Recurrent Networks

A recurrent neural network is an MLP with feedback. The feedback can be local or global. There can be various forms of feedback and hence various architectures. Basically, there are two functional uses of recurrent networks:

- Associative memories
- Input-output mapping networks

To understand working of recurrent neural networks, we define a discrete time variable 't'. At time 't' all units in the network recomputed their outputs, which are then transmitted at time 't+1'. Continuing in this step-by-step fashion, the system produces a sequence of output values when a constant or time varying input is fed into the network.

A recurrent network behaves like a finite automaton. The question is, for input-output mapping, how to train such an automaton to produce desired sequence of output values.

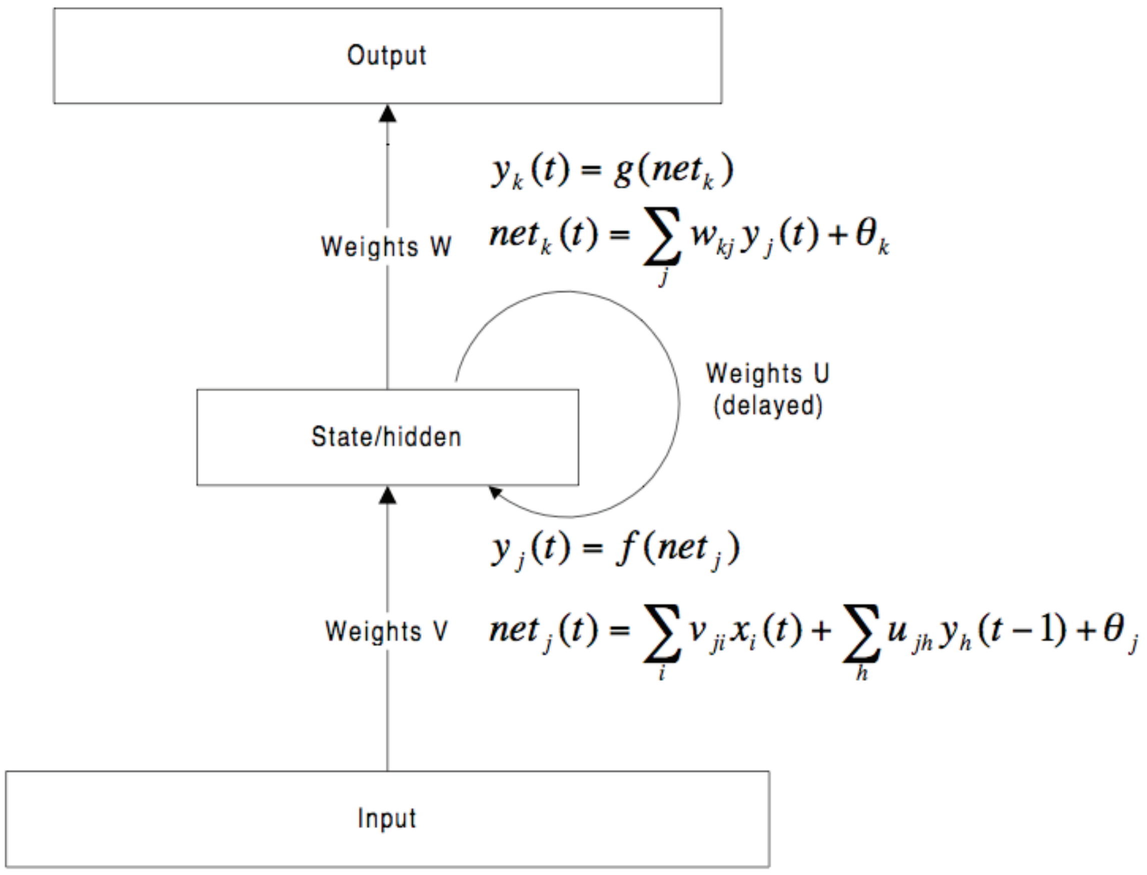

Output
Weights W
$y_k(t) = g(net_k)$
$net_k(t) = \sum_j w_{kj} y_j(t) + \theta_k$
Weights U
(delayed)
State/hidden
$y_j(t) = f(net_j)$
Weights V
$net_j(t) = \sum_i v_{ji} x_i(t) + \sum_h u_{jh} y_h(t-1) + \theta_j$
Input

Fig. 18: Simple Recurrent Neural Network

Refer to Fig. 15.1, Haykin for another architecture of RNN.

The layer that holds the previous state is known as context layer.

Temporal Models (Haykin 648)

Finite duration impulse response (FIR) of order 'p'.

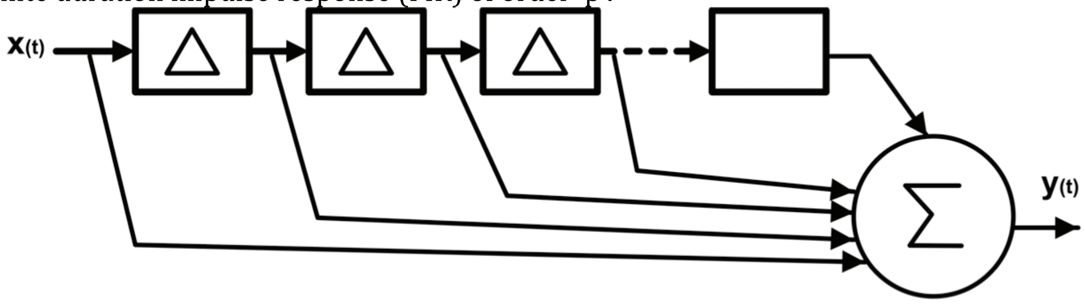


Fig. 19: FIR

Ref. to Section 13.7 (Haykin).

