

Kohonen Self-Organizing Maps

December 2005

Shyam M. Guthikonda
[shyamguth AT gmail DOT com](mailto:shyamguth@gmail.com)

Wittenberg University

I. Table of Contents

I. Introduction	<i>1</i>
Introduction to Neural Networks	<i>1</i>
Introduction to Kohonen Self-Organizing Maps	<i>3</i>
II. Self-Organizing Maps (SOMs)	<i>6</i>
Structure of a SOM	<i>6</i>
The SOM Algorithm	<i>7</i>
III. Applications	<i>11</i>
Color Classification	<i>11</i>
Image Classification	<i>16</i>
IV. Conclusion	<i>18</i>
V. References	<i>19</i>
Publications	<i>19</i>
External Links	<i>19</i>

I. Introduction

Introduction to Neural Networks

Neural networks are a fascinating concept. The first neural network, "Perceptron", was created in 1956 by Frank Rosenblatt. Thirteen years later in 1969, a publication known as *Perceptrons*, by Minsky and Papert, brought a devastating blow to neural network research. In this book, it formalized the concept neural networks, but also pointed out some serious limitations in the original architecture [Noye92]. Specifically, it showed that Perceptron could not perform a basic logical computation of a XOR (exclusive-or). This nearly brought neural network research to a halt. Fortunately, research has resumed, and at the time of this writing, neural network popularity has increased dramatically as a tool to provide solutions to difficult problems [Fitz97].

Designed around the brain-paradigm of Artificial Intelligence, neural networks attempt to model the biological brain. Neural networks are very different from most standard computer science concepts. In a typical program, data is stored in some structure such as frames, which are then stored within a centralized database, such as with an Expert System or a Natural Language Processor. In neural networks, however, information is *distributed* throughout the network [Noye92]. This mirrors the biological brain, which stores its information (memories) throughout its' synapses.

Neural networks have features that make them appealing to both connectionist researchers and individuals needing ways to solve complex problems. Neural networks can be extremely fast and efficient. This facilitates the handling of large amounts of data [Noye92]. The reason for this is that each node in a neural network is essentially its own autonomous entity. Each performs only a small computation in the grand-scheme of the problem. The aggregate of all these nodes, the entire network, is where the true capability lies. This architecture allows for parallel implementation, much like the biological brain, which performs nearly all of its' tasks in parallel. Neural networks are also fault

tolerant [Noye92]. This is a fundamental property. A small portion of bad data or a sector of non-functional nodes will not cripple the network. It will learn to adjust. This does have a limit, though, as too much 'bad' data will cause the network to produce incorrect results. The same can be said of the human brain. A slight head trauma may produce no noticeable effects, but a major head trauma (or a slight head trauma to the correct spot) may leave the person incapacitated. Also, as pointed out in many studies, we can still understand a sentence when the letters of a word are out of order. This proves that our brain can manage with some corrupt input data.

How do neural networks learn? First off, we must define learning. Biologically, learning is an experience that changes the state of an organism such that the new state leads to an improved performance in subsequent situations. Mechanically, it is similar: Computational methods for acquiring and organizing new knowledge that will lead to new skills [Noye92]. Before the network can become useful, it must learn about the information at hand. After training, it can then be used for pragmatic purposes. In general, there are two flavors of learning [Egge98]:

- Supervised Learning. The correct answers are known and this information is used to train the network for the given problem. This type of learning utilizes both input vectors and output vectors. The input vectors are used to provide the starting data, and the output vectors can be used to compare with the input vectors to determine some error. In a special type of supervised learning, *reinforcement learning*, the network is only told if its output is right or wrong. Back-propagation algorithms make use of this style.
- Unsupervised Learning. The correct answers are not known (or just not told to the network). The network must try on its own to discover patterns in the input data. The input vectors are used solely. Output vectors generated will not be used to learn from. Also, and possibly most importantly: no

human interaction is needed for unsupervised learning. This can be an extremely important feature, especially when dealing with a large and/or complex data set that would be time-consuming or difficult to a human to compute. Self-Organizing Maps utilize this style of learning.

As can be seen from the above descriptions, a neural network can have many different features from another neural network. Because of this, there are many different types of neural networks. One in particular, the Self-Organizing Map, is discussed next.

