CS545: Assignment 8

Christopher Mullins

December 16, 2009

Contents

1 Introduction **1**

2 Definitions and Setup **1**

3 The SOM Algorithm **2**

4 R Code **3**

5 Color Data **5**

6 Applications **9**

7 Conclusions **10**

1 Introduction

A Self-Organizing Map (SOM) is a type of artificial neural network used to transform a data set of vectors into a set of lower dimensional vectors. The applications of this are wider than one might expect, but this is the heart of its purpose. The data are usually some set of vectors $\{x \in \mathbb{R}^n\}$, but a SOM can work with any set of vectors that have a well-defined distance metric. The canonical examples and demonstrations of SOMs reduce an n-dimensional data set to a 2-D data set.

In section 2, I describe SOMs in more detail. In section 3, I present the traditional iterative algorithm used to construct Self-organizing maps. I include the R code that I use to generate the data, run the SOM algorithm iteratively, and display the results in section 4. Section 5 includes several examples of SOMs constructed from data sets in $[0, 1] \times [0, 1] \times [0, 1]$ interpreted as color data (where each dimension is the intensity of one of red, blue, or green). Applications of SOMs are discussed in section 6, and conclusions are included in section 7

2 Definitions and Setup

The SOM algorithm is an unsupervised learning procedure. This means that while it is "learning," it is not given any feedback about how it is performing, and it is not attempting to learn some mapping of input values to output values. Other examples of unsupervised learning procedures include clustering algorithms (putting vectors from data sets into discrete "buckets") and other techniques that reduce the dimensionality of data, such as Fischer's method and principle component analysis.

In many examples, the original data set is 3D, interpreted as colors ($\{(x_1, x_2, x_3) : 0 \leq x_i \leq 255\}$ – this is the standard way that colors are represented within computers, with the the additional restriction that $x_i \in \mathbb{Z}$, and each component represents the "amount" of red, green, or blue in the resulting color). This

is useful because then a vector in the original data set can be displayed as a color, while a vector in the resulting data set can be a spot in a 2D grid. In this way, we can visually represent both the original vectors and the resulting vectors in a 2D grid of colored cells. An example is shown in figure 1. From here on, SOMs will be assumed to be a mapping from $\{x \in \mathbb{R}^n\}$ to $\{y \in \mathbb{R}^2\}$, although the SOM algorithm is capable of mapping to an arbitrary dimension $m < n$.

It is important to note that SOMs do not explicitly create a function that maps $x \in \mathbb{R}^n \to y \in \mathbb{R}^2$. In most cases, the cells in the 2D grid are "close" to a vector in the original data set, but is almost never an actual original data point. We can define a map such that $x \in \mathbb{R}^n$ is mapped to the vector $y \in \mathbb{R}^2$ that corresponds to the vector in the SOM that is closest to x, but notice that this is not a one-to-one function.

Figure 1: An example of a SOM organizing 3D vectors in a 2D map

3 The SOM Algorithm

The construction of a SOM is an iterative process. The number of iterations used is slightly arbitrary, and is dependent on the size of the data set, the size of the grid that the vectors are being mapped to (which is not necessarily the same size as the original data set), and the nature of the data. The algorithm to construct a SOM is as follows:

1. Initialize the map with random sampling of vectors from the original data (assign a random vector to each cell in the grid).
2. Randomly select a vector $\vec{m}$ from the original data set
3. Find the cell (x, y) in the grid with the shortest Euclidean distance to the selected vector.
4. Update all of the cells (i, j) in the grid according to the following formula:

$$\vec{v}_{i,j}(t+1) = \vec{v}_{i,j}(t) + \alpha(t)h(i,j)(\vec{m} - v_{i,j}(t)) \quad (1)$$

5. Repeat (2) some predetermined number of times.

In equation 1, $\vec{v}_{x,y}(t)$ denotes the value of the vector at position (x, y) at time t). $\alpha(t)$ is a function from $\mathbb{N} \to (0, 1]$ that specifies a "learning rate". When t is small, we want $\alpha(t)$ to be close to 1. When t is near the maximum number of iterations, it should be closer to 0. In the examples I give, α is a linear function with $\alpha(0) = 0.05$ and $\alpha(k) = 0.01$, where k is the maximum number of iterations.

