

# **Neural Networks in Electrical Engineering**

**Dr. Howard Silver**  
**School of Computer Sciences and Engineering**  
**Fairleigh Dickinson University**  
**Teaneck, NJ 07666**

*Session 1A - Electrical & Computer Engineering*

Neural networks are a subject thought by some to be a branch of artificial intelligence (AI). Although neural network research dates back about sixty years, it is generally considered a newer problem solving technique than AI. Whereas AI is applied in the form of “if-then” rules, neural networks attempt to model the structure of the human brain and are based on self-learning. The structure is highly parallel, resulting in the ability to self-organize to represent information and rapidly solve problems in real time. However, even with today’s high-speed computers, artificial neural networks are limited by the fact that they can only replicate a small fraction of the brain’s total structure.

Early work in neural networks was researched primarily by neuroscientists and psychologists, whose interests were in learning more about the brain. Soon after, mathematicians, physicists and computer scientists joined in. Engineers were attracted to the subject by recognizing its potential for solving problems for which other techniques, such as traditional computer programming, may not have been feasible or economical. Neural networks tend to do well at recognizing patterns within seemingly random data, as opposed to applications involving rules of logic. As a result, they have been applied to areas of business such as evaluation of loan applications and stock market pattern prediction, and to other areas such as handwriting and speech recognition and the detection of signals in a noisy environment.

This paper will present a brief overview of a simple two-layer neural network structure and a supervised learning algorithm called Perceptron. In supervised learning the network is trained to map a given set of input patterns to known outputs. After the patterns have been “learned” the network can be tested for patterns with errors present. With the aid of MATLAB programs created by the author, Perceptron learning is first applied to a problem of alphabetic character recognition, which relates to the application of handwriting analysis. The algorithm is then used to train a network to distinguish among square, triangular and sinusoidal waveforms. The network is then tested by, superimposing a given level of “random” noise on each waveform, and determining whether the “noisy” waveforms can still be distinguished from each other. A third example

replaces the waveforms by three sinusoids at separate frequencies. Learning and testing is again done in the presence of noise for decreasing separation of the waveform frequencies.

Neural networks often need to recognize patterns among information presented to it. Unsupervised learning is a type of training to find patterns for input data without attempting to map them to specific targets. A Kohonen Self Organizing Map (SOM) will be introduced and applied to the classical Traveling Salesman problem, which is analogous to the important problem of minimizing the wiring distance in a complex integrated circuit.

## I. Introduction to Neural Networks

The model used to simulate artificial neural networks is based on the biological nerve cell or neuron shown in Figure 1. Electrical signals arising from impulses from our receptor organs (e.g. eyes, ears) are carried into neurons on dendrites. Neuron outputs are transmitted along axon branches and join with outputs from other neurons at contact points called synapses. Each signal entering a synapse is "weighted" and summed with all the other inputs to that synapse. If the sum of the weighted inputs exceeds a particular threshold signal level for the neuron, the neuron is said to "fire".

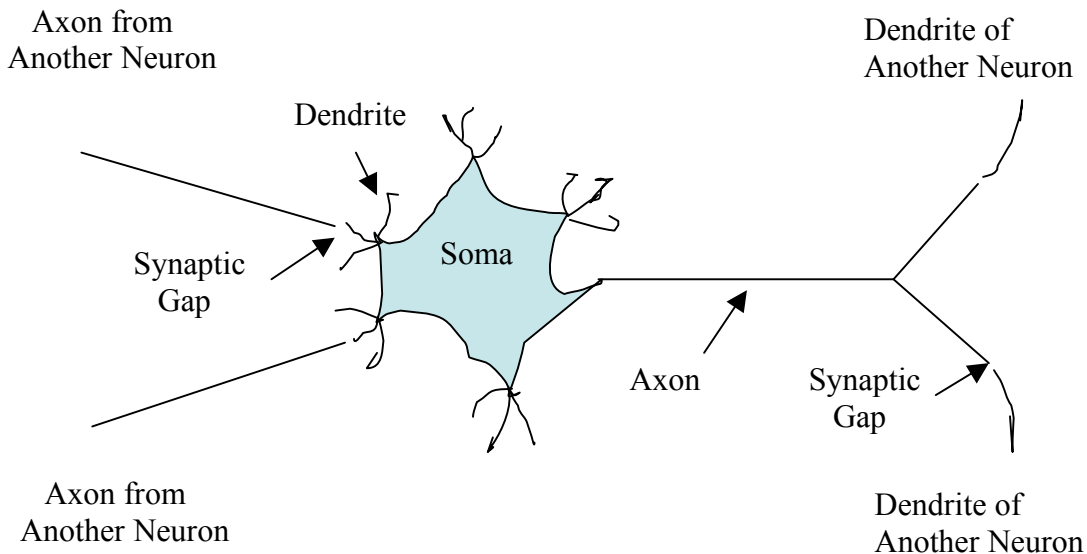
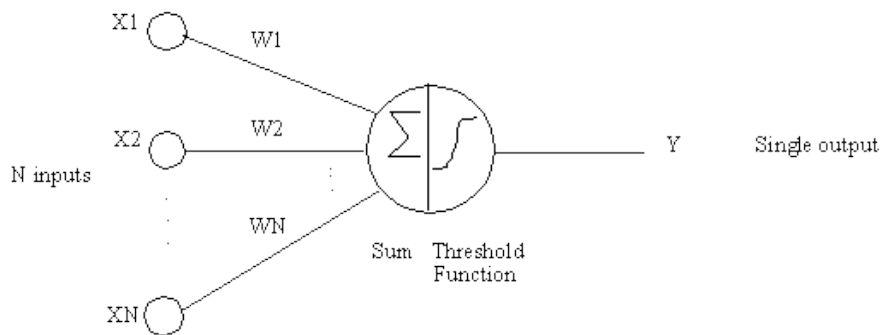


