NEURAL NETWORKS

Hopfield Neural Networks

- Effectively, Hopfield Neural Networks are like **associative memory**:
- Wish to store **p** patterns ζ^{μ}_{i} so that given a new input pattern ζ_i, the network responds by producing the stored pattern most closely resembling ζ_i.
- Stored patterns are indexed by superscript μ=**1**,⋯,**p** while the nodes in the neural network are labeled **i = 1**,⋯, **n**.
- Since knowledge is encoded in the network during design and not learnt, this type of network uses **unsupervised learning**.

Example (1 of 4)

- Consider designing a neural network to classify the hand written digits 0 through 9 that are presented on a 16x16 grid of pixels.

How would….

- A feed-forward Neural Network using back-propagation be designed and operate?

- A Hopfield Neural Network be designed and operate?

Example (2 of 4) – A Back-Propagation Network

Back-propagation Neural Network might be designed and operate as follows:

- Consist of 16x16 input nodes. (256, one for each pixel)
- Consist of 10 output nodes. (one for each digit to recognize)
- Consist of some number of hidden nodes.

- Require training using **sample digits** to adapt/adjust the weights (**learning**).

- Process new digits after training.

- Given a set of 16x16 input pixels, 1 of 10 outputs would become active, indicating the classification of the input digit.

This is classification using a **functional representation** (i.e., there is a non-linear function encoded in the network mapping 16x16 inputs to 10 outputs).

Example (3 of 4) – A Hopfield Network

A Hopfield Neural Network might be designed and operate as follows:

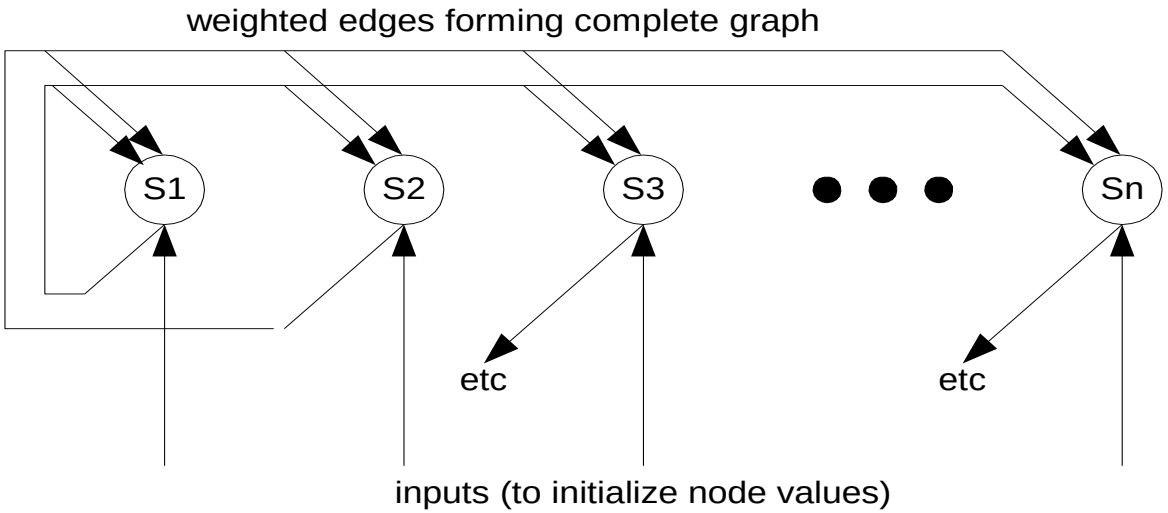
- A 16x16 array of nodes is created.
- The array of nodes is fully connected (edges between all pairs of nodes).
- **Each node is an input and an output. (one layer of neurons only!)**
- Edge weights are determined **a priori based on ideal patterns** for digits 0 through 9; patterns are stored in the network.
  - Continued, next page…

Example (4 of 4) – A Hopfield Network

- Given an unknown input, we initialize the output of each node (i.e., the output at time t=0).
- The network is allowed to **evolve over time until the node outputs become stable**.
  - Will not always stabilize.
- The final output values will (hopefully) correspond to one of the memorized patterns of ideal digits we stored in our network.
  - When might it not? more later …
- This is classification using an **associative representation** in that the network tries to find the **memorized pattern** most closely representing the input.

Hopfield Neural Network Topology

- Basically, a complete weighted graph:



- Each node has an activation function (assume sign function so outputs are -1 or +1):

$$S_i(t) = sign\left(\sum_{j=1}^{N} w_{ij} S_j(t-1)\right)$$

- **Note**: outputs are a **function of time**. The value at time t depends on values at time (t-1).

Algorithm (for "running" the network)

1. Assume network weights are determined and input pattern ζ given.

2. Set $S_i(0) = \zeta_i$ /* initialize network. */

3. Set $t = 1$.

4. Compute $S_i(t) = sign(\sum_{j=1}^{N} w_{ij} S_j(t-1))$.

5. If $S_i(t) == S_i(t-1)$ then STOP and GOTO step 6; otherwise $t = t+1$ and GOTO step 4.

6. The pattern most resembling the input is now available (as the output of the nodes)

Storing Single Patterns (1 of 2) (Stability)

