Chapter 11

Linear Neural Networks

In this chapter, we introduce the concept of the linear neural network. As we will see, a neural network is a biologically inspired algorithm that allows us to build a functional representation from data. In statistical terms, a neural network generally represents nonlinear regression. In this chapter, however, we start with a linear network and examine its properties and shortcomings.

11.1 A Model of Learning

D.O. Hebb (1904-1985) was a physiological psychologist at McGill University. In Hebb's view, learning could be described physiologically: There is some physical change in the nervous system to accommodate learning, and that change is summarized by what we now call Hebb's postulate (from his 1949 book):

> When an axon of cell A is near enough to excite a cell B and repeatedly takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.

As with many named theorems and postulates, this was not an idea that was completely new, but he does give the postulate in a form that can be used as a basis for machine learning. Here are some questions you might think about:

1. Hebb's postulate describes a strengthing or weakening of connections. How is this biologically or chemically done? What exactly is that "physical change"?
2. There is also near instantaneous learning- Circumstances that are so emotionally intense, that we do not require repeated exposure. But perhaps internally repeating the exerience is enough.
3. The postulate does not give any consideration to *feedback*. If the action causes pain, will the neuron connections still be strengthened? How does emotion in general mitigate or assist in these constructions? Again, we can't give authoritative answers to these questions.

We'll leave these questions for you to think about, and move into something we can answer. How do you mathematically model Hebb's postulate?

11.2 Linear Neural Nets

We will go into the formal details later for defining *neural nets*, but this is a good place to get a feel for what they're all about.

Let us first build a simple model for a neuron. A neuron has three body parts- The dendrites, which carry information to the cell body, the cell body, and the axon, which carries information away from the cell body.

Multiple signals come in to the cell body from the dendrites. Mathematically, we will assume they all arrive at the same time, and the action of the dendrites (or the arrival site of the cell body) is that each signal is changed by the physiology of the cell. That is, if x_i is information along dendrite i, arrival at the cell body changes it to $w_i x_i$, where w_i is some real value. Next, the cell body collates this information by summing these signals together. So far, then, this action is simply a dot product of the vector $\boldsymbol{w}$ (the *weights*) to the signal $\boldsymbol{x}$. An additional value is added to the result, which we can think of as the resting state of the cell (or in statistical terms, the bias). In Figure 11.1, we graphically depict the flow of information from the input layer to the output layer.

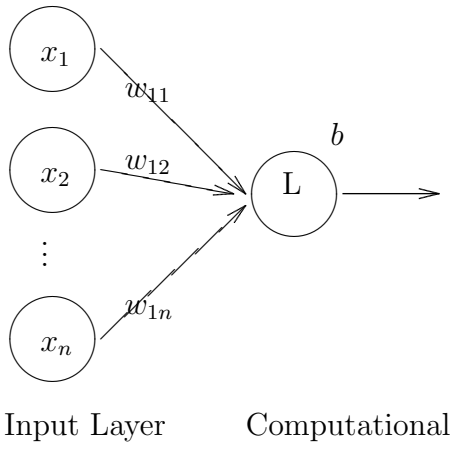


Figure 11.1: The Linear Node. Information travels from left to right along the edges.

Some vocabulary that we'll use:

- $x_1, \ldots, x_n$ are presented to the "input layer". Some researchers call this an actual layer, some do not (which makes some counts of the network different).
- The w's are called the "weights", and we will also denote the edge by the same notation. When a signal travels along an edge, the result is that the signal is multiplied by the weight.
- 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, which is also called the bias term.