Figure 1 - Biological Neuron

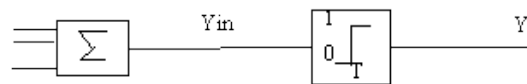
A simple mathematical representation of a neuron is shown in Figure 2. The notation to be used denotes the inputs as  $x$ , the weights as  $w$ , the summed output (pre-threshold) as  $y_{in}$ , and the final output as  $y$ . Subscripts follow these letters where necessary to

distinguish among multiple inputs, weights, etc. Thus, for the single neuron shown with N inputs, the inputs are denoted by  $x_1, x_2, \dots, x_N$  with corresponding weights  $w_1, w_2, \dots, w_N$ . The pre-threshold output  $y_{in}$  is a sum of each input multiplied by the weight on its line. The threshold function will arbitrarily produce binary outputs for  $y$  - i.e. zero if the sum is less than a defined threshold value  $\theta$ , or one if  $y_{in}$  is equal to or exceeds the threshold.

For supervised learning, neural networks are trained to map input patterns to desired outputs (i.e. target values). A modified threshold function, which proves more useful with various learning algorithms, produces bipolar (-1 or +1) rather than binary outputs. Also, an additional fixed input of +1 may be provided with the weight on the connecting line referred to as a bias value (b). This allows flexibility in setting of the threshold value  $\theta$ .



Summation and Threshold Operations (no hidden layer case)



$$Y_{in} = x_1 w_1 + x_2 w_2 + \dots + x_N w_N$$

$$Y = 0 \text{ if } Y_{in} < \theta(T)$$

$$Y = 1 \text{ if } Y_{in} \geq \theta(T) \quad (\text{Neuron "fires"})$$

Special Case: Hard limiting threshold function

Figure 2 – Mathematical Representation of a Single Neuron

The notation for a neural net structure for multiple output neurons is shown in Figure 3. The inputs are  $x_1, x_2, x_3, \dots$  as before. For the output neurons  $y_{in1}, y_{in2}, y_{in3}, \dots$  represent the pre-threshold values of output neurons 1, 2, 3, ... with final outputs  $y_1, y_2, y_3, \dots$  after the threshold function is applied. That is,  $y_j = f(y_{in_j})$  for each output  $j$ , where  $f$  is the threshold function. The weight on the connection from input neuron  $x_i$  to output neuron  $y_j$  is labeled  $w_{ij}$ , and the pre-threshold of the  $j$ th output neuron is the product of each  $x_i$  and  $w_{ij}$ , summed over all  $i$ . The objective is to match each output  $y_j$  to a target value  $t_j$ .

A more general neural network structure is shown in Figure 4. The layer of neurons between the input and output layers is called a hidden layer. The brain's structure is composed on many such layers. However, the examples to be shown in this paper will utilize the simpler structure without any hidden layers.

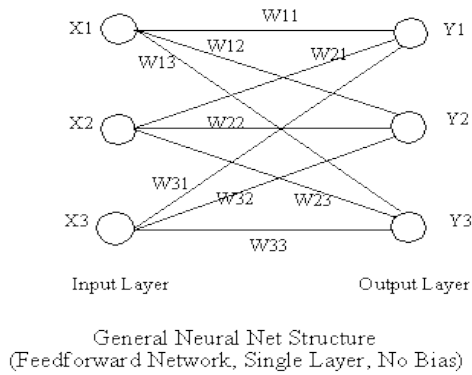


Figure 3 – Neural Network Structure (Input and Output Layers only)

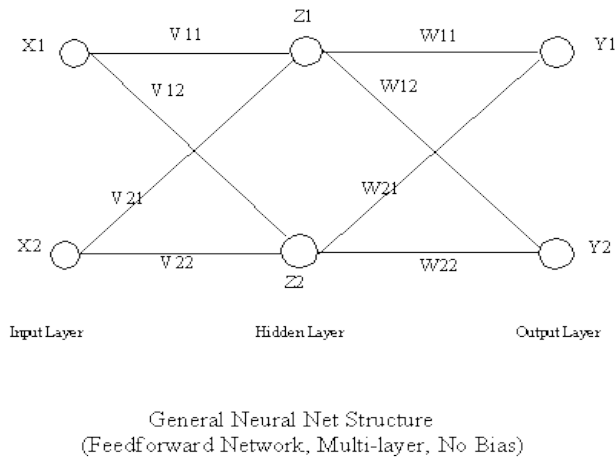


Figure 4 – Neural Network Structure (Hidden Layer included)

## I. Supervised Learning

### A. Introduction to Perceptron

Perceptron refers to a class of neural networks developed more than forty years ago. The learning rule associated with it is a modification of an earlier algorithm called the Hebb rule. A learning rule is a means of finding a suitable set of weights to match the input patterns to their target outputs. The process is an iterative one in which the sequence of patterns are presented to the network and the weights, initially set to zero or to random values, are updated according to the particular rule used. The patterns are repeated until the outputs match or are arbitrarily close to the target values. The number of iterations is referred to as training epochs.

The Perceptron algorithm is shown below. It assumes bipolar outputs and is based on the rationale that by adding to or subtracting from each weight the product of the input and target output values on the associated line, the outputs move in a direction toward their target values.