Step 3: derivation of a training algorithm for an FIR MLP

$$\Delta w_{(t)} = -\eta \, \frac{\partial E_{(t)}}{\partial w_{(t)}}$$

$$w_{(t+1)} = w_{(t)} + \Delta w_{(t)}$$

$$\Delta w_{(t)} = \eta \;\; f'(x_{(t)}) \;\; z^{l-1}{}_{(t-j)} \;\; e_{(t)}$$

$$1 \le j \le b \qquad order \;\; 'b'$$

Temporal difference learning

Temporal difference (TD) learning is an approach to learning how to predict a quantity that depends on future values of a given signal. The name TD derives from its use of changes, or differences, in predictions over successive time steps to drive the learning process. The prediction at any given time step is updated to bring it closer to the prediction of the same quantity at the next time step. It is a supervised learning process in which the training signal for a prediction is a future prediction. TD algorithms are often used in reinforcement learning to predict a measure of the total amount of reward expected over the future, but they can be used to predict other quantities as well. Continuous-time TD algorithms have also been developed.

Suppose a system receives as input a time sequence of vectors (x_t, y_t), $t = 0, 1, 2, \ldots$, where each x_t is an arbitrary signal and y_t is a real number. TD learning applies to the problem of producing at each discrete time step t, an estimate, or prediction, p_t, of the following quantity:

$$Y_t = y_{t+1} + \gamma y_{t+2} + \gamma^2 y_{t+3} + \cdots = \sum_{i=1}^{\infty} \gamma^{i-1} y_{t+i},$$

where γ is a discount factor, with $0 \le \gamma < 1$. Each estimate is a prediction because it involves future values of y. When $\gamma = 0$, p_t predicts just the next value of y, that is, y_{t+1}. As γ increases toward one, values of y in the more distant future become more significant.

The Simplest TD Algorithm

A good way to explain TD learning is to start with a simple case and then extend it to the full algorithm. Consider first the problem of making a one-step-ahead prediction, i.e., the above problem with $\gamma = 0$, meaning that one wants $p_t = y_{t+1}$ for each t. Incremental error-correction supervised learning can be used to update the prediction function as inputs arrive. Letting P_t denote the prediction function at step t, the algorithm updates P_t to a new prediction function P_{t+1} at each step. To do this, it uses the error between the current prediction, p_t, and the prediction target (the quantity being predicted), y_{t+1}. This error can be obtained by computing p_t by applying the current prediction function, P_t, to x_t, waiting one time step while remembering p_t, then observing y_{t+1} to obtain the information required to compute the error $y_{t+1} - p_t = y_{t+1} - P_t(x_t)$.

The simplest example of a prediction function is one implemented as a lookup table. Suppose x_t can take only a finite number of values and that there is an entry in a lookup table to store a prediction for each of these values. At step t, the table entry for input x_t is $p_t = P_t(x_t)$. When y_{t+1} is observed, the table entry for x_t is changed from its current value of p_t to $p_t + \alpha(y_{t+1} - p_t) = (1-\alpha)p_t + \alpha y_{t+1}$, where α is a positive fraction that controls how quickly new data is incorporated into the table and old data forgotten. Only the table entry corresponding to x_t is changed from step t to step $t+1$, to produce the a table, P_{t+1}. The fraction α is called the learning rate or step-size parameter.

To extend this approach to the full prediction problem with $\gamma \neq 0$, the prediction target would be $Y_t = y_{t+1} + \gamma y_{t+2} + \gamma^2 y_{t+3} + \cdots$ instead of just y_{t+1}. The algorithm above would have to update the table entry for x_t by changing its value from p_t to $p_t + \alpha(Y_t - p_t) = (1-\alpha)p_t + \alpha Y_t$. But to do this would require waiting for the arrival of all the future values of y instead of waiting for just the next value. This would prevent the approach from forming a useful learning rule. TD algorithms use the following observation to get around this problem. Notice that

$$\begin{aligned} Y_t &= y_{t+1} + \gamma y_{t+2} + \gamma^2 y_{t+3} + \cdots \\ &= y_{t+1} + \gamma[y_{t+2} + \gamma y_{t+3} + \gamma^2 y_{t+4} + \cdots] \\ &= y_{t+1} + \gamma Y_{t+1}, \end{aligned} \qquad (1)$$

for $t = 0, 1, 2, \ldots$.