We see that mathematically, this single node of a linear network is an affine function from $\mathbb{R}^n$ to $\mathbb{R}$:

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

If we have m neurons and $\mathbf{x}$ is a vector in $\mathbb{R}^n$, then W is a $m \times n$ matrix, and each row corresponds to a signal neuron's weights (that is, W_{ij} refers to the weight taking x_j to neuron i). Graphically, we see this in Figure 11.2. The full mapping is now formally an affine map from $\mathbb{R}^n$ to $\mathbb{R}^m$ (affine because we're adding the bias terms in $\boldsymbol{b}$).

$$\boldsymbol{x} \to W\boldsymbol{x} + \boldsymbol{b}$$

As we know, problems that are *linear* are usually easier to work with, so we can use a "trick" from computer graphics to convert our affine map to a linear map by going up one dimension. First an example of how this will be done:

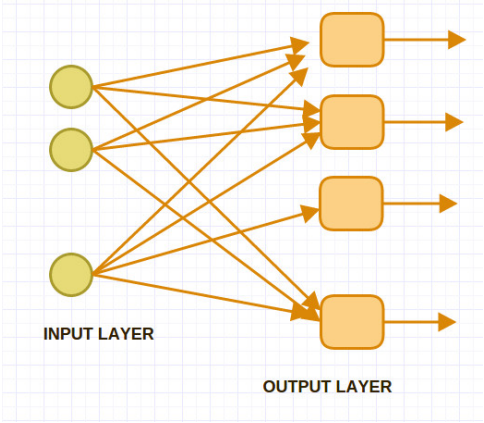


Figure 11.2: The Linear Neural Network is an affine mapping from $\mathbb{R}^n$ to $\mathbb{R}^m$

Example (Convert Affine to Linear)

Suppose that we have the 2×2 affine problem:

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

If we put the vector $\boldsymbol{b}$ as the last column of the matrix, we just need to add a dimension to the vector $\boldsymbol{x}$ by putting a 1 in that position. That is, you should verify that our affine map is equivalent to the following linear map:

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

As a summary, we have the following conversion: Given the affine problem

$$A\boldsymbol{x} + \boldsymbol{b} = \boldsymbol{y}$$

define $\hat{A} = [A \quad \boldsymbol{b}]$ and $\hat{\boldsymbol{x}} = [\boldsymbol{x},\ 1]^T$. Then the affine map is equivalent to the linear map

$$\hat{A}\hat{\boldsymbol{x}} = \boldsymbol{y}.$$

11.3 Training a Network

So far, we've seen that a linear neural network is modeled by $W\boldsymbol{x} + \boldsymbol{b}$. In this case, we would call the weights and biases the **model parameters**. We now need to determine the model parameters given some data, and that is what is called **training**.

To set up the problem, we are given p data pairs (where t is for known "targets").

$$(\boldsymbol{x}^{(i)}, \boldsymbol{t}^{(i)})$$

where our goal is to determine the weights W and biases b so that

$$W\mathbf{x} + \mathbf{b} = \boldsymbol{y} \approx \boldsymbol{t}$$

There are many ways one may perform the training, but there is one way of classification that is helpful: On-line or Batch.

- On-line training (or on-line learning) is an adaptive approach where the weights and biases are modified after looking at a single data pair $(\boldsymbol{x}^{(i)}, \boldsymbol{t}^{(i)})$. A network with this kind of training is called an **Adaline** net (for "Adaptive Linear").

- Batch training is a one-step training method where the weights and biases are determined after the entire data set has been analyzed.

As it turns out, both training processes are designed to minimize an error function, and the most common way to construct an error function is to take the sum of squares error (SSE). If we assume that we have p data pairs, then the sum of squares error is defined as the following.

$$E(W, \boldsymbol{b}) = \sum_{j=1}^{p} \|\boldsymbol{t}^{(j)} - \boldsymbol{y}^{(j)}\|^2 = \sum_{j=1}^{p} \|\boldsymbol{t}^{(j)} - (W\boldsymbol{x}^{(j)} - \boldsymbol{b})\|^2$$

Some formulations of E will multiply it by $1/2$ so the derivative doesn't have a "2" in it, or will take the average error (so multiply by $1/p$). If we multiply E by a constant, the values of W and $\boldsymbol{b}$ that give us the optimal value will be the same, so we may do so if it makes our computations easier.

In the next section, we'll talk about online training, and later, we'll discuss batch training.