- Storing a pattern is equivalent to asking "how do we pick network weights?"
- Assume we wish to store a pattern:

$$\zeta \text{ with bits } \zeta_i = \pm 1 \quad \forall i = 1, \cdots, N$$

- **Observation: If we were to present the stored pattern as input to the network (i.e., hit it exactly), the outputs of the network should not change!**
  - Since the input equals the desired output!
- We have a stability condition given by:

$$\zeta_i = sign(\sum_{j=1}^{N} w_{ij}\zeta_j) \text{ since } S_i = \zeta_i \text{ for all time}$$

Storing Single Patterns (2 of 2) (Weight Selection)

- Consider the following selection for weights:

$$w_{ij} = \frac{\zeta_i \zeta_j}{N}$$

- If we substitute these weights into the stability equation we get:

$$sign(\sum_{j=1}^{N} w_{ij}\zeta_j) = sign(\sum_{j=1}^{N} \frac{\zeta_i \zeta_j}{N}\zeta_j) = sign(\zeta_i \sum_{j=1}^{N} \frac{\zeta_j^2}{N}) = sign(\zeta_i) = \zeta_i$$

- So, the output will not change if the input pattern is exact.

Storing Multiple Patterns

- How can we store p patterns? i.e., we have patterns

$$\zeta_1, \zeta_2, \cdots, \zeta_p$$

- Solution: pick weights as a superposition of the stored patterns:

$$w_{ij} = \frac{1}{N} \sum_{k=1}^{p} \zeta_i^k \zeta_j^k$$

- In other words, compute the ideal weight for each pattern stored individually, and then average them out to pick the final, single weight for each edge.

Limitations of Hopfield Networks

- Severely limited in the number of patterns p that can be stored reliably in N nodes.
- Generally, the maximum number of patterns must be below 0.15N for reliable performance;
  - Avalanche in error occurs above 0.138N.
- Too many patterns will result in spurious outputs; i.e., outputs not corresponding to any stored pattern.
- Storing similar patterns can cause errors in output.

Energy Functions

- There is a relationship between dynamics (convergence to a stored piece of information) of Hopfield Network and an **Energy Function**.
- Consider the function:

$$H = -\frac{1}{2}\sum_{i,j=1}^{N} w_{ij} S_i S_j$$

- Consider that our **weights are picked** such that the **stored patterns represent local minimums** of the function H.
  - **Matching an input to a stored pattern is like minimization of H**; "fall into the closest minimum".
- The stability condition is akin to being stuck in the local minimum.

Optimization Problems

- The usefulness of the energy function is that it makes neural networks potentially useful for optimization problems.
- Say we have a minimization problem…
- If we can formulate our minimization problem in the form of the energy function, then an algorithm for minimization is the Hopfield algorithm.
- Hopefully, running the Hopfield Neural Network algorithm will converge to a local minimum, and hopefully a good solution.

Example Optimization Problem (1 of 3) – Graph Bisection

- Consider graph bisection: divide two sets of graph nodes into two blocks while trying to minimize the interconnection (edges) between the blocks.

- Let:

$$S_i = \begin{cases} -1 & \text{if in block 0} \\ +1 & \text{otherwise (in block1)} \end{cases}$$

$$C_{ij} = \begin{cases} 1 & \text{if edge between i and j} \\ 0 & \text{otherwise} \end{cases}$$

- Goal is to solve the problem:

$$\min L = -\sum_{i,j} C_{ij} S_i S_j : s.t. \sum_i S_i = 0$$

Example Optimization Problem (2 of 3) – Graph Bisection

- Consider the following Energy Function (our problem, but with balance constraint as a penalty):

$$H = -\sum_{i,j} C_{ij} S_i S_j + \mu(\sum_i S_i)^2 \quad \mu > 0$$

- Constraints are now "soft" but penalized with a positive scalar to "help" enforce an equal balance of nodes into partitions.

- We can rearrange to make the above equation look like:

$$H = d - \sum_{i,j} w_{ij} S_i S_j, \quad \text{d constant}$$

Example Optimization Problem (3 of 3) – Graph Bisection

$$H = d - \sum_{i,j} w_{ij} S_i S_j, \quad \text{d constant}$$

- Ignoring the constant, we have an Energy Function that can be minimized using a Hopfield Network.
- **The weights of the network are a function of the problem being solved**.
- For bi-sectioning, weights depend on connectivity of graph, and magnitude of the balance penalty.

Kohonen Self-Organizing Maps (SOM)

- Self-organizing maps (SOM) are neural networks that organize themselves (learn) based on similarity to training data.
  - Also called Kohonen maps after their inventor, Tuevo Kohonen.

- Unlike perceptrons, a SOM is not trained using known input-output pairs (supervised learning).

- Trained using only inputs (unsupervised learning)

  - Called unsupervised because we have no knowledge of "correct" outputs.

SOM Concept

- The idea behind SOMs is to discover patterns in the input data
  - Cf., perceptrons, which detect known patterns.
- Achieves this by grouping similar inputs together.
  - Neurons are normally spatially arranged, so that discovered patterns and groups are clearly visible.
- When a new input is presented to the network (after training), the most similar value in the network is returned.

