Section 7 Self-organizing maps (SOM)

A self-organizing map (SOM) can be viewed as a constrained version of *K*-means clustering. It is the discrete, neural network version of principal curves and surfaces (which are the nonlinear extensions of principal components). Original high-dimensional observations are mapped down onto a lower-dimensional coordinate system hence it can be considered as a dimension-reducing mechanism. We will map onto a two-dimensional coordinate system.

K prototype vectors (similar to the centroids in *k*-means) $m_j \in R^p$ (where *p* is the dimension of the data space - called the *input* space) are scattered at random or initialized to lie in the two-dimensional principal component plane of the data in the input space.

The *K* prototype vectors are also placed on a two-dimensional grid (called the *output* space). We will use a hexagonal grid although a rectangular grid may also be used. See Figure 1.

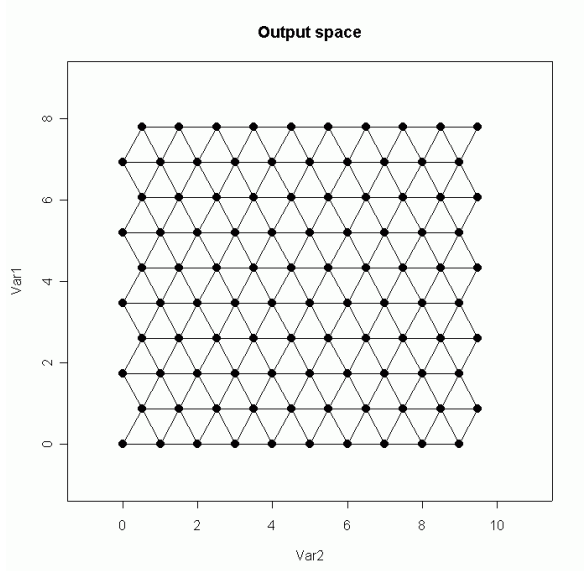

Figure 1.

Each of the *K* prototypes can be parametrized with respect to an integer coordinate pair $l_j \in Q_1 \times Q_2$ where $Q_1 = \{1, 2, ..., q_1\}$, $Q_2 = \{1, 2, ..., q_2\}$ and $K = q1 \cdot q2$.

We find the closest prototype m_b (referred to as the Best Matching Unit - BMU) to x_i (an observation) in Euclidean distance in R^p and then move all neighbours m_j of m_b toward x_i, using

$$m_j \leftarrow m_j + \alpha(t)\, h(\| r_j - r_b \|)(x_i - m_j)$$

where $\| r_j - r_b \|$ is the distance between the two prototype vectors.

Note: The neighbours of m_b are defined to be all m_j that close in the **output** space (less than some threshold or neighbourhood radius σ). The distance is defined in the space $Q_1 \times Q_2$ of integer topological coordinates of the prototypes, rather than in the feature space R^p.

Updating moves the prototypes closer to the data and maintains a smooth two-dimensional spatial relationship between the prototypes. Typically $\alpha(t)$ is decreased from 1.0 to 0.0 and *r* is decreased from some starting value to 1 over a few thousand iterations. The neighbourhood function $h(\| r_j - r_b \|)$ is used to give more weight to prototypes m_j that are closer to m_b than to those further away. If we take σ small

enough so that each neighborhood contains only one prototype vector, the spatial connection between prototypes is lost and SOM is equivalent to *k*-means clustering.

Load in some useful files:

```
> drive <- "D:"
> code.dir <- paste(drive, "DATA/Data Mining R-Code", sep="/")
> data.dir <- paste(drive, "DATA/Data Mining Data", sep="/")
> #
> source(paste(code.dir, "xy_grid.r", sep="/"))
> source(paste(code.dir, "hexagon.r", sep="/"))
> source(paste(code.dir, "pt_to_array.r", sep="/"))
> source(paste(code.dir, "array2arraydist.r", sep="/"))
> source(paste(code.dir, "It_Train_Som.r", sep="/"))
> source(paste(code.dir, "Bat_Train_Som.r", sep="/"))
> source(paste(code.dir, "creategaussians.r", sep="/"))
> source(paste(code.dir, "HexConnect.r", sep="/"))
> library(stats)
```

and some data:

```
> source(paste(data.dir, "SomData.dat", sep="/"))
```

We are going to be concerned about which prototype (codebook) vectors are closest to each datum. The following two functions allow us to do that:

```
> Array2ArrayDist <- function(Source, Data) {
>    #=======================================
>    # Find the distance^2 from the
>    # Source vectors to Data vectors
>    #=======================================
>    n.Source <- dim(Source)[1]
>    n.Data <- dim(Data)[1]
>       # 2*the datum for use in cosine law
>    Dx2 <- t(2*Data)
>       # Length^2 of each datum vector
>    Data.const <- apply(Data*Data, 1, sum)
>       # Length^2 of each source vector
>    Source.const <- apply(Source*Source, 1, sum)
>       # Source^2 - 2*Source*Data + Data^2
>       # Note that the first matrix has one col. for each datum.
>    matrix(Source.const, n.Source, n.Data) - Source%*%Dx2 +
>                matrix(Data.const, n.Source, n.Data, byrow=T)
> }

> min.dist <- function(pts, array) {
>       # Find the minimum distance from source vectors
>       # to the points in an array
>       # Get the distance (has one col. of distances for each datum
>    Dist <- Array2ArrayDist(pts, array)
>       # min.index gives the number of the source point
>       # closest to eack data point.
>    min.dist <- apply(Dist, 2, min)
>    min.index <- apply(Dist, 2, order)[1,]
>    return(min=min.dist, where=min.index)
```

```
> }
```

We are using two dimensional vectors to illustrate the concepts (from the `SomData` file).

```
> Z <- cbind(X, Y)
> pv <- cbind(X1, Y1)
```

While it is possible to use rectangular grids in our output space, we will use the hexagonal grids.

```
> #=============================================
> # for the hexagonal lattice, find the x-y positions
> # for the grid
> #=============================================
> get.hex.grid <- function(grid.size) {
>       # Create a rectangular grid of the required size
>       # expand.grid creates an array of grid points
>       # corresponding to specifies x and y values.
>       # The first col. changes more rapidly than the second.
>    hex.pos <- expand.grid(1:grid.size[2],1:grid.size[1])- c(1,1)
>       # Swap the cols so the second changes faster.
>    hex.pos <- hex.pos[,c(2,1)]
>       # Shift every other row by 0.5 to create hexagon
>    hex.pos[,1] <- hex.pos[,1]+rep(c(0,.5), (grid.size[2]+1)/2)[1:grid.size[2]]
>       # make distances to all neighboring units equal
>    hex.pos[,2] <- hex.pos[,2]*sqrt(0.75)
>    return(hex.pos)
> }
> #=======================================
> # Given a set of points (2d) plot the points
> # and connect to all neighbours.
> #=======================================
> connect.Map <- function(width, height, data, YL=c(0,1), XL=c(0,1), main="") {
>    plot(data, pch=16, cex=1.5, xlim=XL, ylim=YL, main=main)
>       # Right zig-zag
>    for (i in 1:width) {
>       lines(data[(1:height)+(i-1)*height, ])
>    }
>       # Horizontal
>    for (i in 0:(height- 1)) {
>       lines(data[seq(1,(width*height), by=height) + i, ])
>    }
>       # Left zig-zag
>    for (i in 1:(width-1)) {
>       lines(data[(1:height)+rep(c(height,0),
>             (height+1)/2)[1:height]+(i-1)*height, ])
>    }
> }
> grid.size <- c(10, 10)
> lattice.points <- prod(grid.size)      # Number of points on lattice
```

We determine the hexagonal lattice points

```
> H.coords <- get.hex.grid(grid.size)
```

and plot and connect them.

```
> x11(width=20, height=10)
> split.screen(c(1,2)) # split display into two screens
> connect.Map(10, 10, H.coords,c(-1,9),c(-1,11), main="Output space")
```

```
> screen(2)
> connect.Map(10, 10, pv, main="Input space")
> points(Z[,1],Z[,2],col="red",pch="+")
> close.screen(all = TRUE)
```

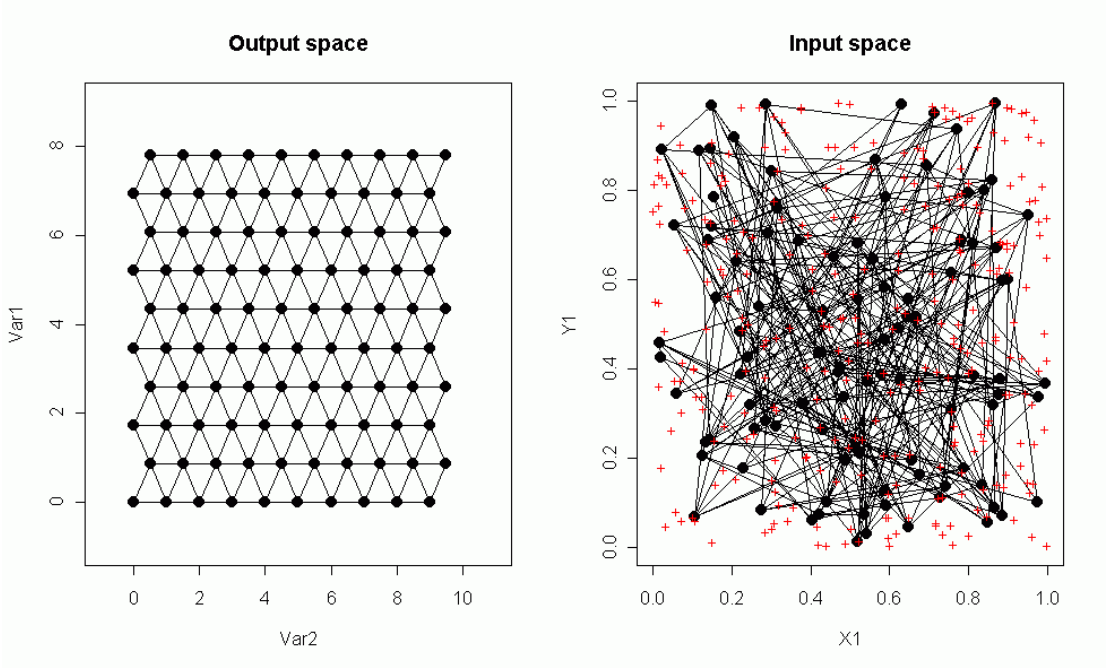

Figure 2.

We need to determine neighbours in the output space so we use the following which determines those vectors that are immediate neighbours of any given vector.

```
> f.H.neighbours <- function (r, height, width) {
>    offset <- -1
>    if (r[1]%%2 == 0) {
>       offset <- 1
>    }
>    res <- matrix(c(r[1]+1,r[2]+offset, r[1]+1,r[2],
>                    r[1],r[2]-1, r[1],r[2]+1,
>                    r[1]-1,r[2]+offset, r[1]-1,r[2]), ncol=2,byrow=T)
>    res <- res[res[,1] != 0,]
>    res <- res[res[,1] <= height,]
>    res <- res[res[,2] <= width,]
>    res[res[,2] != 0,]
> }
```

This gets the row and column positions in a grid based on its linear index.

```
> split <- function (x, base=10) {
>    tens <- (x-1)%/%base
>    units <- x - tens*base
>    cbind(units, tens+1)
> }
> split(36, 10)
     units
[1,]     6 4
> width <- 10
> height <- 10
```

```
> pos <- 36
> x <- split(pos, height)
> (n.pos <- f.H.neighbours(split(pos, height), height, width))
     [,1] [,2]
[1,]    7    5
[2,]    7    4
[3,]    6    3
[4,]    6    5
[5,]    5    5
[6,]    5    4
> (n.ind <- (n.pos[,2]-1)*height+n.pos[,1])      # Convert from grid to linear
[1] 47 37 26 46 45 35
```

We can display the neighbours (green points) of a prototype vector (yellow point) in both the output and input space.

```
> x11(width=20, height=10)
> pv <- cbind(X1, Y1)
> split.screen(c(1,2)) # split display into two screens
> connect.Map(10, 10, H.coords,c(-1,9),c(-1,11), main="Output space")
> points(H.coords[pos,1], H.coords[pos,2], pch=16, col="yellow")
> points(H.coords[n.ind,1], H.coords[n.ind,2], pch=16, col="green")
> screen(2)
> connect.Map(10, 10, pv, main="Input space")
> points(pv[pos,1], pv[pos,2], pch=16, col="yellow")
> points(pv[n.ind,1], pv[n.ind,2], pch=16, col="rgreen")
> points(Z[,1],Z[,2],col="red",pch="+")
> close.screen(all = TRUE)
```

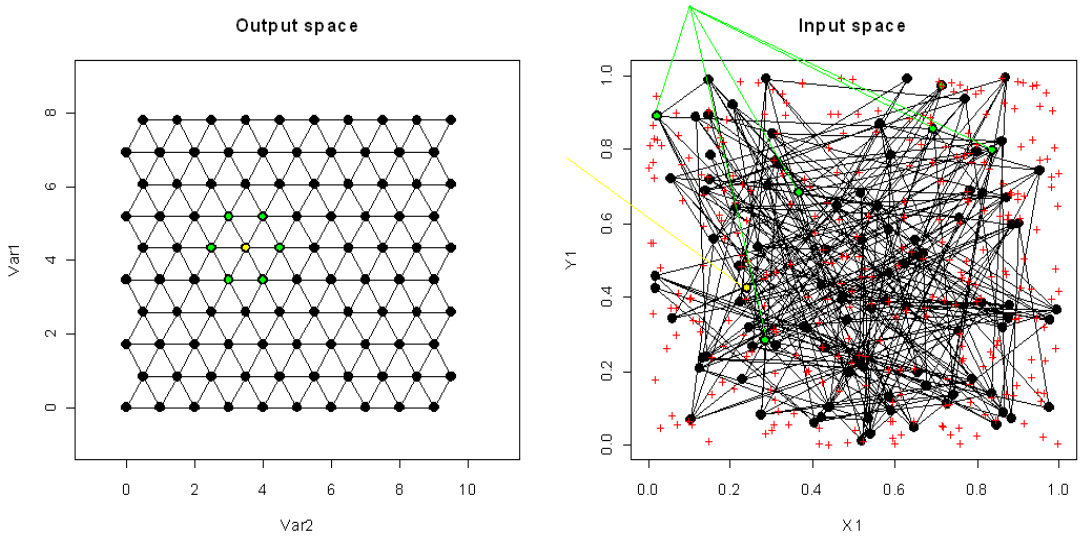

Figure 3.

We need to know the distance between a prototype vector and all other prototype vectors to enable us to weigh the effect of changes to one vector on the others. As one vector changes to move to the center of those data points that were closest to it (similar to *k* –means), its neighbours are influenced to a lesser extent. For example the influence may be restricted to those vectors within a certain distance or neighbourhood radius (called the bubble, it is constant out to a certain radius and zero elsewhere) or the influence may drop off exponentially (i.e. a two dimensional Gaussian window). We will use the Gaussian window

$$e^{-\frac{\|r_j - r_b\|^2}{2\sigma^2(t)}}$$

where $\|r_j - r_b\|$ is the distance between the current prototype vector and the BMU on the output map and σ is the neighbourhood radius.

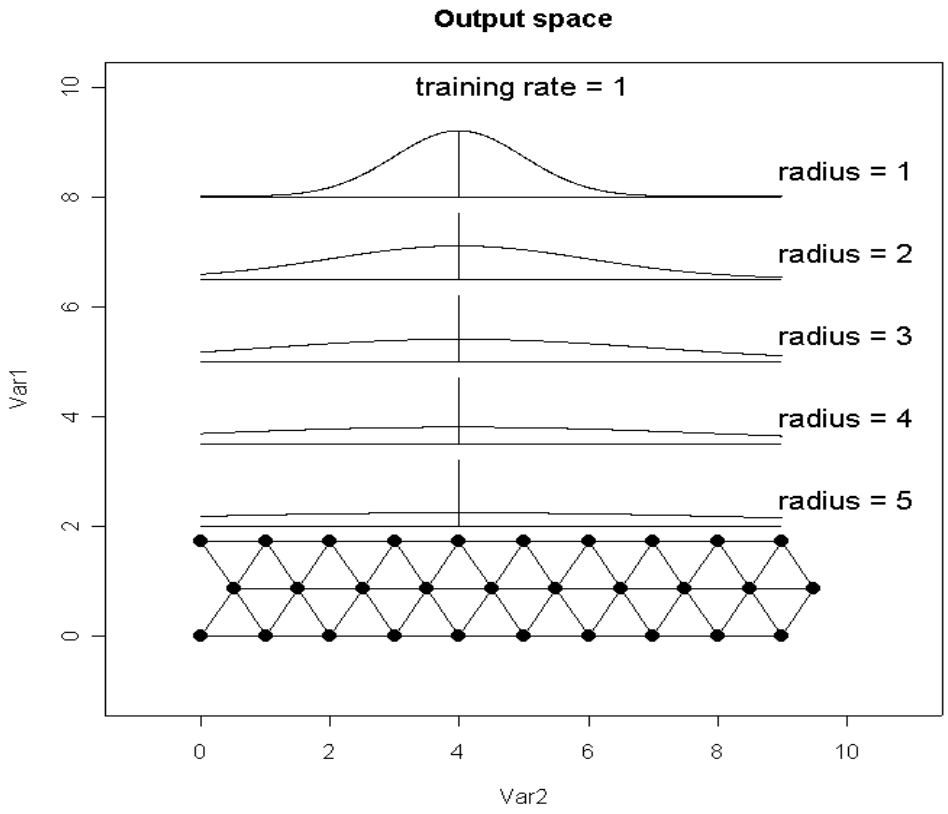

Figure 4.

With a large radius the remote PVs move (as a fraction) almost as much as the BMU while for small radius they move very little.

We need to have the distance (actually distance squared) between the prototype vectors on the output map.

```
> (H.distance <- as.matrix(dist(H.coords))^2)
      1   2   3   4   5  6   7   8   9  10  11  12  13 14 15 16 17 18  19  20
1     0   1   3   7  12 19  27  37  48  61   1   3   4  9 13 21 28 39  49  63
2     1   0   1   3   7 12  19  27  37  48   1   1   1  4  7 13 19 28  37  49
3     3   1   0   1   3  7  12  19  27  37   4   3   1  3  4  9 13 21  28  39
4     7   3   1   0   1  3   7  12  19  27   7   4   1  1  1  4  7 13  19  28
5    12   7   3   1   0  1   3   7  12  19  13   9   4  3  1  3  4  9  13  21
6    19  12   7   3   1  0   1   3   7  12  19  13   7  4  1  1  1  4   7  13
7    27  19  12   7   3  1   0   1   3   7  28  21  13  9  4  3  1  3   4   9
8    37  27  19  12   7  3   1   0   1   3  37  28  19 13  7  4  1  1   1   4
9    48  37  27  19  12  7   3   1   0   1  49  39  28 21 13  9  4  3   1   3
10   61  48  37  27  19 12   7   3   1   0  61  49  37 28 19 13  7  4   1   1
11    1   1   4   7  13 19  28  37  49  61   0   1   3  7 12 19 27 37  48  61
12    3   1   3   4   9 13  21  28  39  49   1   0   1  3  7 12 19 27  37  48
13    4   1   1   1   4  7  13  19  28  37   3   1   0  1  3  7 12 19  27  37
14    9   4   3   1   3  4   9  13  21  28   7   3   1  0  1  3  7 12  19  27
15   13   7   4   1   1  1   4   7  13  19  12   7   3  1  0  1  3  7  12  19
16   21  13   9   4   3  1   3   4   9  13  19  12   7  3  1  0  1  3   7  12
17   28  19  13   7   4  1   1   1   4   7  27  19  12  7  3  1  0  1   3   7
```

```
18   39  28  21  13   9  4   3   1   3   4  37  27  19 12  7  3  1  0   1   3
19   49  37  28  19  13  7   4   1   1   1  48  37  27 19 12  7  3  1   0   1
20   63  49  39  28  21 13   9   4   3   1  61  48  37 27 19 12  7  3   1   0
  .
  .
  .
90  133 112 109  91  91 76  79  67  73  64 117  97  93 76 75 61 63 52  57  49
91   81  73  84  79  93 91 108 109 129 133  64  57  67 63 76 75 91 93 112 117
92   91  81  91  84  97 93 109 108 127 129  73  64  73 67 79 76 91 91 109 112
93   84  73  81  73  84 79  93  91 108 109  67  57  64 57 67 63 76 75  91  93
94   97  84  91  81  91 84  97  93 109 108  79  67  73 64 73 67 79 76  91  91
95   93  79  84  73  81 73  84  79  93  91  76  63  67 57 64 57 67 63  76  75
96  109  93  97  84  91 81  91  84  97  93  91  76  79 67 73 64 73 67  79  76
97  108  91  93  79  84 73  81  73  84  79  91  75  76 63 67 57 64 57  67  63
98  127 108 109  93  97 84  91  81  91  84 109  91  91 76 79 67 73 64  73  67
99  129 109 108  91  93 79  84  73  81  73 112  93  91 75 76 63 67 57  64  57
100 151 129 127 108 109 93  97  84  91  81 133 112 109 91 91 76 79 67  73  64
  .
  .
  .
```

Rather than use all the data in the initial training we use a random sample of the data (100 cases in the following). The distance between each of these cases and each prototype vector is found (`Dx`) and the minimum distance is computed to give the best matching unit (BMU). The update amount `h` is found as

$$h = \alpha(t)\, e^{-\frac{\|r_j - r_b\|^2}{2\sigma^2(t)}}$$

and the prototype vectors are changed by

$$\alpha(t)\, e^{-\frac{\|r_j - r_b\|^2}{2\sigma^2(t)}}\, (x_i - m_j)$$

```
> iterative.train.SOM <- function(data, prototype, radius,
>                trainlen, alpha, Coords, H.distance) {
>    #=====================================
>    # data - the data on which the SOM is to be run
>    # prototype - the prototype vectors
>    # radius - the radius (in output space)
>    # train.steps
>    # alpha
>    # Coords
>    # H.distance
>    #=====================================
>    update.size <- 100
>       # number of cases in data
>    data.len <- dim(data)[1]
>    steps <- 1:trainlen
>    samples <- sample(1:data.len, update.size)
>    radius <- radius^2
>    radius[radius == 0] <- 2.2204e-016 # ~Machine eps
>    for (t in 1:trainlen) {
>       x <- data[samples[t], ]       # pick one sample vector
>       Dx <- t(t(prototype) - x)     # get the difference from prototype vectors
>       bmu <- it.min.dist(x, prototype)$where    # Find the closest to the case
>          # For the BMU get an adjustment to the values for the prototype vectors
>          # based on the proximity to the BMU in the output space
```

```
>       h <- alpha[t] * exp(-H.distance[,bmu]/(2*radius[t]))
>          # update prototype
>       prototype <- prototype - h*Dx
>    }
>    return(prototype)
> }
```

(There is a second version of this file `N.iterative.train.SOM` that is used in the demonstration of SOM. The code is also in "it_train_som.r".)

