Chapter 10

Linear Neural Networks

In this chapter, we introduce the concept of the linear neural network.

10.1 Introduction and Notation

1. The linear neural "cell", or "node" has the schematic form as shown in Figure 10.1 where information flows from left to right in the following way:

   - Present real numbers $x_1, \ldots, x_n$ to the "input layer", as shown.
   - The "w_{1j}" corresponds to a "weight". A weighted edge corresponds to multiplication by w_{1j}. In general, weight w_{ij} corresponds to the edge going FROM node j TO node i.
   - At node L, the sum of the incoming signals is taken, and added to a value, b. We think of b as the "resting state" of the cell.
   - The result is passed along the axon.

2. Mathematically, the end result is:

$$\boldsymbol{x} \mapsto (w_{11}, w_{12}, \ldots, w_{1n}) \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} + b$$

   or

$$\boldsymbol{x} \mapsto w_{11}x_1 + w_{12}x_2 + \ldots + w_{1n}x_n + b$$

   which we can visualize as an $n-$dimensional plane, with normal vector $(w_{11}, \ldots, w_{1n})^T$.

3. We therefore can think of a linear neural node as an affine map.

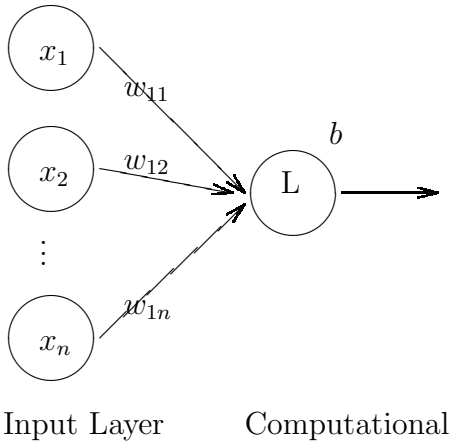

Figure 10.1: The Linear Node.

Figure 10.2: The Linear Neural Network is an affine mapping from $\mathbb{R}^n$ to $\mathbb{R}^k$

4. We can hook multiple computational nodes together to obtain a *linear neural network*, as shown in Figure 10.2.

5. Exercise: Show this mapping corresponds with the affine transformation $A\boldsymbol{x} + \boldsymbol{b}$, for an appropriate matrix A and vector b.

6. **Theorem:** A multilayer linear neural network is equivalent to a single layer linear neural network.

   Proof: Suppose that the network has "n" nodes in the input layer, and has $N_1, N_2, \ldots, N_k$ nodes in the k hidden layers, with m nodes in the output layer. Then the mapping from the input layer to the first layer is an affine mapping from $\mathbb{R}^n$ to $\mathbb{R}^{N_1}$:

   $$\boldsymbol{x} \mapsto A_1\boldsymbol{x} + \boldsymbol{b}_1$$

   where A is $N_1 \times n$. Going to the next layer is realized as an affine mapping from $\mathbb{R}^{N_1}$ to $\mathbb{R}^{N_2}$

   $$A_2(A_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2 \doteq \hat{A}_1\boldsymbol{x} + \hat{\boldsymbol{b}}_1$$

   Continuing to the end, we get one affine mapping from $\mathbb{R}^n$ to $\mathbb{R}^m$:

   $$\hat{A}\boldsymbol{x} + \hat{\boldsymbol{b}}$$

where $\hat{A}$ is $m \times n$, and $\hat{\boldsymbol{b}}$ is $m \times 1$.

7. The affine problem $A\boldsymbol{x} + \boldsymbol{b} = \boldsymbol{y}$ is equalivalent to solving a linear problem, $\hat{A}\hat{\boldsymbol{x}} = \boldsymbol{y}$.

8. This technique (affine to linear by adding dimensions) is used extensively in computer graphics. See, for example, the section on homogeneous coordinates from David Lay's Linear Algebra.

9. Example: Suppose that we have the 2×2 affine problem:

$$\begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

This is equivalent to the linear problem:

$$\begin{pmatrix} a_1 & a_2 & b_1 \\ a_3 & a_4 & b_2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

If we only want a two dimensional output, we can leave off the last row (the last row is there to do "perspective projections").