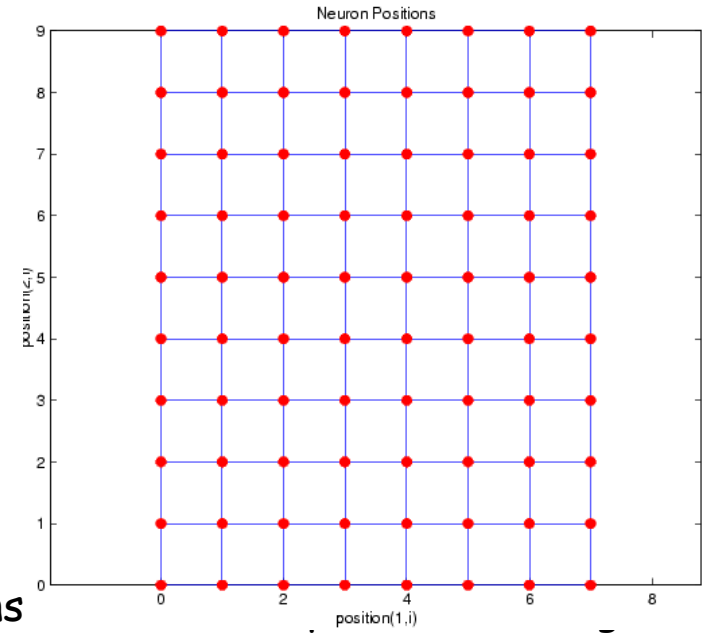
Uses and Applications

- Common uses are:
  - Finding similarities in input data.
  - Dimensionality reduction; n-dimension input space → 2 or 3-dimensional output.
    - Allows for visualization of high-dimensional data.

- Applications include:
  - Pattern recognition and signal processing (e.g., speech processing – phoneme maps; grouping similar sounds together)
  - Data mining (e.g., text mining – grouping similar documents)
  - Optimization
  - Image Analysis and Vision

SOM Structure

- A typical SOM has spatially arranged neurons (a neuron map)
  - e.g., a 2-D SOM with a grid structure
    - 10x8 grid
  - Also hexagonal, random layouts



- Each neuron has:
  - a weight; a vector of values equal in dimens can change).
  - a position in the map (position does not change).

Training in a SOM*

- Neurons compete for the right to represent the input sample
  - Competitive learning environment
  - Neuron is chosen whose weight vector is most similar to the input vector.

- Similarity is typically computed using Euclidean distance:

$$d_j = \sqrt{\sum_i \left(w_{j,i} - x_i\right)^2}$$

Where:

- $w_{j,i}$ = i^{th} element of weight vector for neuron j.
- x_i = i^{th} element of input vector x.
- d_j = similarity of neuron j to input vector x

Updating weight vectors

- The "winning" neuron is "rewarded" (updated) by becoming more like the input sample.
- Neighboring neurons are also rewarded (helps group similar inputs together).

$$\mathbf{w}_j(t+1) = \mathbf{w}_j(t) + \left(\mathbf{x} - \mathbf{w}_j(t)\right) \cdot \theta(t, r) \cdot \alpha(t)$$

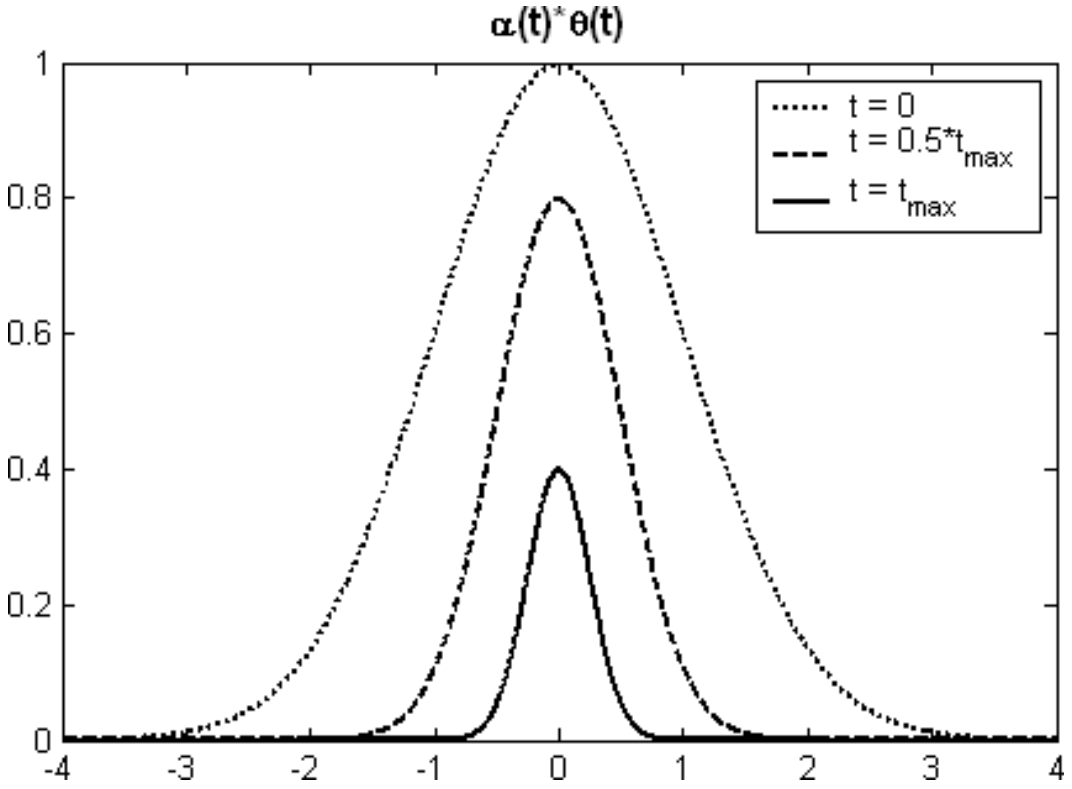
Where:

- t = current training iteration
- θ = neighborhood function
- α = time-dependent learning function
- $\mathbf{w}_j$ = weight vector of neuron j

Neighborhood function

- Neighborhood fn. will reward more those neurons physically closest to the winning neuron.
  - Decreases with distance; e.g., a Gaussian fn.
- May become narrower with time.
  - More neurons updated early in training; fewer as the network is exposed to more training data.
  - cf. simulated annealing, genetic algorithms.
- Magnitude may decrease with time (effect of $\alpha(t)$ fn.).
  - $\alpha(t)$ decreases with time; neurons are not rewarded as much as training progresses.