The following finds the minimum distance from a point to an array and returns the minimum along with the index of the minimum.

```
> it.min.dist <- function(pt, array) {
>    D <- sqrt(apply((t(t(array)-as.vector(pt,mode="numeric")))^2, 1, sum))
>    return(list(min=min(D),where=which(min(D)==D)))
> }
> pv <- cbind(X1, Y1)
```

To illustrate the behaviour of an iterative SOM, we create a 5×5 hexagonal lattice

```
> grid.size <- c(5, 5)
> lattice.points <- prod(grid.size) # Number of points on lattice
> H.coords <- get.hex.grid(grid.size)
```

We can show the squared distances between the various prototype vectors in the output space -

```
> (H.distance <- as.matrix(dist(H.coords))^2)
    1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
1   0  1  3  7 12  1  3  4  9 13  4  7  7 13 16  9 13 12 19 21 16 21 19 27 28
2   1  0  1  3  7  1  1  1  4  7  3  4  3  7  9  7  9  7 12 13 13 16 13 19 19
3   3  1  0  1  3  4  3  1  3  4  7  7  4  7  7 12 13  9 13 12 19 21 16 21 19
4   7  3  1  0  1  7  4  1  1  1  9  7  3  4  3 13 12  7  9  7 19 19 13 16 13
5  12  7  3  1  0 13  9  4  3  1 16 13  7  7  4 21 19 12 13  9 28 27 19 21 16
6   1  1  4  7 13  0  1  3  7 12  1  3  4  9 13  4  7  7 13 16  9 13 12 19 21
7   3  1  3  4  9  1  0  1  3  7  1  1  1  4  7  3  4  3  7  9  7  9  7 12 13
8   4  1  1  1  4  3  1  0  1  3  4  3  1  3  4  7  7  4  7  7 12 13  9 13 12
9   9  4  3  1  3  7  3  1  0  1  7  4  1  1  1  9  7  3  4  3 13 12  7  9  7
10 13  7  4  1  1 12  7  3  1  0 13  9  4  3  1 16 13  7  7  4 21 19 12 13  9
11  4  3  7  9 16  1  1  4  7 13  0  1  3  7 12  1  3  4  9 13  4  7  7 13 16
12  7  4  7  7 13  3  1  3  4  9  1  0  1  3  7  1  1  1  4  7  3  4  3  7  9
13  7  3  4  3  7  4  1  1  1  4  3  1  0  1  3  4  3  1  3  4  7  7  4  7  7
14 13  7  7  4  7  9  4  3  1  3  7  3  1  0  1  7  4  1  1  1  9  7  3  4  3
15 16  9  7  3  4 13  7  4  1  1 12  7  3  1  0 13  9  4  3  1 16 13  7  7  4
16  9  7 12 13 21  4  3  7  9 16  1  1  4  7 13  0  1  3  7 12  1  3  4  9 13
17 13  9 13 12 19  7  4  7  7 13  3  1  3  4  9  1  0  1  3  7  1  1  1  4  7
18 12  7  9  7 12  7  3  4  3  7  4  1  1  1  4  3  1  0  1  3  4  3  1  3  4
19 19 12 13  9 13 13  7  7  4  7  9  4  3  1  3  7  3  1  0  1  7  4  1  1  1
20 21 13 12  7  9 16  9  7  3  4 13  7  4  1  1 12  7  3  1  0 13  9  4  3  1
21 16 13 19 19 28  9  7 12 13 21  4  3  7  9 16  1  1  4  7 13  0  1  3  7 12
22 21 16 21 19 27 13  9 13 12 19  7  4  7  7 13  3  1  3  4  9  1  0  1  3  7
23 19 13 16 13 19 12  7  9  7 12  7  3  4  3  7  4  1  1  1  4  3  1  0  1  3
24 27 19 21 16 21 19 12 13  9 13 13  7  7  4  7  9  4  3  1  3  7  3  1  0  1
25 28 19 19 13 16 21 13 12  7  9 16  9  7  3  4 13  7  4  1  1 12  7  3  1  0
```

What we will do in this example is put the PVs in a circle and take a point near the middle as the sample point.

In the iterative SOM, a random set of points is selected (5 in the example - many more in practice) and each point is 'presented' to the PVs which move in accordance with the update function.

After all the points have been presented, the radius is decreased and a new random sample is selected and the process is repeated until the radius is reduced to 1

We will use the neighbourhood radius

```
> (r <- 4*(1-(1:5)/5)+1)
[1] 4.2 3.4 2.6 1.8 1.0
```

Note: 'distance' with a number refers to squared distance; (1) refers to the squared distances shown in blue on the mesh.

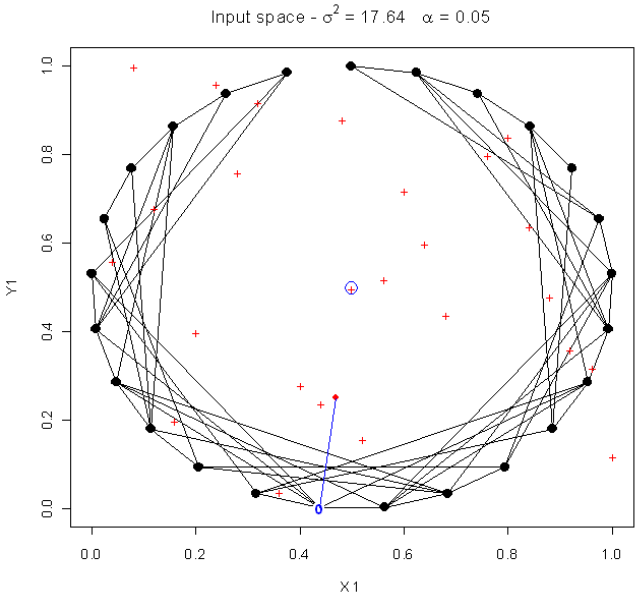

Figure 5. Initial configuration with BMU (0) moving half the distance towards the sample vector.

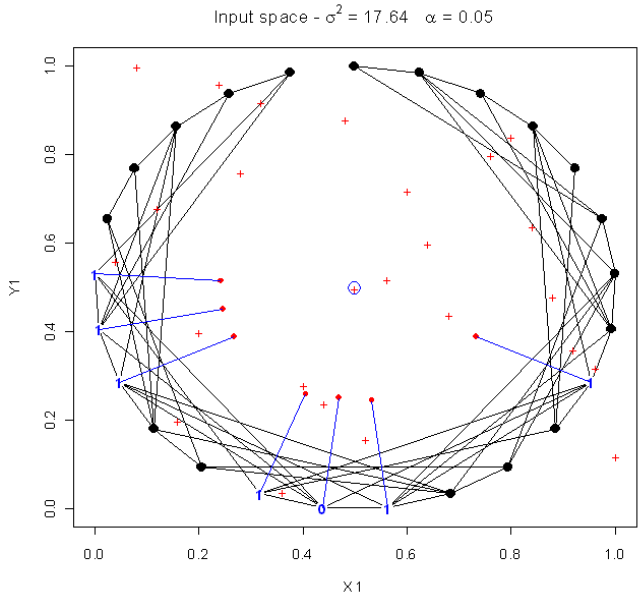

Figure 6. The nearest neighbours (1) moving towards the sample vector. Because of the large radius it also moves about 1/2 the distance.

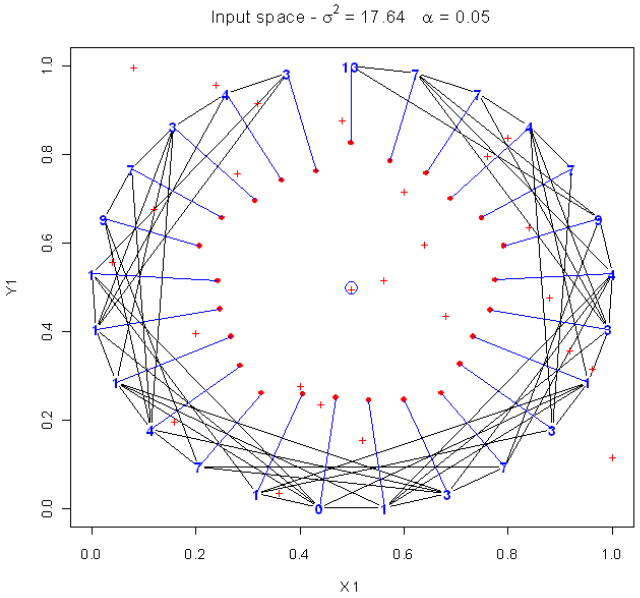

Figure 7. After 1 pass, the new positions for the PVs are shown as red dots. Note that even the distant (13) PVs move about 1/3 of the way

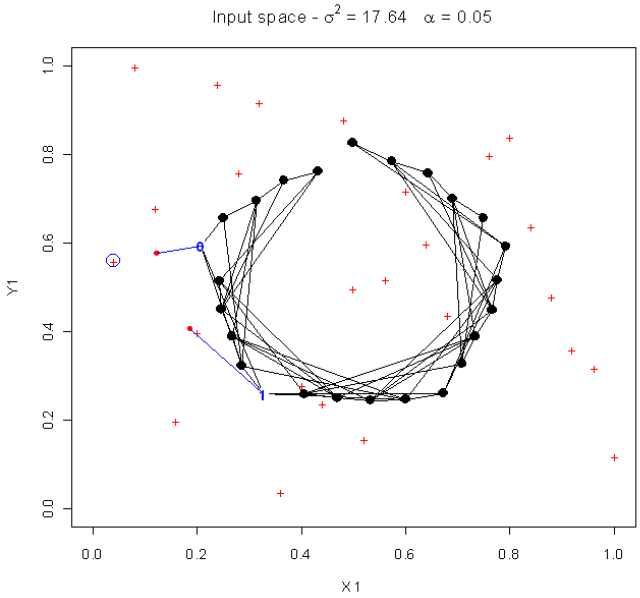

Figure 8. At start of pass 2, the PV at 'distance' 1 has moved further than the BMU. The amount of movement comes from the $(x_i - m_j)$ part of the update,

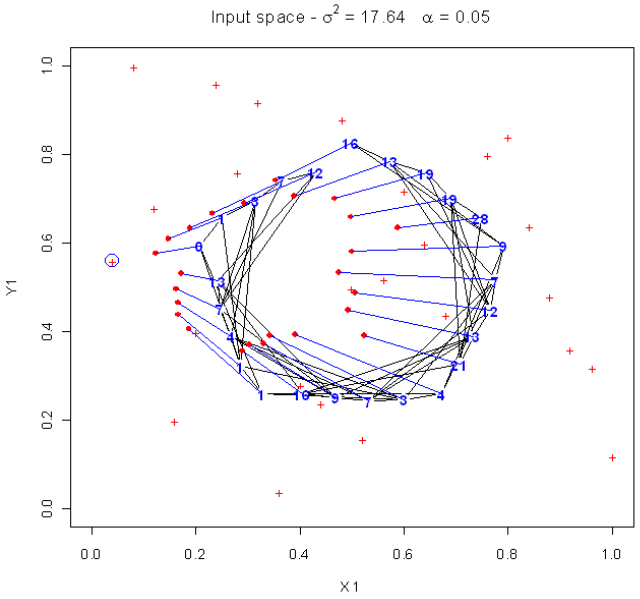

Figure 9. End of pass 2. Again some distant PVs have moved a considereable distance. Compare the movement of the (9) and (28) on the right side.

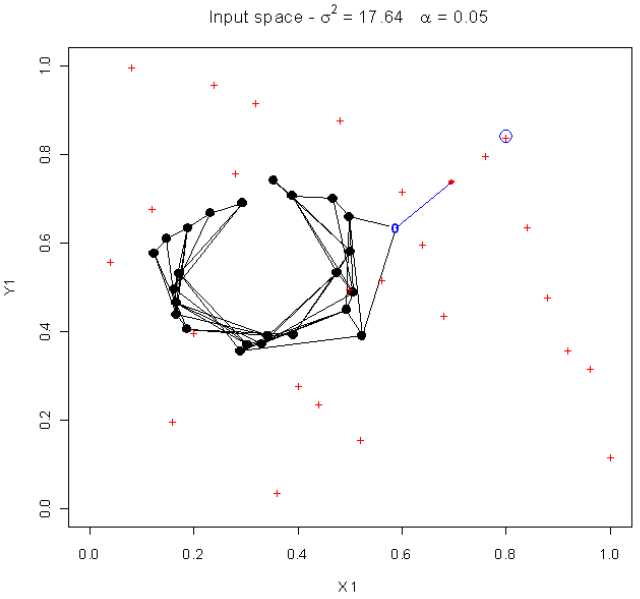

Figure 10. Start of pass 3. The fact that σ is large means that all the neighbours make large moves. This results in a compacting of the PVs.

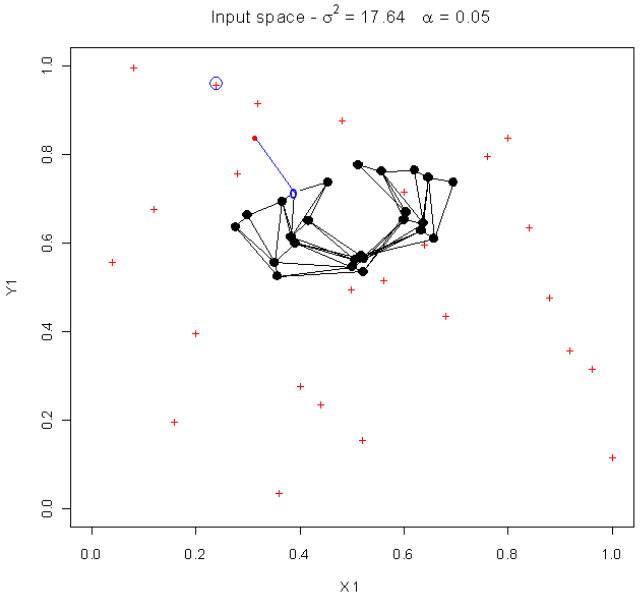

Figure 11.

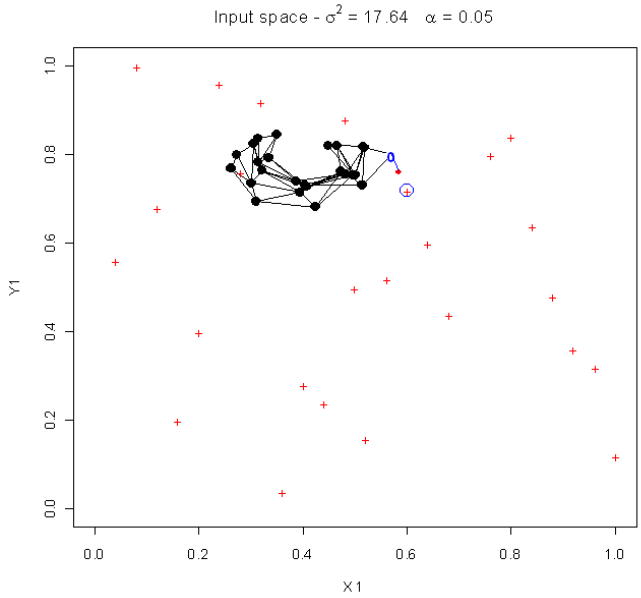

Figure 12.

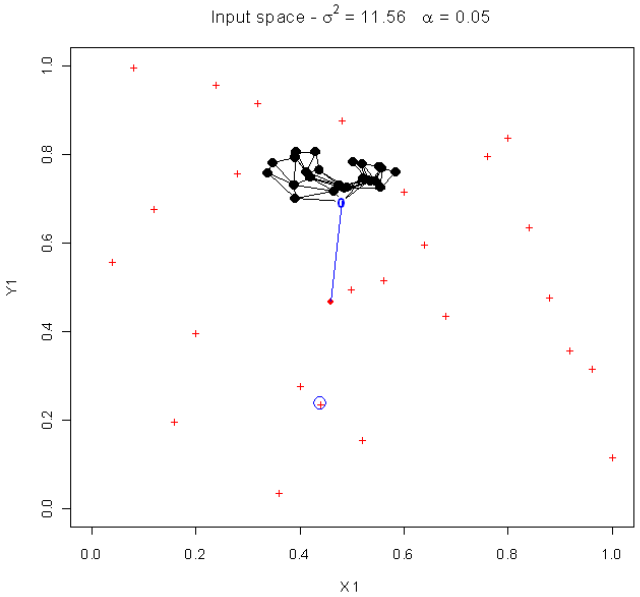

Figure 13. The radius has been reduced. This means the PVs distant from the BMU will not move as much.

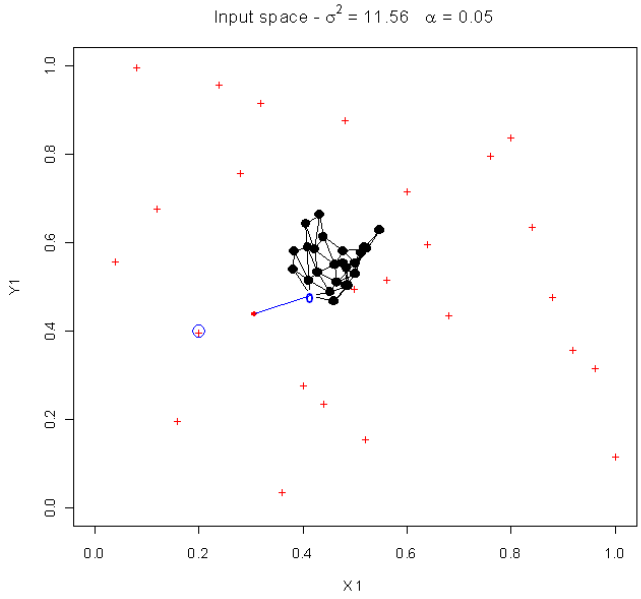

Figure 14. The 'mesh' continues to compact.

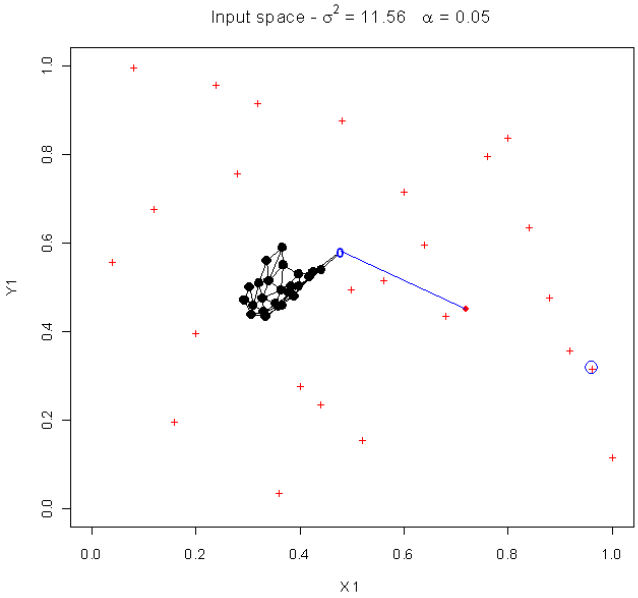

Figure 15.