Introduction to Kohonen Self-Organizing Maps

Kohonen Self-Organizing Maps (or just Self-Organizing Maps, or SOMs for short), are a type of neural network. They were developed in 1982 by Tuevo Kohonen, a professor emeritus of the Academy of Finland [Wiki-01]. Self-Organizing Maps are aptly named. "Self-Organizing" is because no supervision is required. SOMs learn on their own through unsupervised competitive learning. "Maps" is because they attempt to *map* their weights to conform to the given input data. The nodes in a SOM network attempt to become like the inputs presented to them. In this sense, this is how they learn. They can also be called "Feature Maps", as in Self-Organizing Feature Maps. Retaining principle 'features' of the input data is a fundamental principle of SOMs, and one of the things that makes them so valuable. Specifically, the topological relationships between input data are preserved when mapped to a SOM network. This has a pragmatic value of representing complex data. For example [Buck03]:

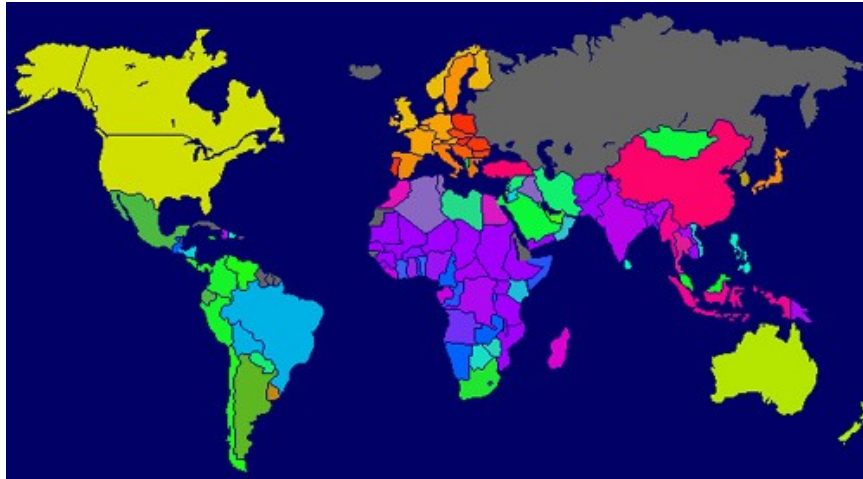


Figure 1

This is a map of the world quality-of-life. Yellows and oranges represent wealthy nations, while purples and blues are the poorer nations. From this view, it can be difficult to visualize the relationships between countries. However, represented by a SOM as shown in Figure 2 [Buck03], it is much easier to see what is going on.

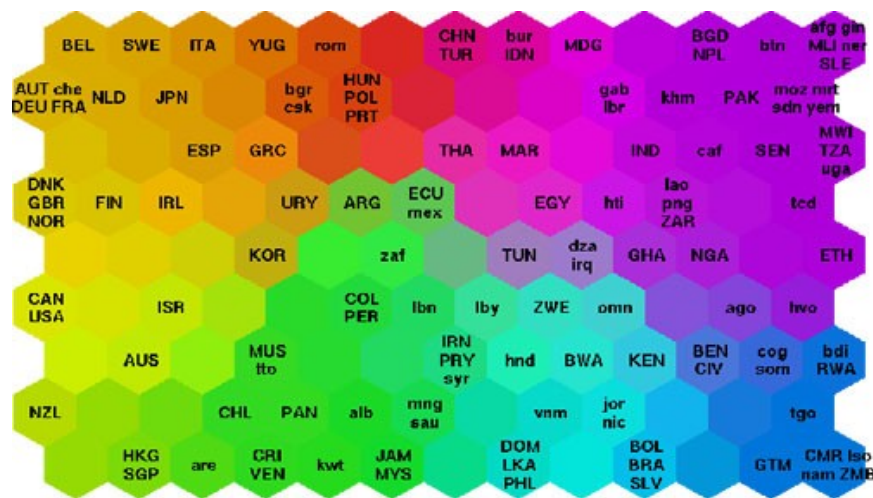


Figure 2

Here we can see the United States, Canada, and Western European countries, on the left side of the network, being the wealthiest countries. The poorest countries, then, can be found on the opposite side of the map (at the point farthest away from the richest countries), represented by the purples and blues.