10. Exercise: Show that the system of affine equations: $AX + B = Y$ is also equivalent to a linear problem: $\hat{A}\hat{X} = Y$

10.2 Training a Linear Net

"To Train" a linear network is to determine weights and biases that best (in the sense of some error) match a given input-output set. There are two distinct types of training: Training when all data is available, and on-line training. If all of the data is available, we will see that "training" corresponds to computing a pseudoinverse. On-line training is a training algorithm that partially updates the weights and biases at each data point, and we slowly evolve the network to best match the data. It is in the latter sense that we can describe a linear network as "adaptive".

1. **Definition:** "ADALINE": Adaptive Linear Node, "MADALINE": An Adaptive Linear Network.

2. **Training with all the data**. Let W be the matrix of weights for the linear neural net, and let $\hat{W}$ be the augmented matrix with the biases in the last column. Let X be the matrix of data[1], and $\hat{X}$ be the augmented

[1] If we use a bias, b, we don't have to mean subtract the data

matrix with ones in the bottom row. Then the mapping from $\mathbb{R}^n$ to $\mathbb{R}^k$ is given by:

$$W\boldsymbol{x}^{(i)} + \boldsymbol{b}$$

which in linear form is given matrix wise:

$$\hat{W}\hat{X}$$

so that to train a linear neural network is to solve:

$$\hat{W}\hat{X} = Y \quad \text{or} \quad \hat{X}^T\hat{W}^T = Y^T$$

In this form, we are solving:

$$AX = B$$

for X. In Matlab, we have some options here:

- `X=A\B` or in original notation: `W=Y/X`
- `X=pinv(A)*B` or `W=Y*pinv(A)`
- Using the SVD, use can construct the pseudoinverse manually. Here are the commands to solve $AX = B$ for X:

```
[U,S,V]=svd(A);
ss=diag(S);
%FIND THE DIMENSION, k using the singular vals in ss
%Once that is determined, continue:
invS=diag(1./ss(1:k));
pinvA=V(:,1:k)*invS*U(:,1:k)';
X=pinvA*B;
```

3. **Remark:** Which method is better? Between Matlab's commands, the backslash command is more efficient that the pseudoinverse command. This is because the pseudoinverse command first computes the actual pseudoinverse, then performs the matrix multiplication. On the other hand, the backslash command takes both matrix $\hat{X}$ and Y into account.

   If you want complete control as to the rank of $\hat{X}$ (i.e., if there's a lot of noise), then you may consider using the SVD so that you can see what the variances look like.

4. **Exercise:** Find the linear neural network for the mapping from X to Y (data is ordered) given below. Use Matlab's matrix division to solve it.

$$X = \left\{ \begin{pmatrix} 2 \\ 2 \end{pmatrix}, \begin{pmatrix} 1 \\ -2 \end{pmatrix}, \begin{pmatrix} -2 \\ 2 \end{pmatrix}, \begin{pmatrix} -1 \\ 1 \end{pmatrix} \right\} \quad Y = \{-1, 1, -1, 1\}$$

   Verify that the weights are approximately: $-0.1523, -0.5076$ and the bias is: 0.3807. The seperating line is the set of points that map to 0:

$$y = -\left(\frac{-0.1523}{-0.5076}\right)x - \left(\frac{0.3807}{-0.5076}\right)$$

5. **Pattern Classification:** In the general pattern classification problem, we know the class labels for each data point, but these will not have meaningful numerical values. For example, if we have classes $1, 2, \ldots, k$, does that necessarily mean that class 2 is "closer" to class 3 than class 10? Does an output of 3.5 mean that the desired target is between classes 3 and 4? We run into interpretation problems.

   One method for labeling that may be preferable is to set the k^{th} pattern label to e_k. In the two label problem, the output for data in pattern 1 would be set to $(1,0)^T$ and for pattern 2 would be $(0,1)^T$. This has an added benefit: we can interpret $(a,b) \to (\frac{a}{a+b}, \frac{b}{a+b})^T$ as a probability. That is, $\boldsymbol{x}$ has probability $\frac{a}{a+b}$ of being in pattern 1, and probability $\frac{b}{a+b}$ of being in pattern 2.