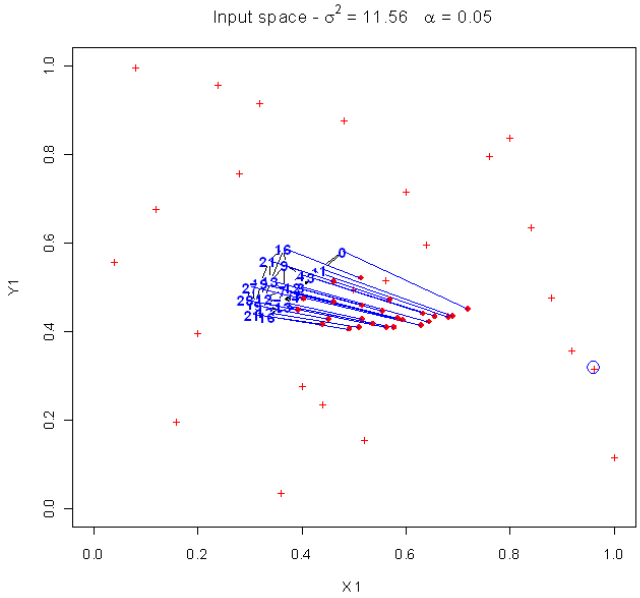

Figure 16.

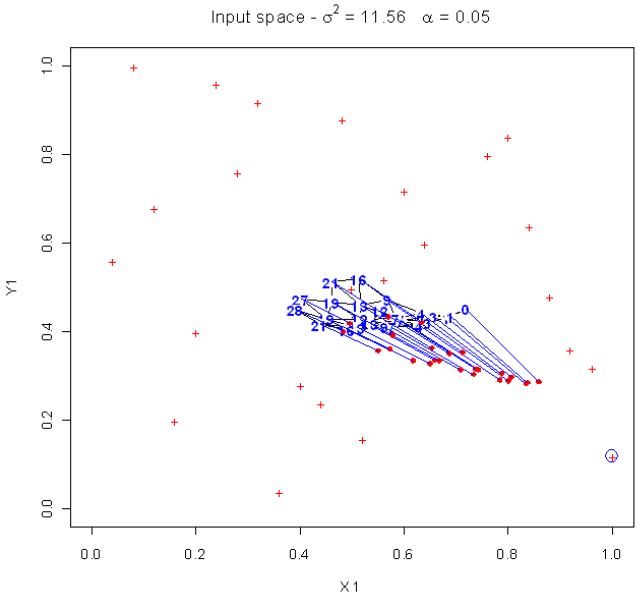

Figure 17.

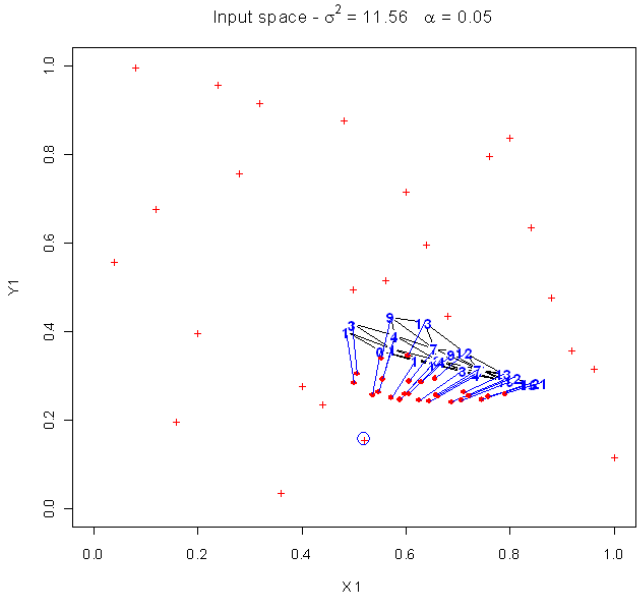

Figure 18.

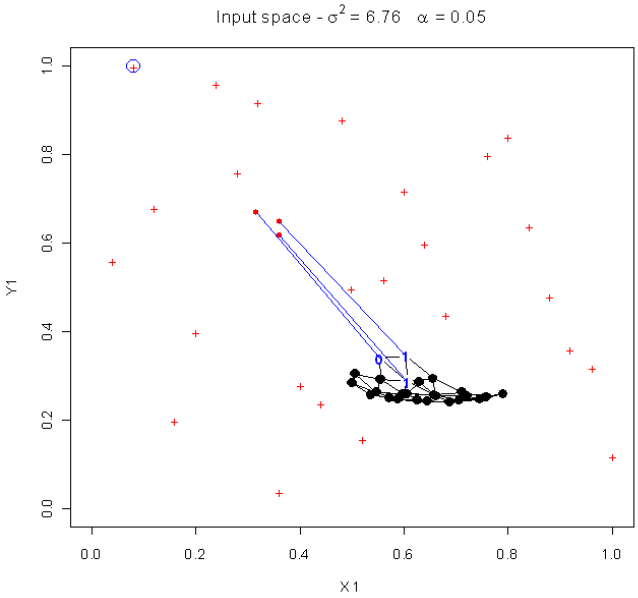

Figure 19. The radius has decreased again.

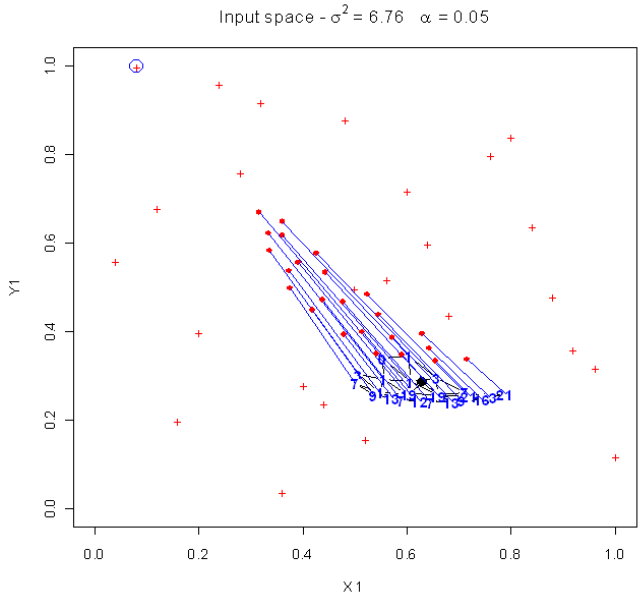

Figure 20.

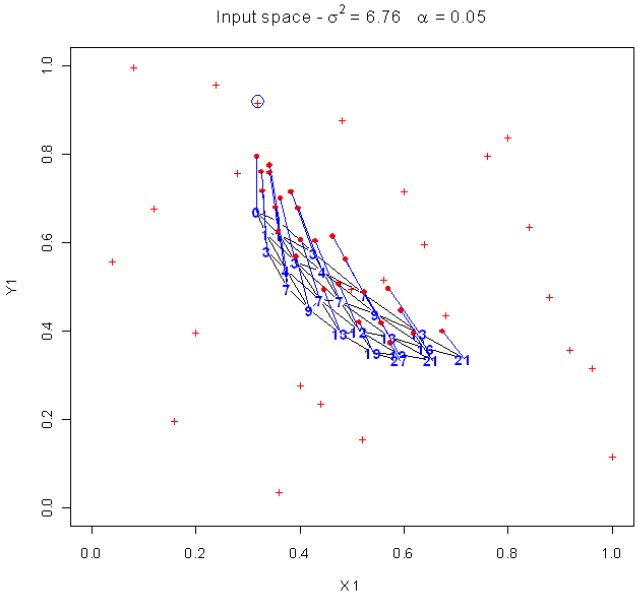

Figure 21.

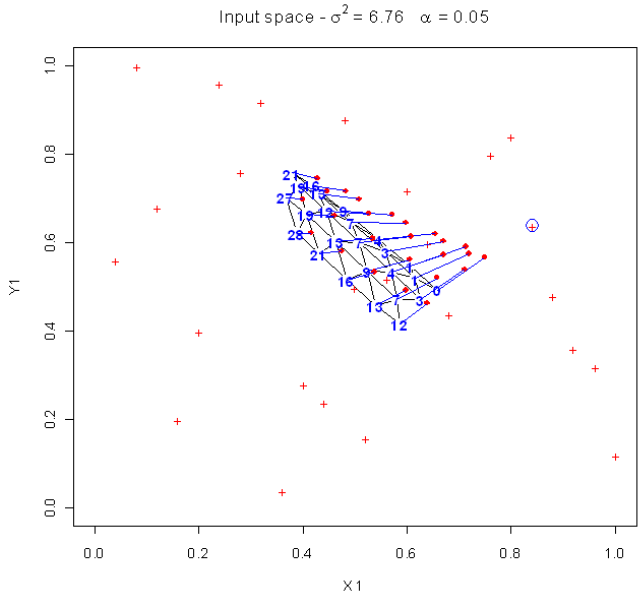

Figure 22. Notice that the PV (28) on the left has moved very little in comparison with the BMU. This will stretch the 'mesh'.

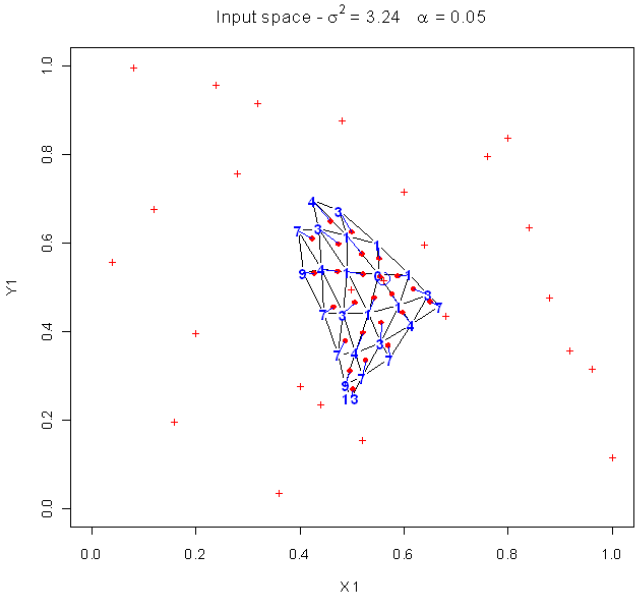

Figure 23. Another reduction in radius. The sample point is inside the 'mesh'

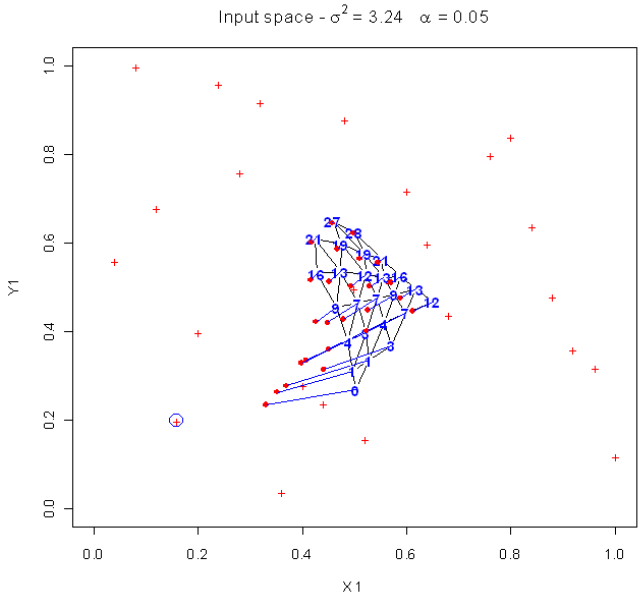

Figure 24. Notice that the PVs at 'distances' from (16) to (28) barely move while the BMU move half way and its near neighbours move a bit less.

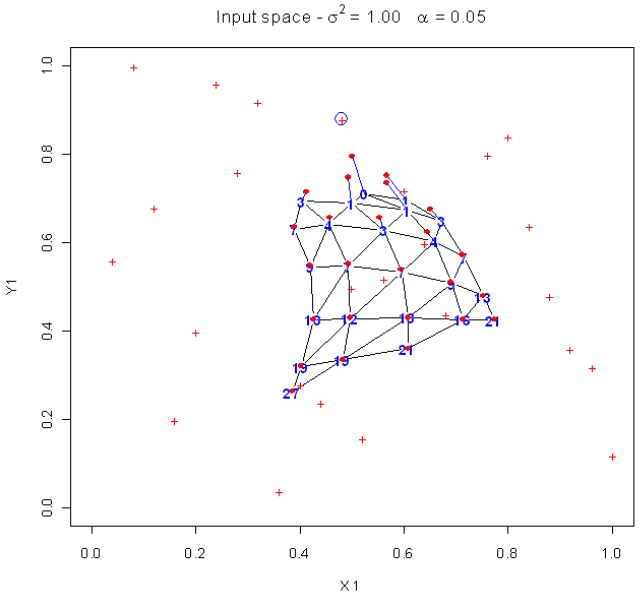

Figure 25. The radius has reduced to 1 and PVs at (1) to (4) are the only ones with much movement.

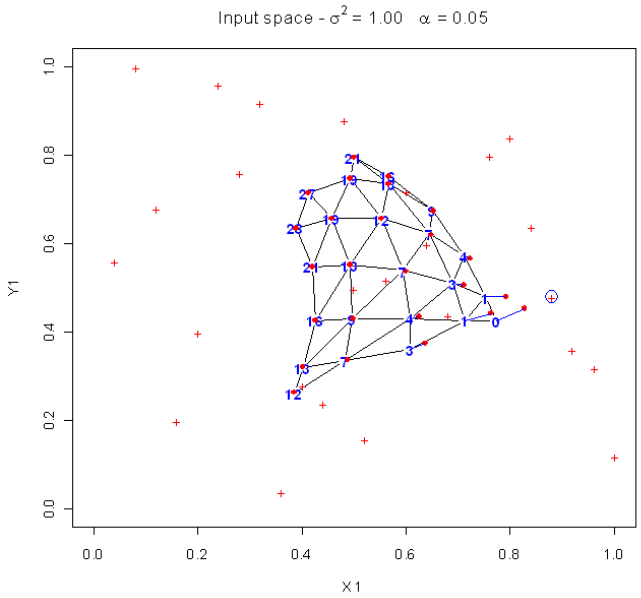

Figure 26.

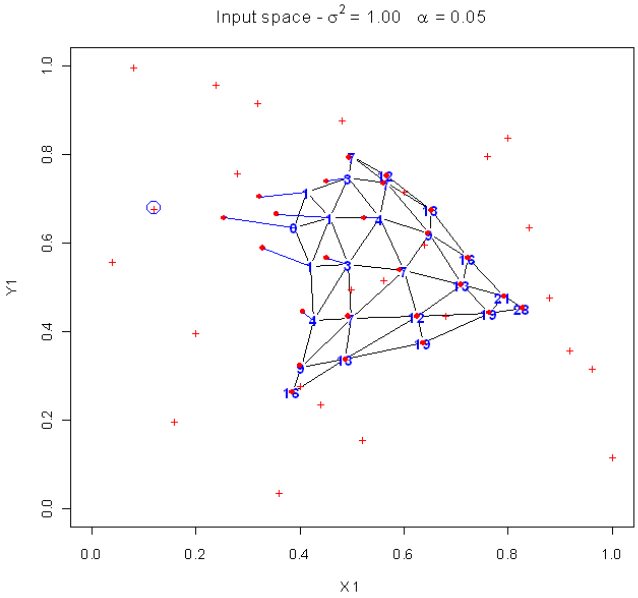

Figure 27.

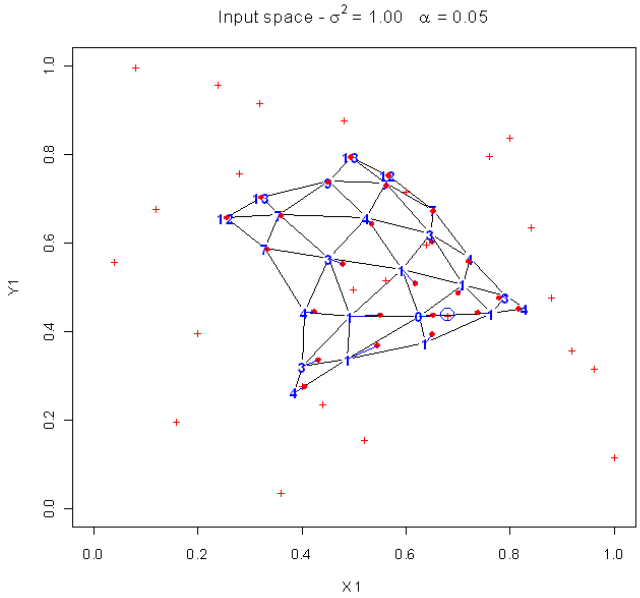

Figure 28.

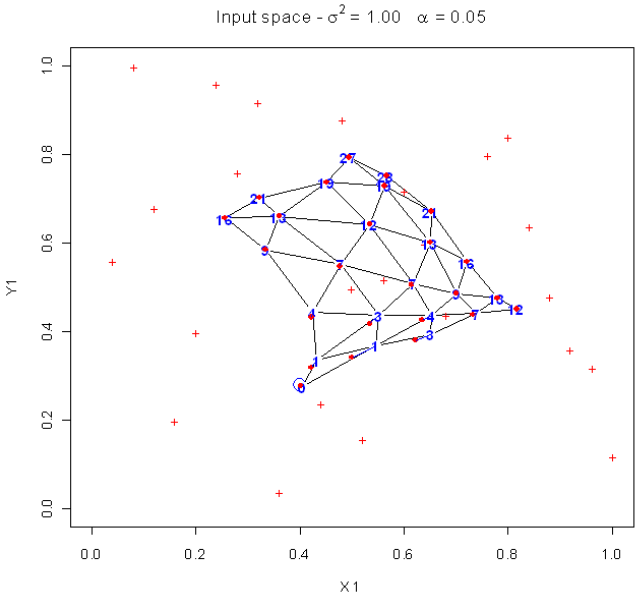

Figure 29.

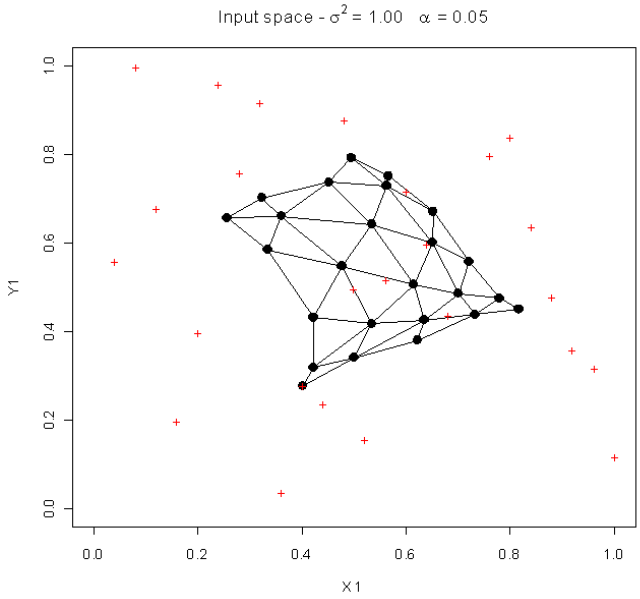

Figure 30. At the end, the 'mesh' is untangled.

With this example in mind, we can look at a larger example.

We let the radius decay more slowly by using the neighbourhood radius

```
> (r <- 4*(1-(1:60)/60) + 1)
 [1] 4.933333 4.866667 4.800000 4.733333 4.666667 4.600000 4.533333 4.466667
 [9] 4.400000 4.333333 4.266667 4.200000 4.133333 4.066667 4.000000 3.933333
[17] 3.866667 3.800000 3.733333 3.666667 3.600000 3.533333 3.466667 3.400000
[25] 3.333333 3.266667 3.200000 3.133333 3.066667 3.000000 2.933333 2.866667
[33] 2.800000 2.733333 2.666667 2.600000 2.533333 2.466667 2.400000 2.333333
[41] 2.266667 2.200000 2.133333 2.066667 2.000000 1.933333 1.866667 1.800000
[49] 1.733333 1.666667 1.600000 1.533333 1.466667 1.400000 1.333333 1.266667
```

```
[57] 1.200000 1.133333 1.066667 1.000000
```

We do 60 iterations of 5 cycles each with the neighbourhood radius going from just less than 5 down to 1. Note that we use 5 repetitions of the radius

```
> for (i in 1:60) {
>    pv <- iterative.train.SOM(Z, pv, rep(r[i], 5), 5, rep(0.1, 5), Coords, H.distance)
>    connect.Map(10, 10, pv, main=paste(5*i, "Epochs - radius = ", floor(r[i]*100)/100))
>    points(Z[,1],Z[,2],col="red",pch="+")
>       # Show neighbours
>    points(pv[pos,1], pv[pos,2], pch=16, col="yellow")
>    points(pv[n.ind,1], pv[n.ind,2], pch=16, col="green")
>    points(Z[,1],Z[,2],col="red",pch="+")
> }
```

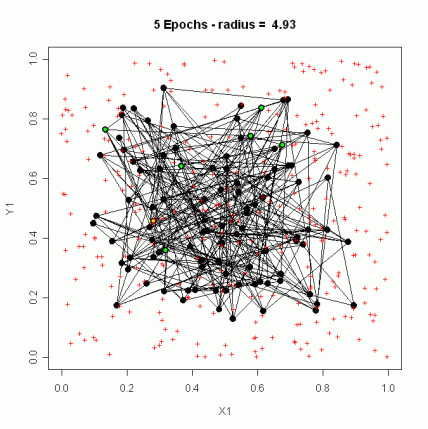

Figure 31.