Neighborhood function example*



- Neighborhood function becomes narrower with time.
- Learning weight $\alpha(t)$ becomes smaller with time.
- Overall result: as the number of iterations increases:
  - fewer neighbors are updated.
  - the degree of update is reduced.

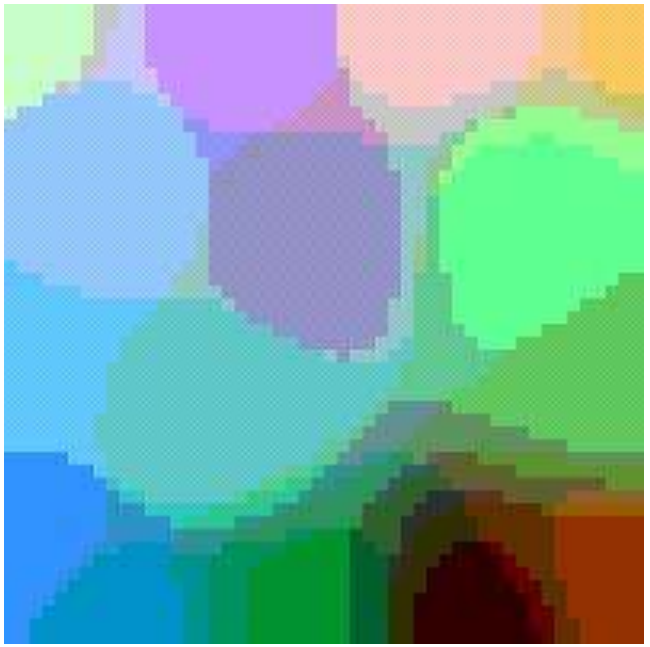
Pseudo-code for SOM Training

```
1.  SOM_init(); // initialize weight vectors  (random or otherwise)

2.  // train until the max # of iterations is reached.
3.  for( t = 0; t < t_max; t++ ) {

4.      x_i = selectRandTrainingSample(); //select a random training sample
5.      n_j = findBestMatch( x_i ); // find the neuron most similar to x_i

6.      // reward neurons close to winner n_j
7.      rewardNeurons( n_j, theta, alpha );

8.  }
```

- **NOTE**: A large number of training iterations are typically required; expose network to the training set as much as 1,000 times. (1,000 epochs)

Example – Color Grouping

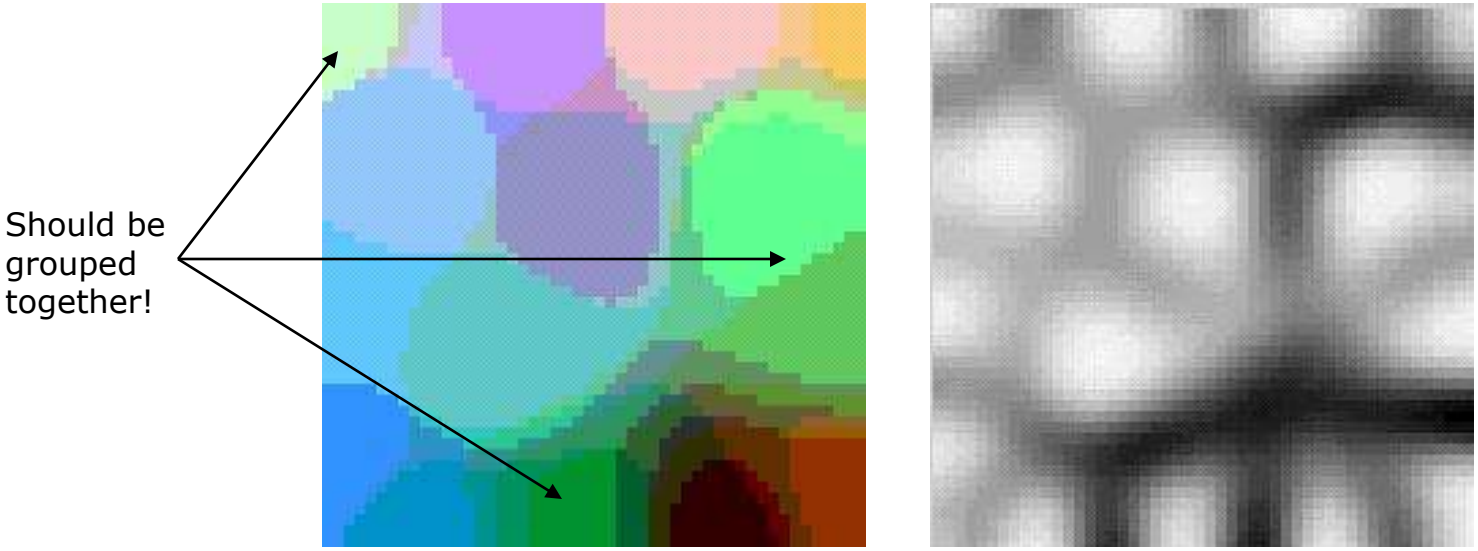
- Given: a set of colors in {R,G,B} format (3 dimensions)
- Goal: arrange them spatially (in 2 dimensions) such that similar colors are grouped together.
  - Essentially performing dimensionality reduction (from 3 dimensions to 2).
- Each pixel in the figure represents a neuron; the weight of each neuron is a {R,G,B} value, shown by its color.
  - Easy example to visualize!