$h(x, y)$ is a "neighborhood function", sometimes referred to by the more mathematical bunch as a smoothing kernel. It is close to 1 when (i, j) is close to the cell (x, y) discovered in step (3). This allows for the ability to update cells close to the chosen one, and have the effect fade out as the distance from the cell increases.

In some implementations, each iteration involves selecting multiple vectors and repeating this process for each vector. Notice that unless the number of vectors selected is equivalent to the number of cells in the grid, this means that if the number of iterations isn't big enough, most of the original data will be under-represented in the grid, even though the grid looks nice.

4 R Code

The examples I plan to run for this paper involve mapping n random color data (3-dimensional) to a 2-dimensional grid with a number of cells equal to the number of points in the data. As such, it's important to have a nice way to visualize a 2D grid filled with colors. This is essentially a $\sqrt{n} \times \sqrt{n}$ bitmap, with each pixel equal to some color assigned by the SOM. Before any iteration has occurred, this is the same bitmap, except colored by a random sampling of the data.

To generate the color data, I use the following code:

```
# Generates n random (R,G,B) triplets, with 0 <= R,G,B <= 1.  Each value
# represents the intensity of the respective color.
generateColorData <- function(n, maxIntensity = 255, gray = FALSE)
{
    if (gray == FALSE)
    {
        # Generate 3*n random numbers between 0 and 255 and scale them
        # appropriately.
        data <- runif(3*n, min = 0, max = maxIntensity) / 255
    }
    else
    {
        # If R = G = B, the resulting color will be a shade of gray.  Generate n
        # random numbers (as done above), but repeat all of them 3 times.
        data <- rep(runif(n, min = 0, max = maxIntensity) / 255, 3)
    }

    return( matrix(data, nrow = n, ncol = 3) )
}
```

Now comes the more important part: plotting the data. CRAN[CRA04] to the rescue again. The `pixmap` package includes a method to plot a bitmap given a matrix of color data. The following code produces a plot of an $n \times 3$ color matrix:

```
# Plots a bitmap given an n x 3 data matrix of (R,G,B) intensity values.  Has
# the ability to plot multiple bitmaps.  Splits the data matrix into n parts,
# puts a maximum of maxrow images per row.
drawColorData <- function(data, n = 1, maxrow = 10)
{
    # Compute number of rows and columns needed
    nr  <- ceiling(n / maxrow)
    nc  <- min(maxrow, n)

    # Compute size of individual images
    t   <- nrow(data) / n

    # Set up the plot device so that we can plot nr*nc images
    dev <- par(mfrow = c(nr, nc), mar = c(0, 0, 0, 0))

    for (i in 0:(n-1))
    {
```

```
        # Compute start/stop positions within the data matrix
        start <- i*t + 1
        stop  <- (i + 1)*t

        # Plot the ith image
        plot(pixmapRGB(data[start:stop,], nrow = sqrt(t), ncol = sqrt(t)))
    }

    # Reset plot device
    par(dev)
}
```

Note that this code has the ability to split up the data into an arbitrary number (n) of images. This will make plotting the progression of the SOM algorithm easier in the future.