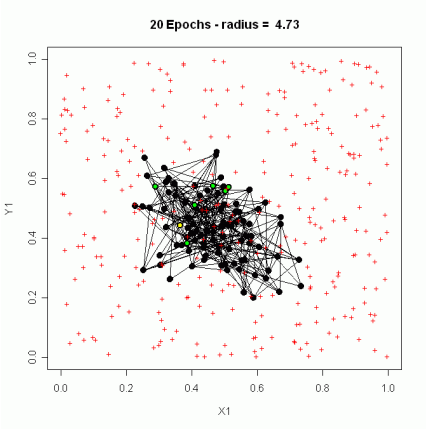

Figure 32.

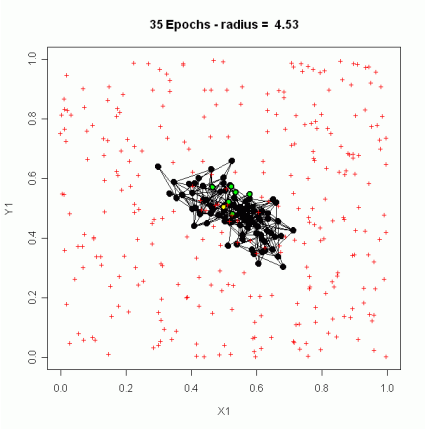

Figure 33.

Notice that as the process begins the prototype vectors are scattered throughout the input space. The lines join the prototype vectors to their neighbours in the output space and form a tangled net. The neighbourhood radius is fairly large so as one prototype vector moves to be more like the input vector for which it is the BMU so do a large number of its neighbours. This results in the consolidation and untangling of the net. As the process continues, the prototype vectors which are neighbours in the output space also have become neighbours in the input space. Note that the green prototype vectors are output space neighbours of the yellow prototype vector.

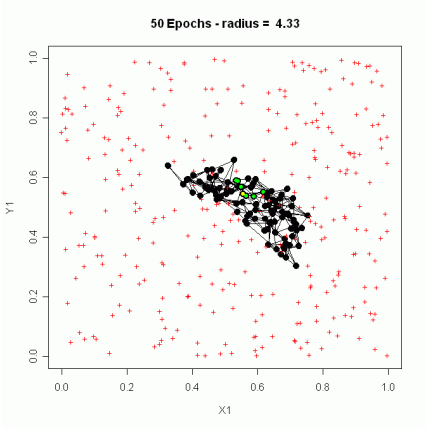

Figure 34.

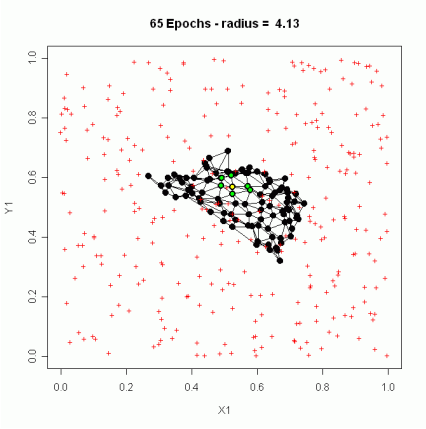

Figure 35.

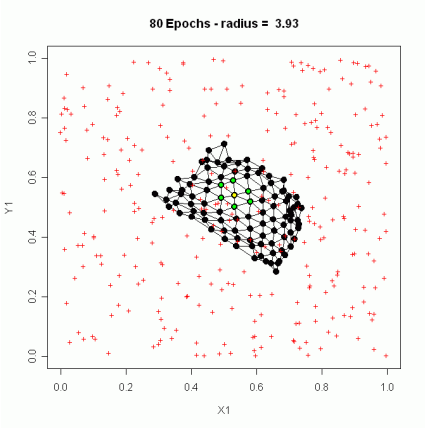

Figure 36.

The result is that the prototype vectors now move together and the untangled net expands.

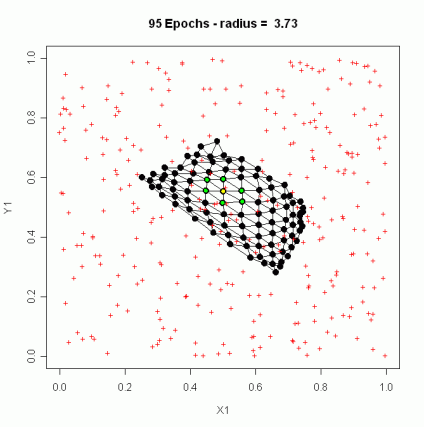

Figure 37.

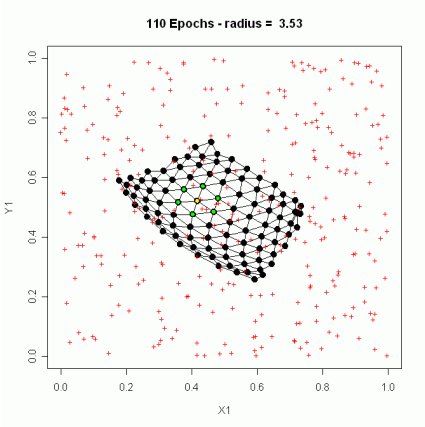

Figure 38.

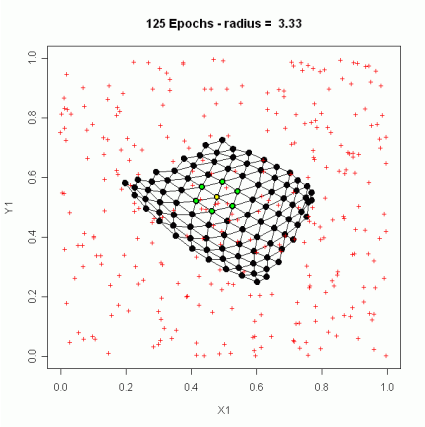

Figure 39.

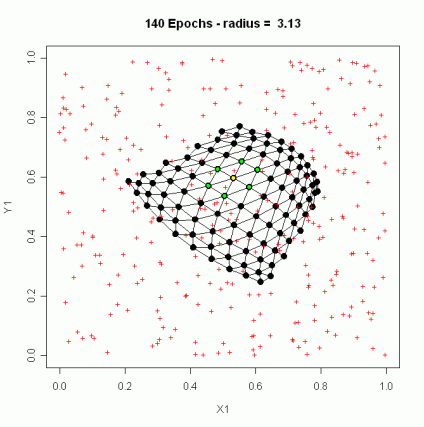

Figure 40.

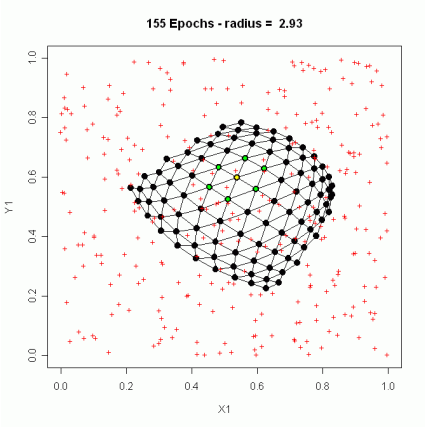

Figure 41.

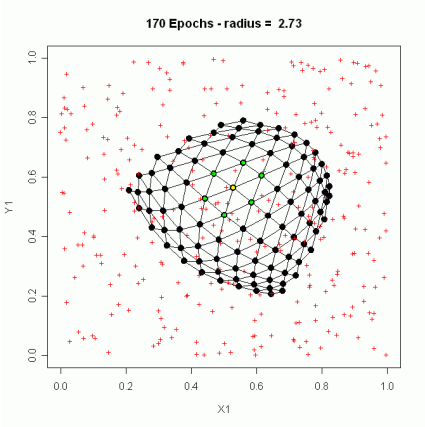

Figure 42.

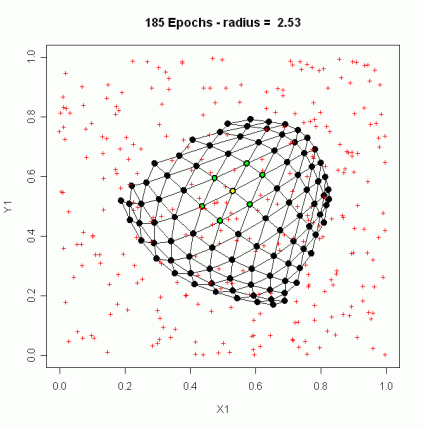

Figure 43.

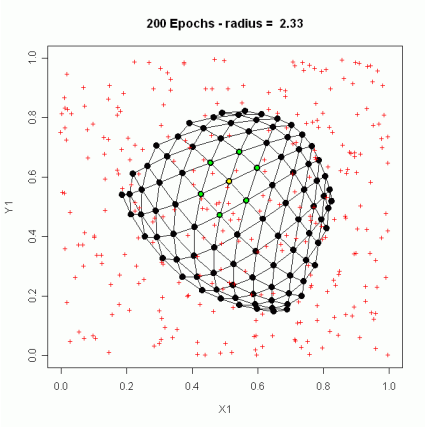

Figure 44.

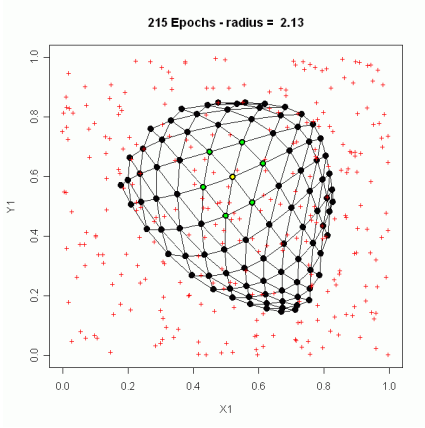

Figure 45.

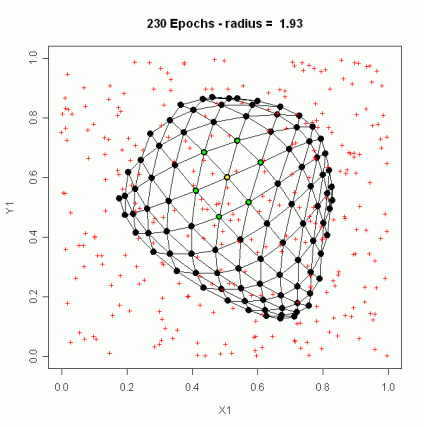

Figure 46.

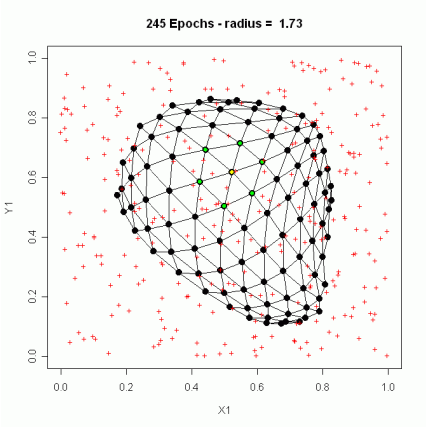

Figure 47.

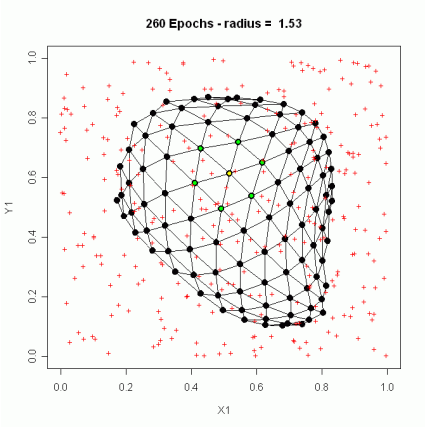

Figure 48.

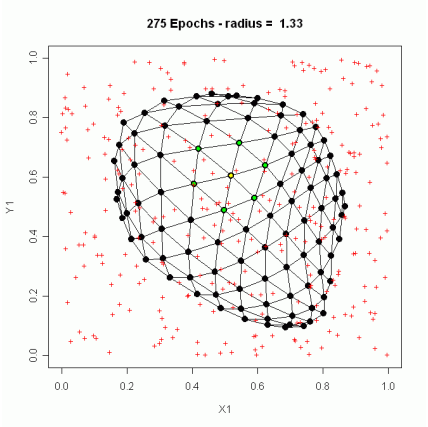

Figure 49.

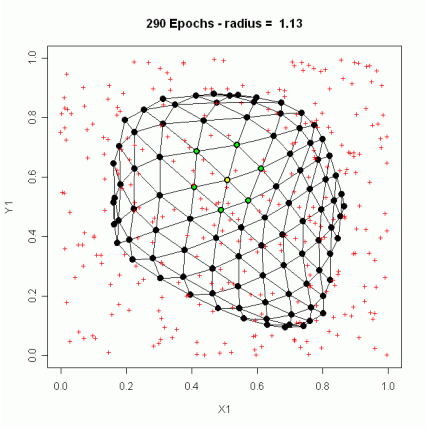

Figure 50.

Example: SOM on Flea Beetles

This uses six dimensional data.

```
> source(paste(rcode.dir, "ReadFleas.r", sep=""))
```

We get the number of cases

```
> n.cases <- dim(d.flea)[1]
```

and put all the information into a structure to keep it all together

```
> SOM.struct <- list(data=d.flea, labels=species, col.names=headers, ave={}, sd={})
> SOM.struct$ave <- apply(SOM.struct$data, 2, mean)
> SOM.struct$sd <- apply(SOM.struct$data, 2, sd)
> SOM.struct
$data
   tars1 tars2 head aede1 aede2 aede3
1    191   131   53   150    15   104
2    185   134   50   147    13   105
3    200   137   52   144    14   102
  .
  .
  .
73   195   110   49   124    13    89
74   187   124   49   129    14    88
$labels
 [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

```
[39] 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
$col.names
[1] "tars1" "tars2" "head"  "aede1" "aede2" "aede3"
$ave
    tars1     tars2      head     aede1     aede2     aede3
177.25676 123.95946  50.35135 134.81081  12.98649  95.37838
$sd
    tars1     tars2      head     aede1     aede2     aede3
29.412541  8.481146  2.751998 10.350932  2.142162 14.304614
```

We normalize (standardize) the data by subtracting the mean of the columns and dividing the result by the standard deviation

```
> SOM.norm <- div.arr.vec(add.arr.vec(SOM.struct$data, -SOM.struct$ave), SOM.struct$sd)
```

We may require the reverse of the process.

```
> SOM.denorm <- add.arr.vec(mult.arr.vec(SOM.norm, SOM.struct$sd), SOM.struct$ave)
> Data <- SOM.norm
```

Rather than putting the prototype vectors in at random we can distribute them uniformly over a grid in the direction in which the variance is greatest (think PCA !).

```
> eigs <- eigen((t(Data)%*%Data)/n.cases)
> Vec <- eigs$vectors[,1:2]
> Val <- eigs$values[1:2]
>    # Get L2 norm
> norm.V <- sqrt(diag(t(Vec)%*%Vec))
>    # Scale the vector
> n.V <- mult.arr.vec(div.arr.vec(Vec, norm.V), sqrt(Val))
>    # Choose a grid size for the prototype vectors.
> g.size <- c(11,6)
> g.units <- prod(g.size)
> pv <- matrix(rep(0, g.units), g.units, n.var, byrow=T)
>    # Establish a grid on [-1, 1] x [-1, 1]
> grid <- expand.grid(((1:g.size[1])*2-1-g.size[1])/(g.size[1]-1),
>                     ((1:g.size[2])*2-1-g.size[2])/(g.size[2]-1))
>    # Project on eigenvector plane
> pv <- pv + t(n.V%*% t(grid))


> Coords <- get.hex.grid(g.size[2:1])
>    # This gives the distances in the output (hexagonal) space
> Hex.distance <- as.matrix(dist(Coords))^2
```

In the batch SOM routine the updating is not done on a case by case basis but rather all cases that are in the Voronoi set of a prototype vector are averaged and the average is used as the new value for the prototype vector. The neighbouring (in the output space) prototype vectors are adjusted based on their distance from the current prototype vector and the neighbourhood function.

```
> batch.train.SOM <- function(trainlen, pv, Data, g.units, n.cases, Hex.distance, radius,
n.var) {
>    Dx2 <- t(2*Data)     # Take 2*distance to the datum for use in cosine law
>       # A constant distance (length of each datum vector)
>    Data.const <- drop(apply(Data*Data, 1, sum))
>    for (t in 1:trainlen) {
>       Dist <- matrix((apply(pv*pv, 1, sum)), g.units, n.cases) -  pv%*%Dx2
>       min.dist <- apply(Dist, 2, min)
>       min.index <- apply(Dist, 2, order)[1,]
>            # Updating step: Compute the average of the data vectors
>            # in the neighbourhood of a prototype vector and use that
```

```
>            # average as the new value for the prototype vector.
>            # The neighbouring prototype vectors are also altered based
>            # on their closeness to the first. The effect is controlled
>            # by the neighborhood function.
>            # This effect is is given by the H matrix.
>            # So, for each prototype vector the new weight vector is
```

$$m = \frac{\sum_i h_i d_i}{\sum_i h_i}$$

```
>            # where i denotes the index of data vector.
>            # The values of neighbourhood function h_i are the same
>            # for all data vectors belonging to the Voronoi set of the
>            # same prototype vector, so we calculate a partition matrix P
>            # with elements p_ij=1 if the BMU of data vector j is i.
>        H <- exp(-Hex.distance/(2*radius[t]))
>        P <- matrix(0, g.units, n.cases)
>        P[cbind(min.index, 1:n.cases)]<-1
>           # The sum of vectors in each Voronoi set are calculated as
>           # (P*Data) and the neighbourhood is taken into account by
>           # calculating a weighted sum of the
>           # Voronoi sum (H*). The "activation" matrix A is the
>           # denominator of the equation above.
>        S <- H%*%(P%*%Data)
>        A <- H%*%matrix(rep(apply(P,1,sum), n.var), g.units, n.var)
>        pv <- S*(1/A)
>    }
>    return(pv)
> }
```

We set our initial training length (the number of times we loop through to move the prototype vectors) and radius (which changes at each loop) for a batch version of the SOM. We start by using a fairly large neighbourhood radius.

```
> trainlen <- 5
> radius <- seq((trainlen-1), 0, by=-1)/(trainlen - 1) +1
> (radius <- radius^2)
[1] 4.0000 3.0625 2.2500 1.5625 1.0000
```

Notice that the radius is decreasing at each stage so that the influence of the change in one prototype vector is felt by a decreasing number of other vectors.

```
> pv <- batch.train.SOM(trainlen, pv, Data, g.units, n.cases, Hex.distance, radius, n.var)
```

Once we have done the initial training we fine-tune our results by running a number of iterations with a unit radius.

```
> trainlen <- 18
> radius <- rep(1, trainlen)
> pv <- batch.train.SOM(trainlen, pv, Data, g.units, n.cases, Hex.distance, radius, n.var)
```

We now need to display our results.

One way is with a U-matrix which displays the relative density of the data. To create the U-matrix we need to find several distances.

First we create a 3-dimensional array for the prototype vectors (each level of the array contains the values

for one variable of the data).