6. **Example:** The following example gives the solution to the pattern classification problem from Chapter 1 (See Figure **??**). The first set of commands creates the data that we will classify. This script file will reproduce (some of the data is random) the image in Figure 10.3.

```
1  X1=0.6*randn(2,300)+repmat([2;2],1,300);
2  X2=0.6*randn(2,300)+repmat([1;-2],1,300);
3  X=[X1 X2];
4  X(3,:)=ones(1,600);
5
6  Y=[repmat([1;0],1,300) repmat([0;1],1,300)];
7  C=Y/X;
8
9  %Plotting routines:
10 plot(X1(1,:),X1(2,:),'o',X2(1,:),X2(2,:),'x');
11 hold on
12 n1=min(X(1,:));n2=max(X(1,:));
14 t=linspace(n1,n2);
15 L1=(-C(1,1)*t+(-C(1,3)+0.5))./C(1,2);
16 L2=(-C(2,1)*t+(-C(2,3)+0.5))./C(2,2);
17 plot(t,L1,t,L2);
```

   - Lines 1-3 set up the data set X. We will take the first 300 points ($X1$) as pattern 1, and the last 300 points as pattern 2.
   - Line 4 sets up the augmented matrix for the bias.
   - Line 6 sets up the targets.
   - Line 7 is the training. The weights and biases are in the 2×3 matrix C.
   - Line 10: Plot the patterns
   - Line 12-17: Compute the seperating lines.

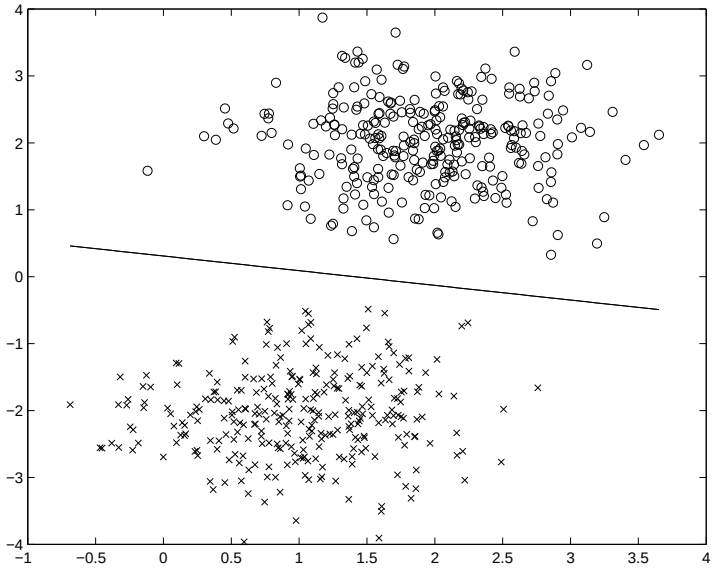

Figure 10.3: The Two Pattern Classification Problem from Chapter 1. The line is the preimage of $(0.5, 0.5)^T$.

- Line 18: Plot the seperating lines. (They are identical in theory, in practice they are very, very close).

7. **Online Training** We will derive the **Widrow-Hoff Learning Rule** for one dimensional outputs, then describe the generalization to higher dimensional output.

   We can describe the Widrow-Hoff Learning rule as an approximation to minimizing a function via gradient descent.

8. **Remark:** As we discussed in Chapter 2, the direction of largest decrease of a function $f : \mathbb{R}^n \to \mathbb{R}$ is in the direction of the negative of the gradient of f. If we view f as our Error Function, performing gradient descent will ideally result in minimizing the error.

9. **General Gradient Descent** Let $\boldsymbol{x} \in \mathbb{R}^n$, and let $f : \mathbb{R}^n \to \mathbb{R}$. Then performing gradient descent is performed by moving (updating) the original $\boldsymbol{x}$ in the following way:

   $$\boldsymbol{x} \to \boldsymbol{x} + \alpha \nabla f(\boldsymbol{x})$$

   Therefore, the Widrow-Hoff rule is an iterative procedure requiring an extra parameter. This parameter gives how far in the direction of the negative gradient we travel before stopping to recompute. In neural network terminology, this parameter is called the *learning rate*.

10. **Remark:** For more on the numerical aspects of general gradient descent and its implementation in C, consult "Numerical Recipes in C" [29].

11. **Derivation of the Widrow-Hoff rule:** Let $k = 1, 2, \ldots, p$ index the data. Let t_k denote the target for $\boldsymbol{x}_k$, and let y_k denote the output of the linear neural network:
$$\boldsymbol{w}^T \boldsymbol{x}_k + b = y_k \approx t_k$$
Then we wish to minimize the mean square error with respect to the weights and biases:
$$E_{\text{mse}} = \frac{1}{p} \sum_{k=1}^{p} e^2(k) \doteq \frac{1}{p} \sum_{k=1}^{p} (t_k - y_k)^2$$

12. **Exercise:** Verify that:
$$\frac{\partial e^2(k)}{\partial w_{1j}} = 2e(k) \frac{\partial e(k)}{\partial w_{1j}}$$
and
$$\frac{\partial e^2(k)}{\partial b} = 2e(k) \frac{\partial e(k)}{\partial b}$$

13. **Exercise:** Show that:
$$\frac{\partial e(k)}{\partial w_{1j}} = -x_j(k) \text{ and } \frac{\partial e(k)}{\partial b} = -1$$
where $x_j(k)$ refers to the j^{th} coordinate of the k^{th} data point.