Quality of SOM Result

- We expect similar data to be close together in the final mapping.
  - E.g., Color mapping: since purple is blue + red, would expect to pass through purple region when moving from red to blue.
  - Since the network is unsupervised, this may not always be the case. (see figure, previous page).
- Can check the quality of SOM by examining the similarity between the weights of neighboring neurons
  - use Euclidean distance again.
- Result can be color-coded for visualization in a "similarity map"
  - If the average similarity of a neuron to its neighbors is low (i.e., avg. dist. is large), assign a dark color.
  - If the average similarity is high (i.e., avg. dist. is small), assign a light color.

SOM Quality illustrated



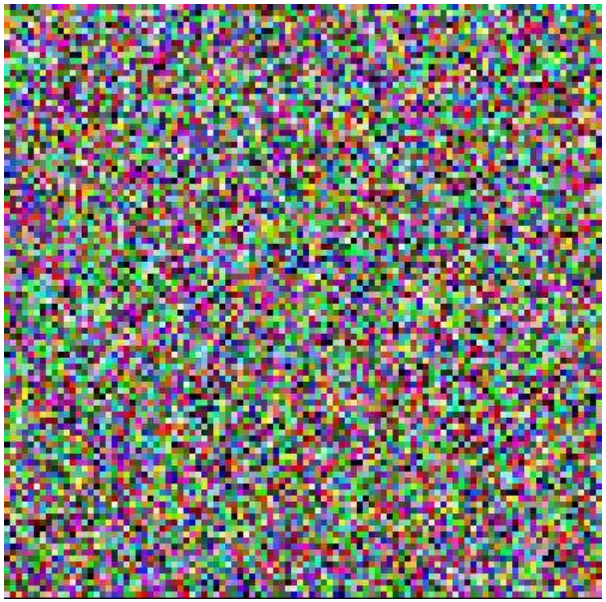
- A mapping with similar values arranged next to each other will have a "similarity map" dominated by light grays and whites (smooth transitions).
- Dark gray/black valleys in the "similarity map" (as above) indicate dissimilar neighboring regions
  - In this case, red and green are not similar, nor are black and green. Also, many green regions are separated from each other.

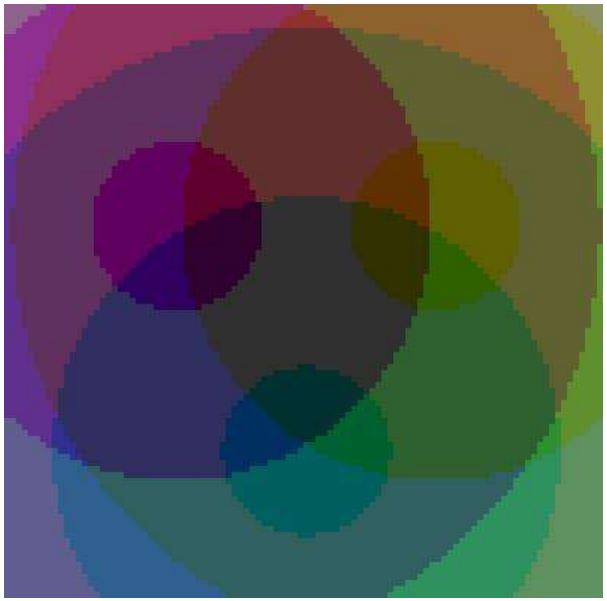
Improving SOM quality

How to improve SOM quality?

- May need to generate multiple maps separately, then merge them to get one good final map!
  - How do we choose which parts of each map to keep? Difficult problem … (will not discuss further)
- Alternatively, use a more intelligent initialization scheme for network weights (i.e., not random)
  - Use some prior knowledge (if available).
  - This can also reduce the number of iterations (epochs) required to get a good map!

Initializing Network Weights





- Random vs. intelligent weight initialization
  - Left: Random initial values
  - Right: Intelligent initial values for color mapping problem
  - Intelligent approach should keep similar colors next to each other as the map evolves!

Advantages/Disadvantages of SOM

Advantages

- Conceptually simple, easy to train.
- Generally give good results.

Disadvantages

- Need full data vectors for each training sample (not always available).
- Similar samples are not always grouped next to each other.
  - May need to merge multiple maps and/or add prior knowledge.
- **Computationally expensive!**
  - Especially a problem as the number of input dimensions increases.

Radial Basis Function (RBF) Networks

- Differ from multi-layer perceptrons in the type of neurons used in the hidden layer.
  - RBF neurons incorporate the idea of distance from a center point into the activation fn. (hence "radial").
  - Center point for each neuron represents its weight.
    - **Note**: center point has the same number of dimensions as there are network inputs.
  - Typically use Euclidean distance: $d_j = \sqrt{\sum_i \left(x_i - c_{i,j}\right)^2}$
    - $c_{i,j}$ = center point for i^{th} input of hidden neuron j.
    - x_i = i^{th} network input
    - d_j = distance from input **x** to center of j^{th} neuron.
  - Cf. perceptron weights which are multiplied by the inputs.

