);
net.trainParam.show=25;
net.trainParam.epochs=300;
[net,tr]=train(net,p,t,[],[],v);
```

Here we also see the added output argument to `train`, a vector `tr`, which is a training record of the errors. The added empty vectors are for "input delay" arguments, which we will not use in this course.

When run, the training should stop early due to an increase of the error on the validation set.

7.5.2 A second method is to perform some analysis on the errors that the network produces. There are several methods to do this (including some statistics that are beyond the scope of this course). We discuss three commonly used procedures:

(a) Matlab has a `postreg` function to analyze errors for 1-dimensional output. This function performs a linear regression between the neural output of the function and the target values. The general idea is that the higher the correlation, the better the training for the network. Here is an example:

```
p=[-1:0.05:1];
t=sin(2*pi*p)+0.1*randn(size(p));

v.P=[-0.975:0.05:0.975];
v.T=[sin(2*pi*v.P)+0.1*randn(size(v.P))];

net=newff(minmax(p),[10,1]);
net.trainParam.show=25;
net.trainParam.epochs=200;
[net,tr]=train(net,p,t);
an=sim(net,v.P);
[m,b,r]=postreg(an,v.T);
```

with the resulting graph and correlation as shown in Figure 7.3.

(b) A second method is taken from the idea that if the training is good, then the error should be uncorrelated. Put another way, all correlations should be explained by the neural network model. We can therefore look at the correlations in the error. Here's an example:

```
x=3*rand(500,3)-1;
y(:,1)=x(:,1).^2-x(:,2);
y(:,2)=y(:,1)-exp(x(:,3));
y(:,3)=sin(x(:,1));
net=newff(minmax(x'),[20,3],{'tansig','purelin'});
```

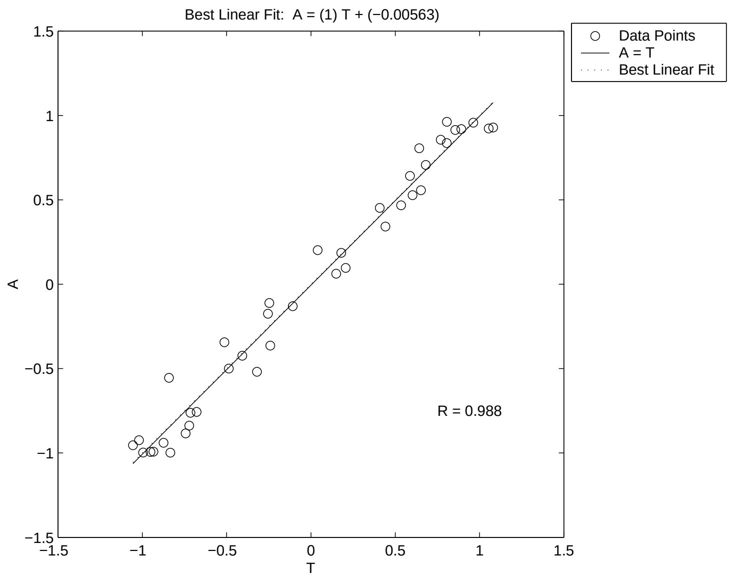

Figure 7.3: The result of using post regression on the targets. Obtaining a correlation as shown means that the network has trained well.

```
net.trainParam.epochs=75;
net=train(net,x',y');
vp=3*rand(200,3)-1;
vt(:,1)=vp(:,1).^2-vp(:,2);
vt(:,2)=vt(:,1)-exp(vp(:,3));
vt(:,3)=sin(vp(:,1));
an=sim(net,vp');
G=an-vt';
[U,S,V]=svd((1/200)*G*G');
S=
   1.0e-06 *