#### Steps in Applying

##### Perceptron:

- ! Initialize the weights and bias to 0.
- ! Set a learning rate  $\alpha$  ( $0 < \alpha \leq 1$ ) and threshold  $\theta$ .
- ! For each input pattern,
- ! \_\_ Compute for each output
- !  $y_{in} = b + x_1 * w_1 + x_2 * w_2 + x_3 * w_3 + \dots$
- ! and set
- ! \_\_  $y = 1$  for  $y_{in} < -\theta$
- ! \_\_  $y = 0$  for  $-\theta \leq y_{in} \leq \theta$
- ! \_\_  $y = 1$  for  $y_{in} > \theta$
- ! \_\_ If the  $j$ th output  $y_j$  is not equal to  $t_j$ , set
- ! \_\_  $w_{ij(new)} = w_{ij(old)} + \alpha * x_i * t_j$
- ! \_\_  $b_{j(new)} = b_{j(old)} + \alpha * t_j$
- ! (else no change in  $w_{ij}$  and  $b_j$ )

### B. Perceptron Applied to Character Recognition

The ability to recognize and associate character patterns makes neural nets useful in an area such as handwriting analysis. We will illustrate this with an example in which we train a network to learn a pattern for each of the 26 letters of the alphabet. To keep the number of inputs to a reasonable size, the characters will be upper case and defined by a 5x5 array of dots. A more practical size array for character display is 9x7, but this would necessitate 63 rather than 25 inputs. An alternative problem, to be illustrated later with unsupervised learning, would be to train the network to learn several patterns for each

character (e.g. different font styles) and fewer characters overall (to reduce the number of patterns to be learned).

For example, the letter "A" can be represented by the 5x5 dot array below - each position highlighted is shown graphically by the pound sign (#) for which the corresponding input value is +1. The non-highlighted positions, shown by a dot (.), have values 0 for binary or -1 for bipolar.

Character Inputs for "A" (binary display shown)

```
..#.. 00100
.#.#. 01010
#...# 10001
##### 11111
#...# 10001
```

The string of bits, from top to bottom row and left to right column becomes

```
'0010001010100011111110001'
```

corresponding to neural net inputs  $x_1$  to  $x_{25}$ . Each letter will of course have a unique input pattern. We will define 26 separate outputs, one identifying each letter in order A to Z. Thus, if the pattern above for "A" is presented, the binary target output string is '100000000000000000000000' ( $t_1$  to  $t_{26}$ ). Likewise, for "B" only the second output is high and the rest are 0's, etc.

The objective is to train our network to learn the 26 character patterns. If learning is successful, the network can be tested by entering input patterns "close to" a learned character, and see if the net can associate with that character.0

MATLAB [1,2] is a widely used matrix based equation solving program. A MATLAB program was written to apply Perceptron to train a neural network for this requirement. The output when the program is run is shown below. For only five training iterations (epochs) most of the character patterns have already been associated with the appropriate output line. All 26 alphabetic characters are learned after 17 epochs, resulting in a unit 26x26 matrix displayed. Note the algorithm is applied with a bipolar threshold function, after which the outputs are converted to binary for convenience in presentation.

A separate MATLAB program was also created to enable the testing of “error patterns”, to see how well they are mapped to the learned results. The program enables entry of an arbitrary 25-bit string in binary representing the test pattern. The string is converted to numeric bipolar form (i.e. each 0 replaced by -1) and the outputs are computed using the final weights from the training program.

Shown below are samples of test patterns representing the letter “A” with one or more errors. Only the test pattern outputs are displayed in this case, again in binary form. As an added feature, the program also numerically sorted the pre-threshold outputs (i.e.  $y_{in}$  values) with the letter associated with the strongest output shown first. Some insight can be gained into correlation between output strength and “closeness” of the error pattern to learned patterns. For the first pattern with a single error (upper left corner), only output “A” was activated. For two errors (upper left and right corners), the output “A” is still the strongest, but the “?” shown in place of “1” indicates that its pre-threshold input is right at the threshold value. The third pattern illustrates a problem neural networks have in classification. Although the character looks to the brain like “A”, it is actually a pattern with six errors in comparison with the learned pattern. The outputs show that several output lines associated with other letters are above threshold, but the line for “A” is not.

```
Enter a test pattern (1s and 0s in quotes)
'1010001010100011111110001'
Character entered:
#.#..
.#.#.
#...#
#####
#...#
```



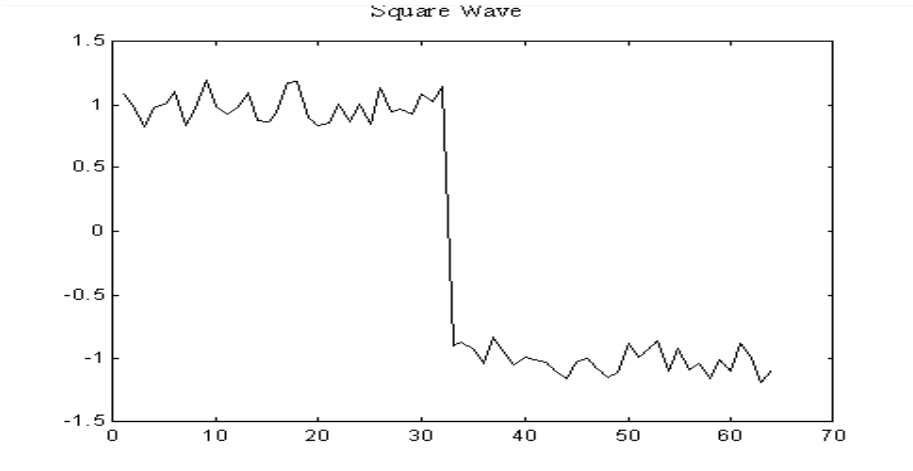


Figure 5(a) – Signal Classification with Perceptron (Square Wave)

