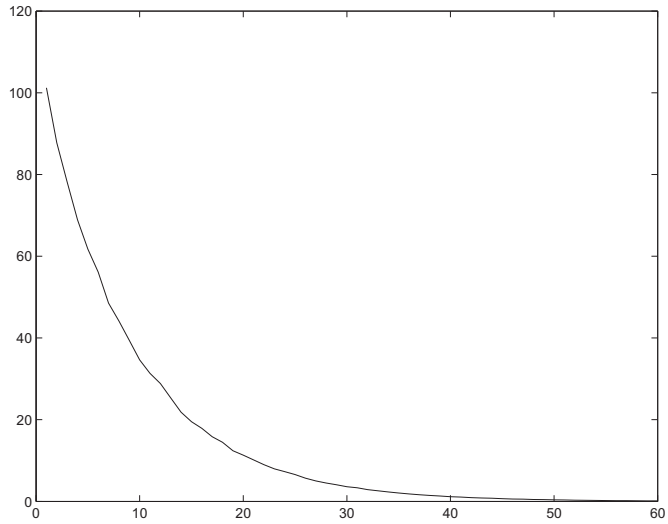

Figure 2: The errors for the Widrow-Hoff rule applied to letter recognition (or associative memory. After 60 passes through the data, the associations are very good.

```
    for j=1:NumPoints
        ThisOut=W*X(:,idx(j))+b;
        ThisErr=T(idx(j))-ThisOut;
    %Update the weights and biases
        W=W+alpha*ThisErr*X(:,idx(j))';
        b=b+alpha*ThisErr;
    end
    EpochErr(k)=norm((W*X+b*ones(1,6))-T);
end
figure(2)
plot(EpochErr);
```

The plot of the error is shown in Figure 2. The horizontal axis counts the number of passes through the data, and the vertical axis gives the sum of the squared errors. Note that after 60 passes, we get very good classification of the letters!

You should put this code into Matlab and reproduce the figures.

Exercise: Show that the system of equations $A\mathbf{x} + \mathbf{b}$ can be written as a linear system:

$$\hat{A}\hat{\mathbf{x}}$$

for an appropriate matrix $\hat{A}$ and $\hat{\mathbf{x}}$. This means that Hebb's rule could be performed directly without training $\mathbf{b}$ separately.

Vocabulary of Learning

"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 have **batch training**, and this means that we need to solve some system of equations (least squares) for the weights and biases. Finding a line of best fit is **batch training**.

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", and Hebb's rule was **on-line**.

Example

Find the linear neural network for the mapping from X to Y (data is ordered) given below. Use batch training instead of Hebb's rule.

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

SOLUTION: We'll build the system of equations as:

$$WX = Y$$

where

$$X = \begin{bmatrix} 2 & 1 & -2 & -1 \\ 2 & -2 & 2 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \qquad Y = [-1,\ 1,\ -1,\ 1]$$

In Matlab, the weight matrix W is found as: `Y/X`, as we see below. We will also illustrate the solution by plotting it. For your reference, the weight matrix was:

$$W = [-0.1523,\ -0.5076,\ 0.3807]$$

And WX gave:

```
 -0.9391    1.2437   -0.3299    0.0254
```

Here is the Matlab code:

```
X=[2 1 -2 -1;2 -2 2 1];
X(3,:)=ones(1,4);
Y=[-1 1 -1 1];
W=Y/X;
plot(X(1,[1,3]),X(2,[1,3]),'^',X(1,[2,4]),X(2,[2,4]),'x');
x1=-3;x2=3;y1=-3;y2=3;
t=linspace(x1,x2);
%Seperating line is ax+by+c=0
y=-(W(1)/W(2))*t-(W(3)/W(2));
hold on
plot(X(1,[1,3]),X(2,[1,3]),'o',X(1,[2,4]),X(2,[2,4]),'o');
plot(t,y);
axis([x1 x2 y1 y2]);
```

Pattern Classification

In the general pattern classification problem, we are given samples of data that represent each class. These samples are translated into vectors $\mathbf{x}_i \in \mathbb{R}^n$. Each sample is identified with a class label. Our goal is to build a function that will input a sample and output the class to which the sample belongs.

In fact, we have seen that a set of linear neurons may be able to build such a classifier. Before we continue, you may note that the class labels do not really have any intrinsic numerical value- They simply refer to which class the sample belongs, and we have indexed the classes.

For example, if we have classes $1, 2, 3, 4, 5$, does that necessarily mean that class 1 is "closer" to class 2 than class 5? Does an output of 3.5 mean that the desired target is between classes 3 and 4? You see, these numbers have no meaning.

Therefore, we should translate the classes into vectors for which we can assign some meaning. 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.

Example:

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 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.

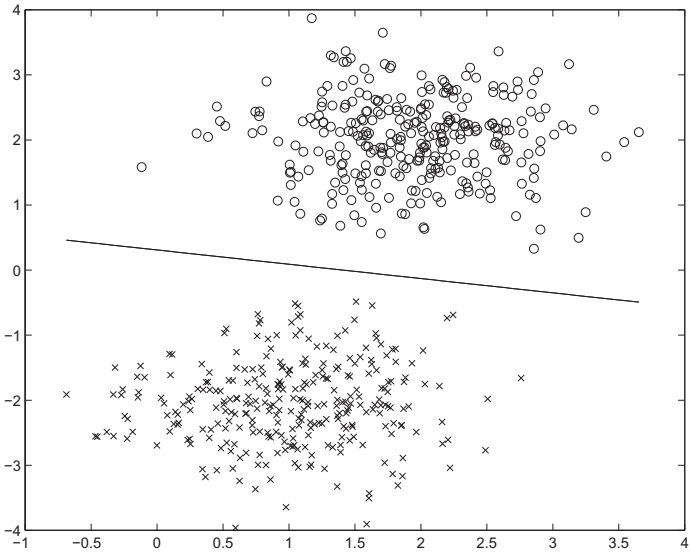

Figure 3: The Two Pattern Classification Problem. The line is the preimage of $(0.5, 0.5)^T$.