14. **Remark:** The standard way of performing gradient descent would mean that we adjust the vector $\boldsymbol{w}$ and scalar b by:
$$\text{New } w_{1j} = \text{Old } w_{1j} + \alpha \frac{\partial E_{\text{mse}}}{\partial w_{1j}} \text{ and New } b = \text{Old } b + \alpha \frac{\partial E_{\text{mse}}}{\partial b}$$
where α is our learning rate. We will *estimate* E_{mse} at data point k:
$$w_{1j}(k+1) = w_{1j}(k) + \alpha \frac{\partial e^2(k)}{\partial w_{1j}} \text{ and } b(k+1) = b(k) + \alpha \frac{\partial e^2(k)}{\partial b}$$

15. **Exercise:** Show that the Widrow-Hoff rule is given by:
$$\boldsymbol{w}(k+1) = \boldsymbol{w} + 2\alpha(t_k - y_k)\boldsymbol{x}^{(k)}$$
$$b(k+1) = b(k) + 2\alpha(t_k - y_k)$$

16. **Extensions:** For multidimensional output, this update extends to:
$$W(k+1) = W(k) + 2\alpha \boldsymbol{e}(k) \left(\boldsymbol{x}^{(k)}\right)^T$$
$$\boldsymbol{b}(k+1) = \boldsymbol{b}(k) + 2\alpha \boldsymbol{e}(k)$$

17. **Widrow-Hoff in Matlab**

    For easy programming, we'll call 2α = lr for *learning rate*. Then, the function call to train the linear neural network will look like:

```
function [W,b,err]=wid_hoff1(X,Y,lr,iters)
%FUNCTION [W,b,err]=wid_hoff1(X,Y,lr,iters)
%This function trains a linear neural network
%using the Widrow-Hoff training algorithm.  This
%is a steepest descent method, and so will need
%a learning rate, lr (for example, lr=0.1)
%
%        Input:  Data sets X, Y (for input, output)
%                Dimensions:  number of points x dimension
%                lr:     Learning rate
%                iters:  Number of times to run through
%                        the data
%        Output: Weight matrix W and bias vector b so
%                that Wx+b approximates y.
%                err:  Training record for the error

%It's convenient to work with X and Y as dimension
%by number of points
X=X';
Y=Y';
[m1,m2]=size(X);
[n1,n2]=size(Y);

%Initialize W and b to zero
W=zeros(n1,m1);
b=zeros(n1,1);

for i=1:iters              %Number of times through data
  for j=1:m2               %Go through every data point
    e=(Y(:,j)-(W*X(:,j)+b));  %Target - Network Output
    dW=lr*e*X(:,j)';
    W=W+dW;
    b=b+lr*e;
    err(i,j)=norm(e);       %Store error for later
  end
end
```

18. We could also add a "momentum" term to try to speed up the gradient descent. A useful example of how the learning rate and momentum effect the convergence can be found in Matlab: `nnd10nc`, which we will also

look at in the next section. The general form of gradient descent with a momentum term μ and learning rate α is given by:

$$\Delta \boldsymbol{x} = \mu \Delta \boldsymbol{x} + (1-\mu)\alpha \nabla f(\boldsymbol{x}) \tag{10.1}$$

$$\boldsymbol{x} = \boldsymbol{x} + \Delta \boldsymbol{x} \tag{10.2}$$

From these equations, we see that if the momentum term is set so that $\mu = 0$, we have standard gradient descent (which may be too fast), and if we set $\mu = 1$, then since $\Delta \boldsymbol{x}$ is usually set to 0 to start, then $\Delta \boldsymbol{x}$ will be zero for all iterations.

10.3 Time Series and Linear Networks

We've already seen one application of linear networks: If data is linearly seperable, then a linear network can do pattern classification. Furthermore, if the input-output relationship is linear, then a linear net can approximate that relationship. Here, we will see that a linear neural network can be used in signal processing.