Figure 2 is a hexagonal grid. Each hexagon represents a node in the neural

network. This is typically called a unified distance matrix, or a u-matrix [Buck03], and is probably the most popular method of displaying SOMs.

Another intrinsic property of SOMs is known as vector quantization. This is a data compression technique. SOMs provide a way of representing multi-dimensional data in a much lower dimensional space – typically one or two dimensions [Buck03]. This aids in their visualization benefit, as humans are more proficient at comprehending data in lower dimensions than higher dimensions, as can be seen in the comparison of Figure 1 to Figure 2.

The above examples show how SOMs are a valuable tool in dealing with complex or vast amounts of data. In particular, they are extremely useful for the visualization and representation of these complex or large quantities of data in manner that is most easily understood by the human brain.

II. Self-Organizing Maps (SOMs)

Structure of a SOM

The structure of a SOM is fairly simple, and is best understood with the use of an illustration such as Figure 3 [Ches04].

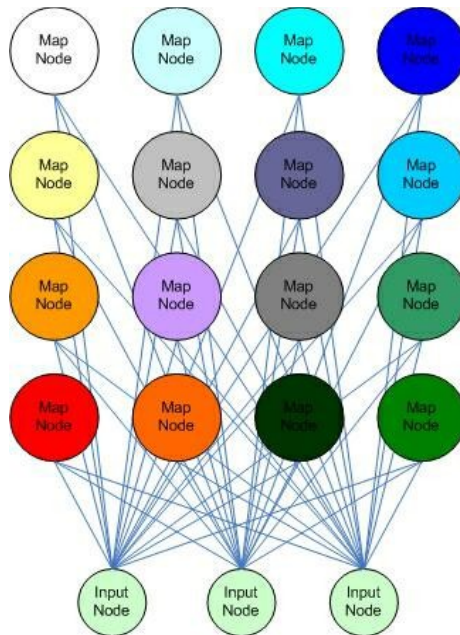


Figure 3

Figure 3 is a 4x4 SOM network (4 nodes down, 4 nodes across). It is easy to overlook this structure as being trivial, but there are a few key things to notice. First, each map node is connected to each input node. For this small 4x4 node network, that is $4 \times 4 \times 3 = 48$ connections. Secondly, notice that *map nodes are not connected to each other*. The nodes are organized in this manner, as a 2-D grid makes it easy to visualize the results. This representation is also useful when the SOM algorithm is used. In this configuration, each map node has a unique (i,j) coordinate. This makes it easy to reference a node in the network, and to calculate the distances between nodes. Because of the connections only to the input nodes, the map nodes are oblivious as to what values their neighbors have. A map node will only update its' weights (explained next) based on what the input vector tells it.

The following relationships describe what a node essentially is:

1. $network \subset mapNode \subset float\ weights[numWeights]$
2. $inputVectors \subset inputVector \subset float\ weights[numWeights]$

1 says that the network (the 4x4 grid above) contains map nodes. A single map node contains an array of floats, or its' weights. numWeights will become more apparent during application discussion. The only other common item that a map node should contain is it's (i,j) position in the network. 2 says that the collection of input vectors (or input nodes) contains individual input vectors. Each input vector contains an array of floats, or its' weights. Note that numWeights is the same for both weight vectors. The weight vectors must be the same for map nodes and input vectors or the algorithm will not work.

The SOM Algorithm

The Self-Organizing Map algorithm can be broken up into 6 steps [Buck03].

- 1). Each node's weights are initialized.
- 2). A vector is chosen at random from the set of training data and presented to the network.
- 3). Every node in the network is examined to calculate which ones' weights are most like the input vector. The winning node is commonly known as the *Best Matching Unit* (BMU). (Equation 1).
- 4). The radius of the neighborhood of the BMU is calculated. This value starts large. Typically it is set to be the radius of the network, diminishing each time-step. (Equation 2a, 2b).
- 5). Any nodes found within the radius of the BMU, calculated in 4), are adjusted to make them more like the input vector (Equation 3a, 3b). The closer a node is to the BMU, the more its' weights are altered (Equation 3c).
- 6). Repeat 2) for N iterations.