11.4 Hebbian Learning (On-line training)

We'll recall that the linear network inputs a pattern, $\boldsymbol{x} \in \mathbb{R}^n$, and it outputs a pattern, $\boldsymbol{y} \in \mathbb{R}^m$ (remember that in our notation, $\boldsymbol{y}$ represents the output of the network, and $\boldsymbol{t}$ is our known target data). In terms of the weights, matrix W is $m \times n$, and so individually,

W_{ij} connects the j^{th} value of the input to the i^{th} value of the output.

Thus we might take the following as Hebb's Rule: The change in the weight connecting the j^{th} input to the i^{th} cell is given by:

$$\Delta W_{ij} = \alpha y_i x_j$$

where α is called **the learning rate**.

If both x_j and y_i match in sign, then W_{ij} becomes larger, and if there is a mismatch in sign, W_{ij} gets smaller. This is the **unsupervised Hebbian rule**. Let's take a closer look at this using some linear algebra. With $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{y} \in \mathbb{R}^m$, then the outer produce $\mathbf{y}\mathbf{x}^T$ produces an $m \times n$ matrix (same dimensions as W):

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} [x_1, x_2, \ldots, x_n] = \begin{bmatrix} y_1x_1 & y_1x_2 & \ldots & y_1x_n \\ y_2x_1 & y_2x_2 & \ldots & y_2x_n \\ \vdots & & & \vdots \\ y_mx_1 & y_mx_2 & \ldots & y_mx_n \end{bmatrix}$$

You should verify that in this case, we can compactly write Hebb's rule as:

$$W_{\text{new}} = W_{\text{old}} + \alpha \mathbf{y}\mathbf{x}^T$$

and that this change is valid for a single $\mathbf{x}, \mathbf{y}$ stimulus-response pair.

There are some difficulties with unsupervised Hebbian learning- in particular, if α stays fixed, then the update rule will "blow up" on us- We need to incorporate the targets.

11.4.1 Widrow-Hoff Learning

We want to define the update rule in such a way as to go to zero as the output $\boldsymbol{y}$ gets closer to the target $\boldsymbol{t}$. Here is one way to accomplish this, and it is called the Widrow-Hoff learning rule[1]:

$$\Delta W_{ij} = \alpha(t_j - y_j)x_i$$

If we put this in matrix form, the learning rule becomes:

$$W_{\text{new}} = W_{\text{old}} + \alpha\left(\mathbf{t} - \mathbf{y}\right)\mathbf{x}^T \tag{11.1}$$

where $(\mathbf{x}, \mathbf{t})$ is a desired input-output relation, and $\mathbf{y} = W\mathbf{x} + \boldsymbol{b}$, and the update rule for the bias vector is similar:

$$\mathbf{b}_{\text{new}} = \mathbf{b}_{\text{old}} + \alpha\left(\mathbf{t} - \mathbf{y}\right) \tag{11.2}$$

11.4.2 Derivation of Widrow-Hoff (Exercises)

In this series of exercises, you'll see that the Widrow-Hoff update is really an approximation to gradient descent on our error function - in fact, we've seen it in stochastic gradient descent. First, we'll look at one-dimensional output, and then extend that to multidimensional output.

For notation, let $k = 1, 2, \ldots, p$ index the data. Let $(\boldsymbol{x}^{(k)}, t^{(k)})$ denote the input, target pairs for the linear network, and the output of the network is $y^{(k)} = \boldsymbol{w}^T\boldsymbol{x}^{(k)} + b$, and E is the SSE as defined earlier.