Instead of writing my own SOM in R, I chose to see if someone else already had. As it turns out, there's a package in the Comprehensive R Archive Network (CRAN) [CRA04] that implements the SOM algorithm. As this is more of an exploratory paper, I wanted figures that showed what the 2D grid looked like at each iteration. Unfortunately, this implementation does not offer such a feature. It ran for a number of iterations, and returned the result. Fortunately, as the parameters for a given iteration are determinable from the parameters of a previous iteration (with this implementation, anyway), it is possible to write a wrapper for the existing R project to force it to keep track of its progression.

```
# Using the CRAN SOM package, iteratively train a Self-Organizing Map, and keep
# a history of the SOM at each iteration.  (By default, the som function runs
# through all iterations with no way to access history).
iterativeTrain <- function(data, iterations = 10, updates = FALSE,
    alpha = c(0.05, 0.01))
{
    if (updates == TRUE)
        drawColorData(data)

    n <- sqrt(nrow(data))
    g <- somgrid(n,n)

    # Construct a list of the alpha values to use at each iteration.  This is a
    # linear decrease from the max to the min over n iterations.
    if (length(alpha) != (iterations + 1))
        a <- seq(max(alpha), min(alpha), length = (iterations + 1))
    # Otherwise, just use alpha as it is.
    else
        a <- alpha

    # Run one iteration of the SOM learning algorithm.
    s <- som(data, grid = g, toroidal = FALSE, rlen = 1, alpha = alpha[1:2])
    data <- rbind(data, s$codes)

    if (updates == TRUE)
        drawColorData(data, 2)

    for (i in 2:iterations)
    {
        # Run an iteration of the SOM learning algorithm.
        s <- som(data, grid = g, toroidal = FALSE, rlen = 1,
            init = s$codes, alpha = a[i:(i+1)], keep.data = FALSE
        )
```

```
        # Store the results in the data matrix.
        data <- rbind(data, s$codes)

        if (updates == TRUE)
            drawColorData(data, i+1)
    }

    # Plot the results
    drawColorData(data, iterations + 1)
}
```

Notice since the wrapper is responsible for passing α values to the real SOM function, it becomes possible to experiment with other ways to modify the α_is (for example, an exponential decay).

5 Color Data

Given a random set of color data, we might expect a self-organizing map (with "good" parameters) to produce something similar to a color wheel (example shown in figure 2), where we observe a somewhat continuous change from one color to another.

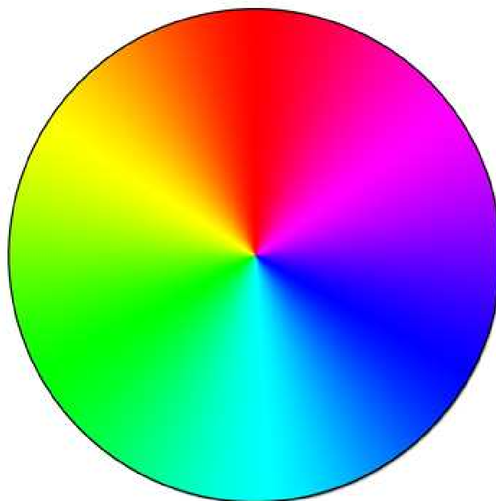

Figure 2: Example of a "continuous" color wheel.

To get an idea of what the progression looks like, I include two independent runs of the SOM algorithm on similar data sets in figures 3 and 4. The SOM code is provided in an R package [Yan04]. The data set is 1024 vectors in $\mathbb{R}^3$ (which can conveniently be represented as a 32x32 image of RGB colors). The SOM grid is 32x32. A rendition of the original data is shown in the first square.

Notice that as the algorithm progresses, the resulting grid gets increasingly homogeneous. If the data is homogeneous, then this is desirable. However, if the data is sporadic (such as completely random data with high variation), this is potentially bad thing, especially if it is obvious that there are samples in the data that are not well represented in the SOM. This suggests we should use fewer iterations, or should adjust α or h so that the learning is more drastic (otherwise vectors that are not well represented in the grid will just get washed out). The result of making α a linear function from 1 to 0.001 (instead of 0.05 to 0.01) is shown in figure 5. An example of the SOM algorithm running using an exponential decay of the α_is is shown in