The equations utilized by the algorithm are as follows:

Equation 1 – Calculate the BMU.

$$DistFromInput^2 = \sum_{i=0}^{i=n} (I_i - W_i)^2$$

I = current input vector
W = node's weight vector
n = number of weights

Equation 2a – Radius of the neighborhood.

$$\sigma(t) = \sigma_0 e^{(-t/\lambda)}$$

t = current iteration
 λ = time constant (Equation 2b)
 σ_0 = radius of the map

Equation 2b – Time constant

$$\lambda = numIterations / mapRadius$$

Equation 3a – New weight of a node.

$$W(t+1) = W(t) + \Theta(t) L(t) (I(t) - W(t))$$

Equation 3b – Learning rate.

$$L(t) = L_0 e^{(-t/\lambda)}$$

Equation 3c – Distance from BMU.

$$\Theta(t) = e^{(-distFromBMU^2 / (2\sigma^2(t)))}$$

There are some things to note about these formulas. Equation 1 is simply the Euclidean distance formula, squared. It is squared because we are not concerned with the actual numerical distance from the input. We just need some sort of uniform scale in order to compare each node to the input vector. This equation provides that, eliminating the need for a computationally expensive square root operation for every node in the network.

Equations 2a and 3b utilize exponential decay. At t=0 they are at their max. As t (the current iteration number) increases, they approach zero. This is exactly what we want. In 2a, the radius should start out as the radius of the lattice, and approach zero, at which time the radius is simply the BMU node (see Figure 4).

Equation 2b is almost arbitrary. Any constant value can be chosen. This provides a good value, though, as it depends directly on the map size and the number of iterations to perform.