1. Find an expression for $\partial E/\partial w_j$ (be sure to substitute the function in for y), where E is our sum of squares error.
2. Find an expression for $\partial E/\partial b$.
3. Using the previous two answers, what would our update rule look like if we performed gradient descent on the error function?
4. Instead of using the full error function, we will estimate the full error by using only one data point. That is, for the k^{th} data point,
$$E(\boldsymbol{w}, b) \approx (t^{(k)} - y^{(k)})^2$$
Show that using the approximation gives us the Widrow-Hoff rule, where α is a constant.
$$\boldsymbol{w}_{\text{new}} = \boldsymbol{w}_{\text{old}} + \alpha(t_k - y_k)\boldsymbol{x}^{(k)} \qquad \text{and} \qquad b_{\text{new}} = b_{\text{old}} + \alpha(t_k - y_k).$$
5. Show that the multidimensional extension leads us to Equation 11.1 and 11.2.

11.4.3 Looking at the Learning Rate

In order to optimize the learning rate, it is possible to show that, if matrix H is the Hessian of our error function, then as long as α is bounded as shown:

$$0 < \alpha < \frac{1}{\lambda_{\max}}$$

then the Widrow-Hoff algorithm will converge. For us, we can show that the Hessian on the full error will be XX^T, if X is $n \times p$ (so a scaled covariance matrix). Since we're approximating the error with the single vector $\boldsymbol{x}$, then:

1. As an exercise, show that $\|\boldsymbol{x}\|^2$ is the non-zero eigenvalue for the rank 1 matrix $\boldsymbol{x}\boldsymbol{x}^T$.
2. Matlab uses the following estimate for the maximum learning rate:
$$\alpha = \frac{0.9999}{\texttt{max(eig(}\mathbf{x}\mathbf{x}^T\texttt{))}} = \frac{0.9999}{\|\boldsymbol{x}\|^2}$$

[1] Also goes by the names Least Mean Squares rule, and the delta rule.

11.4.4 Implementation Details

We'll provide Matlab/Python implementations separately, but let's consider the details using pseudo-code.

In our code, we'll assume that the data is in matrix X, the desired targets are in matrix T, and the training parameters are weight matrix W and vector $\mathbf{b}$. We'll assume that X is $n \times p$, and that sets up the dimensions for the other matrices and vectors:

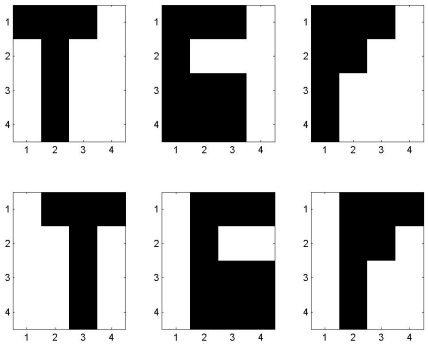
$$X \text{ is } n \times p \quad T \text{ is } m \times p \quad W \text{ is } m \times n \quad \mathbf{b} \text{ is } m \times 1$$

- Randomly initialize `W` and `b` (`randn` is better than `rand` for this).
- Loop for i from 1 to the number of epochs (an epoch is one pass through the data)
  - Randomly permute the training data.
  - For j from 1 to the number of points p:
    - Compute the output y using the j^{th} input.
    - Compute the error: $t - y$ for the j^{th} input.
    - Update the weights: $W_{\text{new}} = W_{\text{old}} + \alpha(t - y)x^T$
    - Update the b: $b_{\text{new}} = b_{\text{old}} + \alpha(t - y)$.
    - End of j loop.
  - Compute the SSE for this epoch:
    - `temp`$= T - (WX + \mathbf{b})$.
    - SSE= sum(temp. $*$ temp).

11.4.5 Example: Associative Memory

Here we will reproduce an experiment by Widrow and Hoff[2] who built an actual machine to do this (we'll do a computer simulation).

We'll have three letters as input, T, G and F. We'll associate these letters to the numbers $-60, 0, 60$ respectively. We want our network to perform the association using the Widrow-Hoff learning rule.
The letters will be defined by 4×4 arrays of numbers, where 1 corresponds to the color black, and -1 corresponds to the color white. In this example, we'll have two samples of each letter, as shown in the figure to the right.