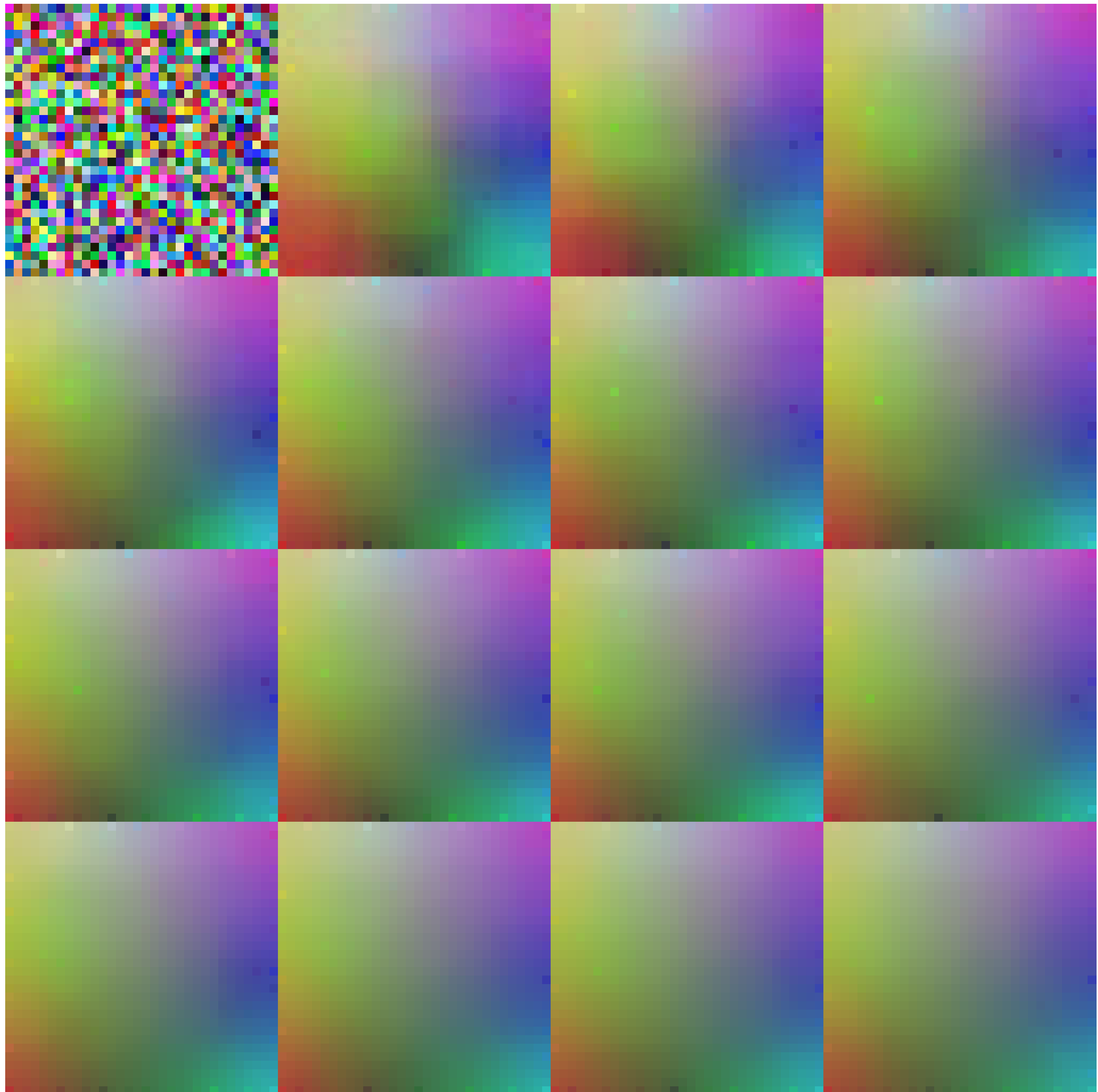

Figure 3: Progression of construction of a SOM (1). Notice that the whole grid looks slightly "washed out," and does not include more saturated colors.