Activation function "width"

- RBF activation fn.'s also incorporate the idea of width or spread.
  - Idea is to place higher emphasis on inputs closer to the center point of the neuron; lower emphasis as this distance increases.
  - Gaussian fn. is the most popular for this purpose, and the only one we will consider.

$$h(\mathbf{x}) = \exp\left[-\frac{\|\mathbf{x}-\mathbf{c}\|^2}{r^2}\right]$$

  - h(**x**) = the neuron output.

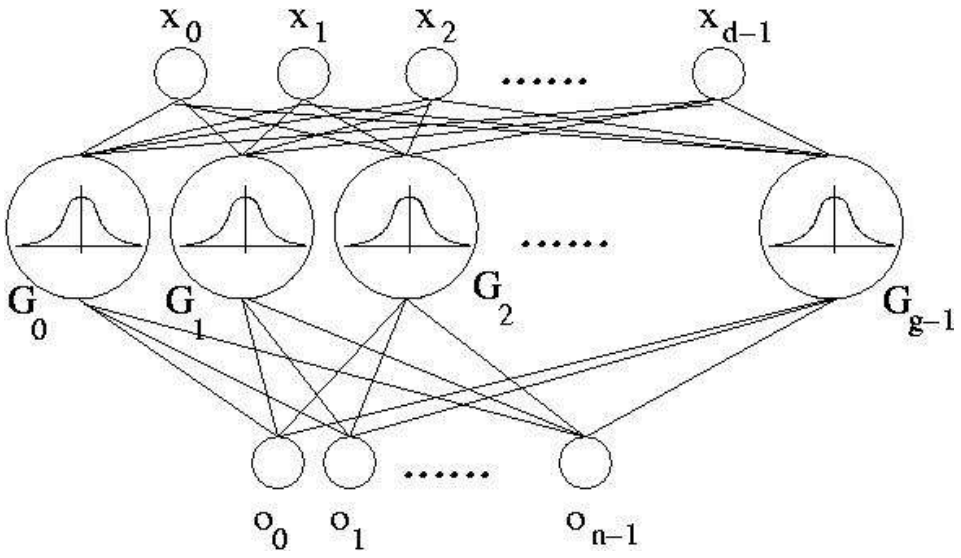
RBF Network Architecture

- RBF networks typically use a 3 layer architecture:

  1. Input layer.
  2. Hidden layer with radial activation functions.
  3. Output layer.

- Output layer typically uses neurons with linear activation:

$$O_i = \sum_j W_{j,i} A_j$$



Note: Gaussians shown within neurons identify a RBF network.

Network Architecture Comments

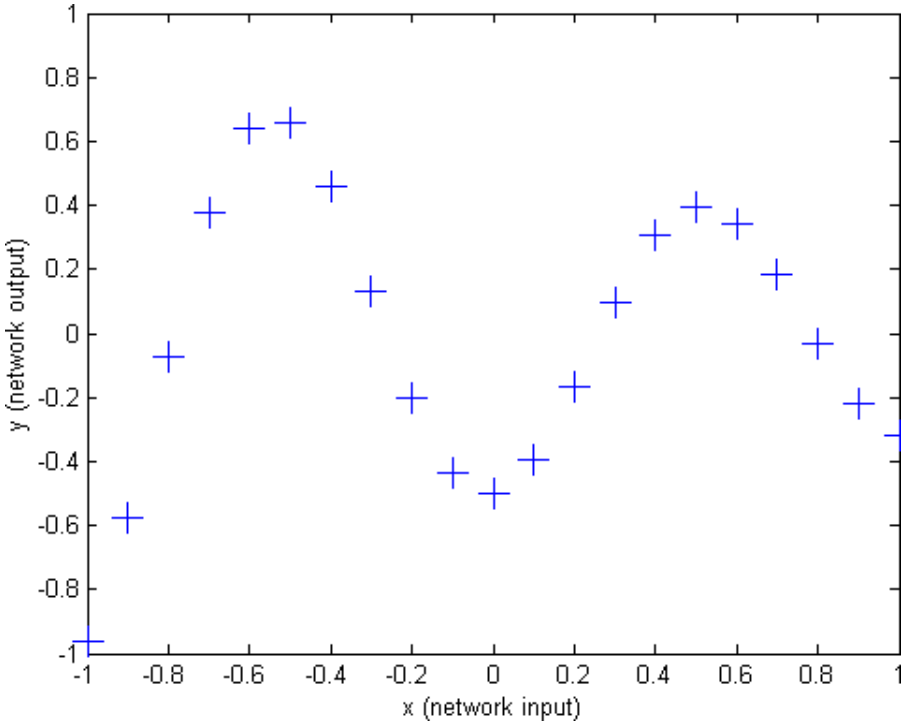
- This architecture is particularly used for regression applications
  - i.e., function approximation
  - Linear output nodes allow the network to model functions as linear combinations of Gaussians with different centers.
- Sigmoidal activation fn.'s at the output layer are sometimes used for classification applications.
  - Sigmoid reduces the output to the domain [0,1] to express the input's similarity to a known class.