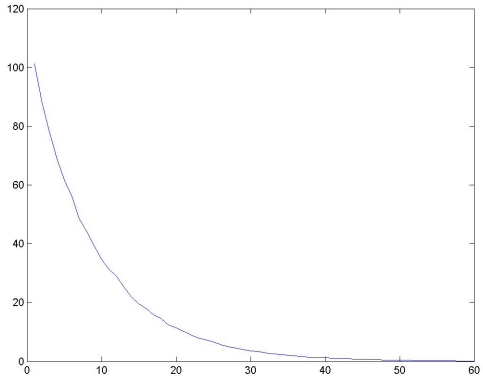


Implementation and problem specification:

- Dimensions:
$$X \text{ is } 16 \times 6, \qquad T \text{ is } 1 \times 6, \qquad W \text{ is } 1 \times 16, \qquad b \text{ is a scalar}$$
- We'll use an $\alpha = 0.03$.
- We'll take 60 passes through the data (60 training epochs).

[2] See "Adaptive Switching Circuits" by B. Widrow and M.E. Hoff, in 1960 IRE WESCON Convention Record, New York: IRE, Part 4, p. 96-104. You'll find reprints on the internet.

The plot of the error is shown in the figure to the right. The horizontal axis counts the number of passes through the data, and the vertical axis gives the sum of the squared errors. Since we only had 6 points, the error graph (taken on the full set) is just to show that the training is succeeding, it does not represent the error of the classifier.



11.5 One step training

In this section, we'll assume that all of the data is available to us at the beginning of training. Further, rather than affine form $(Wx+b)$, we'll convert the output to the linear form by creating $\hat{W}$ and $\hat{X}$. If you'll recall, we start with:

$$W\left[\boldsymbol{x}^{(1)},\ \boldsymbol{x}^{(2)},\ \ldots,\ \boldsymbol{x}^{(p)}\right] + [\boldsymbol{b},\ \boldsymbol{b},\ \ldots,\ \boldsymbol{b}] = \left[\boldsymbol{y}^{(1)},\ \boldsymbol{y}^{(2)},\ \ldots,\ \boldsymbol{y}^{(p)}\right],$$

and then we transform this into a linear problem by appending $\boldsymbol{b}$ to the last column of W and we put a row of 1's as the bottom row in the data:

$$\hat{W} = [W \quad \boldsymbol{b}\,], \qquad \hat{X} = \begin{bmatrix} \boldsymbol{x}^{(1)} & \boldsymbol{x}^{(2)} & \ldots & \boldsymbol{x}^{(p)} \\ 1 & 1 & & 1 \end{bmatrix}$$

Now the output of the linear network is simply $Y = \hat{W}\hat{X}$, and we can solve the system of equations for $\hat{W}$ directly

$$\hat{W}\hat{X} = T$$

In particular, we can use the pseudo-inverse from the **reduced** SVD of $\hat{X}$:

$$\hat{X} = U\Sigma V^T \quad \Rightarrow \quad \hat{W} = TV\Sigma^{-1}U^T$$

If $\hat{X}$ has rank k, be sure you can identify the dimensions of all the matrices above.

11.6 Small Batch Training

Between training using a single point at a time versus using all the data, we might update the parameters after gathering some small number of data points together. One advantage to this is that the direction of gradient descent will be better established than using a single point.

Furthermore, rather than completing training as with one-step learning, regular online learning keeps the network flexible and adaptable to new changes in the data.

In small batch training, the algorithm proceeds as with the single point training, except t, y, and x are replaced by their averages taken over the small group. Further, the value of α can be approximated by

$$\alpha \approx \frac{0.99}{\max(\text{eig}(XX^T))}$$

where X in this context is the small group of points $\boldsymbol{x}$ collected in a temporary matrix X.

11.7 Time Series and Linear Networks

In this section, we'll see that linear neural networks can play a role in signal processing. In that context, we're thinking of an incoming signal parameterized by t and sampled digitally to create a sequence of points. We'll start there.