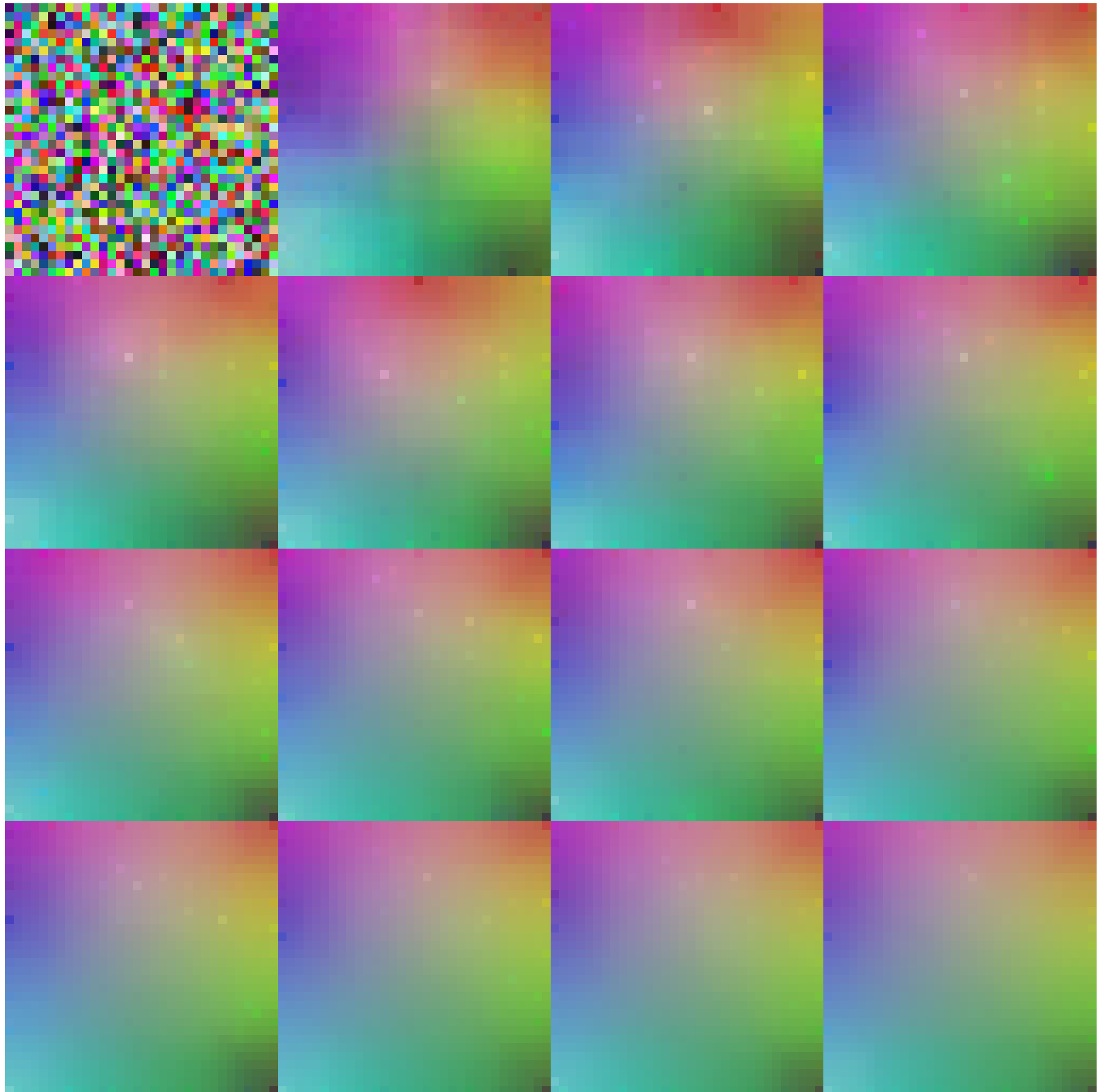

Figure 4: Progression of construction of a SOM (2).