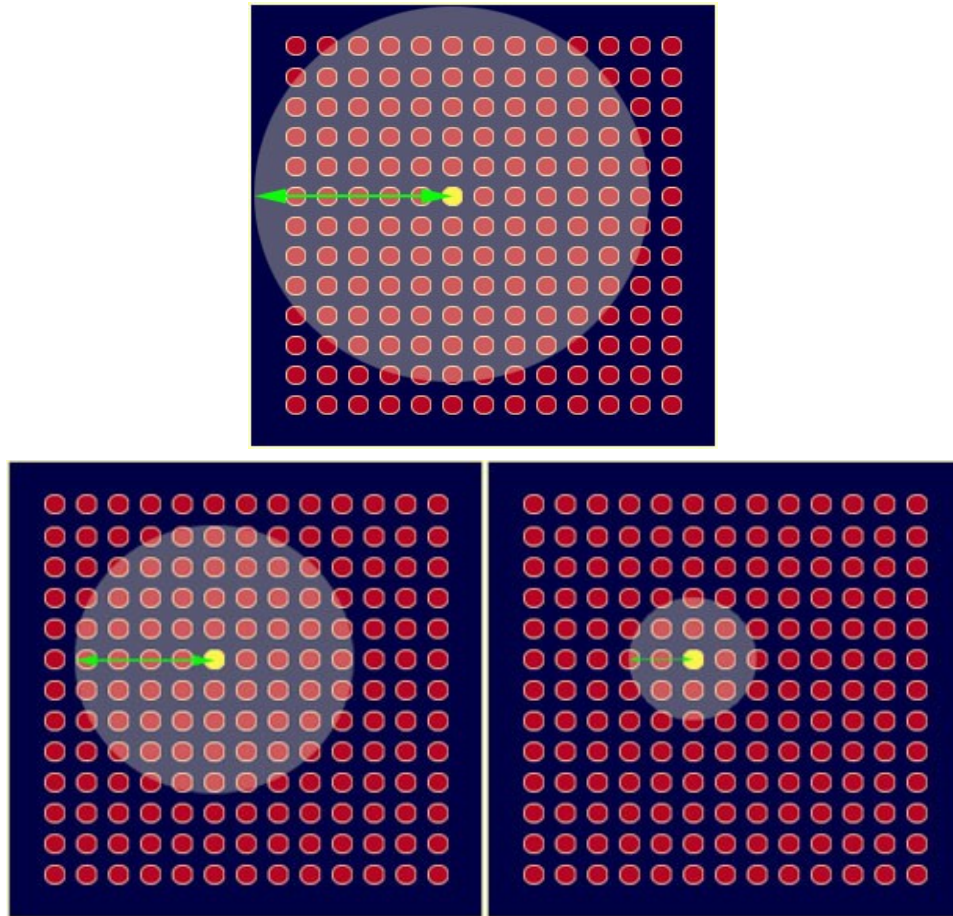


Figure 4

Equation 3a is the main learning function. $W(t+1)$ is the new, 'educated', weight value of the given node. Over time, this equation essentially makes a given node weight more like the currently selected input vector, I . A node that is very different from the current input vector will learn more than a node very similar to the current input vector. The difference between the node weight and the input vector are then scaled by the current learning rate of the SOM, and by $\Theta(t)$.

$\Theta(t)$, Equation 3c, is used to make nodes closer to the BMU learn more than nodes on the outskirts of the current neighborhood radius. Nodes outside of the

neighborhood radius are skipped completely. $distFromBMU$ is the actual number of nodes between the current node and the BMU, easily calculated as:

$$distFromBMU^2 = (bmuI - nodeI)^2 + (bmuJ - nodeJ)^2$$

This can be done since the node network is just a 2-D grid of nodes. With this in mind, nodes on the very fringe of the neighborhood radius will learn some fraction less 1.0. As $distFromBMU$ decreases, $\Theta(t)$ approaches 1.0. The BMU itself will have a $distFromBMU$ equal to 0, which gives $\Theta(t)$ its maximum value of 1.0. Again, this Euclidean distance remains squared to avoid the square root operation.

There exists a lot of variation regarding the equations used with the SOM algorithm. There is also a lot of research being done on the optimal parameters. Some things of particular heavy debate are the number of iterations, the learning rate, and the neighborhood radius. It has been suggested by Kohonen himself, however, that the training should be split into two phases. Phase 1 will reduce the learning coefficient from 0.9 to 0.1, and the neighborhood radius from half the diameter of the lattice to the immediately surrounding nodes. Phase 2 will reduce the learning rate from 0.1 to 0.0, but over double or more the number of iterations in Phase 1. In Phase 2, the neighborhood radius value should remain fixed at 1 (the BMU only). Analyzing these parameters, Phase 1 allows the network to quickly 'fill out the space', while Phase 2 performs the 'fine-tuning' of the network to a more accurate representation [Matt04].

III. Applications

The following are descriptions of applications utilizing SOMs. The first, Color Classification helps demonstrate the concept of SOMs. It is not very practical on its own. However, the framework presented can be used for other extremely pragmatic applications. One of these such useful applications is described in the second section, Image Classification. To go from the Color to Image Classification, all one needs to really change is the weight vector calculation, as the algorithms used are exactly the same.

Color Classification

This is considered the "Hello World" of SOMs. Most tutorials published on SOMs use the color classification SOM as their primary example. This is for a very good reason. By going through this example, the concept of SOMs can be solidly grasped. The reason color classification is fairly easy to understand is because of the relatively small amount of data utilized, as well as the visual aspect of the data.

Color classification SOMs only use three weights per map and input nodes. These weights represent the (r,g,b) triplet for the color. For example, colors may be presented to the network – (1,0,0) for red, (0,1,0) for green, etc. The goal for the network here, is to learn how to represent all of these input colors on it's 2-D grid while maintaining the intrinsic properties of a SOM such as retaining the topological relationships between input vectors. With this in mind, if dark blue and light blue are presented to the SOM, they should end up next to each other on the network grid.

To illustrate the process, we will step through the algorithm for the color classification application. Step 1 is the initialization of the network. Figure 5 shows a newly initialized network. Each square is a node in the network.

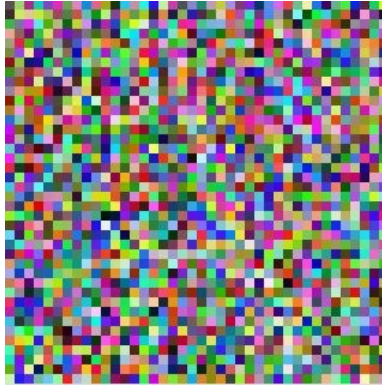


Figure 5

The initialization method used here is to assign a random value between 0.0 and 1.0 for each component (r, g, and b) of each node.