Definitions

A **time series** is a sequence of real numbers. We denote a time series in the usual way:

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

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

$$\hat{x}_1 = \begin{pmatrix} x(1) \\ x(2) \\ \vdots \\ x(k) \end{pmatrix}, \quad \hat{x}_2 = \begin{pmatrix} x(2) \\ x(3) \\ \vdots \\ x(k+1) \end{pmatrix}, \quad \hat{x}_3 = \begin{pmatrix} x(3) \\ x(4) \\ \vdots \\ x(k+2) \end{pmatrix}, \ \ldots$$

This is also called a **lagged vector (lag k)**. We would say that the time series has been **embedded in** $\mathbb{R}^k$.

Consider the Matlab code that would embed the data into $\mathbb{R}^5$.

```
p=length(S);
X=zeros(k,p-k);
for j=1:k
    X(j,:)=S(j:p-(k+1-j));
end
T=S(k+1:p);
```

This code is the basis for the function `lagX.m` on our class website. The function creates our domain vectors as the columns of matrix X, and the set of targets (in this case, 1 dimensional) in vector T. From here, we can apply the neural network and train it. That is, our goal is to build a time series predictor. We're going to predict the value of x_6 based on x_1, x_2, x_3, x_4, x_5. Then we predict the value of x_7 based on x_2 through x_6, and so on. Actually, this kind of a function is called a filter.

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

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

In particular, 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 $\boldsymbol{x} = (x_{i-1}, x_{i-2}, \ldots, x_{i-k})^T$.

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

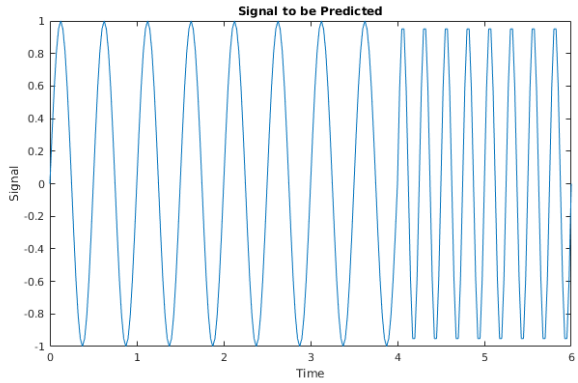
11.7.1 Example: Adapting to a Changing Signal

In this example, we build a time series from two time series- the signal is:

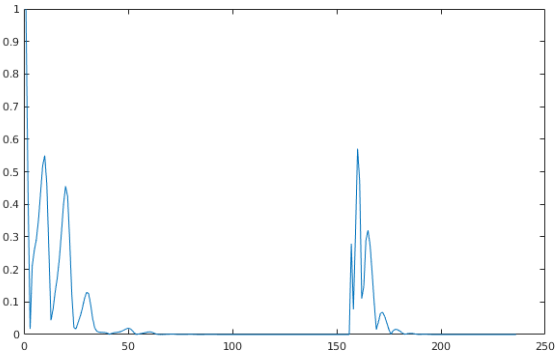
$$S(t) = \begin{cases} \sin(4\pi t) & \text{if } 0 \le t < 4 \\ \sin(8\pi t) & \text{if } 4 \le t \le 6 \end{cases}$$

The idea is that the online training will find the weights and biases for the first part of the signal quite rapidly. The hope is that, when the signal changes, the Widrow-Hoff algorithm will allow the function to rapidly adapt to the new signal (and it does!).

Below is the original signal.



The value of α in this example was fixed. We're doing that so we can see the result of changing the learning rate. For this example, we found that the optimal fixed value was approximately 0.153, but you might change that in the script for yourself. Below is shown the output error as a function of the time index. Indeed, we see that the error rapidly goes to zero, but then pops up and adapts as the signal changes.



Application: Noise removal

This idea is presented by Matlab. I haven't tried it myself yet, but it would be interesting to see what kinds of results you would get.

- 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.

Homework

The homework will consist of a computer lab where you'll practice training the linear neural network. Since the specific implementation will depend on the coding language used, we'll keep that as a separate document.