Let's define the following temporal difference error (or TD error):

$$\delta_{t+1} = y_{t+1} + \gamma P_t(x_{t+1}) - P_t(x_t).$$

where, $y_{t+1} + \gamma \; P_t(x_{t+1}) = y_t$
$P_t(x_t) = P_t$

Then the simplest TD algorithm updates a lookup-table as follows: for each $t = 0, 1, 2, \ldots$, upon observing (x_{t+1}, y_{t+1}):

$$P_{t+1}(x) = \begin{cases} P_t(x) + \alpha\, \delta_{t+1} & \text{if } x = x_t \\ P_t(x) & \text{otherwise,} \end{cases}$$

where x denotes any possible input signal.

Back-propagation through time (BPTT)
This is an extension of the standard back-propagation algorithm. It is derived by unfolding the temporal operation of the network into a layered feedforward network, the topology of which grows by one layer at every time step.

We will see two different approaches to train RNN using BPTT.

- Epochwise back-propagation through time
- Truncated back-propagation through time

Epochwise BPTT (Haykin page 753)
The data sets used to train recurrent networks are partitioned into independent epochs with each epoch representing a temporal pattern of interest.

Let's say **n₀** denote the start time of an epoch and **n₁** denote its end time. Given this epoch, we may define the cost function;

$$\varepsilon_{total}(n_0, n_1) = \frac{1}{2} \sum_{n=n_0}^{n_1} \sum_{j \in A} e_j^2(n)$$

where, A = set of indices 'j' pertaining to those neurons in the network for which desired responses are specified.

Truncated BPTT (Haykin page 754)
It uses instantaneous values of sum of squared errors.

$$\varepsilon(n) = \frac{1}{2} \sum_{j=A} e_j^2(n)$$

Save the historical data – input and network states. Let the depth be 'h'.

$$\delta_j(l) = -\frac{\partial \varepsilon(l)}{\partial v_j(l)}$$

$$\textit{where} \quad j \in A$$
$$n - h < l \le n$$

Unsupervised learning

In machine learning, unsupervised learning is a class of problems in which one seeks to determine how the data are organized. It is distinguished from supervised learning (and reinforcement learning) in that the learner is given only unlabeled examples.

Among neural network models, the Self-Organizing Map (SOM) and Adaptive resonance theory (ART) are commonly used unsupervised learning algorithms. The SOM is a topographic organization in which nearby locations in the map represent inputs with similar properties.

Unsupervised learning allegedly involves no target values. In fact, for most varieties of unsupervised learning, the targets are the same as the inputs. In other words, unsupervised learning usually performs the same task as an auto-associative network, compressing the information from the inputs. Unsupervised learning is very useful for data visualization although the NN literature generally ignores this application.

Unsupervised competitive learning is used in a wide variety of fields under a wide variety of names, the most common of which is "cluster analysis" (see the Classification Society of North America's web site for more information on cluster analysis, including software, at http://www.pitt.edu/~csna/.) The main form of competitive learning in the NN literature is vector quantization (VQ, also called a "Kohonen network", although Kohonen invented several other types of networks as well.

Competitive Learning

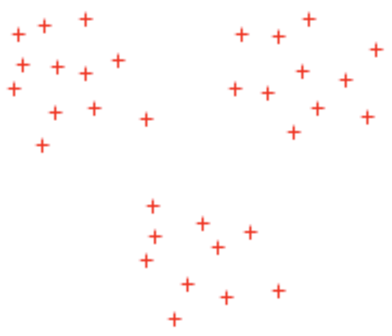
Competitive learning is a rule based on the idea that only one neuron from a given iteration in a given layer will fire at a time. Weights are adjusted such that only one neuron in a layer, for instance the output layer, fire. Competitive learning is useful for classification of input patterns into a discrete set of output classes. The "winner" of each iteration, element i* , is the element whose total weighted input is the largest. Using this notation, one example of a competitive learning rule can be defined mathematically as:

$$w_{ij}[n+1] = w_{ij}[n] + \Delta w_{ij}[n]$$

$$\Delta w_{ij}[n] = \begin{cases} \eta(x_i - w_{ij}) & if \quad i = j \\ 0 & otherwise \end{cases}$$

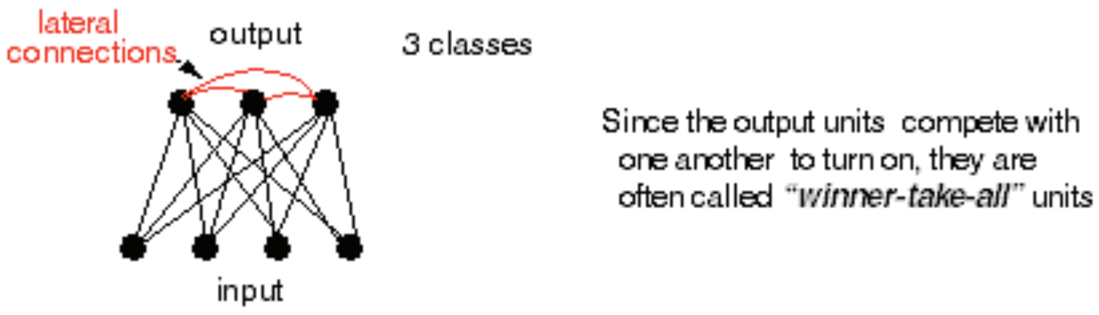
In competitive networks, output units compete for the right to respond.

Goal: method of clustering - divide the data into a number of clusters such that the inputs in the same cluster are in some sense similar.