Example – Regression (1 of 4)

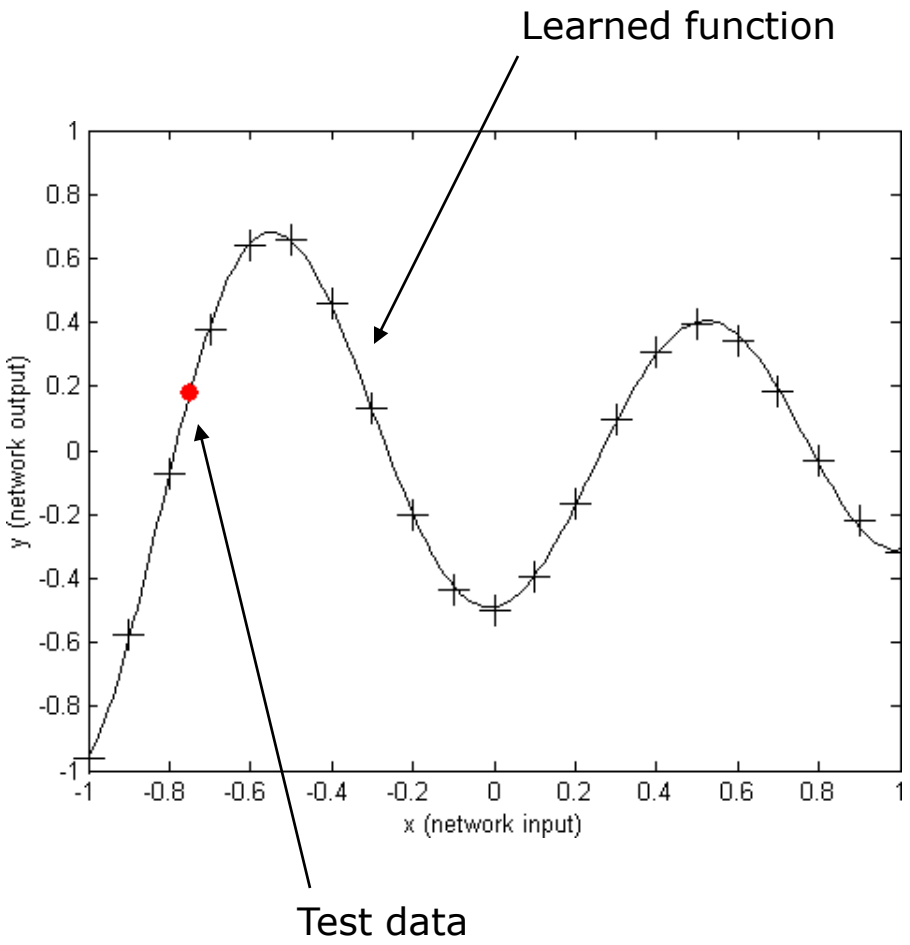
- Regression problem description:
  - Given training data (i.e., inputs and target outputs),*
  - Can we learn the underlying process generating these known outputs from these inputs?
    - Essentially, can we fit a function to the data?*
- How accurately can the trained network (fitted curve) predict the output for new data? (i.e., data not in the training set?)
- We will use a 1-D example for illustration, but the method is equally powerful for higher dimensions.
  - Can fit any function with any desired accuracy given appropriate weights and enough hidden nodes.

Example – Regression (2 of 4)

- Given: the training data shown here:
  - The network input is the independent variable (x).
- Goal: to train the network to mimic the process generating the dependent variable (y).
  - i.e., fit a curve to the data points.



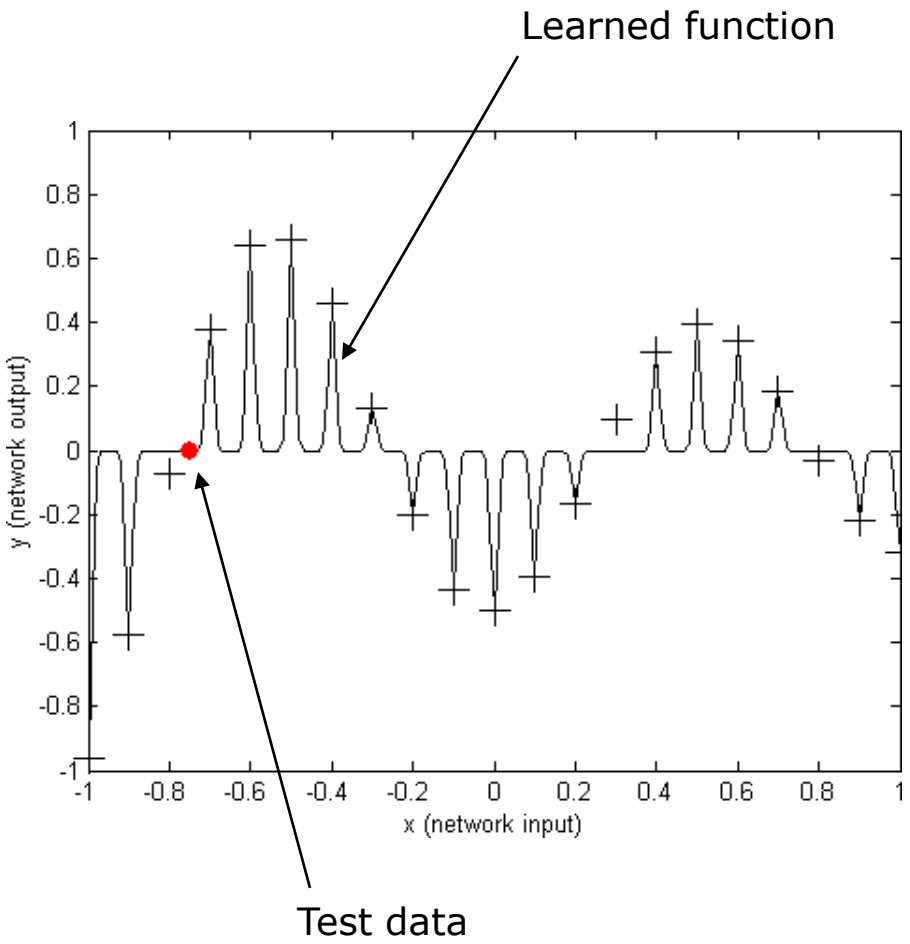
Example – Regression (3 of 4)