```
> (M <- array(pv, c(g.size[1], g.size[2], n.var)))
, , 1
            [,1]       [,2]        [,3]       [,4]         [,5]      [,6]
 [1,]  0.3813057  0.4550991  0.61037985  0.8550628  1.100176324 1.2163432
 [2,]  0.3824255  0.4424574  0.60809320  0.8866356  1.160373679 1.2666272
 [3,]  0.3437388  0.3264197  0.41409977  0.6044414  0.957487607 1.2956688
 [4,]  0.1365874  0.1988923  0.38284771  0.6077559  1.066498980 1.4180894
 [5,] -0.4583689 -0.2738356  0.03273612  0.2633002  0.514392172 0.9772603
 [6,] -1.0759355 -0.6002975 -0.08896322  0.1387178  0.354039479 0.6759806
 [7,] -1.4978912 -1.2132654 -0.64628587 -0.1192350  0.127060374 0.2720340
 [8,] -1.4491475 -1.1711385 -0.72448742 -0.1179095  0.169185209 0.3163078
 [9,] -1.4809989 -1.3569755 -1.18290938 -0.7169288 -0.002860286 0.2940061
[10,] -1.3376625 -1.2656667 -1.07634444 -0.4618620  0.235215816 0.4439037
[11,] -1.2598891 -1.2437860 -1.13582210 -0.8422796 -0.093772051 0.4259545
, , 2
             [,1]        [,2]       [,3]       [,4]       [,5]        [,6]
 [1,] -1.40148270 -1.02815336 -0.5247861 -0.2653503 -0.1640307 -0.09431428
 [2,] -1.25776481 -0.77412414 -0.3946837 -0.2172795 -0.1273898 -0.13707279
   .
   .
   .
, , 6
              [,1]        [,2]        [,3]        [,4]       [,5]        [,6]
 [1,] -1.698794697 -1.49859015 -1.37577733 -1.39627082 -1.2520778 -0.89079650
 [2,] -1.254144568 -1.16923388 -1.19664764 -1.17112228 -0.8877749 -0.62292021
 [3,] -0.788941278 -0.76251421 -0.83328655 -0.86727435 -0.7653947 -0.62900487
 [4,] -0.376886219 -0.42625721 -0.51179198 -0.50008541 -0.5509208 -0.62079882
 [5,] -0.001987346 -0.03660223 -0.09736228 -0.07823276 -0.1500199 -0.40657057
 [6,]  0.362400838  0.37602321  0.42628394  0.38725653  0.2182051 -0.01932081
 [7,]  0.563073547  0.64351491  0.75492536  0.73467494  0.6610705  0.61729956
 [8,]  0.755653357  0.86223163  0.91550992  0.87846432  0.8346176  0.82161920
 [9,]  0.805083864  0.85563906  0.92852877  0.99401332  0.9357082  0.82673759
[10,]  0.719140413  0.82144199  0.97454551  1.02778734  0.8290597  0.69497004
[11,]  0.568148004  0.64896960  0.86311190  1.06303868  0.9372813  0.71489390
```

The code

```
> apply(M[,,1], 1, diff)
           [,1]       [,2]        [,3]       [,4]      [,5]      [,6]      [,7]
[1,] 0.07379341 0.06003182 -0.01731914 0.06230487 0.1845332 0.4756381 0.2846258
[2,] 0.15528072 0.16563584  0.08768011 0.18395546 0.3065717 0.5113343 0.5669795
[3,] 0.24468297 0.27854244  0.19034165 0.22490817 0.2305641 0.2276810 0.5270509
[4,] 0.24511351 0.27373804  0.35304619 0.45874310 0.2510919 0.2153217 0.2462954
[5,] 0.11616689 0.10625356  0.33818116 0.35159037 0.4628682 0.3219412 0.1449737
          [,8]      [,9]      [,10]      [,11]
[1,] 0.2780091 0.1240234 0.07199585 0.01610311
[2,] 0.4466510 0.1740661 0.18932224 0.10796388
[3,] 0.6065779 0.4659806 0.61448246 0.29354248
[4,] 0.2870947 0.7140685 0.69707780 0.74850757
[5,] 0.1471225 0.2968664 0.20868785 0.51972660
```

finds the difference between elements in the same row at level 1 (the first variable) so, when we square and add at all levels, we get the distance between the prototypes in the same row. We can also get the distances in the columns.

Note that the code used is to demonstrate the concepts. For example, the following assumes a dataset with 6 columns.

```
> dx <- sqrt(apply(M[,,1], 1, diff)^2+apply(M[,,2], 1, diff)^2 +
>            apply(M[,,3], 1, diff)^2+apply(M[,,4], 1, diff)^2 +
>            apply(M[,,5], 1, diff)^2+apply(M[,,6], 1, diff)^2)
```

A more robust form might be

```
> tmp <- 0
> for (i in 1:6) {
>    tmp <- tmp + apply(M[,,i], 1, diff)^2
> }
> dx <- sqrt(tmp)
```

or, in the form of a function

```
> M.xy.diff <- function(M, row.col, n.vars) {
>    tmp <- 0
>    for (i in 1:n.vars) {
>       tmp <- tmp + apply(M[,,i], row.col, diff)^2
>    }
>    return( sqrt(tmp))
> }
> dx <- M.diff(M, 1, 6)
> dy <- M.diff(M, 2, 6)
```

These (and the following) are used in the construction of the *U*-matrix as illustrated in the figures below.

The position in a rectangular array is shown by the figure on the left while that in a hexagonal array is shown in the figure on the right. The circles represent the prototype vectors and the arrows represent the various distances between them.

The U-matrix is $2m - 1 \times 2n - 1$ where m and n are the row and column dimensions of the prototype vector array. Once the distances are computed the positions indicated by the circles are filled with the median values of the distances connected to it. In the figures, the output space is a 3×3 array and the *U*-matrix is a 5×5 array constructed as shown in the figure on the right.

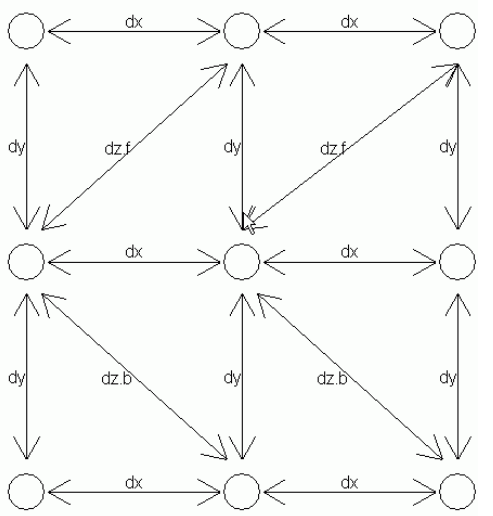

Figure 51.

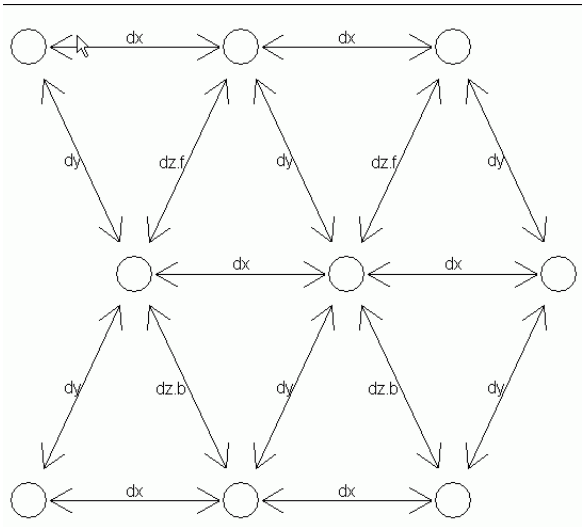

Figure 52.

The distance for diagonal neighbours (forward and backwards) is more complicated and requires selecting every other value

```
> (ind.dz.f1 <- seq(1, g.size[1]-1, by=2))
[1] 1 3 5 7 9
> (ind.dz.f2 <- 2:g.size[2])
[1] 2 3 4 5 6
> (ind.dz.f3 <- seq(2,g.size[1], by=2))
[1] 2  4  6  8 10
> (ind.dz.f4 <- 1:(g.size[2]-1))
```

```
[1] 1 2 3 4 5
> M.f.diff <- function(M, g.size, n.vars) {
>    ind.dz.f1 <- seq(1,g.size[1]-1, by=2)
>    ind.dz.f2 <- 2:g.size[2]
>    ind.dz.f3 <- seq(2,g.size[1], by=2)
>    ind.dz.f4 <- 1:(g.size[2]-1)
>    tmp <- 0
>    for (i in 1:n.vars) {
>       tmp <- tmp + (M[,,i][ind.dz.f1, ind.dz.f2] - M[,,i][ind.dz.f3, ind.dz.f4])^2
>    }
>    return(sqrt(tmp))
> }
> dz.f <- M.f.diff(M, g.size, 6)
> M.b.diff <- function(M, g.size, n.vars) {
>    ind.dz.b1 <- seq(2,g.size[1],by=2)
>    ind.dz.b2 <- 1:(g.size[2]-1)
>    ind.dz.b3 <- seq(3,g.size[1],by=2)
>    ind.dz.b4 <- 2:g.size[2]
>    tmp <- 0
>    for (i in 1:n.vars) {
>       tmp <- tmp + (M[,,i][ind.dz.b1, ind.dz.b2]-M[,,i][ind.dz.b3, ind.dz.b4])^2
>    }
>    return( sqrt(tmp))
> }
> dz.b <- M.b.diff(M, g.size, 6)
> Usize <- c(2*g.size[1]-1, 2*g.size[2]-1)
> U <- matrix(0, Usize[1], Usize[2])
> U[seq(1,Usize[1],by=2), seq(2,Usize[2],by=2)] <- t(dx)
> U[seq(2,Usize[1],by=2), seq(1,Usize[2],by=2)] <- dy
> U[seq(2,Usize[1],by=4), seq(2,Usize[2],by=2)] <- dz.f
> U[seq(4,Usize[1],by=4), seq(2,Usize[2],by=2)] <- dz.b
```

The positions of the medians are

```
> med.pos <- as.matrix(expand.grid(seq(1, Usize[1], by=2), seq(1, Usize[2], by=2)))
```

The neighbours for the medians are

```
> the.neighbours <- apply(med.pos, 1, f.H.neighbours, Usize[1], Usize[2])
> for (i in 1:dim(med.pos)[1]) {
>    U[med.pos[i,1], med.pos[i,2]] <- median(U[the.neighbours[[i]]])
> }
```

For plotting this we need a grid with the first row as is, second row shifted by 0.5, the third by 1, and the fourth by 0.5 (and repeated as often as needed.

```
> get.Uhex.grid <- function(grid.size) {
>       # Create a grid of the required size (see get.hex.grid)
>    hex.pos <- expand.grid(1:grid.size[2],1:grid.size[1])-c(1,1)
>      # Swap the rows/cols
>    hex.pos <- hex.pos[,c(2,1)]
>      # Shift every other row by 0.5 to create hexagon
>      # We need multiples of 4 for row shifts of (0,.5,1,.5)
>    m.4 <- (grid.size[2]%/%4 + 1)
>    hex.pos[,1] <- hex.pos[,1]+rep(c(0,.5,1,.5), m.4)[1:grid.size[2]]
>    hex.pos[,2] <- hex.pos[,2]*sqrt(0.75)   # make distances to all neighboring units equal
>    return(hex.pos)
```

```
> }
> U.mat.pos <- get.Uhex.grid(c(2*g.size[2]-1, 2*g.size[1]-1))
> U.mat.pos[,2] <- abs(max(U.mat.pos[,2]) - U.mat.pos[,2])
>    # Scale
> V <- (U-min(U))/(max(U)-min(U))
> oldpar <- par(mfrow = c(3, 3))
> nf <- layout(matrix(1:18, 3, 6, byrow = TRUE),rep(c(10,1),9))
> zz <- (255:0)/255
> draw.hexagon(U.mat.pos, gray(1-V), 0, gray, main="U-matrix")
> image(1,seq(min(U),max(U),len=256),t(cbind(zz)), col=grey(1-zz), ylim=range(U))
>
```

In the U-matrix (below) the shading indicates distance. White is close - black is far. Note that the U-matrix array has cells between the prototypes. The greyness of the cell is determined by the distance to the neighbouring prototype vector, and the greyness of the prototype is the median of the distances to the neighbours.

For the components, the standard hexagonal grid is used and the prototype vector value of that variable is displayed. The grey scale key to indicates the range of values in the display. It should be noted that the appearance of similar patterns at similar positions on different component planes indicates correlations between variables.

```
> coord.pos <- get.hex.grid(c(g.size[2], g.size[1]))
> for (i in 1:6) {
>    C1 <- (pv[,i]-min(pv[,i]))/(max(pv[,i])-min(pv[,i]))
>       # Flip
>    coord.pos[,2] <- abs(max(coord.pos[,2]) - coord.pos[,2])
>    draw.hexagon(coord.pos, gray(1-C1), 0, gray, main=headers[i])
>    image(1,seq(min(pv[,i]),max(pv[,i]),len=256),t(cbind(zz)), col=grey(1-zz),
>          ylim=range(pv[,i]))
> }
> par(oldpar)
```

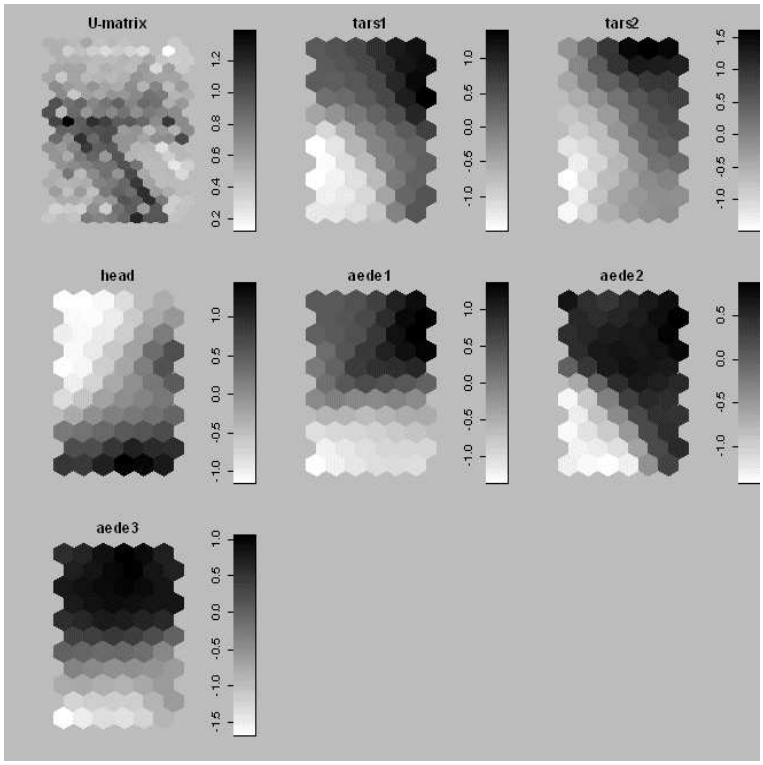

Figure 53.

Two measures of quality are:

- average (mean) quantization error, where for each data vector we find the distance between this data vector and the closest prototype vector called the Best Matching Unit (BMU) and average those distances and
- topographic error, where for each data vector, we consider its first and second BMUs. We find the proportion of all data vectors for which the first and second BMU are not adjacent

```
> Dx2 <- t(2*Data)
> Data.const <- drop(apply(Data*Data, 1, sum))
>    # The distance (less a constant) from each datum to each prototype
> Dist <- matrix((apply(pv*pv, 1, sum)), g.units, n.cases) - pv%*%Dx2
>    # Get two smallest distances in each col (i.e. each datum)
> sort.Dist <- apply(Dist, 2, sort)[1:2,]
>    # Compute the average distance from data to BMU
> mqe <- mean(sqrt(sort.Dist[1,] + Data.const))
>    # Get the 2 minima
> mins.ind <- apply(Dist, 2, order)[1:2,]
>    # Have to determine if the 1st & 2nd mins are neighbours
>    # Get the row/col indices for the array
> H.grid <- as.matrix(expand.grid(1:g.size[1], 1:g.size[2]))
>    # Find all the neighbours of the BMU for each datum
> smallest.neighbours <- apply(H.grid [mins.ind [1,],], 1, f.H.neighbours, g.size[1],
g.size[2])
>    # Get the indices for the next best
> nxt.smallest.inds <- H.grid [mins.ind [2,],]
>    # Count the number that are not adjacent
> count <- 0
> for (i in 1:n.cases) {
```

```
>    adjacent <- any(apply(smallest.neighbours[[i]],1,"==", nxt.smallest.inds[i,]))
>    if (adjacent) count <- count+1
> }
The mean quantization error 0.817371
and the topographic error 0
```

The next method of display is called a "hit" matrix. (Note that the class information was not used in producing the SOM. It is used here to see if the clusters in SOM can be related to the classes.) The known classes are displayed as coloured hexagons and the size of the hexagon is determined by the number of that class associated with that prototype vector. The counts associated with a prototype vector are determined and the maximum for each class is found. The counts for the prototype vectors are normalized and that class is displayed on the hit matrix with the size of the hexagon reflecting that normalized value. The dark band between prototypes indicates a large distance between neighbours.

Count the hits for each class:

```
> n.classes <- 3
> class.length <- c(21, 22, 31)
> hits <- matrix(0, g.units, n.classes)
> class.start <- 0
> for (i in 1:n.classes) {
>    for (j in 1:class.length[i]) {
>       hits[mins.ind[1,j+class.start], i] <- hits[mins.ind[1,j+class.start], i] + 1
>    }
>    class.start <- class.start + class.length[i]
> }
```

Count the maximum number of hits/class:

```
> max.hits <- apply(hits, 2, max)
>    # Max at least 1
> max.hits[max.hits==0] <- 1
> hex.scale <- sqrt(t(t(hits)/max.hits))       # max hits gives 1
```

Make one vector with prototype number and class attached:

```
> hex.scale <- cbind(rep(1:g.units,n.classes),
c(rep(1,g.units),rep(2,g.units),rep(3,g.units)), as.vector(hex.scale))
```

Eliminate those that are 0.

```
> hex.scale <- hex.scale[hex.scale[,3]!=0,]
```

Now sort so the largest are plotted first (so as to not wipe out the smaller):

```
> hex.scale <- hex.scale[order(-hex.scale[,3]),]
```

We need to select the prototype positions in `U.mat.pos`.

```
> prototype.ind <- {}
> for (i in seq(1,2*g.size[2]-1,by=2)) {
>    prototype.ind <- c(prototype.ind, seq(1,2*g.size[1]-1,by=2)+(2*g.size[1]-1)*(i-1))
> }
> draw.hexagon(U.mat.pos, gray(1-V),0,gray)
```

This gives the positions of the prototype on the U-matrix:

```
> prototype.pos <- U.mat.pos[prototype.ind,]
>    # hex.scale[,1] - gives the prototype number
> pos.col.scale <- cbind(prototype.pos[hex.scale[,1],], hex.scale[,2:3])
```

We have drawn the hexagonal lattice and need to update it with the 'hit' information.

```
> update.hexagon <- function (pos, Col="white", Border=1,Scale=1) {
>    hex<-matrix(c(0, 0.5, 0.5, 0, -0.5, -0.5, 0,
```

```
>                1.732, 0.866, -0.866, -1.732, -0.866, 0.866, 1.732), 7, 2)
>    oldpar <- par(pty = "s", bg="gray", bty="n", xaxt="n", yaxt="n", mar=c(1,1,1,1))
>    x.r <- range(pos[,1])+c(-2, 2)
>    y.r <- range(pos[,2])*3+c(-2, 2)
>    a <- 3
>    if (length(Scale) != dim(pos)[1]) {
>       scale <- rep(1, dim(pos)[1])
>    } else {
>    scale <- Scale
>    }
>    if (Border == 0) {
>       for (i in 1:dim(pos)[1]) {
>          polygon(pos[i,1]+scale[i]*hex[,1], a*pos[i,2]+scale[i]*hex[,2], col=Col[i],
border=Col[i])
>       }
>    } else {
>       for (i in 1:dim(pos)[1]) {
>          polygon(pos[i,1]+scale[i]*hex[,1], a*pos[i,2]+scale[i]*hex[,2], col=Col[i],
border=Border)
>       }
>    }
>    par(oldpar)
> }

> update.hexagon(pos.col.scale[,1:2], Col = pos.col.scale[,3]+1, Scale=2*pos.col.scale[,4],
Bord=0)
> CC <- t(t(Coords)-c(0,max(Coords[,2])))
> CC[,2] <- -CC[,2]
> draw.hexagon.3count(CC, hits, Col="white", BG="white")
```

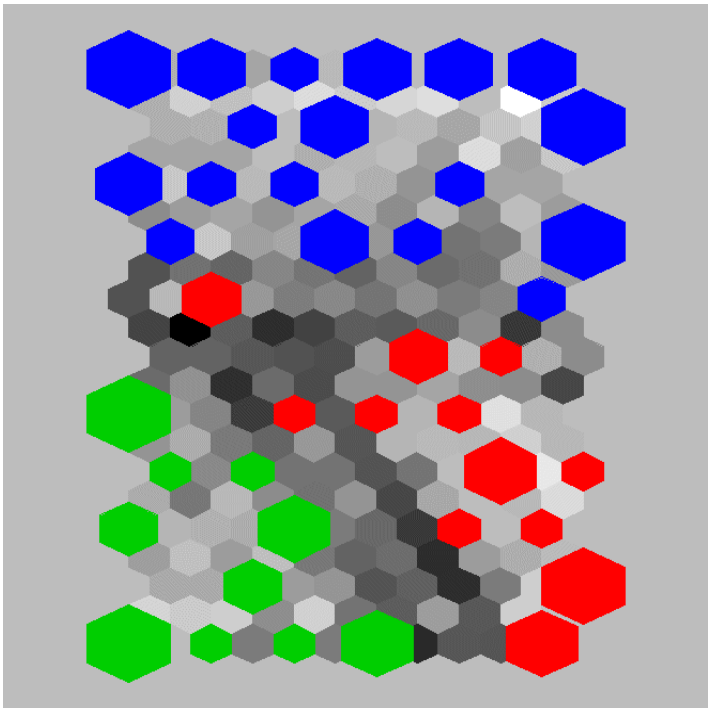

Figure 54.

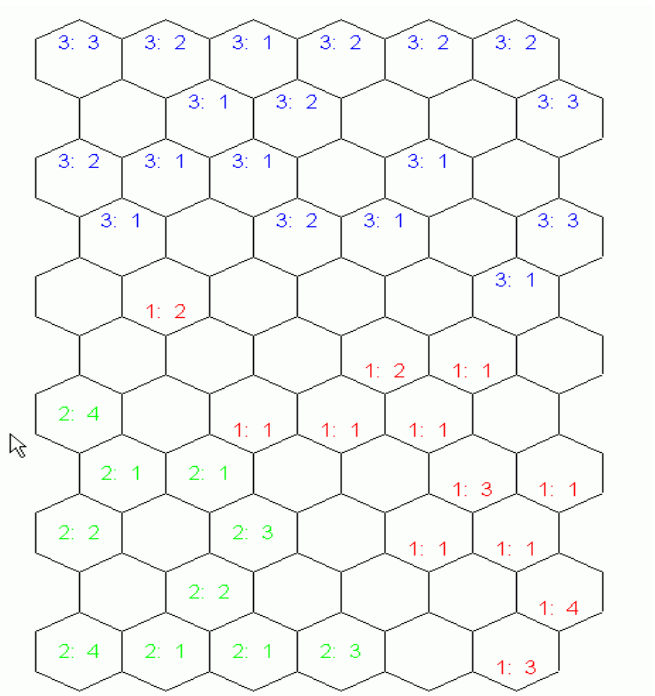

Figure 55.

Bear in mind that, in the hit matrix, the large hexagons mean only that the cell has the maximum number of

hits possible for that class. Hence, the size of the hexagon relates only to that class and cannot be used to compare classes. The figure on the right contains the actual counts for each class and it can be seen that the large hexagons for the 'blue' class represent 3 hits, while for the 'red' and 'green' they represent 4 hits.

There is a package in R called `som` that will do similar computations,