Step 2 is to choose a vector at random from the input vectors. Eight input vectors are used in this example, ranging from red to yellow to dark green. Next, Step 3 goes through every node and finds the BMU, as described earlier. Figure 6 [Ches04] shows the BMU being selected in the 4x4 network. Step 4 of the algorithm calculates the neighborhood radius. This is also shown in Figure 6. All the nodes tinted red are within the radius. Step 5 then applies the learning functions to all of these nodes. It is based on their distance from the BMU. The BMU (dark red) learns the most, while nodes on the outskirts of the radius (light pink) learn the least. Nodes outside of the radius (white) don't learn at all.

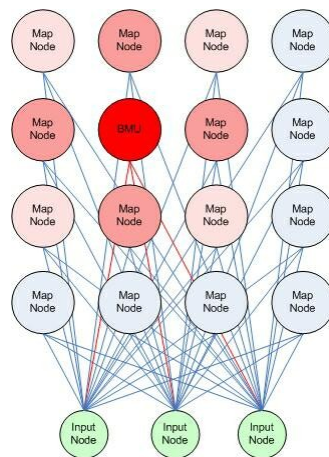


Figure 6

We then go back to Step 2 and repeat. Figure 7 shows a trained SOM,

representing all eight input colors. Notice how light green is next to dark green, and red is next to orange. An ideal map would probably have light blue next to dark blue. This is where the Error Map comes into play, which is described next.

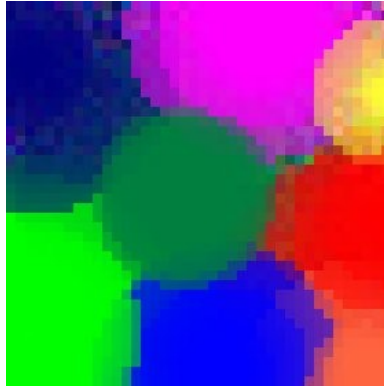


Figure 7

There are two other windows in the color classification application. These are the BMU Window and the Error Map – the bottom left and bottom right windows of the User-Interface, respectively. These windows are not active until after the network is trained. They are not necessary, but they provide some interesting information. First we describe the BMU window. Upon successful SOM training, this window will show small white dots. These white dots represent the N most frequently used BMU nodes, where N is the number of input vectors (unless $N < \#$ of iterations. Then, $N = \#$ of iterations). These nodes have been deemed to be a BMU the most times out of all the nodes in the network, presenting the least distance possible between a map node and the selected input vector for the given iteration. Figure 8 shows this window. Notice how Figure 8 could be placed on top of Figure 7, and the dots would correspond to the centers of the circles.

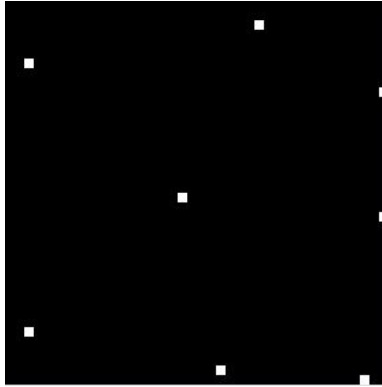


Figure 8

Next is the Error Map. Each time a SOM is trained, it can produce a completely different result given the same input data. This is because the network is initialized with random colors, presenting a unique setup prior to each training session. Also, this occurs because input vectors to be presented to the network are chosen at random. With this in mind, some SOMs may turn out 'better' than others, where 'better' is a measure of how well the topological data is preserved. One method for gauging the superiority of one SOM over another is to calculate an error map [Germ99]. This will give us some numeric value. SOMs with lower values can be said to be 'better' mappings. A SOM with a value of zero would be a perfect mapping, where the entire network is the exact same color.