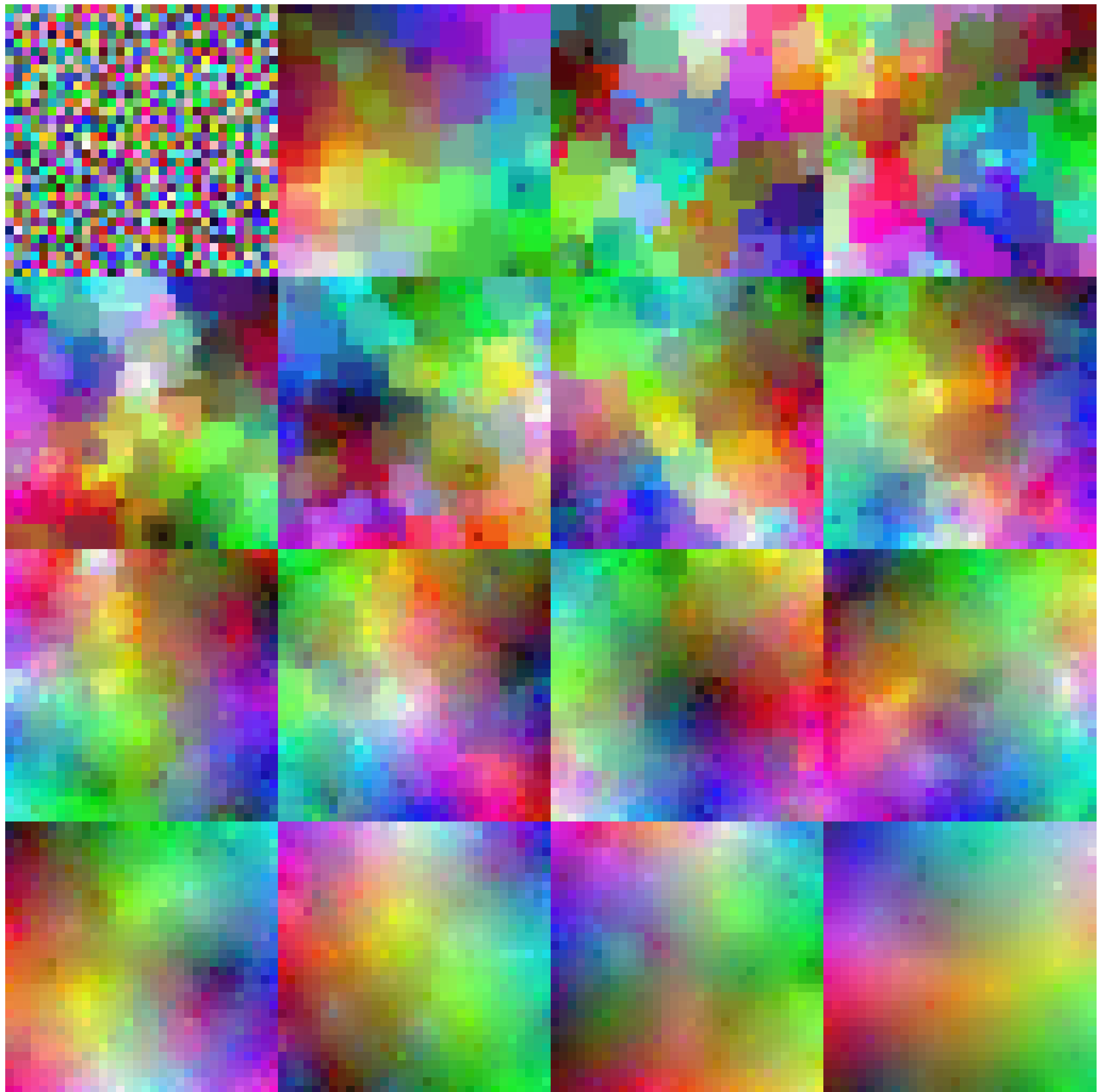

Figure 5: Progression of a SOM where α is a linear function with bounds 1 and 0.001. Notice that the result is not nearly as homogeneous, but is much more representative of the original data.

figure 6. Notice that this approach seems to be composed of more unsaturated colors, but appears to be an improvement over the initial approach despite remaining reasonably homogeneous.