1. **Definition:** A time series is a sequence of real (or complex) numbers. We denote a time series in the usual way:

$$X = \{x(1), x(2), x(3), \ldots, x(t), \ldots\}$$

2. **Definition:** A tapped delay line with k taps is constructed from a time series:

$$\hat{x}_1 = \begin{pmatrix} x(k) \\ x(k-1) \\ \vdots \\ x(1) \end{pmatrix}, \quad \hat{x}_2 = \begin{pmatrix} x(k+1) \\ x(k) \\ \vdots \\ x(2) \end{pmatrix}, \ \ldots$$

3. **Remark:** This is also called a time series with *lag* k.

4. **Remark:** This is also called an *embedding* of the time series to $\mathbb{R}^k$.

5. **Remark:** In Matlab, we can do the embedding in the following way. Here, we embed to $\mathbb{R}^5$, columnwise:

```
Q=length(X);
P=zeros(5,Q);
P(1,2:Q)=X(1:(Q-1));
P(2,3:Q)=X(1:(Q-2));
P(3,4:Q)=X(1:(Q-3));
P(4,5:Q)=X(1:(Q-4));
P(5,6:Q)=X(1:(Q-5));
```

Look these lines over carefully so that you see what is being done- we're doing some padding with zeros so that we don't decrease the number of data points being used.

6. **Exercise:** Use the last remark as the basis for a function you write, call `lag`, whose input is a time series (length n), and input the number of taps, k. Output the $k \times n$ matrix of embedded points.

7. **Definition:** A *filter* with $k-$taps is a function on the time series so that:

$$x_i = f(x_{i-1}, x_{i-2}, \ldots, x_{i-k})$$

8. **Remark:** We can think of a filter as performing a prediction on x_i using the past k time series values.

9. **Definition:** A *linear filter* will have the form:

$$\boldsymbol{w}^T \boldsymbol{x} + b = x_i$$

where $\boldsymbol{w}$ are the weights, b is the bias, and $\boldsymbol{x} = (x_{i-1}, x_{i-2}, \ldots, x_{i-k})^T$.

10. **Remark:** In signals processing dialect, the linear filter above is called a Finite Impulse Response (FIR) filter.

11. **Exercise:** Run the demonstration `applin1`. The filter will be constructed using Matlab's `newlind` command, which uses a least squares approximation. Try solving the problem directly using the P and T data sets that were constructed by the demo, and compare the weights, bias and error.

12. **Application: Noise removal**

    - Background: There is a signal that we would like to have as pure as possible, but there is some noise contaminating it. For example, a pilot's voice may be contaminated by engine noise. We would like to remove the noise using a linear neural network. We assume that the noise *source* is available for sampling, but the noise contamination is an unknown function of the noise source.
    - GOAL: Filter out the noise, given only access to the noise source.
    - Idea for the solution:

      Suppose that the noise source is input (using time delays) to a linear filter. What can the linear network do? It can only form linear combinations of its past values, and therefore can only estimate signals that are (at least) correlated to the noise source. The pilot's voice (or signal of interest) should NOT be correlated to the noise.

      If we ask a linear network to model the noise PLUS the pilot's voice, the linear network will only be capable of modeling the noise.

      This gives us an easily implemented algorithm:

Let v_k be the main signal (or voice) sampled at time k. Let n_k be the sample of the noise source at time k. The contaminated signal is then: $v_k + f(n_k)$, where f is a (unknown) model of how the noise is transformed.

We will design the linear network so that:

- INPUT: $n_k, n_{k-1}, \ldots, n_{k-(m-1)}$ (m lags)
- DESIRED OUTPUT: $v_k + f(n_k) = c_k$
- ACTUAL NETWORK OUTPUT: a_k

Algorithm: At time k, input the lagged vector, and compute a_k. The error is $a_k - c_k$. Use the Widrow-Hoff learning rule to update the weights and bias.

- Use either Matlab's built in training routines (exmained in the next section), or modify `wid_hoff1.m` by changing the error measure.
- A sample is given in `nnd10nc` and `nnd10eeg`, where there is an interactive selection of the learning rates and the results. Kind of fun!

10.4 Script file: APPLIN2

This program shows what an *adaptive* network can do, versus a network that was trained using least squares, with no additional training. We will see that an adaptive network can respond rapidly to a changing input signal. The learning algorithm is Widrow-Hoff, so we also need a learning rate.