To calculate an error map, loop through every map node of the network. Add up the distance (not the physical distance, but the weight distance. This is exactly the same as how the BMU is calculated) from the node we're currently evaluating, to each of it's neighbors. Average this distance. Multiply this by 3 (the number of weights used), assuming no square root is used to calculate the distance between adjacent nodes. If the square root operation is used, multiply by $\sqrt{3}$ instead. Assign this value to the node. This gives each map node a nice value between 0.0 and 1.0. These values can then be used as the r=g=b values for each square of the Error Map window. Pure white represents the maximum possible distance between adjacent nodes, while black shows that adjacent nodes are all the same color. Shades of gray in between give an even finer explanation,

with darker grays being a better map than a map with light grays. Figure 9 shows an example. Notice the lines and how they line up with Figure 7.

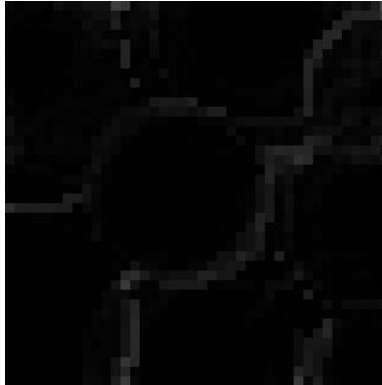


Figure 9

Given the above error map, one might think there are better ways to initialize a given SOM network. As it turns out, there are numerous methods available. For the color SOM, three initializations were tested: Random (already shown), Gradient, and a "Corner" Method. Figure 10 shows the Gradient initialization on the left, and the "Corner" Method on the right. The Gradient initialization simply linearly interpolates (LERP) from black (all weights = 0.0) in the top left corner to white (all weights = 1.0) in the bottom right corner. The "Corners" method places pure black in the top left corner, red in the top right, blue in the bottom left, and green in the bottom right. It then interpolates between these colors in all directions.

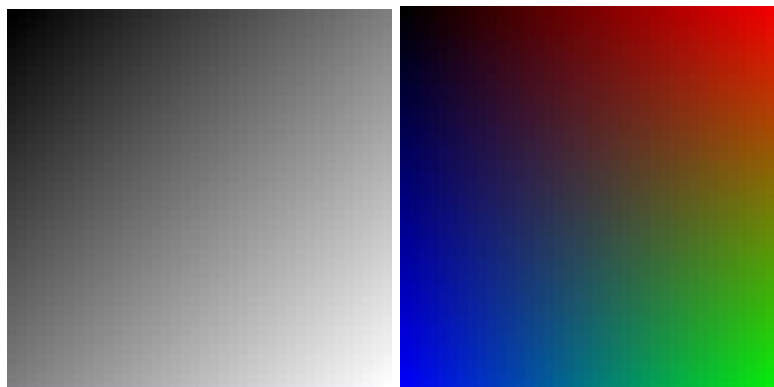


Figure 10

As it turns out, based on numerous (11) runs of the application with a learning

rate of 0.9 for 50 iterations, the "Corners" Method provides the best mapping. The following table summarizes the results:

	<u>Random</u>	<u>Gradient</u>	<u>"Corners"</u>
Avg. Error	29.15393	16.67887	12.68077

Image Classification

The purpose of this application is to show a useful implementation of the SOM algorithm. Given some database of pictures, this SOM will organize them based on content. Utilizing the intrinsic properties of SOMs, related pictures will be found close to each other in the network. Using a database of 200-300 images, a network of size 10x10 is adequate. An ideally spaced network will then give 2-3 images per node [Ches04].

As mentioned earlier, the Color Classification application provides the framework needed for many SOM applications. The key difference between the two applications is the way the weight vectors are calculated. Calculating the weight vectors will be one of the most difficult aspects in SOM applications. If a good weight vector algorithm is not chosen, the SOMs produced by the application will be poor.

For the color classification application, weight vector calculations required no thought. The weight vector was essentially provided by the input vector (the r,g,b values of the input color). These nice vector quantities are not provided by images. So, we must come up with a way to create our own. The two weight vectors used in this application are the Histogram Data Vector and the Area Data Vector [Ches04]. Combined, they will give us a weight vector of length 43.

The histogram data vector is the more complicated of the two. It has a total of 16 weights. This vector essentially represents the brightness of an image. To calculate this vector, we create 16 'bins' that are initially empty. Next, convert the images to gray-scale ($r=g=b$, with byte values from 0-255). We then perform integer division, dividing the pixel value by 16 (this works out nicely as 16^2 is 256). This gives us a number from 0-15 for each pixel. We then go through each

of these computed 0-15 values and 'toss' the pixel into it's associated base-0 bin (no tossing is actually done. Just increment the bin's counter). If you then divide each of these bin counters by the total number of pixels in the image, this normalizes the vector, giving you a fairly unique 16-length vector, with elements ranging from 0.0-1.0, describing the brightness of the image – perfect for use as a SOM weight vector.