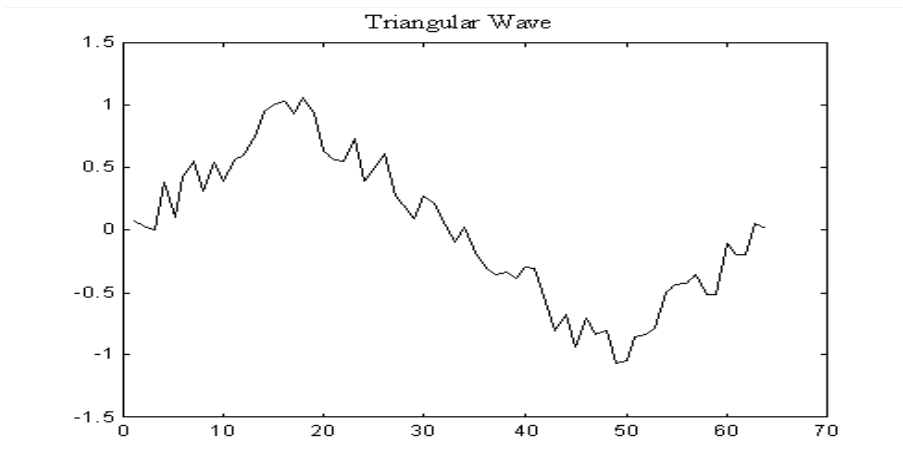


Figure 5(b) – Signal Classification with Perceptron (Triangular Wave)

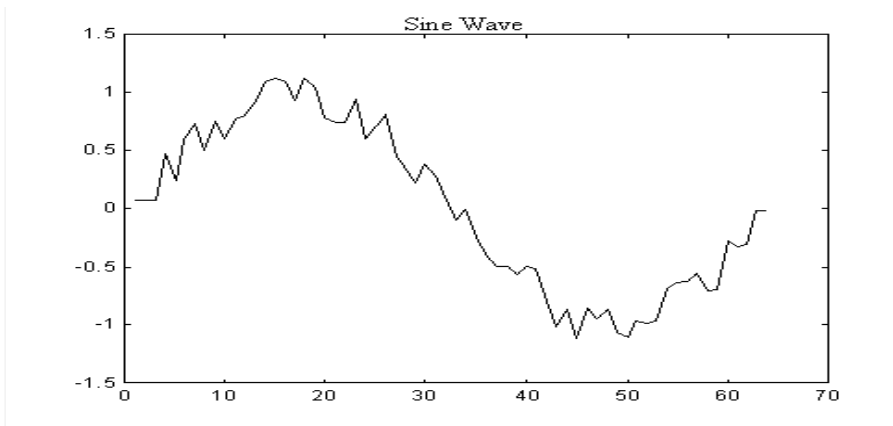


Figure 5(c) – Signal Classification with Perceptron (Sine Wave)

For this simulation each input signal is sampled an arbitrarily chosen 64 times over a single period. Each sampled value is then a separate input to the network. As with the alphabetic character example, a separate output is provided for each signal (square, triangular, and sine). However, the signals with noise are represented by inputs, which are no longer two-valued (binary or bipolar). Thus, our network has 64 multi-valued inputs and 3 bipolar outputs. Thus, a square wave input gives us the outputs 1, -1, -1; a triangular wave results in -1, 1, -1; a sine wave produces -1, -1, 1.

A MATLAB routine was written to train the network and then allow testing for arbitrary levels of noise. Portions of the output from several runs of the routine are shown below. The number of epochs was first increased until learning was complete. Then, the trained network was tested with decreasing signal to noise ratios, to see where the noise becomes too large to enable proper classification. Outputs from sample runs are shown below.

```
EDU>> nnet11h
Enter number of training epochs  20
Enter learning rate (□) 1
Enter threshold value (□) 0.2
Final outputs after training:
100
011
000
:
EDU>> nnet11h
Enter number of training epochs  30
Enter learning rate (□) 1
Enter threshold value (□) 0.2
Final outputs after training:
101
010
000
:
EDU>> nnet11h
Enter number of training epochs  40
Enter learning rate (□) 1
Enter threshold value (□) 0.2
Final outputs after training:
100
010
001
Enter signal to noise ratio 100
Classification of signals embedded in noise
100
010
001
:
Enter signal to noise ratio 10
Classification of signals embedded in noise
100
010
001
:
```

```

Enter signal to noise ratio 5
Classification of signals embedded in noise
100
010
000
:
Enter signal to noise ratio 2
Classification of signals embedded in noise
100
010
000
:
Enter signal to noise ratio 1
Classification of signals embedded in noise
100
010
010
:
Enter signal to noise ratio 1
Classification of signals embedded in noise
100
011
001

```

The desired output is of course the 3x3 identity matrix, matching the bipolar target matrix in the routine. The results show that the network has not fully learned the noiseless patterns after 20 and 30 epochs, but has achieved success after 40 iterations. After learning is complete the net is able to correctly classify the noisy signals for signal to noise ratios of 100 and 10, but has problems when this ratio is 5 or less.

Note that even for very large noise values the first row of the output matrix is correct in each case. This result is not unexpected since the square wave differs considerably from the other two waveforms and can therefore more easily be classified even with large noise superimposed. On the other hand, the triangular and sine waves are closer in appearance and are therefore more difficult to distinguish. The similarity is evident in Figures 5(b) and 5(c) with noise superimposed.

Note also that when the program is run a second time with the signal to noise ratio set to 1, different outputs result due to the randomness of the noise. Initializing the random number seed was not done in the program.

#### **D. Signal Frequency Separation using Perceptron**

A modified MATLAB signal classification program trains a neural network to classify three sinusoidal signals of the same amplitude and phase, and separated only in frequency. The middle frequency is  $\Delta$  percent above and the highest frequency  $2\Delta$  percent above the lowest frequency. As for the previous example, after training an attempt is made to associate noisy signals with the learned signals. The parameters

entered upon running the program are the percent frequency separation, the number of samples per period, the number of training epochs and the signal to noise ratio.

Figure 6 shows outputs of sample runs.