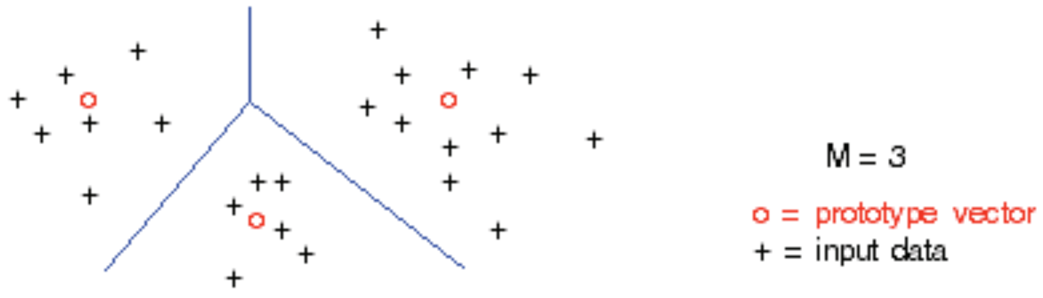
A basic competitive learning network has one layer of input nodes and one layer of output nodes. Binary valued outputs are often (but not always) used. There are as many output nodes as there are classes.

Often (but not always) there are lateral inhibitory connections between the output nodes.(in simulations, the function of the lateral connections can be replaced with a different algorithm)

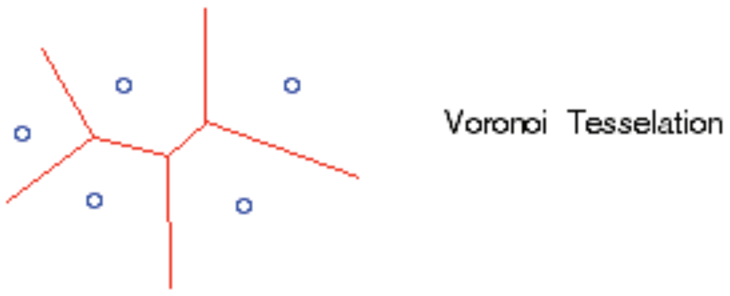

lateral connections
output
3 classes
input
Since the output units compete with one another to turn on, they are often called *"winner-take-all"* units

The output units are also often called ***grandmother cells***. The term grandmother cell comes from discussions as to whether your brain might contain cells that fire only when you encounter your maternal grandmother, or whether such higher level concepts are more distributed.

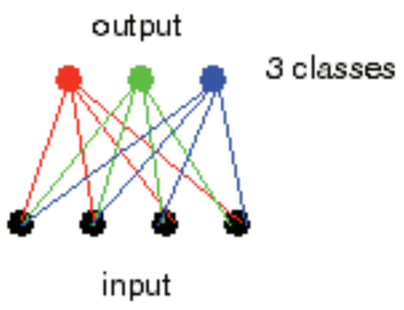
Vector Quantization (VQ)
Vector quantization is one example of competitive learning.
The goal here is to have the network "discover" structure in the data by finding how the data is clustered. The results can be used for data encoding and compression. One such method for doing this is called vector quantization.



In vector quantization, we assume there is a codebook which is defined by a set of M prototype vectors. (M is chosen by the user and the initial prototype vectors are chosen arbitrarily).
An input belongs to cluster i if i is the index of the closest prototype (closest in the sense of the normal euclidean distance). This has the effect of dividing up the input space into a ***Voronoi tesselation***.



Implementing Vector Quantization with a Network



The prototypes are represented by the weight vectors connecting to each output node.
Here there are 3 prototypes:
$w_{\cdot 1}$ = the weights connecting to output 1
$w_{\cdot 2}$ = the weights connecting to output 2
$w_{\cdot 3}$ = the weights connecting to output 3

VQ and Data Compression

Vector quantization can be used for (lossy) data compression. If we are sending information over a phone line, we

- initially send the codebook vectors
- for each input, we send the index of the class that the input belongs

For a large amount of data, this can be a significant reduction. If M=64, then it takes only 6 bits to encode the index. If the data itself consists of floating point numbers (4 bytes) there is an 80% reduction (100*(1 - 6/32)).

Learning Vector Quantization

In a learning vector quantization (LVQ) machine, the input values are compared to the weight vector of each neuron. Neurons who most closely match the input are known as the best match unit (BMU) of the system. The weight vector of the BMU and those of nearby neurons are adjusted to be closer to the input vector by a certain step size. Neurons become trained to be individual feature detectors, and a combination of feature detectors can be used to identify large classes of features from the input space. The LVQ algorithm is a simplified precursor to more advanced learning algorithms, such as the self-organizing map. LVQ training is a type of competitive learning rule.