While the histogram data vector could be used on it's own, it is better to have as many weight values as possible. This distinguishes images from each other even more, which is useful for the SOM during learning. The other weight vector used in this application is the area data vector. This contains 27 weights, but is extremely simple to calculate. Figure 11 shows the basic idea.

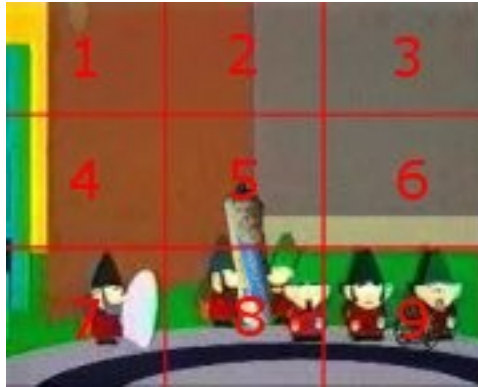


Figure 11

To calculate this vector, we split the image into 9 equal regions. We then calculate the average color for each region. This gives us 27 (r,g,b per square, $3 \times 3 \times 3 = 27$) floats from 0.0-1.0.

Tacking the 27-length area data vector onto our previously calculated 16-length histogram data vector gives a nice 43-length weight vector with all values in the range of 0.0-1.0. This vector is (very close to) a completely unique vector describing an image. We calculate this vector for each image in our database, and pass this data into our SOM as with the color classification application. Using the same algorithm for both applications, we now have a practical program that can sort our pictures based on brightness and colors. This can be expanded even more to detect primitive shapes and textures in an image (see PicSOM).

IV. Conclusion

This paper has just begin to touch on the possibilities of SOMs. A type of neural network, SOMs provide an elegant solution to many arduous problems with large or difficult to interpret data sets. Through their intrinsic properties, such as preserving topological relationships between input data, they allow the visualization of complex data. They are powerful enough to perform extremely computationally expensive operations such as image classification in a relatively short amount of time. Yet, they can be simple enough to code this in a relatively few number of lines, utilizing only a handful of equations. SOMs can be as primitive or as complex as the user desires or requires. SOMs with networks of dimensions greater than 2-D are available (see ET-Map) for even greater flexibility. As more research is done on SOMs, better parameters may be found and more application areas will be created and refined.

V. References

Publications

- Buck03** Mat Buckland... <http://www.ai-junkie.com/ann/som/som1.html>.
- Ches04** Casey Chesnut, <http://www.generation5.org/content/2004/aiSomPic.asp>.
- Egge98** J. Eggermont, "Rule-Extraction and Learning in the BP-SOM Architecture," *Leiden University*, Internal Report IR-98-16 (Masters Thesis), August 1998.
- Fitz97** Don W. Fitzpatrick, "Neural Net Primer: A Brief Introduction to the Use of Neural Networks Suitable for Futures Forecasting," <http://www.jurikres.com/down/nnprimer.txt>, March 1997.
- Germ99** Tom Germano <http://davis.wpi.edu/~matt/courses/soms/>.
- Matt04** James Matthews, <http://www.generation5.org/content/1999/selforganize.asp>.
- Noye92** James L. Noyes, *Artificial Intelligence with Common Lisp: Fundamentals of Symbolic and Numeric Processing*, D.C. Heath, Lexington, MA, 1992.
- Wiki-01** Wikipedia, http://en.wikipedia.org/wiki/Teuvo_Kohonen.

External Links

- ET-Map – "A testbed of 110,000 Internet homepages from the entertainment section of Yahoo! Was gathered by an Internet Spider. An automatic indexing algorithm was applied to the homepages and a... multi-layered Kohonen SOM [was] created," <http://ai.eller.arizona.edu/research/dl/etspace.htm>.
- PicSOM – "An image browsing system based on the Self-Organizing Map," <http://www.cis.hut.fi/picsom/>.