### Classification of Three Sinusoids of Different Frequency

```
>> nnet11i
Enter frequency separation in pct. (del) 100
Number of samples per cycle (xtot) 64
Enter number of training epochs (m) 100
```

```
Enter signal to noise ratio (SN) - zero to exit 1
```

Final outputs after training:

Classification of signals embedded in noise

```
100
010
001
```

```
100
010
001
```

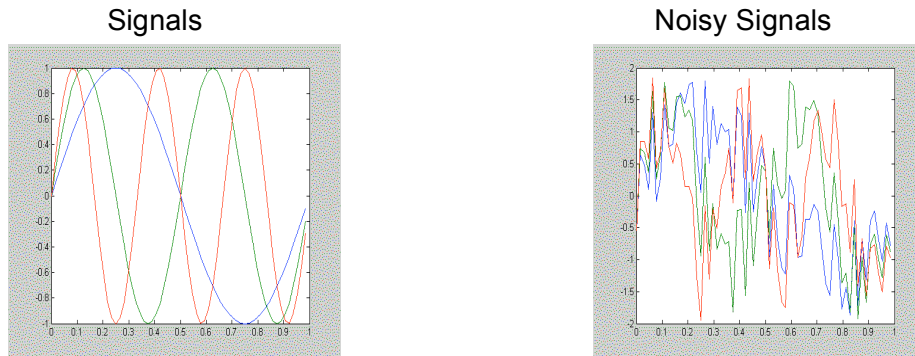


Figure 6(a) Signal Frequency Separation of 100 Percent

```
>> nnet11i
Enter frequency separation in pct. (del) 10
Number of samples per cycle (xtot) 64
Enter number of training epochs (m) 100
```

Final outputs after training:

```
100
010
001
```

Enter signal to noise ratio (SN) - zero to exit 10

Classification of signals embedded in noise

```
100
010
001
```

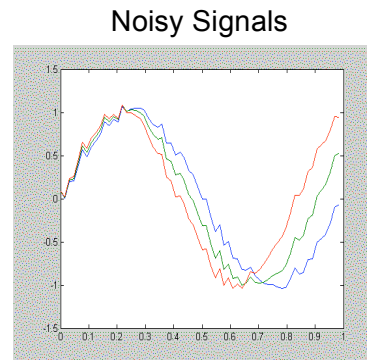
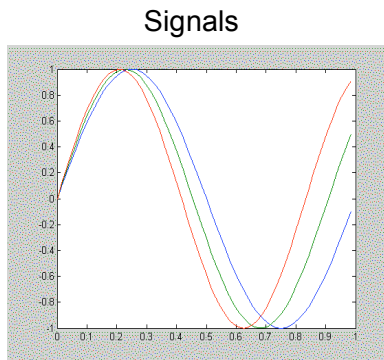


Figure 6(b) Signal Frequency Separation of 10 Percent

```
>> nnet11i
Enter frequency separation in pct. (del) 5
Number of samples per cycle (xtot) 64
Enter number of training epochs (m) 500
```

Final outputs after training:

```
100
010
001
```

Enter signal to noise ratio (SN) - zero to exit 10

Classification of signals embedded in noise

```
0?0
010
001
```

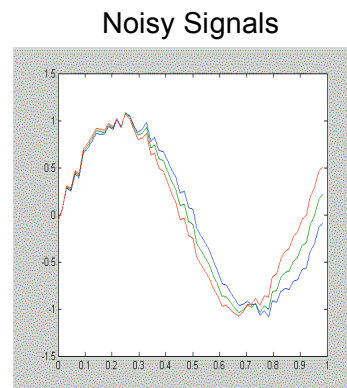
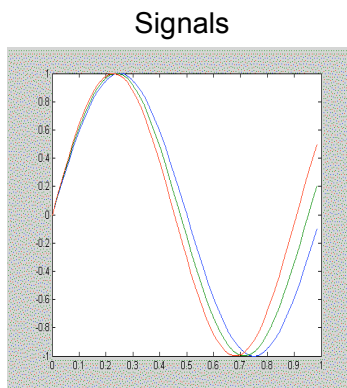


Figure 6(c) Signal Frequency Separation of 5 Percent

```
>> nnet11i
Enter frequency separation in pct. (del) 1
Number of samples per cycle (xtot) 1000
Enter number of training epochs (m) 10000
```

Final outputs after training:

```
100
010
001
```

```
Enter signal to noise ratio (SN) - zero to exit 100
```

Classification of signals embedded in noise

```
100
0?0
001
```

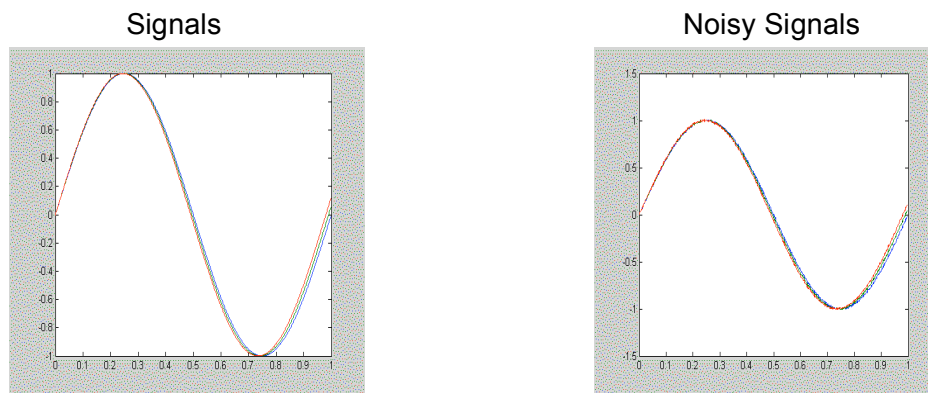


Figure 6(d) Signal Frequency Separation of 1 Percent

As expected, the results show that lower frequency separation requires more samples and/or iterations to train. Also, even if training is successful, proper classification of the noisy signals gets very difficult for smaller frequency separations. For frequency separations of 100% and 10%, training was successful after 100 epochs using 64 samples per cycle. As shown in Figures 6(a) and 6(b) proper classification was made for 100% separation with a signal to noise ratio of 1, and for 10% separation with a signal to noise ratio of 10.