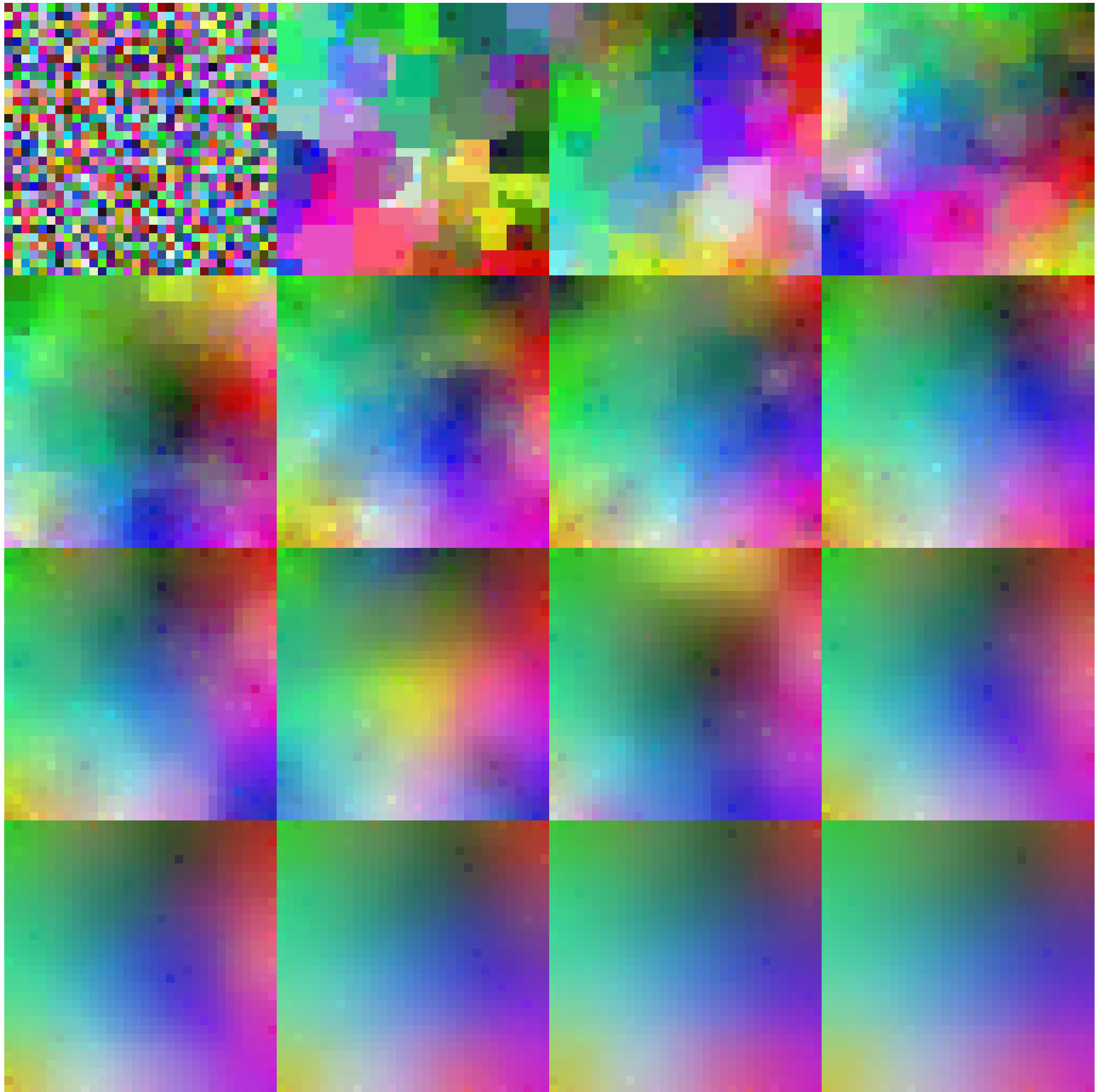

Figure 6: Progression of a SOM where $\alpha_{i+1} = 0.7\alpha_i$.

6 Applications

At this point, applications beyond making pretty pictures might not be obvious. Despite this, SOMs can be used for a variety of interesting and helpful things. First, SOMs grant humans the ability to visualize an approximation of high-dimensional data in a simple way. After a SOM is constructed, it can be re-applied to the data in order to see which data points are similar to one another. One example used in literature is the US congress voting history. Given a series of votes (yes/no on certain bills, so we're mapping vectors of discrete values to $\mathbb{R}^2$), we construct a SOM. Because we certain things about the members of congress (political affiliation, age, etc), we can assign some coloring to the cells and see if similar voters are similar in other ways.

So, in general, SOMs are useful if you have a set of measurements, and set of information about each measurement. You can group together the measurements that are close to one another using a SOM, then see if any of the other measurements are correlated with the measurements being close together.

7 Conclusions

In conclusion, SOMs are interesting (lots of pretty pictures!), and are a very powerful tool for determining which properties relevant to a data set are correlated with the values of the data. The most difficult part of my efforts was understanding the SOM algorithm.

References

[CRA04] CRAN. The comprehensive r archive network, 2004.

[Hay99] Simon Haykin. *Neural Networks.* Prentice Hall, Englewood Cliffs, 1999.

[Koh01] Teuvo Kohonen. *Self-Organizing Maps.* Springer, Berlin, 2001.

[Yan04] Jun Yan. Self-organizing map r package, 2004.