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

Exercise

Let's put all of this together to solve another pattern classification problem using Hebb's rule. Suppose we are given the following associations:

Point	Class
$(1, 1)$	1
$(1, 2)$	1
$(2, -1)$	2
$(2, 0)$	2
$(-1, 2)$	3
$(-2, 1)$	3
$(-1, -1)$	4
$(-2, -2)$	4

Graphically, we can see the classes in the plane in Figure 4. In this example, take Class 1 to be the vector $[-1, -1]^T$, Class 2 as vector $[-1, 1]^T$, Class 3 as $[1, -1]^T$, and Class 4 as $[1, 1]^T$- this puts the 4 classes are on the vertices of a square.

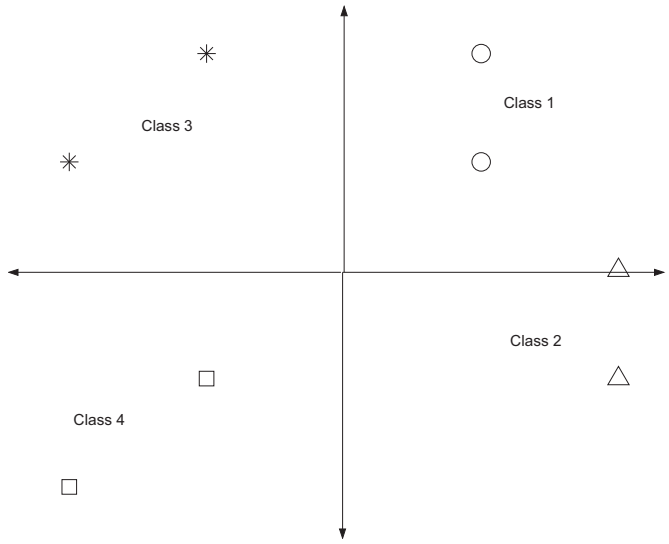

Figure 4: Pattern Classification Problem. Each point is a sample of one of the four classes.

Now for the details of the program. First write the inputs as an 2×8 matrix, with a corresponding output matrix that is also 2×8. Parameters that can be placed first will be the maximum number of times through the data N and the learning rate, a, which we will set to 0.04. We can also set an error bound so that we might stop early. Set the initial weights to the 2×2 identity, and the bias vector b to $[1, 1]^T$.

Be sure you have trained long enough to get a good error, and plot the decision boundaries as well.

Online Training: Widrow Hoff

Before we discuss online training further, be sure you have read through the Appendix on derivatives and the method called gradient descent (for minimizing some error function).

In 1960 Bernard Widrow and his graduate student Marcian Hoff developed a new neural network and a new learning rule which they called the LMS (Least Mean Square) Algorithm- which is an online algorithm designed to minimize the mean square error (MSE). That is, given n stimulus-response pairs, $(\mathbf{x}_i, \mathbf{t}_i)$, and given a matrix W and vector $\mathbf{b}$ so that

$$\mathbf{y}_i \doteq W\mathbf{x}_i + \mathbf{b}$$

the MSE is given by

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

The (online) training rule proposed by Widrow and Hoff is the same as our modification to Hebb's rule:

$$W^{(new)} = W^{(old)} + \alpha\,(\mathbf{t}_i - \mathbf{y}_i)\,\mathbf{x}_i^T$$

Derivation of the Rule

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.

To simplify the derivation, let the matrix W be a row vector so that

$$W\mathbf{x}_i + \mathbf{b} \Rightarrow \mathbf{w}^T\mathbf{x}_i + b$$

Then we wish to minimize the mean square error with respect to the weights and biases:

$$E_{\text{mse}} = \frac{1}{p}\sum_{k=1}^{p} e^2(k) \doteq \frac{1}{p}\sum_{k=1}^{p}(t_k - y_k)^2$$

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

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

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

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

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

5. **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
```

6. 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:

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

$$\boldsymbol{x} = \boldsymbol{x} + \Delta \boldsymbol{x} \quad (2)$$

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.

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!

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.

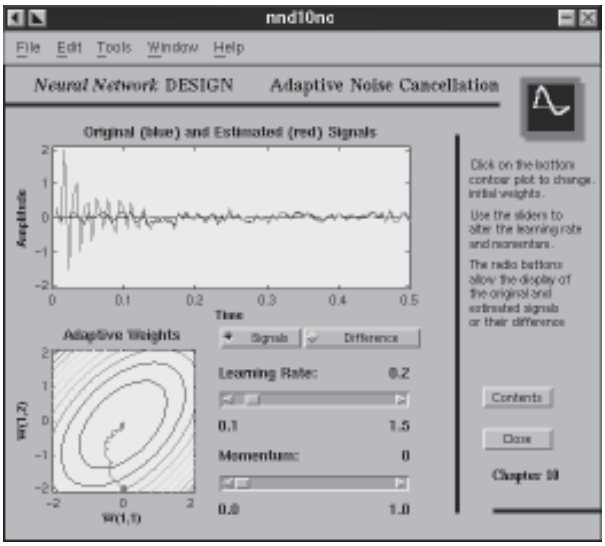

Figure 5: Screenshot of the Matlab toolbox demonstration, nnd10nc.

- 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{:})`

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 5, 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!

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.

Bibliography

References

[1] Fadeley, R. Oregon malignancy pattern physiographically related to Hanford, Washington, Radioisotope Storage. Journal of Environmental Health, 27(6), p. 883–897, June 1965.