Figure 6(c) shows that for a separation of 5% it was necessary to increase the number of epochs to 500 to train the network, but proper classification of only 2 of the 3 signals was made at a signal to noise ratio of 10. Finally Figure 6(d) shows that successful learning required many more samples per cycle (1000) as well as epochs (10000), but one of the 3 signals was still not classified even for a signal to noise ratio as high as 100. As was the case in the previous example, results are not necessarily repeatable due to the randomness of the noise.

### III. Unsupervised Learning

#### A. Introduction to Kohonen Self-Organizing Maps

The Kohonen algorithm is applied to unsupervised learning, where target values are not specified. The associated Self Organizing Map (SOM) is a topological structure made up of cluster units. The training algorithm also builds in "competition" among neurons. Learning is restricted to neurons that are either "winners" (or are neighboring units to "winners") of a competition relating to the closeness of weights to inputs. The SOM uses the competition among neurons to learn, in an unsupervised way, how to group input data into clusters. A cluster unit is somewhat analogous to an output corresponding to a group of input patterns, in a supervised learning situation. Interestingly, cluster units are a property observed in the brain.

Cluster units can be organized in either a one-dimensional or two-dimensional fashion. The one-dimensional unit is called a linear array whereas a two-dimensional unit can be arranged as either a rectangular or hexagonal grid. Illustrations of a part of a linear array and a rectangular grid are shown below. The "winning unit" of a particular competition is designated by "#" and neighboring units within a "radius" R are shown as "\*".

##### Linear array

```
      #           (R=0)
-----
    * # *       (R=1)
-----
  * * # * *     (R=2)
-----
* * * # * * *   (R=3)
```

##### Rectangular grid

```
      #           (R=0)
-----
    * * *
    * # *       (R=1)
    * * *
-----
  * * * * *
  * * * * *
  * * # * *     (R=2)
  * * * * *
  * * * * *
```

As we see, the value of R determines how many clustering units will learn. The total number of cluster units is chosen for a particular problem as the number of groups into which we want to place a given set of input patterns.

The Kohonen learning steps are as follows:

Initialize the weights (e.g. random values).

Set the neighborhood radius (R) and a learning rate ( $\alpha$ ).

Repeat the steps below until convergence or a maximum number of epochs is reached.

For each input pattern  $X = [x_1 \ x_2 \ x_3 \ \dots]$

Compute a "distance"

$$D(j) = (w_{1j} - x_1)^2 + (w_{2j} - x_2)^2 + (w_{3j} - x_3)^2 + \dots$$

for each cluster (i.e. all j), and find  $j_{min}$ , the value of j corresponding to the minimum  $D(j)$ .

If j is "in the neighborhood of"  $j_{min}$ ,

$$w_{ij}(new) = w_{ij}(old) + \alpha [x_i - w_{ij}(old)] \quad \text{for all } i.$$

Decrease  $\alpha$  (linearly or geometrically) and reduce R (at a specified rate) if  $R > 0$ .

The rule finds a total squared distance (i.e. separation) between inputs and weights for connections to each cluster (i.e. output). It then restricts the updating of weights to the "winner" (i.e. the one with minimum distance) and to any neighboring units if  $R > 0$ . Notice that the Kohonen approach also recommends a dynamic rather than a fixed learning rate as used with other learning algorithms. The process tends to match the weights to the inputs in a way which groups similar input patterns together. Notice that unlike Perceptron, this algorithm does not depend on target values being specified.

## **B. Self-Organizing Maps Applied to Pattern Recognition**

In an example taken from a text by L. Fausett [3] features seven letters (A, B, C, D, E, J, and K) defined by a  $9 \times 7$  array pattern, and each with three font styles. Our net therefore has 63 inputs and thus  $7 \times 3 = 21$  different patterns are to be grouped. The number of clusters chosen is arbitrary and may be fewer or greater than the number of patterns. For this example 25 clusters were specified.

A MATLAB routine was written to generate the Kohonen map for this requirement, assuming a linear (one-dimensional) array of cluster units. Randomizing of the weight matrix causes differences in clustering for different runs. But the results converge rapidly for the case where  $R=0$  (i.e. no topological structure) and more slowly for  $R=1$ . The character patterns and results of two sample runs are shown below.

## Character training set

```

..##... #####. ..###. #####. #####. ....### ##.##
..##... .#...# #...# #...# #...# #...# #...# #...#
..##... #...# #...# #...# #...# #...# #...# #...#
..##... #...# #...# #...# #...# #...# #...# #...#
..##... #####. #...# #...# #...# #...# #...# #...#
#####. #...# #...# #...# #...# #...# #...# #...#
#...# #...# #...# #...# #...# #...# #...# #...#
#...# #...# #...# #...# #...# #...# #...# #...#
###.### #####. ..###. #####. #####. ....### ##.##
    A1      B1      C1      D1      E1      J1      K1

```

```

..##... #####. ..###. #####. #####. ....# #...#
..##... #...# #...# #...# #...# #...# #...# #...#
..##... #...# #...# #...# #...# #...# #...# #...#
..##... #...# #...# #...# #...# #...# #...# #...#
..##... #####. #...# #...# #...# #...# #...# #...#
#####. #...# #...# #...# #...# #...# #...# #...#
#...# #...# #...# #...# #...# #...# #...# #...#
#...# #...# #...# #...# #...# #...# #...# #...#
#...# #...# #...# #...# #...# #...# #...# #...#
..##... #####. ..###. #####. #####. ....### ##.##
    A2      B2      C2      D2      E2      J2      K2