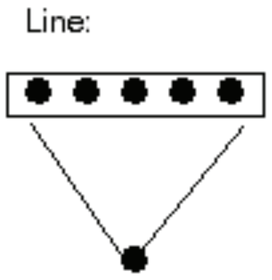
Kohonen Networks

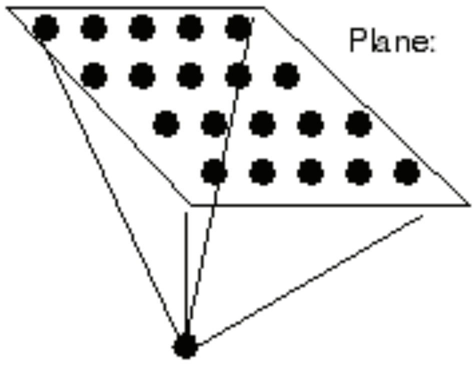
Kohonon's SOMs are a type of unsupervised learning. The goal is to discover some underlying structure of the data. However, the kind of structure we are looking for is very different than, say, PCA or vector quantization.

Kohonen's SOM is called a topology-preserving map because there is a topological structure imposed on the nodes in the network. A topological map is simply a mapping that preserves neighborhood relations.

In the nets we have studied so far, we have ignored the geometrical arrangements of output nodes. Each node in a given layer has been identical in that each is connected with all of the nodes in the upper and/or lower layer. We are now going to take into consideration that physical arrangement of these nodes. Nodes that are "close" together are going to interact differently than nodes that are "far" apart.

What do we mean by "close" and "far"? We can think of organizing the output nodes in a line or in a planar configuration.





The goal is to train the net so that nearby outputs correspond to nearby inputs.

E.g. if x1 and x2 are two input vectors and t1 and t2 are the locations of the corresponding winning output nodes, then t1 and t2 should be close if x1 and x2 are similar. A network that performs this kind of mapping is called a feature map.

In the brain, neurons tend to cluster in groups. The connections within the group are much greater than the connections with the neurons outside of the group. Kohonen's network tries to mimick this in a simple way.

Algorithm for Kohonon's Self Organizing Map

- Assume output nodes are connected in an array (usually 1 or 2 dimensional)
- Assume that the network is fully connected - all nodes in input layer are connected to all nodes in output layer.
- Use the competitive learning algorithm as follows:
  - Randomly choose an input vector x
  - Determine the "winning" output node i, where w_i is the weight vector connecting the inputs to output node i.
    Note: the above equation is equivalent to w_i x >= w_k x only if the weights are normalized.

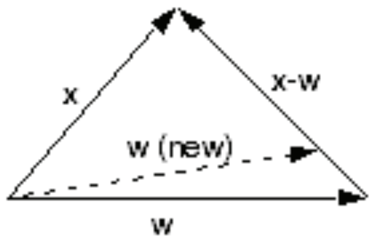
$$|\omega_i - x| \le |\omega_k - x| \qquad \forall k$$

  - Given the winning node i, the weight update is

$$w_k(\text{new}) = w_k(\text{old}) + \mu\, \aleph(i, k)\,(x - w_k)$$

where $\aleph(i, k)$ is called the neighborhood function that has value 1 when i=k and falls off with the distance $|r_k - r_i|$ between units i and k in the output array. Thus, units close to the winner as well as the winner itself, have their weights updated appreciably. Weights associated with far away output nodes do not change significantly. It is here that the toplogical information is supplied. Nearby units receive similar updates and thus end up responding to nearby input patterns.

The above rule drags the weight vector wi and the weights of nearby units towards the input x.



Example of the neighborhood function is:

$$\aleph(i, k) = e^{\left(-|r_k - r_i|^2\right)/(2\sigma^2)}$$

where s^2 is the width parameter that can gradually be decreased over time.

Kohonen's Self organizing Feature Maps
Adapted from ai-junkie.com

Overview
Kohonen Self Organising Feature Maps, or SOMs as I shall be referring to them from now on, are fascinating beasts. They were invented by a man named Teuvo Kohonen, a professor of the Academy of Finland, and they provide a way of representing multidimensional data in much lower dimensional spaces - usually one or two dimensions. This process, of reducing the dimensionality of vectors, is essentially a data compression technique known as *vector quantisation*. In addition, the Kohonen technique creates a network that stores information in such a way that any topological relationships within the training set are maintained.

A common example used to help teach the principals behind SOMs is the mapping of colours from their three dimensional components - red, green and blue, into two dimensions. Figure 20 shows an example of a SOM trained to recognize the eight different colours shown on the right. The colours have been presented to the network as 3D vectors - one dimension for each of the colour components - and the network has learnt to represent them in the 2D space you can see. Notice that in addition to clustering the colours into distinct regions, regions of similar properties are usually found adjacent to each other. This feature of Kohonen maps is often put to good use as you will discover later.

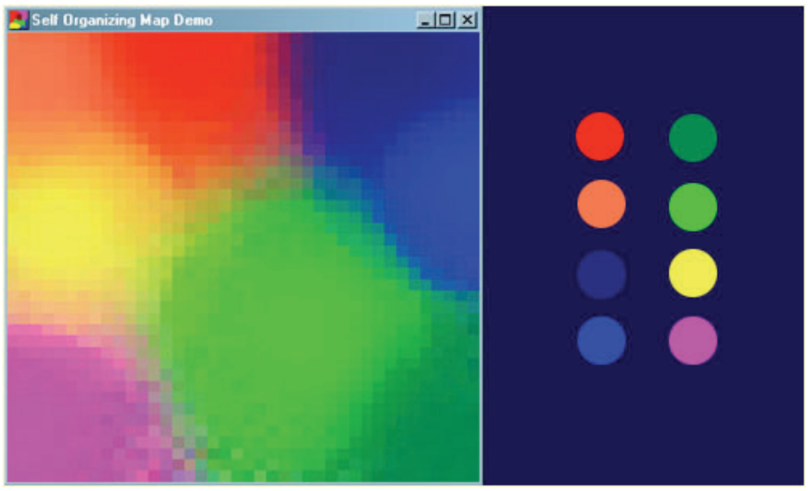


Fig. 20: Screenshot of the demo program (left) and the colours it has classified (right)