The input will be a signal that is static for the first 4 seconds of input, then changes period for the last two seconds. The linear network will be trained to predict x_k from $x_{k-1}, x_{k-2}, \ldots, x_{k-5}$.

The following is Matlab's script file `applin2` with my line numbers and comments (it's so short because I've removed their comments and plotting routines).

```
1   time1 = 0:0.05:4;      % from 0 to 4 seconds
2   time2 = 4.05:0.024:6;  % from 4 to 6 seconds
3   time = [time1 time2];  % from 0 to 6 seconds
4
5   T = con2seq([sin(time1*4*pi) sin(time2*8*pi)]);
6   P = T;
7
8   lr = 0.1;
9   delays = [1 2 3 4 5];
10  net = newlin(minmax(cat(2,P{:})),1,delays,lr);
11  [net,y,e]=adapt(net,P,T);
```

Program Comments

- Lines 1-3 set up a varying time scale to create a sample data set.
- Line 5 does two things. The first thing:

```
[sin(time1*4*pi) sin(time2*8*pi)]
```

sets up the signal, which will be input to the network, then converts the vector to a "cell array". The neural network has been written to work with this data type.

- Line 6: The input pattern is the "same" as the output pattern (but notice the delays defined in Line 10).
- Line 8: Set the learning rate for steepest descent.
- Line 9: Set the delays. Notice that this means that the lag vector is taken as we've stated earlier. If we wanted x_k to be a function of x_k, x_{k-3}, x_{k-5}, this vector would be: [0 3 5].
- Line 10: Creates and initializes the linear neural network. The `minmax(cat(2,P{:}))` command is used to define the minimum and maximum values of the input for each dimension. To see why we are using the `cat` command, see the last comment below. For more options and other types of arguments type `help newlin`
- Line 11 trains the network using Widrow-Hoff. It returns the network output, y and the error e as cell arrays. To plot them or work with them, you can convert them back into regular vectors with the command `cat(2,e{:})`

10.5 Matlab Demonstration

Here we describe the Matlab demonstration `nnd10nc`, which demonstrates the noise removal technique using a two node linear neural network. The screen shot is given in Figure 10.4, where we see a main viewing screen, a smaller screen and two scroll bars.

In the main viewing window, one can see the effect of the noise removal. In the smaller screen, one can see the effects of the learning rate and momentum terms on the convergence of the gradient descent algorithm. The lines on the screen correspond to the level curves of the error function, whose global minimum occurs at the center (this graph is only possible due to only having two input nodes).

Recall that, if the momentum term is set to 1, then there is no gradient descent, and if the momentum term is set to 0, we have a standard gradient descent with learning rate. Note that if the learning rate is set too high, we can miss the minimum!

Also see the related demonstration, `nnd10eeg`, which uses a real Electroencephalogram trace!

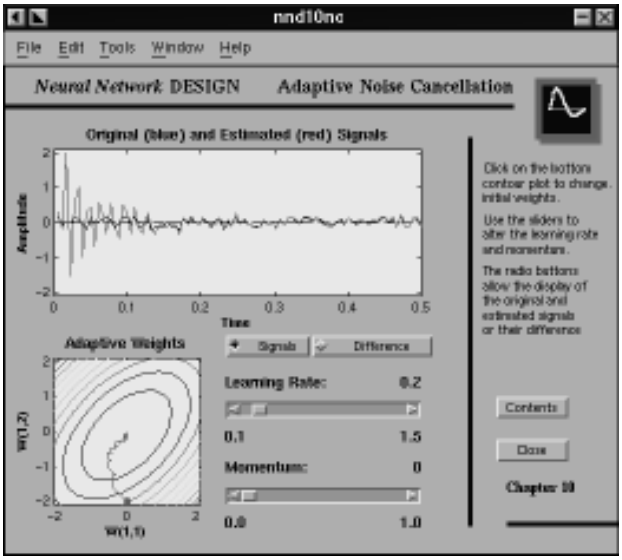

Figure 10.4: Screenshot of the Matlab toolbox demonstration, nnd10nc.

10.6 Summary

A linear neural network is an affine mapping. The network can be trained two ways: Least squares (using the pseudoinverse) if we have all data available, or adaptively, using the Widrow-Hoff algorithm, which is an approximate gradient descent on the least squares error.

Applications to time series include noise removal and prediction, where we first embed the time series to $\mathbb{R}^k$. These applications are widely used in the signals processing community.