    0.3983         0         0
         0    0.3534         0
         0         0    0.0042
```

(c) The last method we discuss is similar to the second. We can examine the histogram of the size of the errors. If training went well, we can expect the errors to be clustering about zero, and rapidly trail off.

7.5.3 Examine the weights of the network. Overtraining will generally lead to very large weights (recall what some of the weights looked like in the RBF network). Extremely large weights are undesireable, because they can lead to instabilities in the network- i.e., small deviations in the domain can lead to huge deviations in the range.

Similarly, weights that are extremely small may suggest that training could have been done with fewer nodes in the hidden layer.

7.5.4 Regularization. A detailed explanation of regularization is beyond the scope of these notes, but we can give some heuristics. The basic idea is to penalize those weights that get very large in favor of small weights. Formally, we change the error function. Let E denote our standard mean squared error. Assume all training parameters (weights and biases) are indexed as: W_i, $i = 1, 2, \ldots, N$. The new error function, MSE_{reg}, is defined as:

$$\text{MSE}_{\text{reg}} = \gamma E + (1-\gamma)\frac{1}{N}\sum_{i=1}^{N} W_i$$

where γ is called the *regularization parameter*. As we can see, if $\gamma = 1$, the weights play no role in the training, but if $\gamma = 0$, the weights play the only role in the training. In general, the smaller the parameter, the more smooth the approximation.

The problem is in knowing what to make the regularization parameter. If it is too large, the network may overfit. If it is too small, we may not be modeling the data very well. Before considering automatic methods to determine the regularization parameter, the following example shows how to implement this new error:

```
net=newff(minmax(X),[5,1]);
net.performFcn='msereg';
net.performParam.ratio=0.5 %This is gamma
```

Matlab has a Bayesian regularization technique installed that tries to use Bayesian statistics to determine the best value of γ. In addition, it outputs a value called "effective number of parameters" of the network. If the effective number of parameters is far less than the number of parameters used, it suggests that you use a simpler network for training. Here is an example of its use:

```
net=newff(minmax(X),[5,1],{'tansig','purelin'},'trainbr');
```

That is, the `trainbr` stands for training with Bayesian regularization.

7.6 Example: Alphabet recognition

In this project, we look at how a neural network can be used for character recognition. This is extremely useful for companies such as the U.S. Postal Service, and they actually use neural networks to accomplish the reading of address information.

We will do a simpler example. We will assume that a letter is represented by a 5×7 matrix of values that are either 1 or 0. Therefore, the domain of the neural network will be 35 dimensional.

The output of the neural network will be 26-dimensional, where there is a 1 in the numerical position corresponding to its letter.

Our goal: Character recognition, even when the characters have been contaminated by noise[2].

Architectural Items

- We will utilize a $35 - 10 - 26$ feed forward neural network using a `tansig` transfer function for both the hidden and output layers.
- We will use a standard gradient descent to speed up training, `traingdx` in place of the default `trainlm`.
- We will use a sum of squares error `sse` in place of the usual mean square error `mse`.
- The training error goal should be set to 0.1.

[2]In real world systems, the contamination is not by noise but by people's sloppy hand-writing!

- Use a maximum of 5000 training epochs.
- Additionally, we need to set a momentum term for the gradient descent by typing: `net.trainParam.mc=0.95`

The Example

Run the example `appcr1`. Print and analyze the file `appcr1.m`. Annotate each line of code to understand what is occuring.

7.7 Project 1: Mushroom Classification

In this project, we use a neural net to perform classification on mushroom characteristics. That is, the neural net should input several characteristics of a mushroom, and output whether or not that mushroom is poisonous or edible.

The data consists of a matrix of 4062 entries each containing 22 characteristics of a mushroom (i.e., cap size, cap shape, stalk color, etc.). The range is either 0 or 1, depending on whether the mushroom is edible or not. Obtain this data by loading `mushroom_pre.mat`.

Project Questions:

7.7.1 Discuss any problems there might be in using an RBF network.

7.7.2 Decide on how you are going to preprocess the data. Run a global KL on the data to see how it is sitting in high dimensional space. Plot the best two dimensional representation.

7.7.3 Retain 1,000 points of data for training, and reserve the rest for validation of the network model. Use as few nodes in the hidden layer as necessary in order to get an error of 0 in classification.

7.7.4 Plot the results of the training. That is, first plot the network results for those mushrooms corresponding to class 1 as blue dots. Plot the network output for those mushrooms corresponding to class 0 as red dots. You should see a wide seperation between red and blue.

7.7.5 Does preprocessing the data help? Rerun the network training, except now use enough dimensions to retain 90% of the energy. Use the projected data for the domain of your network.

7.7.6 Turn in the script file for training the two networks, and any plots you used.

7.8 Autoassociation Neural Networks

In the previous section, we saw that a three layer feedforward neural network is capable of approximating any continuous function on a compact domain.

Suppose now that we want to reduce the dimensionality of the data set X. This can be expressed as constructing homeomorphisms

$$G : X \subset \mathbb{R}^N \to Y \subset \mathbb{R}^M$$
$$H : Y \subset \mathbb{R}^M \to X \subset \mathbb{R}^N$$

We could build two separate neural networks, one for each function, if we knew a priori the target values in the reduced dimensional space Y. For the moment, suppose we are not concerned with the form of the reduced representation. Then the compression paradigm can be reformulated: find functions G and H as before, but we only require that the composition $G \circ H$ is the identity on X. We envision this composition as a single, large network composed of two, three layer networks glued together, as shown in Figure 7.4.

In this figure, the input and output layers have N nodes, the bottleneck layer has $M << N$ nodes, and the two hidden layers have some predetermined number of nodes. It has been shown in [6, 8, 22] that these hidden layers are necessary to perform a nonlinear compression and decompression. That is, without the nonlinear hidden layer, this network performs a linear dimensionality reduction- for which we know the optimal solution is given by its Karhunen-Loève expansion.

This network has been widely used as a compression technique, and in a recent work [14] several criteria were included in the training algorithm to force certain specific representations on the bottleneck layer.

Matlab and the Autoassociator

In Matlab, it is simple to implement an autoassociative network, since we can define any number of hidden layers. In this case, we have a 5 layer network,

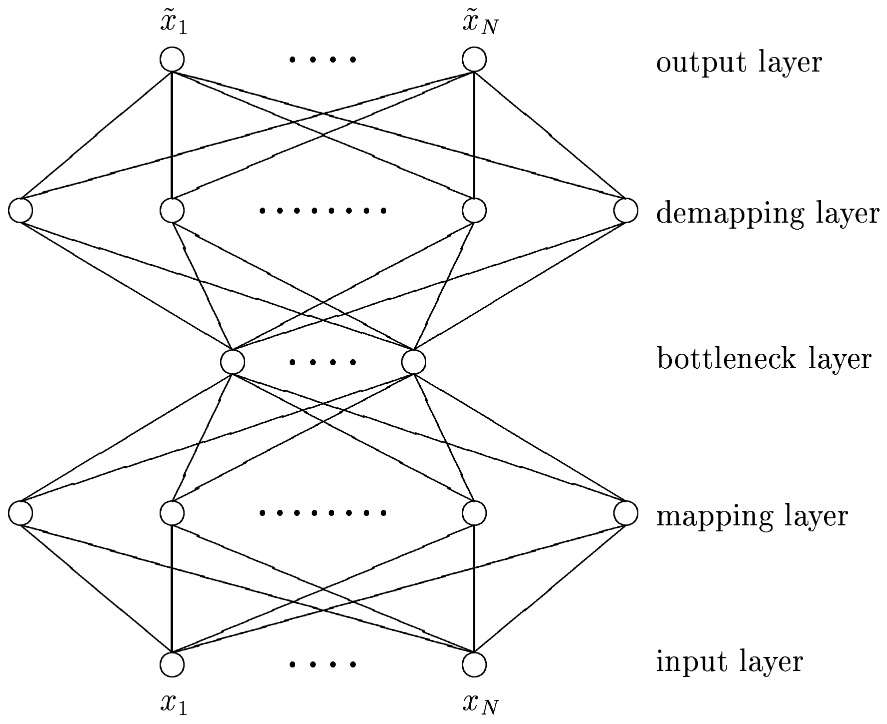

Figure 7.4: The bottleneck network architecture.

so that if the input dimension is n, the hidden layer dimension is k, and the target dimension is m, then the command:

```
net=newff(minmax(P),[k,m,k,n],{'tansig','tansig','tansig','purelin'});
```

will construct the network. Notice that the domain data is the same as the range data.

Question: Why will the neural network not simply construct an identity function, $f(x) = x$?

Project: Unknotting a knot