```

```

..##... #####. ..###. #####. #####. ....### ##.##
..##... #...# #...# #...# #...# #...# #...# #...#
..##... #...# #...# #...# #...# #...# #...# #...#
..##... #...# #...# #...# #...# #...# #...# #...#
..##... #...# #...# #...# #...# #...# #...# #...#
#####. #...# #...# #...# #...# #...# #...# #...#
#...# #...# #...# #...# #...# #...# #...# #...#
#...# #...# #...# #...# #...# #...# #...# #...#
#...# #...# #...# #...# #...# #...# #...# #...#
###.### #####. ..###. #####. #####. ....### ##.##
    A3      B3      C3      D3      E3      J3      K3

```

## Results of First Run

(Initial R=0, 10 epochs)

Unit	Patterns
2	B1, B3, D1, D3, E1, E3, K1, K3
11	A1, A2, A3
14	C1, C2, C3, J1, J2, J3
25	B2, D2, E2, K2

## Results of Second Run

(Initial R=1, 100 epochs)

Unit	Patterns
2	C1
4	C2, C3
6	J1, J2, J3
8	D1, D3
9	B1, B3
10	E1
11	E3
12	K1, K3
14	K1
16	D2
17	B2, E2
19	A1
20	A2
21	A3

Note that when only the winning unit is allowed to learn (i.e.  $R=0$ ) the patterns group themselves together into a small number of clusters. For example, for the case where  $R=0$  initially, A1, A2 and A3 are grouped together in unit 11, B1 and B3 are both in unit 2 while B2 is in unit 25.

For the case where initially  $R=1$ , where neighboring units are learning, the units are linked now topologically. That is, like patterns may wind up in adjacent rather than the same cluster. More cluster units are used as compared to the  $R=0$  case. However, we see a high degree of clustering in that the three fonts of letters A, C, J, and K are in the same unit or in units close together. Two of the three fonts of the other letters are close together as well, so the results are not too different than for the case where only the winning unit learns.

Another MATLAB routine was written which attempt to organize these same character patterns into a two-dimensional rectangular array of cluster units. Here the 25 cluster units are set up as a square 5x5 array.

A sample output of this program after 100 epochs is shown below.

Row:

```

3  3  4
1  5  1
5  5  5
1  5  1
1  5  1
3  3  3
1  4  1

```

Column:

```

2  2  1
4  3  4
1  5  5
5  4  5
3  2  3
4  4  5
2  2  2

```

The clustering is easier to observe from the two dimensional table below.

		Column				
		1	2	3	4	5
Row	1		K1, K3	E1, E3	B1, B3	D1, D3
	2					
	3		A1, A2		J1, J2	J3
	4	A3	K2			
	5	C1	E2	B2	D2	C2, C3

The results are similar to those for the linear array where  $R=1$  initially. The fonts for A and J are together or in a neighboring group, and C2 and C3 are still together (although C1 is not as close as one would expect). As with the linear arrays, B2, E2, D2 and K2 have wandered off from their counterparts, which are in the same cluster. An examination of the 7x9 patterns shows that these patterns differ in a relatively large number of bit positions from the other fonts in their respective groups. The different fonts for A, C, and J, however, are more closely matched.

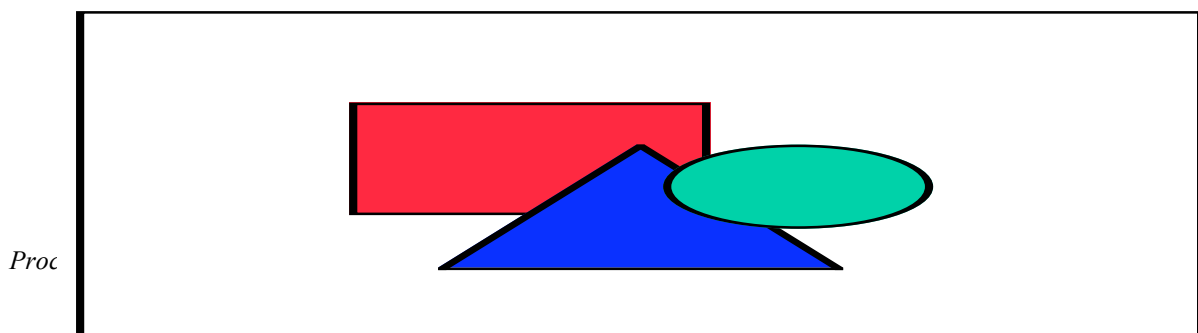
### C. Self-Organizing Maps Applied to the Traveling Salesman Problem

The Kohonen SOM is one of several neural net structures that have been used to solve an optimization problem with constraints. In the classical traveling salesman problem city locations are defined by their two dimensional coordinates. The objective is to find a minimum distance route enabling the salesman to visit each city once and return to the starting point. Any city can serve as the start of the route. This problem has an important analogy in the design of mass produced integrated circuits, namely that of finding an optimum wiring path to connect a large number of nodes together. Without the use of neural nets, computer methods involving exhaustive searches among all path combinations become impractical for a large number of nodes.

The SOM algorithm lends itself very nicely to the traveling salesman problem. Only two input neurons are needed - one each for the x and y coordinate numbers for the cities. Each city's coordinates is then a separate pattern to be learned, and the number of clusters specified is matched to the number of patterns (i.e. cities). Since there are only two input neurons, the distance function associated with the  $j$ th cluster is

$$D(j) = (w_{1j} - x_1)^2 + (w_{2j} - x_2)^2$$

Recalling that the updating of weights serves to move their values toward the inputs, the SOM will attempt to organize the training patterns into individual groups as the process converges. If successful, each column of the weight matrix (i.e.  $w_{1j}$  and  $w_{2j}$  for the  $j$ th row) will approach the coordinates of a single city, rather than an average over a group of cities. The distance  $D(j)$  then truly represents the square of the physical distance between two cities. Although it is not obvious, the clusters will be organized so that the ordering represents the optimum path. We will illustrate this concept first with a simple example for which 4 cities are on the vertices of a square, as illustrated below. Bipolar coordinates will be assumed for simplicity.