```
> library(som)
> lsom.flea <- normalize(d.flea, byrow=FALSE)
> (flea.som <- som(lsom.flea, xdim=6, ydim=11, topol="hexa", init="linear"))
Initialization:  linear
Topology:  hexa
Neighborhood type:  gaussian
Learning rate type:  inverse
Initial learning rate parameter:  0.05 0.02
Initial radius of training area:  6 3
Average quantization error:  1.097787
Average distortion measure:  13.34978 with error radius:  1
The code book is:
   x  y nobs          1           2           3           4           5           6
1  0  0    0 -0.20670700 -1.968373490 -1.87802451 -1.63815937  0.09473000  -1.30886196
2  1  0    1  0.26693589 -1.531249870 -1.55117936 -1.43504918  0.48451020  -1.30325821
3  2  0    2  0.70386110 -1.139395895 -1.26158888 -1.26261890  0.84238101  -1.30968798
4  3  0    0  1.13296898 -0.778393321 -1.00028859 -1.11601406  1.19429176  -1.33322012
5  4  0    0  1.58699986 -0.427421171 -0.75100494 -0.98592078  1.57040585  -1.37807121
6  5  0    0  2.09002250 -0.065903776 -0.49791010 -0.86310169  1.99192426  -1.44602570
7  0  1    0 -0.35084348 -1.633704621 -1.53786920 -1.26397297 -0.06921021  -0.95165334
8  1  1    5  0.10190757 -1.228750304 -1.23909455 -1.08375096  0.30383357  -0.95671373
9  2  1    5  0.51714894 -0.868190899 -0.97662388 -0.93307531  0.64407967  -0.97261758
10 3  1    4  0.92764190 -0.532408307 -0.73511270 -0.80186409  0.98123710  -1.00247055
11 4  1    2  1.36770103 -0.198216301 -0.49717188 -0.68005750  1.34681395  -1.05061814
...
> plot(flea.som)
```

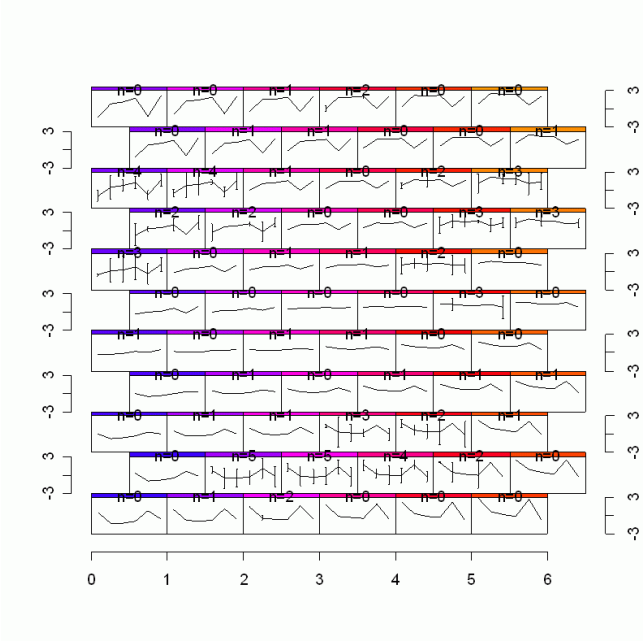

Figure 56. SOM plot

We can also look at how SOM might deal with clusters.

Example: Start with three Gaussians in two dimensions.

```
> grid.size <- c(10, 10)
>    # Draw a hexagonal lattice
> H.coords <- get.hex.grid(c(grid.size[1],grid.size[2]))
> connect.Map(grid.size[1],grid.size[2], H.coords,
>               c(-1,grid.size[1]),c(-1,grid.size[2]))

> Z <- create.gaussians(3, 2)
> plot (Z[,1],Z[,2])
> n.lattice <- prod(grid.size)
>    # Init code book vectors
> X1 <- runif(n.lattice)*20 - 10
> Y1 <- runif(n.lattice)*20 - 10
> pv <- cbind(X1, Y1)

> H.distance <- as.matrix(dist(H.coords))^2
> connect.Map(grid.size[1],grid.size[2], pv, XL=c(-15,15), YL=c(-15,15))
> points(Z[,1],Z[,2],col="red",pch="+")
> trainlen <- 5
> radius <- seq((trainlen-1), 0, by=-1)/(trainlen - 1) +1
> radius <- radius^2
> pv <- batch.train.SOM(trainlen, pv, Z, dim(pv)[1], dim(Z)[1],H.distance, radius, 2)
> connect.Map(grid.size[1],grid.size[2], pv, XL=c(-15,15), YL=c(-15,15))
> points(Z[,1],Z[,2],col="red",pch="+")
```

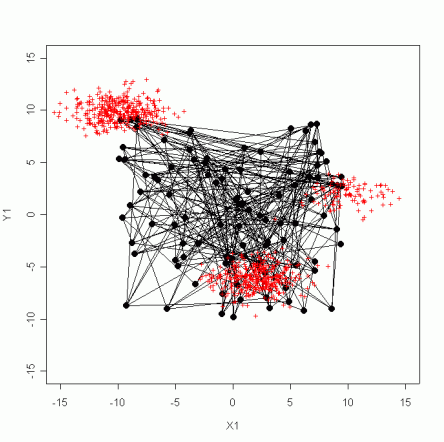

Figure 57. The starting configuration. The prototype vectors are randomly distributed.

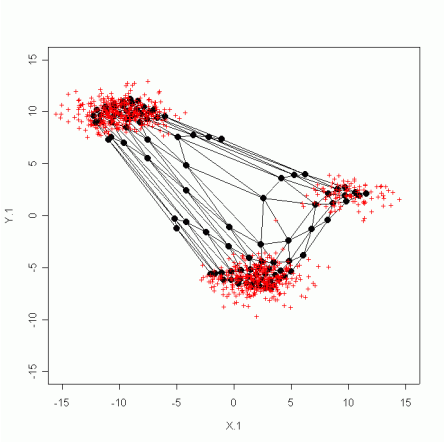

Figure 58 After the initial training phase. Prototype vectors are concentated at the clusters.

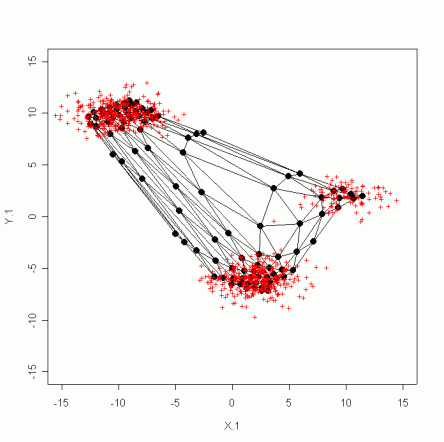

Figure 59. After fine tuning Not much change from the initial training.

The following shows how the SOM quantizes colours. We need a function to convert our data (in $[0,1]$) to a colour.

```
>    # Convert a triple of numbers to RGB colours.
> set.RGB <- function (Z) {
```

```
>    rgb(Z[,1],Z[,2],Z[,3])
> }
```

Now create three random uniform sets of numbers, create a three column matrix, and get the corresponding colours.

```
> n.cases <- 500
> n.var <- 3
> X.C <- runif(n.cases)
> Y.C <- runif(n.cases)
> Z.C <- runif(n.cases)
> Z <- cbind(X.C, Y.C, Z.C)
> Z.col <- set.RGB(Z)
```

As before we will bind our information into a structure.

```
> SOM.struct <- list(data=Z, labels="", col.names=c("R","G","B"), ave={}, sd={})
>    # 'apply' applies the function (say 'mean') to each column (2) of the data.
> SOM.struct$ave <- apply(SOM.struct$data, 2, mean)
> SOM.struct$sd <- apply(SOM.struct$data, 2, sd)
> Data <- SOM.struct$data
```

The following code is similar to that shown earlier.

```
> grid.size <- c(11,6)
> g.units <- prod(grid.size)
```

Establish a grid on $[-1, 1] \times [-1, 1]$.

```
> grid <- expand.grid(((1:grid.size[1])*2-1-grid.size[1])/(grid.size[1]-1),
> ((1:grid.size[2])*2-1-grid.size[2])/(grid.size[2]-1))
> H.coords <- get.hex.grid(grid.size[2:1])
>    # This gives the distances^2 in the output (hexagonal) space
> H.distance <- as.matrix(dist(H.coords))^2
>    # Init code book vectors
> X1 <- runif(g.units)
> Y1 <- runif(g.units)
> Z1 <- runif(g.units)
> pv <- cbind(X1, Y1, Z1)
> pv.col <- set.RGB(pv)
```

Now plot the initial configuration. (`connect.3D` is similar to the `connect.Map`.)

```
> if (dev.cur()!=1) dev.off(dev.cur())
> x11(width=20,height=10)
> par(bg = "white")
> split.screen(c(1,2))
> screen(1)
> draw.hexagon(H.coords, Col=pv.col)
> screen(2)
> connect.3D(grid.size[2], grid.size[1], pv, Col=pv.col,
>          xlab="Red", ylab="Green", zlab="Blue")
```

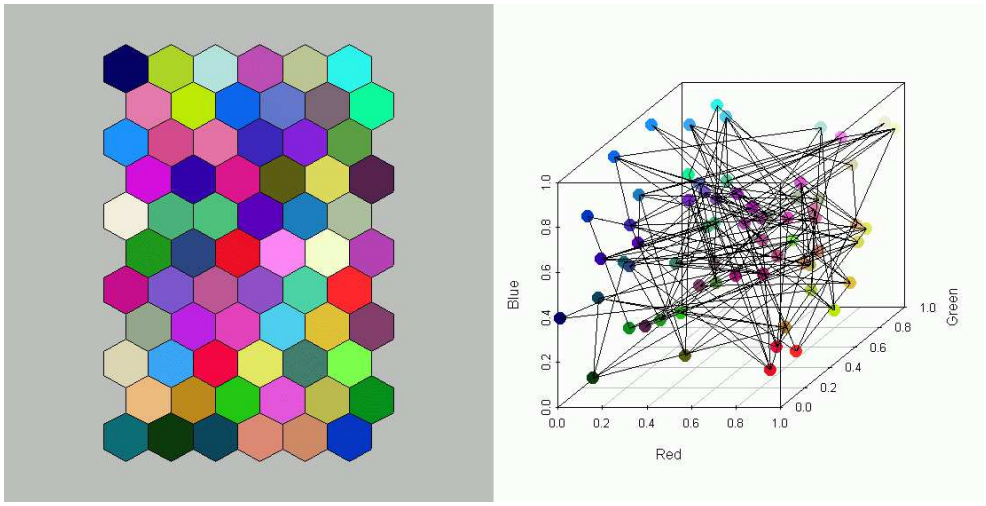

Figure 60.