One of the most interesting aspects of SOMs is that they learn to classify data *without supervision*. You may already be aware of supervised training techniques such as backpropagation where the training data consists of vector pairs - an input vector and a target vector. With this approach an input vector is presented to the network (typically a multilayer feedforward network) and the output is compared with the target vector. If they differ, the weights of the network are altered slightly to reduce the error in the output. This is repeated many times and with many sets of vector pairs until the network gives the desired output. Training a SOM however, requires no target vector. A SOM learns to classify the training data without any external supervision whatsoever. Neat huh?

Network Architecture
For the purposes of this tutorial I'll be discussing a two dimensional SOM. The network is created from a 2D lattice of 'nodes', each of which is fully connected to the input layer. Figure 21 shows a very small Kohonen network of 4 X 4 nodes connected to the input layer (shown in green) representing a two dimensional vector.

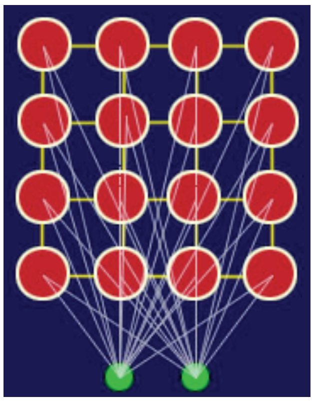


Fig. 21: A simple Kohonen network

Each node has a specific topological position (an x, y coordinate in the lattice) and contains a vector of weights of the same dimension as the input vectors. That is to say, if the training data consists of vectors, *V*, of *n* dimensions:

$V_1, V_2, V_3 \ldots V_n$

Then each node will contain a corresponding weight vector *W,* of *n* dimensions:

$W_1, W_2, W_3 \ldots W_n$

The lines connecting the nodes in Figure 21 are only there to represent adjacency and do not signify a connection as normally indicated when discussing a neural network. *There are no lateral connections between nodes within the lattice.*

The SOM shown in Figure 20 has a default lattice size of 40 X 40. Each node in the lattice has three weights, one for each element of the input vector: red, green and blue. Each node is represented by a rectangular cell when drawn to your display. Figure 22 shows the cells rendered with black outlines so you can clearly see each node.

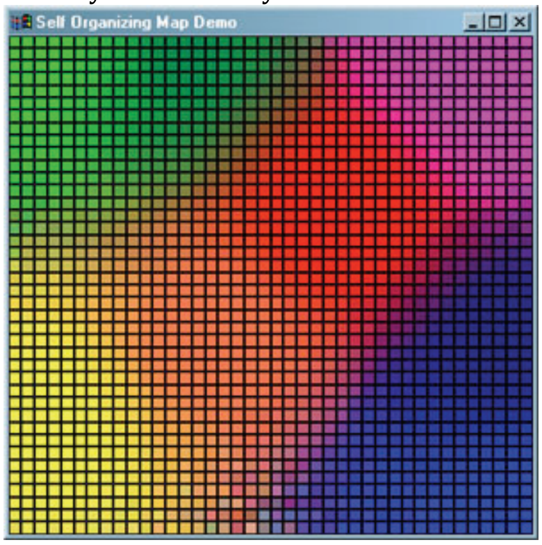


Fig. 22: Each cell represents a node in the lattice

Learning Algorithm Overview

A SOM does not need a target output to be specified unlike many other types of network. Instead, where the node weights match the input vector, that area of the lattice is selectively optimized to more closely resemble the data for the class the input vector is a member of. From an initial distribution of random weights, and over many iterations, the SOM eventually settles into a map of stable zones. Each zone is effectively a feature classifier, so you can think of the graphical output as a type of feature map of the input space. If you take another look at the trained network shown in figure 1, the blocks of similar colour represent the individual zones. Any new, previously unseen input vectors presented to the network will stimulate nodes in the zone with similar weight vectors.

Training occurs in several steps and over many iterations:

1. Each node's weights are initialized.
2. A vector is chosen at random from the set of training data and presented to the lattice.
3. Every node is examined to calculate which one's weights are most like the input vector. The winning node is commonly known as the Best Matching Unit (BMU).
4. The radius of the neighborhood of the BMU is now calculated. This is a value that starts large, typically set to the 'radius' of the lattice, but diminishes each time-step. Any nodes found within this radius are deemed to be inside the BMU's neighborhood.
5. Each neighboring node's (the nodes found in step 4) weights are adjusted to make them more like the input vector. The closer a node is to the BMU, the more its weights get altered.
6. Repeat step 2 for N iterations.