The coordinates of the cities (identified by asterisks) are as follows:

- City A: -1, -1
- City B: -1, 1
- City C: 1, -1
- City D: 1, 1

It is obvious that a minimum distance route goes around the outside of the square formed by the cities, for a total distance of  $4(2) = 8$  units. Optimum routes, starting from A for example, would be ABDC or ACDB and return to A. Going diagonally from A to D or from B to C would lead to a greater total distance.

The MATLAB routine for this example allows cluster units near the winning unit to learn (i.e. for  $R > 0$ ), and accounts for the fact that the salesman must return to the starting city by considering the first and last cluster unit adjacent. When this routine is run for the example of 4 cities on the vertices of a square, the following weight matrices resulted from three different runs:

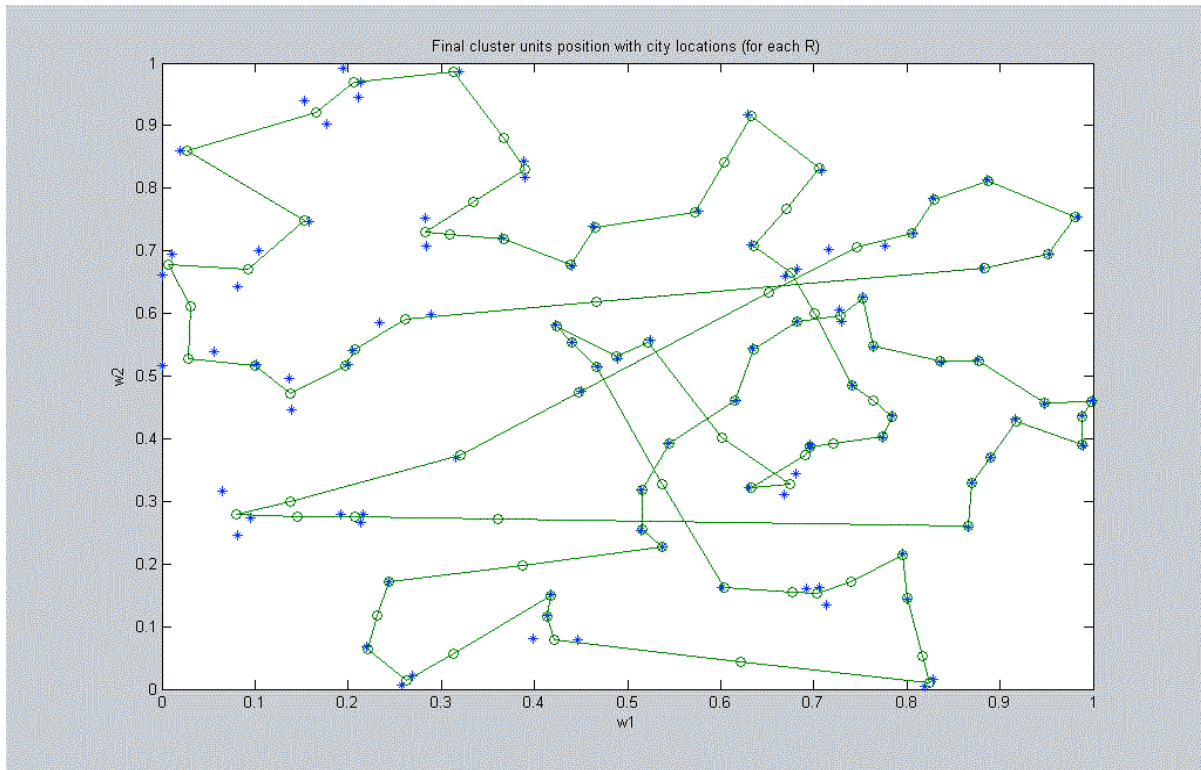
$$W = \begin{bmatrix} 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

$$W = \begin{bmatrix} 1 & -1 & -1 & 1 \\ 1 & 1 & -1 & -1 \end{bmatrix}$$

$$W = \begin{bmatrix} -1 & 1 & 1 & -1 \\ -1 & -1 & 1 & 1 \end{bmatrix}$$

The ordering of the columns of each weight matrix match the input vectors DCAB, DBAC, and ACDB respectively. Each of these orders matches to a minimum path around the edges of the square. There are 8 possible solutions, including 4 clockwise and 4 counterclockwise paths starting from any city. All these solutions should come up upon repeated runs of the program, again due to the initial randomness of the weights.

Minor changes in the MATLAB routine were made to select random coordinates for an arbitrary number of cities. The graphical result of a sample run for 100 cities is shown in Figure 7. Each city location is denoted by "\*" and the final weights which determine the path are shown by "o". The path appears to be a near optimum solution to achieving minimum overall distance.



$0.1 < \alpha < 0.25$ , 100 epochs, Initial R = 2

Figure 7 – Near Optimum Path for 100 Randomly Located Cities

## References

- [1] William Palm III, “*Introduction to MATLAB 7 for Engineers*”, McGraw Hill, 2005.
- [2] Delores Etter and David Kuncicky, “*Introduction to MATLAB 7*”, Pearson Prentice Hall, 2005.
- [3] Laurene Fausett; *Fundamentals of Neural Networks*; Prentice Hall; 1994.

## Biography

Dr. HOWARD SILVER is currently Professor of Electrical Engineering and Deputy Director of the School of Computer Sciences and Engineering at Fairleigh Dickinson University. Dr. Silver has co-authored textbooks on 32-bit Microprocessors and Computer Based Instrumentation. His current areas of interests are neural networks, fuzzy logic and computer simulation of engineering problems.

Address: 1000 River Road; Teaneck, NJ 07666

Telephone: (201) 692-2830

Email: silver@fdu.edu