```
>    # Neighbourhood radius
> r <- 4*(1-(1:60)/60)+1
> for (i in 1:60) {
>    pv <- iterative.train.SOM(Z, pv, rep(r[i], 5), 5, rep(0.1, 5), Coords, H.distance)
>    pv.col <- set.RGB(pv)
>    screen(1)
>    draw.hexagon(H.coords, Col=pv.col, main=paste(i*5, "Epochs"))
>    screen(2)
>    connect.3D(grid.size[2], grid.size[1], pv, Col=pv.col,
>    xlab="Red", ylab="Green", zlab="Blue")
>    readline("Press Enter...")
> }
> close.screen(all=T)
```

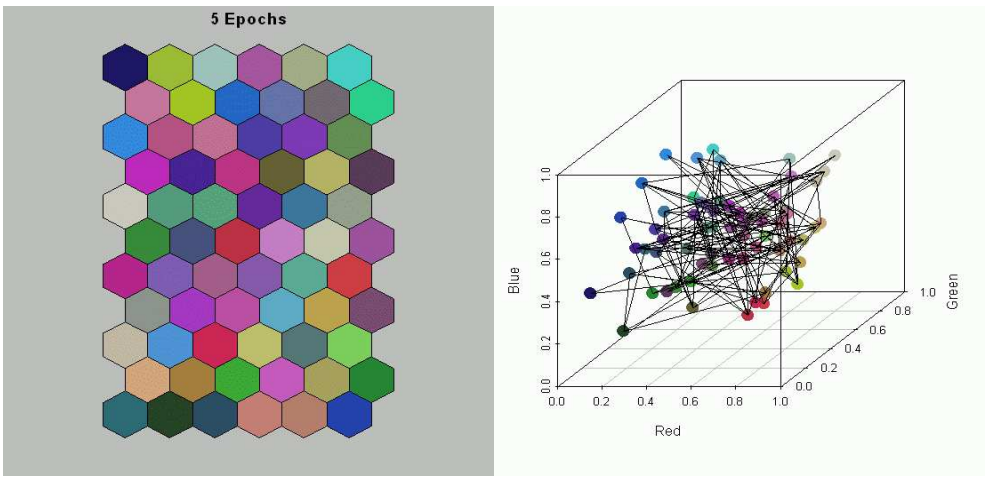

Figure 61.

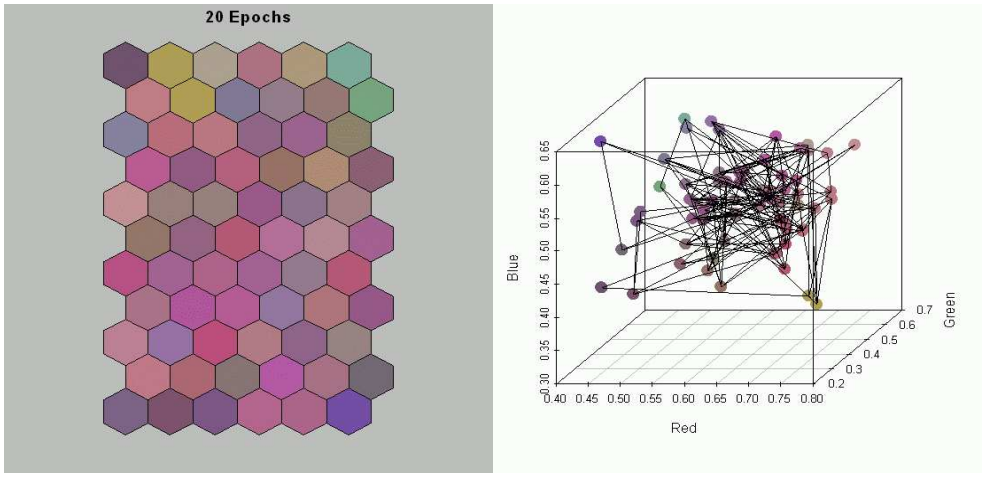

Figure 62.

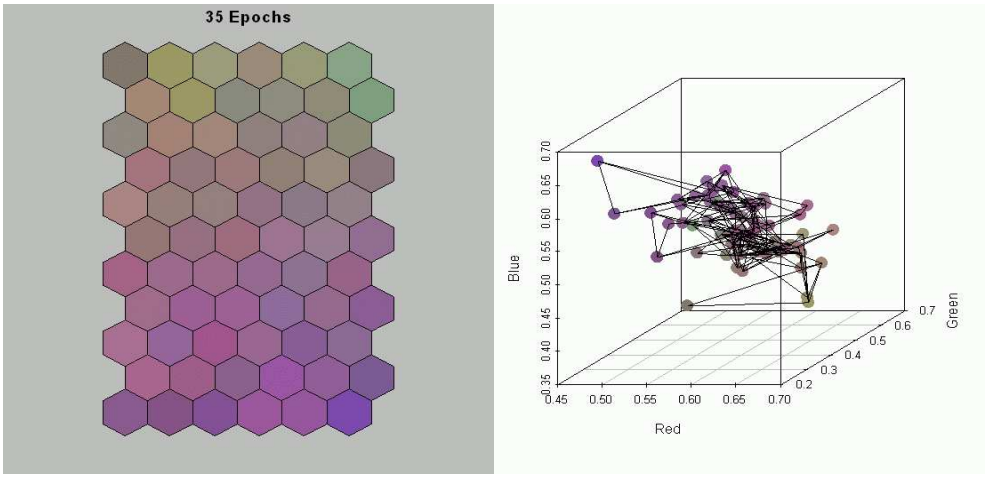

Figure 63.

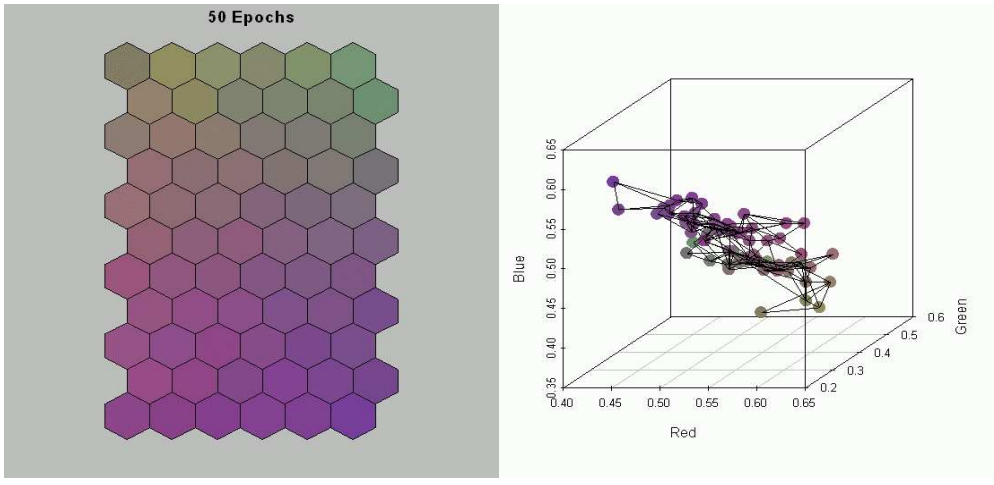

Figure 64.

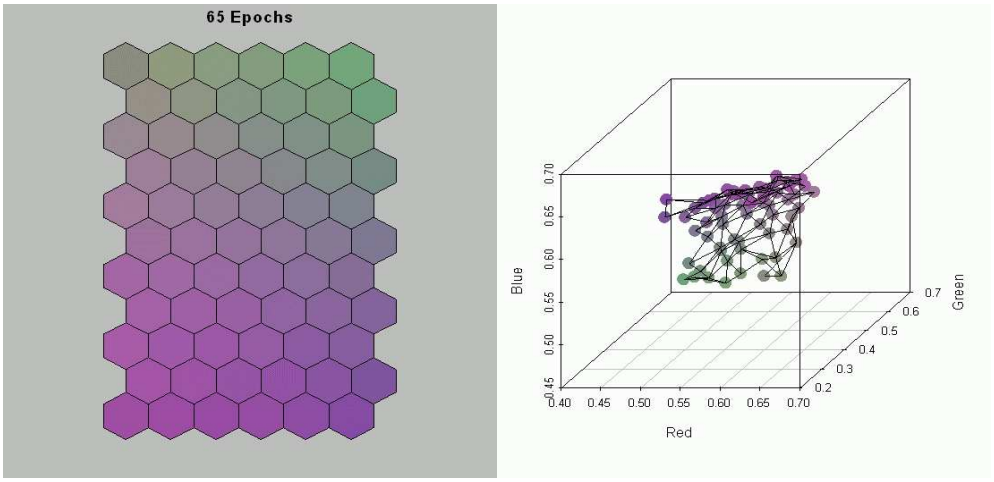

Figure 65.

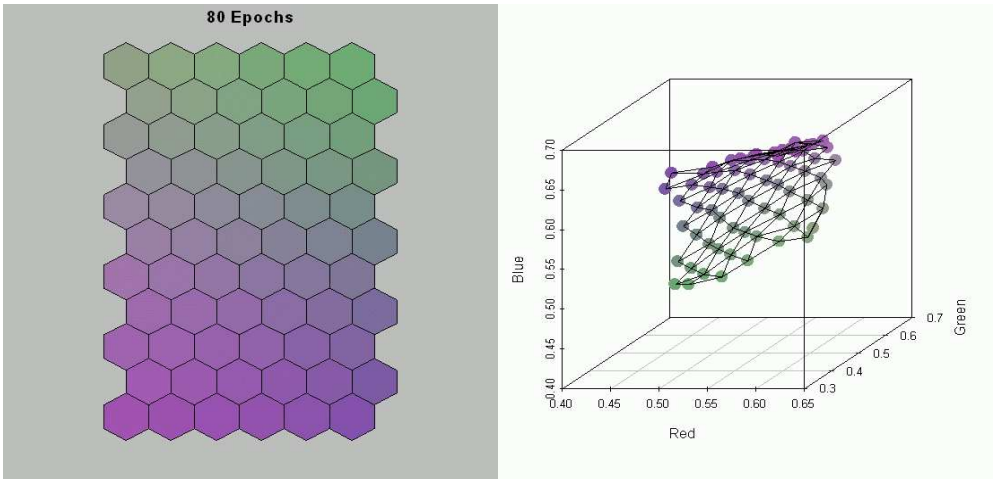

Figure 66.

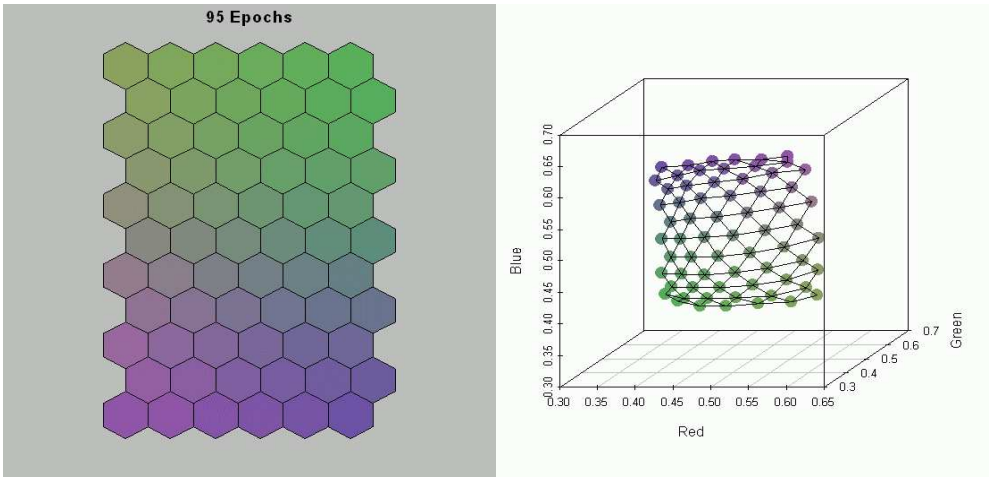

Figure 67.

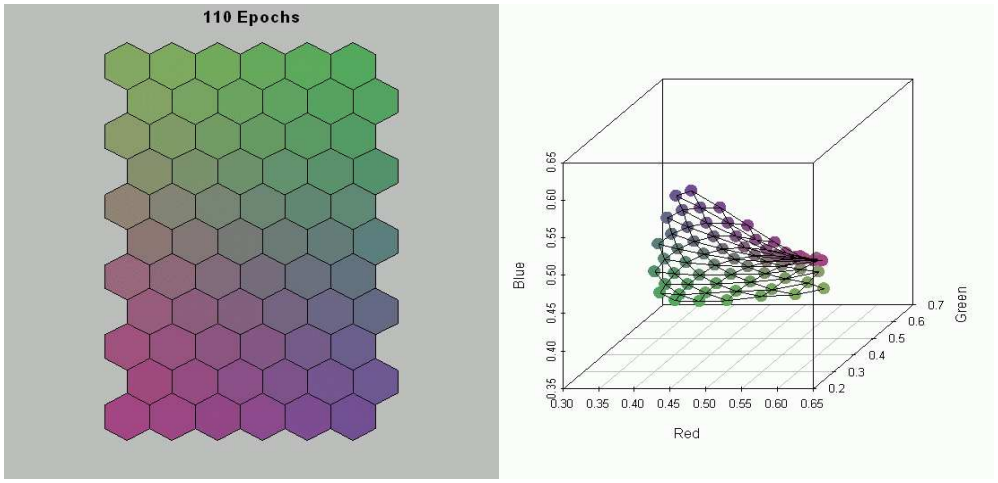

Figure 68.

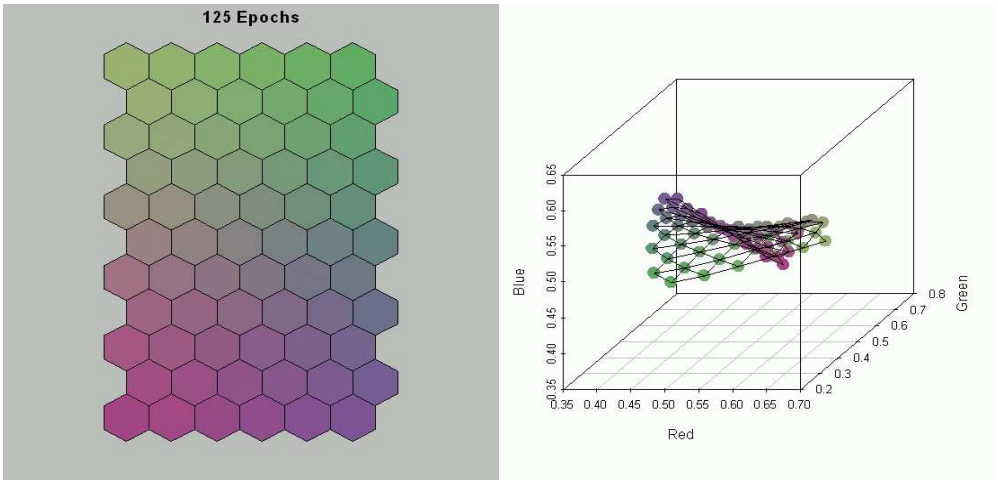

Figure 69.

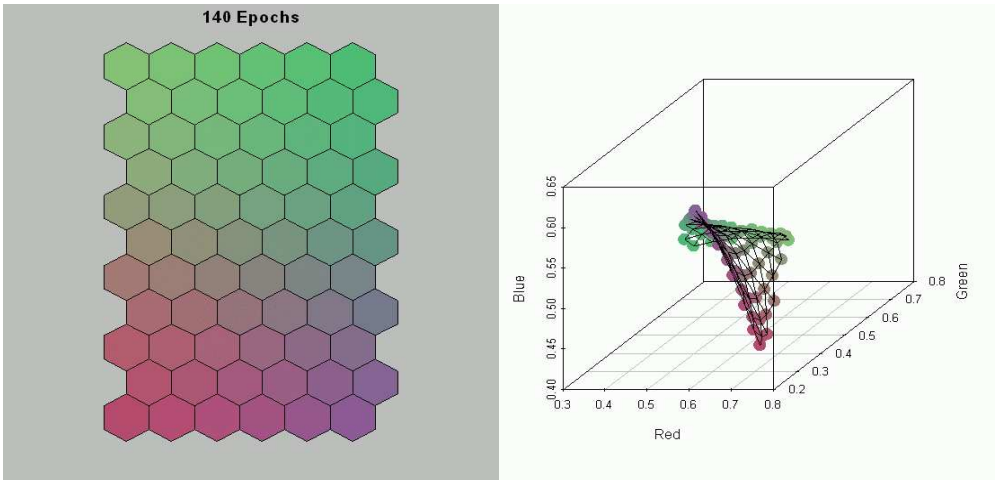

Figure 70.

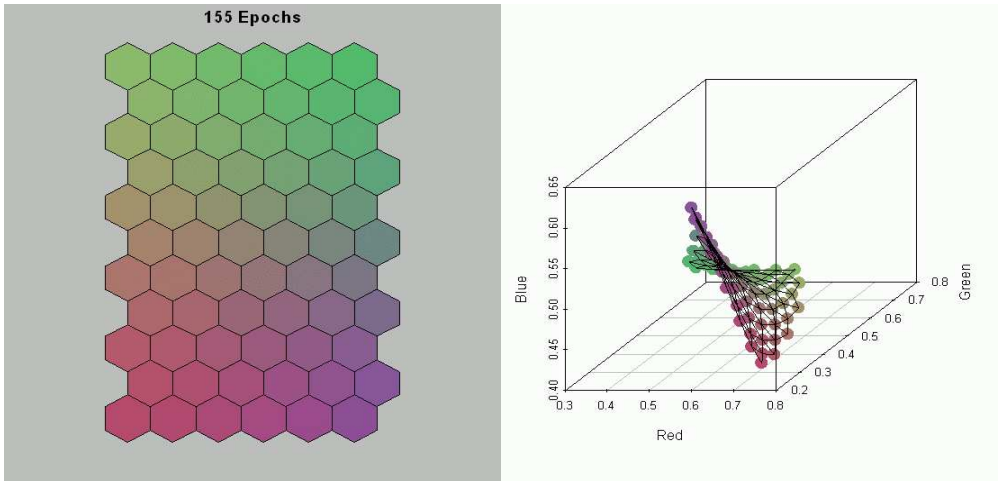

Figure 71.

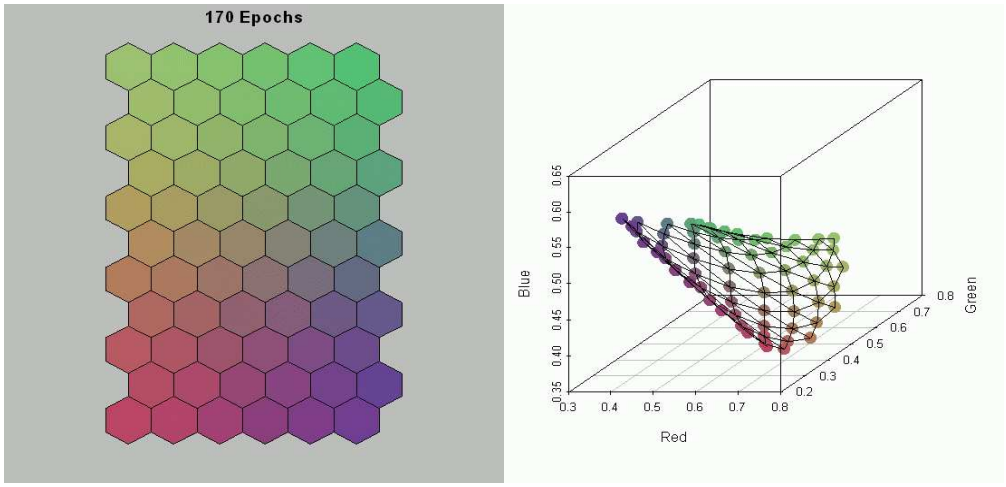

Figure 72.

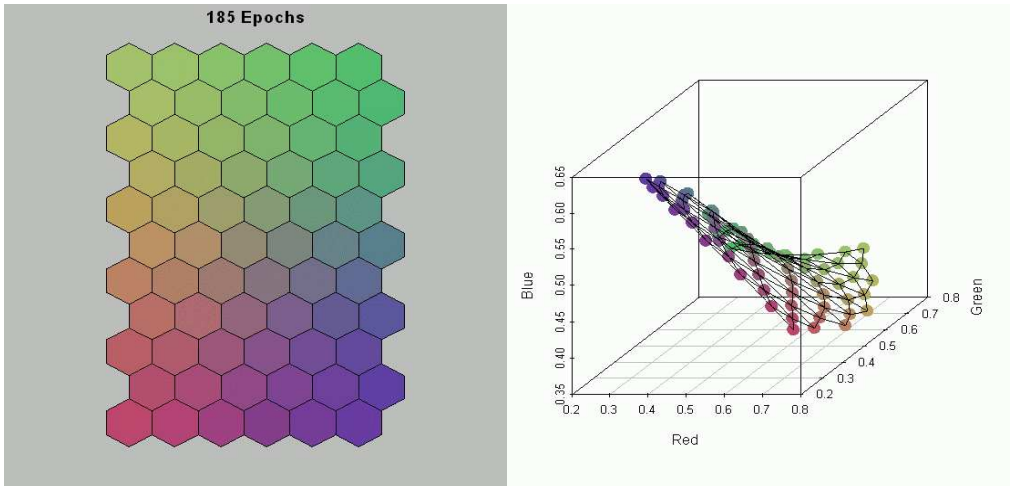

Figure 73.

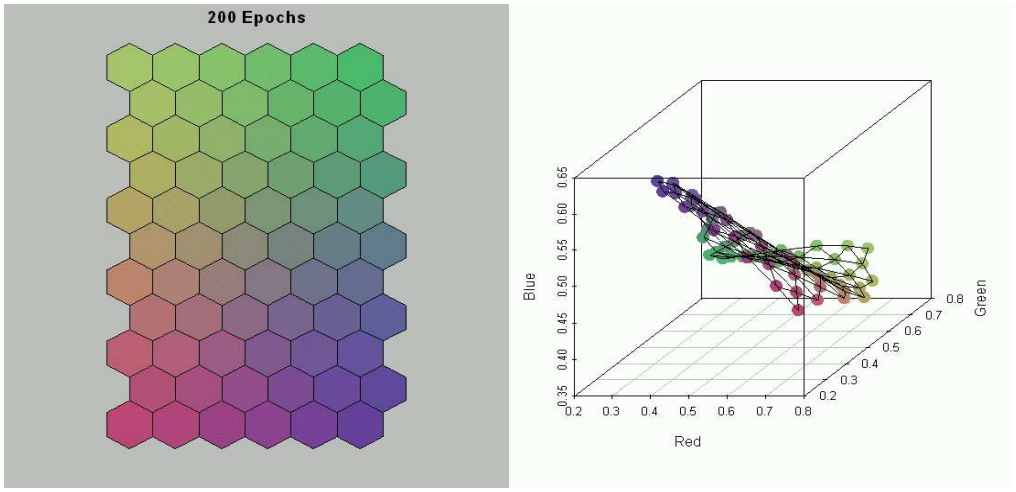

Figure 74.

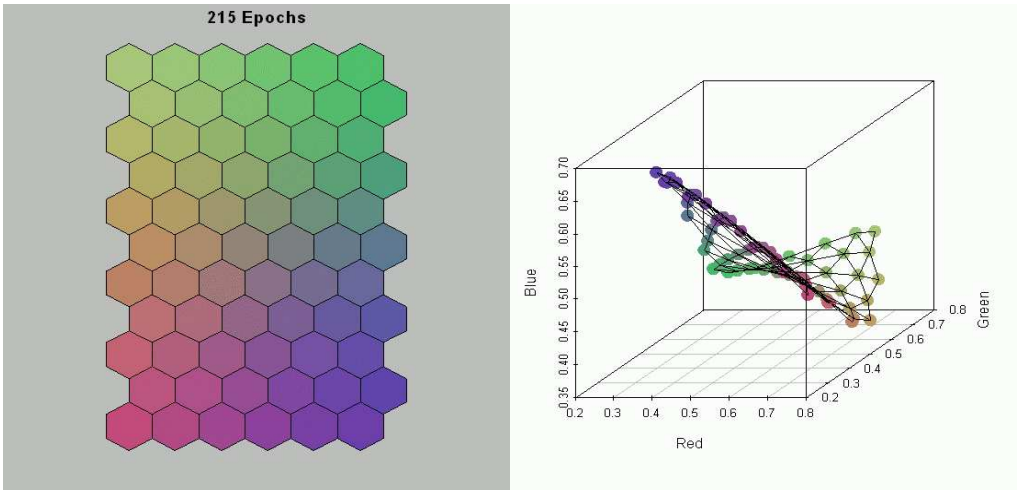

Figure 75.

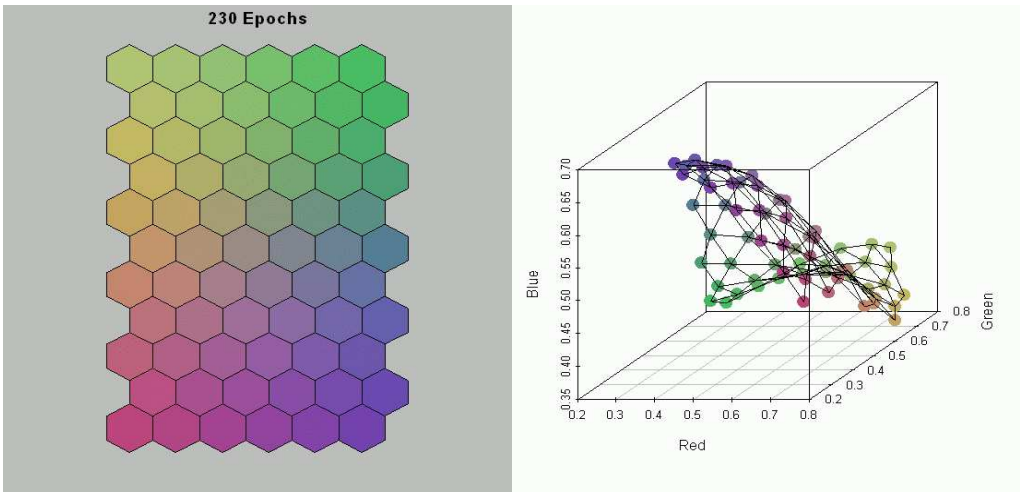

Figure 76.

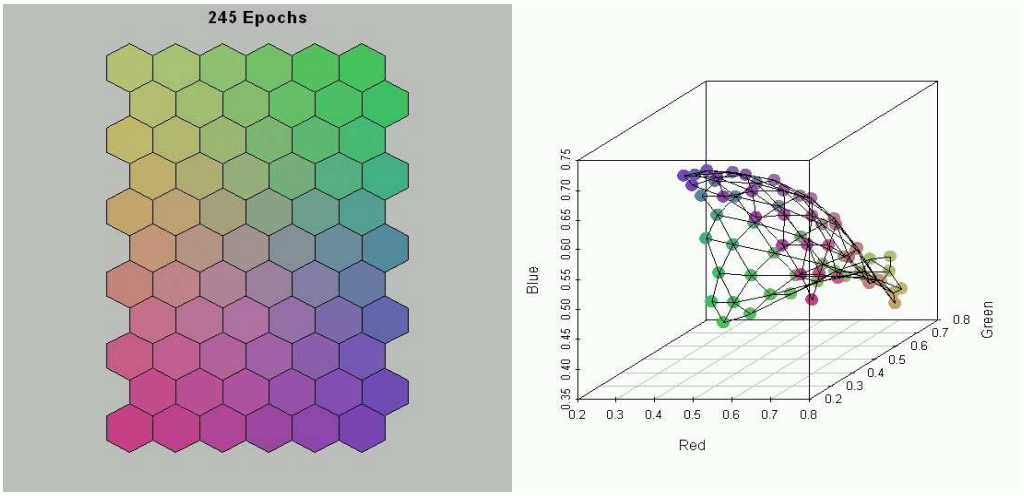

Figure 77.

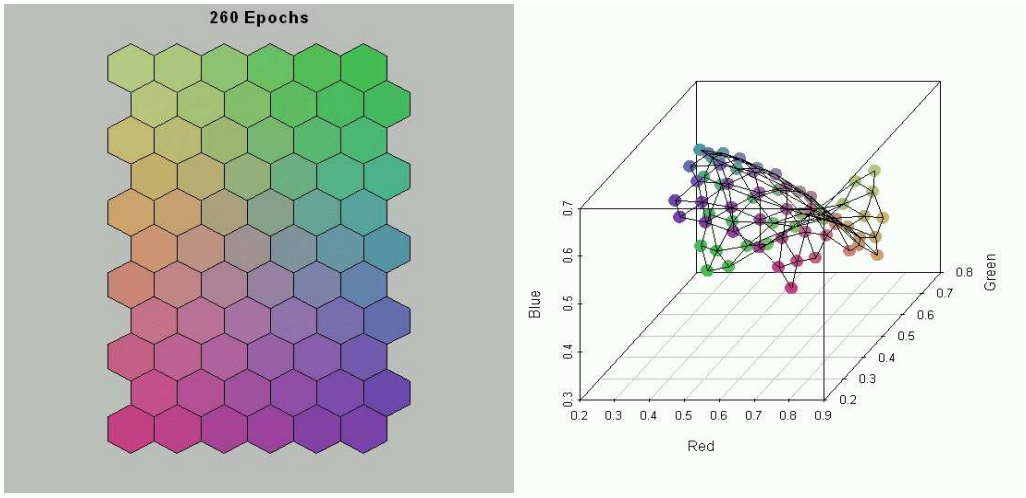

Figure 78.

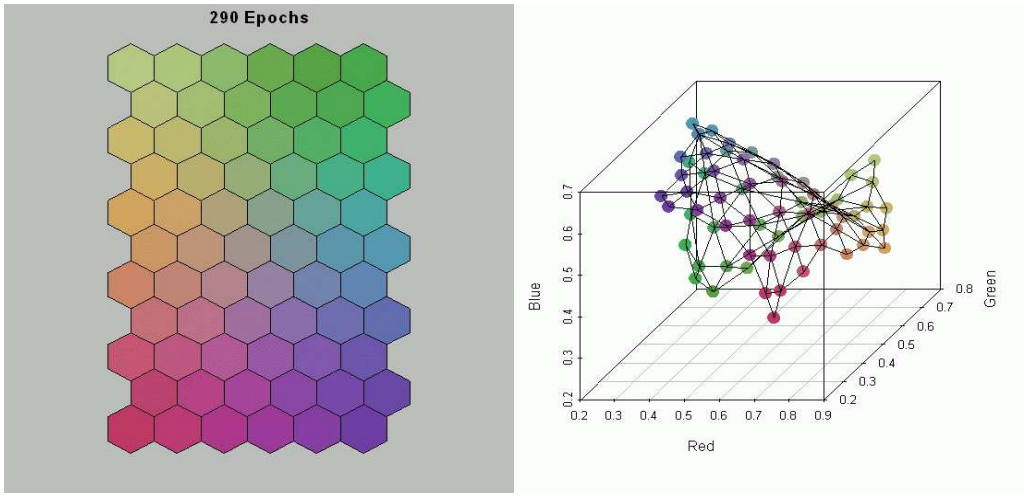

Figure 79.

In any process such as this, It is important to keep in mind that different initializations can lead to quite different results. It is a good idea to repeat the SOM several times to make sure that you get an appropriate result. The problem is that processes such as this can take a lot of time (for large problems) so you might be

tempted to settle for your first result.

Vector Quantization with SOM - Lena revisited

The SOM algorithm is similar in many respects to the *k*-means algorithm so it seems reasonable to see how it functions in the Vector Quantization sense.

To do this we use the same process as we used in the previous lecture - convert the 256×256 picture into a $16,384 \times 4$ array by grouping blocks of 4 pixels.