The Learning Algorithm In Detail

Now let's take a look at each step in detail. I'll supplement my descriptions with extracts from the source code where appropriate. Hopefully, for those of you who can program, the code will help reinforce your understanding of the principles involved.

Initializing The Weights

Prior to training, each node's weights must be initialized. Typically these will be set to small standardized random values. The weights in the SOM from the accompanying code project are initialized so that 0 < w < 1. Nodes are defined in the source code by the class CNode. Here are the relevant parts of that class:

```
class CNode
{
private:
  //this node's weights
  vector<double>     m_dWeights;

  //its position within the lattice
  double             m_dX,
                     m_dY;

  //the edges of this node's cell. Each node, when draw to the client
  //area, is represented as a rectangular cell. The colour of the cell
  //is set to the RGB value its weights represent.
  int                m_iLeft;
  int                m_iTop;
  int                m_iRight;
  int                m_iBottom;


public:
  CNode(int lft, int rgt, int top, int bot, int NumWeights):m_iLeft(lft),
                                                        m_iRight(rgt),
                                                        m_iBottom(bot),
                          m_iTop(top)
  {
   //initialize the weights to small random variables
   for (int w=0; w<NumWeights; ++w)
   {
    m_dWeights.push_back(RandFloat());
   }

   //calculate the node's center
   m_dX = m_iLeft + (double)(m_iRight - m_iLeft)/2;
   m_dY = m_iTop  + (double)(m_iBottom - m_iTop)/2;
  }

  ...

 };
```

You can see that the weights are initialized automatically by the constructor when a CNode object is instantiated. The member variables m_iLeft, m_iRight, m_iTop and m_iBottom are only used to render the network to the display area - each node is represented by a rectangular cell described by these values, as shown previously in Figure 22.

Calculating the Best Matching Unit
To determine the best matching unit, one method is to iterate through all the nodes and calculate the Euclidean distance between each node's weight vector and the current input vector. The node with a weight vector closest to the input vector is tagged as the BMU.
The Euclidean distance is given as:

$$Dist = \sqrt{\sum_{i=0}^{i=n} (V_i - W_i)^2} \quad \text{Equation 1}$$

where ***V*** is the current input vector and ***W*** is the node's weight vector. In code this equation translates to:

```
double CNode::GetDistance(const vector<double> &InputVector)
{
 double distance = 0;

 for (int i=0; i<m_dWeights.size(); ++i)
 {
  distance += (InputVector[i] - m_dWeights[i]) * (InputVector[i] - m_dWeights[i]);
 }

 return sqrt(distance);
}
```

As an example, to calculate the distance between the vector for the colour red (1, 0, 0) with an arbitrary weight vector (0.1, 0.4, 0.5)

distance = sqrt($(1 - 0.1)^2 + (0 - 0.4)^2 + (0 - 0.5)^2$)

$= \text{sqrt}((0.9)^2 + (-0.4)^2 + (-0.5)^2)$

$= \text{sqrt}(0.81 + 0.16 + 0.25)$

$= \text{sqrt}(1.22)$

distance = 1.106

Determining the Best Matching Unit's Local Neighbourhood

This is where things start to get more interesting! Each iteration, after the BMU has been determined, the next step is to calculate which of the other nodes are within the BMU's neighborhood. All these nodes will have their weight vectors altered in the next step. So how do we do that? Well it's not too difficult... first you calculate what the radius of the neighborhood should be and then it's a simple application of good ol' Pythagoras to determine if each node is within the radial distance or not. Figure 23 shows an example of the size of a typical neighborhood close to the commencement of training.

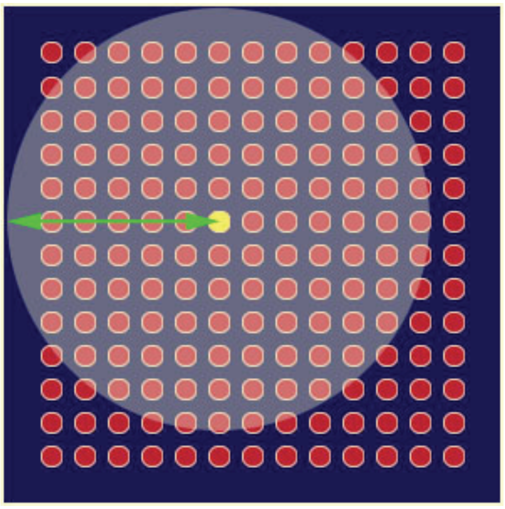


Fig. 23: The BMU's neighbourhood

You can see that the neighbourhood shown above is centered around the BMU (coloured yellow) and encompasses most of the other nodes. The green arrow shows the radius.