- This solution was found with:
  - 6 neurons in the hidden layer
  - centers in the range [-1,1] **(changed)**
  - Width r = 1.

- **Note**: An appropriate output (red dot) was produced when a new input value was given to the network.
  - The learned function effectively interpolates between the training data.
  - Shows network's ability to generalize.

Example – Regression (4 of 4)



- This solution was found with:
  - 18 neurons in the hidden layer
  - centers in the range [-1,1] **(changed)**
  - Width r = 0.01 (100x smaller)

- **Note**: Network has overfit the data!
  - Has not effectively interpolated between training points.
  - Network's response to new data no longer reflects the underlying process.

- Cause of problem: RBF width was too small!!
  - More on this later …

RBF Network Training

- Two aspects of training in RBF networks:

  1. RBF nodes' centers and widths.

  2. Weights connecting RBF nodes to output nodes.

Training Output Weights (1 of 2)

- Training output weights is simple when output neurons use linear activation:
  - Minimize squared error between target and network outputs:

$$E = \frac{1}{2}\sum_i (T_i - O_i)^2 = \frac{1}{2}\sum_i T_i^2 - \sum_i T_i O_i + \frac{1}{2}\sum_i O_i^2$$

$$= \frac{1}{2}\sum_i T_i^2 - \sum_i T_i \sum_j W_{j,i} A_j + \frac{1}{2}\sum_i \left(\sum_j W_{j,i} A_j \right)^2$$

  - Take the derivative w.r.t. the network weights

$$\frac{dE}{dW_{j,i}} = -\sum_i T_i \sum_j A_j + \sum_i \left(\sum_j W_{j,i} A_j \right) \sum_j A_j$$

$$= \sum_j A_j \left[\sum_i \sum_j W_{j,i} A_j - \sum_i T_i \right]$$

Training Output Weights (2 of 2)

- Setting the derivative to zero gives the following equation to solve for $W_{j,i}$

$$\sum_i \sum_j W_{j,i} A_j = \sum_i T_i$$

- Expressing in matrix notation, we have:

  where:

  $$\mathbf{AW} = \mathbf{T}$$

  - **W** = hidden-to-output weights
  - ***A*** = RBF layer outputs (each column of ***A*** is one set of RBF outputs)
  - **T** = target outputs (each column of **T** is the target outputs for each column of ***A***)

- Solving gives:

  $$\mathbf{W} = \mathbf{A}^{-1}\mathbf{T}$$

  - Can find the minimum error solution for all training inputs in one step!

RBF Layer Training

- To train the hidden layer with RBF nodes, we need to determine:

  1. Number of hidden neurons required.
  2. Center of each neuron's activation fn.
  3. Width of each neuron's activation fn.

- The process is simplified if we impose the condition that the widths of all activation fn.'s be equal, and known.

RBF Layer Training Algorithm

- The following algorithm can now be used to determine the number of neurons required.

1. Initialize the network with zero hidden neurons.
2. Find the training data that produces the greatest error.
3. Add a neuron a distance of zero from this data vector.
4. Train the output weights (linear system) to minimize overall error for the entire training set.
5. If error < ε (predefined max error), quit, otherwise goto step 2 and repeat.

- **NOTE:** May need to repeat many times with different widths to get a good result.

Pseudo-code for RBF training

```
1.  trainRBF( in, out, width, MaxError, data ) {
2.      hidden = 0;
3.      net = initRBFNetwork( in, out, hidden ); // init network nodes.
4.
5.      do {
6.          // find the data vector that produces the largest error
7.          i = findMaxNetworkError( data, net ); // i = index of vector

8.          // add neuron to the RBF layer at same point as the above data vector
9.          addRBFNeuron( net, width, data(i) ); // data(i) = center point

10.         // find overall network error
11.         NetError = trainOutputWeights( net, data );

12.     } while( NetError > MaxError );
13. }
```

Advantages of RBF Networks

- Linear output nodes give two advantages over MLPs:

1. Ease of training

   - Hidden-to-output weights trained in one step (linear system).

2. Not susceptible to getting stuck in local minima.

   - In training, wish to minimize MSE.
   - Linearity makes the MSE surface quadratic, with only one minimum point!

Disadvantages of RBF Networks

- Matrix inversion (for solving linear system) becomes expensive as:
  1. More hidden neurons are added.
  2. More training data is used.

  - For this reason, RBF networks are not particularly useful for large problem spaces.

- Gaussian activation fn.'s must adequately "cover" the input data.

  - Prone to overfitting; unseen input data may not be "close" to the centers of Gaussian functions (see previous examples)
  - Depends mostly on width parameter.

  - A number of intelligent techniques (e.g., supervised learning, clustering) have been tried to "learn" appropriate widths for activations from the training data.
    - don't need to know these