A unique feature of the Kohonen learning algorithm is that the area of the neighbourhood shrinks over time. This is accomplished by making the radius of the neighbourhood shrink over time (that was hard to work out eh? ;0)). To do this I use the exponential decay function:

$$\sigma(t) = \sigma_0 \exp\left(-\frac{t}{\lambda}\right) \qquad t = 1, 2, 3\ldots$$ equation 2

where the Greek letter sigma, σ_0, denotes the width of the lattice at time t_0 and the Greek letter lambda, λ, denotes a time constant. t is the current time-step (iteration of the loop). In my code I name the value σ, m_dMapRadius, and it is equal to σ_0 at the commencement of training. This is how I calculate σ_0.

```
m_dMapRadius = max(constWindowWidth, constWindowHeight)/2;
```

The value of λ is dependent on σ and the number of iterations chosen for the algorithm to run. I have named λ, m_dTimeConstant and it is calculated by the line:

```
m_dTimeConstant = m_iNumIterations/log(m_dMapRadius);
```

m_iNumIterations is the number of iterations the learning algorithm will perform. This value is set by the user in 'constants.h'.
We can use m_dTimeConstant and m_dMapRadius to calculate the neighbourhood radius for each iteration of the algorithm using Equation 2. The line that does this in the source code can be found inside Csom::Epoch and it looks like this:

```
m_dNeighbourhoodRadius = m_dMapRadius * exp(-(double)m_iIterationCount/m_dTimeConstant);
```

Figure 5 shows how the neighbourhood in Figure 23 decreases over time (the figure is drawn assuming the neighbourhood remains centered on the same node, in practice the BMU will move around according to the input vector being presented to the network) .

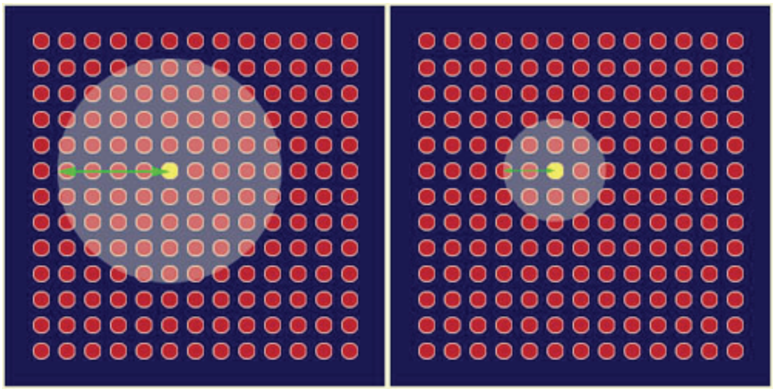


Fig. 24: The ever shrinking radius

Over time the neighborhood will shrink to the size of just one node... the BMU.

Now we know the radius, it's a simple matter to iterate through all the nodes in the lattice to determine if they lay within the radius or not. If a node is found to be within the neighborhood then its weight vector is adjusted as follows.

Adjusting the Weights

Every node within the BMU's neighbourhood (including the BMU) has its weight vector adjusted according to the following equation:

$$W(t+1) = W(t) + L(t)(V(t) - W(t))$$ Equation 3

Where t represents the time-step and L is a small variable called the learning rate, which decreases with time. Basically, what this equation is saying, is that the new adjusted weight for the node is equal to the old weight (W), plus a fraction of the difference (L) between the old weight and the input vector (V). My, that was a bit of a mouthful, no wonder mathematicians prefer symbols!

The decay of the learning rate is calculated each iteration using the following equation:

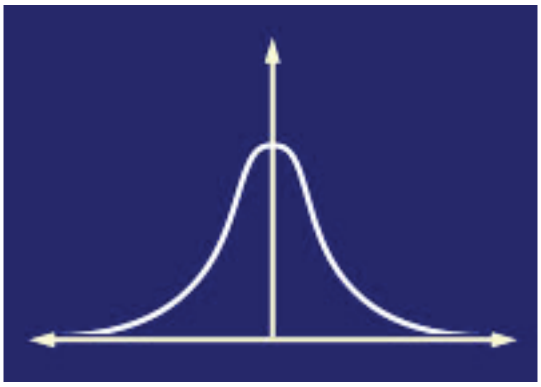
$$L(t) = L_0 \exp\left(-\frac{t}{\lambda}\right) \qquad t = 1,2,3\ldots$$ Equation 4

The more observant amongst you will notice that this is the same as the exponential decay function described in Equation 2, except this time I'm using it to decay the learning rate. In code it looks like this:

```
m_dLearningRate = constStartLearningRate * exp(-(double)m_iIterationCount/m_iNumIterations);
```

The learning rate at the start of training, constStartLearningRate, is set in the 'constants.h' file as 0.1. It then gradually decays over time so that during the last few iterations it is close to zero.

But Wait! I've not been completely honest with you. Equation 3 is incorrect! You see, not only does the learning rate have to decay over time, but also, the effect of learning should be proportional to the distance a node is from the BMU. Indeed, at the edges of the BMUs neighbourhood, the learning process should have barely any effect at all! Ideally, the amount of learning should fade over distance similar to the Gaussian decay shown below.



To achieve this, all it takes is a slight adjustment to Equation 3.

$$W(t+1) = W(t) + \Theta(t)L(t)(V(t) - W(t))$$ Equation 5

I've used the Greek capital letter theta, Θ, to represent the amount of influence a node's distance from the BMU has on its learning. Θ(t) is given by Equation 6.

$$\Theta(t) = \exp\left(-\frac{dist^2}{2\sigma^2(t)}\right) \qquad t = 1,2,3\ldots$$ Equation 6

Where dist is the distance a node is from the BMU and σ, is the width of the neighbourhood function as calculated by Equation 2. Additionally, please note that Θ also decays over time.