```
> source(paste(rcode.dir, "VQ_helpers.r", sep=""))
> library(pixmap)
> d.file <- paste(data.dir, "lena_bw.pgm", sep="")
> x <- read.pnm(d.file)
> if (dev.cur()[[1]]!=1) bringToTop(which=dev.cur())
> plot(x)
> x@size
> pix <- x@grey
>    #============ 1/4 of data ==============
> blocked.4 <- deconstruct.4(pix)
> Rows <- dim(pix)[1]
> Cols <- dim(pix)[2]
```

Codebook from *k*-means (see VQ)

```
> numb.centers <- 100
> classes.100.4 <- {}
> while (length(classes.100.4) == 0) {
>    res <- try(kmeans(blocked.4, numb.centers, 50))
>    if (length(res) > 1) classes.100.4 <- res
> }
>    # Get integer values for the centroids.
>    # Each of the blocks has an associated centroid so we have
>    # 16384 indices to the 100 4-D codebook vectors
> new.bytes.4 <- floor(classes.100.4$centers[classes.100.4$cluster,]*256)/256
```

We now do a SOM on the blocked data (the code is the same as we saw earlier).

```
>    # Get the number of cases
> n.cases <- dim(blocked.4)[1]
>    # Put all the info into a structure
> SOM.struct <- list(data=blocked.4, labels="",
>                    col.names="", ave={}, sd={})
>    # 'apply' applies the function (say 'mean') to each column (2) of the data.
> SOM.struct$ave <- apply(SOM.struct$data, 2, mean)
> SOM.struct$sd  <- apply(SOM.struct$data, 2, sd)
>
```

Put the prototype vectors on the principal components plane.

```
> Data <- div.arr.vec(add.arr.vec(SOM.struct$data, -SOM.struct$ave), SOM.struct$sd)
> eigs <- eigen((t(Data)%*%Data)/n.cases)
> Vec <- eigs$vectors[,1:2]
> Val <- eigs$values[1:2]
>    # Get L2 norm
> norm.V <- sqrt(diag(t(Vec)%*%Vec))
>    # Scale the vector
> n.V <- mult.arr.vec(div.arr.vec(Vec, norm.V), sqrt(Val))
```

Now choose a grid size for the prototype vectors.

```
> g.size <- c(10, 10)
> g.units <- prod(g.size)
>
> cb <- matrix(rep(c(0,0,0,0), g.units), g.units, 4, byrow=T)
>    # Establish a grid on [-1, 1] x [-1, 1]
> grid <- expand.grid(((1:g.size[1])*2-1-g.size[1])/(g.size[1]-1),
>                     ((1:g.size[2])*2-1-g.size[2])/(g.size[2]-1))
```

Project on the eigenvector plane.

```
> cb <- cb + t(n.V%*% t(grid))
> lattice.points <- prod(g.size)   #  Number of points on lattice
> H.coords <- get.hex.grid(g.size)
> H.distance <- as.matrix(dist(H.coords))^2
> trainlen <- 5
> radius <- seq((trainlen-1), 0, by=-1)/(trainlen - 1) +1
> radius <- radius^2
```

Do the initial training

```
> cb <- batch.train.SOM(trainlen,  cb, Data, g.units, n.cases, H.distance, radius, 4)
```

and the fine tuning.

```
> trainlen <- 18
> radius <- rep(1, trainlen)
> cb <- batch.train.SOM(trainlen, cb, Data, g.units, n.cases, H.distance, radius, 4)
```

Find the BMU.

```
> Dx2 <- t(2*Data)
> Dist <- matrix((apply(cb*cb, 1, sum)), g.units, n.cases) -  cb%*%Dx2
> min.index <- apply(Dist, 2, order)[1,]
> new.bytes.4.SOM <- floor(cb[min.index,]*256)/256
>
```

We need to denormalize for plotting.

```
> new.bytes.4.SOM.denorm <- add.arr.vec(mult.arr.vec(new.bytes.4.SOM, SOM.struct$sd), SOM.struct
>
```

We next find the densities of the hits on each centroid/prototype vector so that we can plot the densest first.

```
> dens.k <- {}
> for (i in 1:100) {
>    dens.k <- c(dens.k, sum(sort(classes.100.4$cluster)==i))
> }
> dens.k <- cbind(1:100, dens.k)
> dens.k <- dens.k[order(-dens.k[,2]),]
> dens <- {}
> for (i in 1:100) {
>    dens <- c(dens, sum(sort(min.index)==i))
> }
> dens <- cbind(1:100, dens)
> dens <- dens[order(-dens[,2]),]
> print(cbind(dens.k, dens))
          dens.k     dens
  [1,]  30    590   1 1145
  [2,]  60    525 100  957
  [3,]  98    507  34  394
  [4,]  37    487  47  392
  [5,]  64    469   2  376
  [6,]  82    468  38  360
  [7,]  83    464  11  341
```

```
  [8,]  38    437  97  339
  [9,]  69    434  88  318
 [10,]  55    421  99  313
 [11,]  80    399  85  308
 [12,]  66    395  43  289
 [13,]  73    392  86  287
 [14,]  94    390   3  286
 [15,]   8    385  21  266
 [16,]  45    369  36  249
 [17,]  61    363  77  240
 [18,]  90    363  90  234
 [19,]  39    331  35  232
 [20,]   3    327  45  231
 [21,]  18    322  46  213
 [22,]  91    321  59  211
 [23,]  31    311  58  209
 [24,]  47    311  64  207
 [25,]  40    309  68  207
 [26,]  57    289  66  204
 [27,]   1    286  69  204
 [28,]  19    246  31  199
 [29,]  43    227  50  199
 [30,]   2    224  24  198
 [31,]  34    224  98  191
  ...
 [49,]   5    105  32  119
 [50,]  67    101  44  114
 [51,]  88     94  61  113
 [52,]  53     88  37  109
 [53,]  20     84  41  107
 [54,]  46     76  95  106
 [55,]  86     74  60  105
  ...
 [90,]  65     23  17   60
 [91,]  56     21  20   54
 [92,]  71     21  29   54
 [93,]  44     19  91   54
 [94,]  85     18  81   51
 [95,]  92     18   8   49
 [96,]  24     15  62   45
 [97,] 100     15  48   43
 [98,]  48     13  92   41
 [99,]   9     11  82   35
[100,]  84     11  19   27
```

Note that the SOM hits are quite large on the first two prototype vectors, but are then fewer until the middle and remain higher until the end. If we look at the total number of pixels that are painted (see below), we see that the SOM method paints more for the first 11 prototypes and then falls behind. As we will see, this shows up in the form of more white pixels for the SOM until the very end at which point all pixels are painted.

```
    K-means SOM
 [1,]   590 1145
 [2,]  1115 2102
 [3,]  1622 2496
 [4,]  2109 2888
 [5,]  2578 3264
 [6,]  3046 3624
 [7,]  3510 3965
 [8,]  3947 4304
 [9,]  4381 4622
[10,]  4802 4935
[11,]  5201 5243
[12,]  5596 5532
```

```
[13,]  5988 5819
[14,]  6378 6105
[15,]  6763 6371
[16,]  7132 6620
[17,]  7495 6860
[18,]  7858 7094
[19,]  8189 7326
[20,]  8516 7557
[21,]  8838 7770
[22,]  9159 7981
[23,]  9470 8190
[24,]  9781 8397
[25,] 10090 8604
[26,] 10379 8808
[27,] 10665 9012
[28,] 10911 9211
[29,] 11138 9410
[30,] 11362 9608
[31,] 11586 9799
> x11(width=8, height=4)
> #=========================================
> # A function to produce those indices NOT in a set. Use this to get the test sample.
> #=========================================
> "%w/o%" <- function(x,y) x[!x %in% y]
```

We will look at the reconstruction of the image as each new centroid or codebook vector is added in.

We scale the centroids (and codebook vectors) for plotting

```
> tmp <- floor(classes.100.4$centers*256)/256
```

and put them in order of number of hits

```
> centroid.ord <- tmp[dens.k[,1],]
```

Do the same for the codebook vectors

```
> tmp <- add.arr.vec(mult.arr.vec(floor(cb*256)/256, SOM.struct$sd),
>                    SOM.struct$ave)
> cb.ord <- tmp[dens[,1],]
```

This will enable us to display the structure of the new elements being added to the image at each step.

We will plot in order of the number of positions in the image associated with each centroid (codebook vector).

`ind` and `ind.k` give the indices the items currently used in the construction.

```
> ind <- {}                # Index of points to plot for SOM
> ind.k <- {}              # Index of points to plot for k-means
> oldpar <- par(mar=c(1,1,1,1))
> for (p in 1:100) {
>    ind.k <- c(ind.k, (classes.100.4$cluster==format(dens.k[p,1]))*(1:dim(Data)[1]))
>    not.ind.k <- (1:dim(Data)[1])%w/o%ind.k
>    temp <- new.bytes.4
>       # Block out all the points that are not in the current plot
>    temp[not.ind.k,]<-1
>    z.4 <- x
>    z.4@grey <- reconstruct.4(temp, Rows, Cols)
>
>    ind <- c(ind,
(dimnames(new.bytes.4.SOM.denorm)[[1]]==format(dens[p,1]))*(1:dim(Data)[1]))
>    not.ind <- (1:dim(Data)[1])%w/o%ind
>    temp <- new.bytes.4.denorm
>       # Block out all the points that are not in the current plot
```

```
>     temp[not.ind,]<-1
>     z.4.SOM <- x
>     z.4.SOM@grey <- reconstruct.4(temp, Rows, Cols)
>     close.screen(all = TRUE)
>     split.screen(c(1,2))            # split display into two screens
>     screen(1)
>     plot(z.4, main=paste("k-means of 4, # of cb =",p))
>     screen(2)
>     plot(z.4.SOM, main=paste("SOM of 4, # of cb =",p))
>        # Show the blocks of 4
>     plot(pixmapGrey(matrix(centroid.ord[p,], 2, 2), nrow=2))
>        #
>     plot(pixmapGrey(matrix(cb.ord[p,], 2, 2), nrow=2))
> }
> par(oldpar)
```

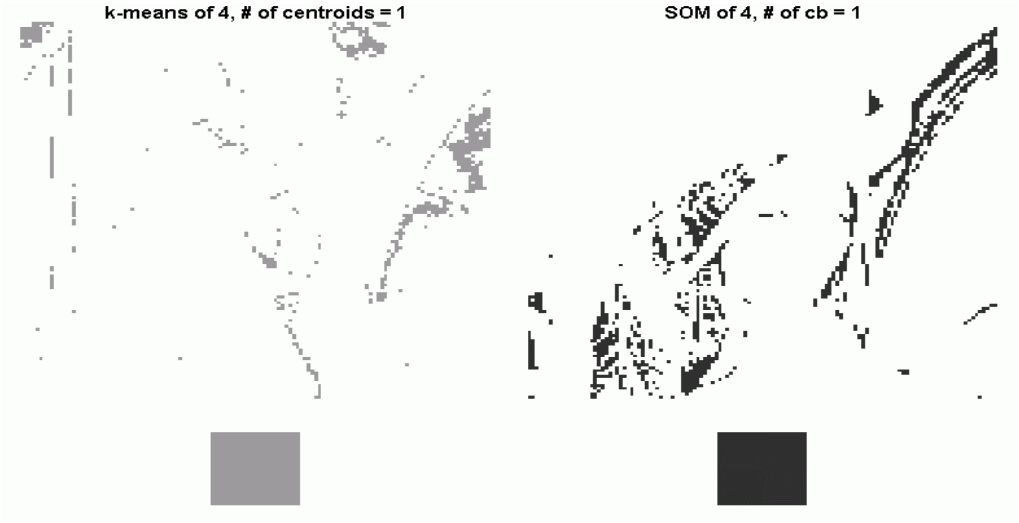

Figure 80.

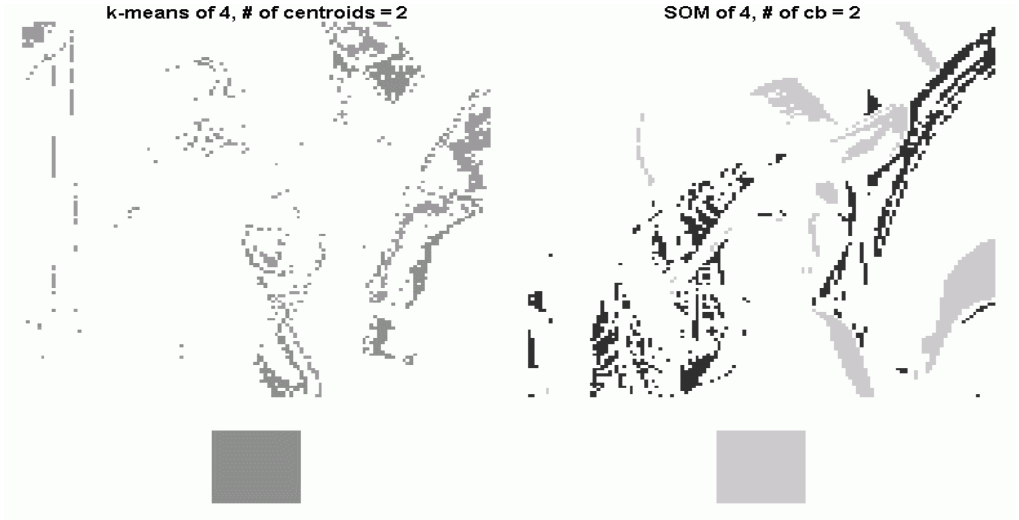

Figure 81.

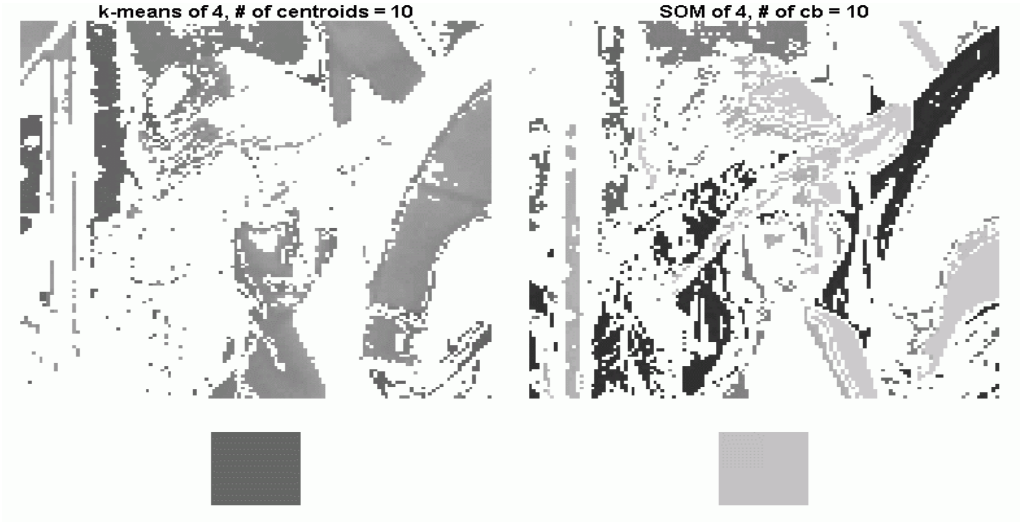

Figure 82.

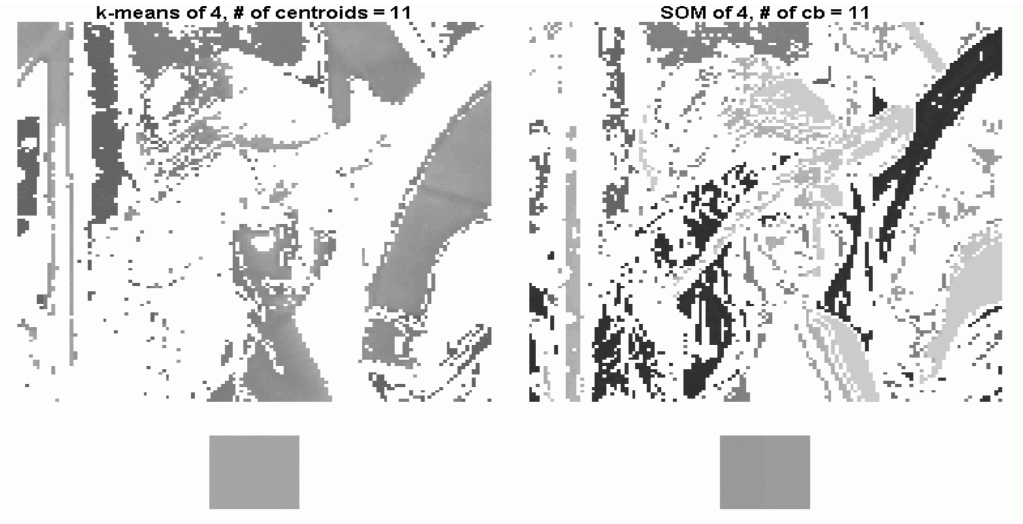

Figure 83.

Figure 84.

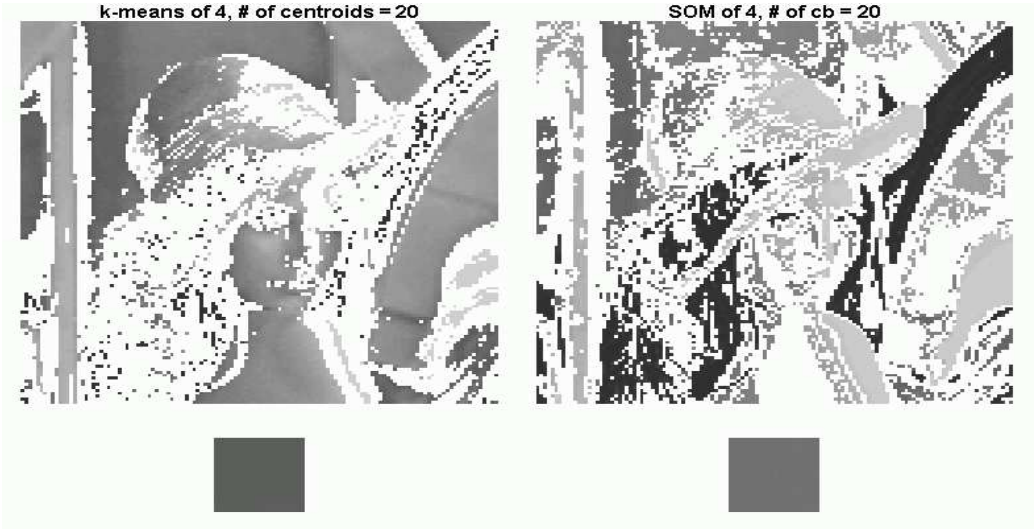

Figure 85.

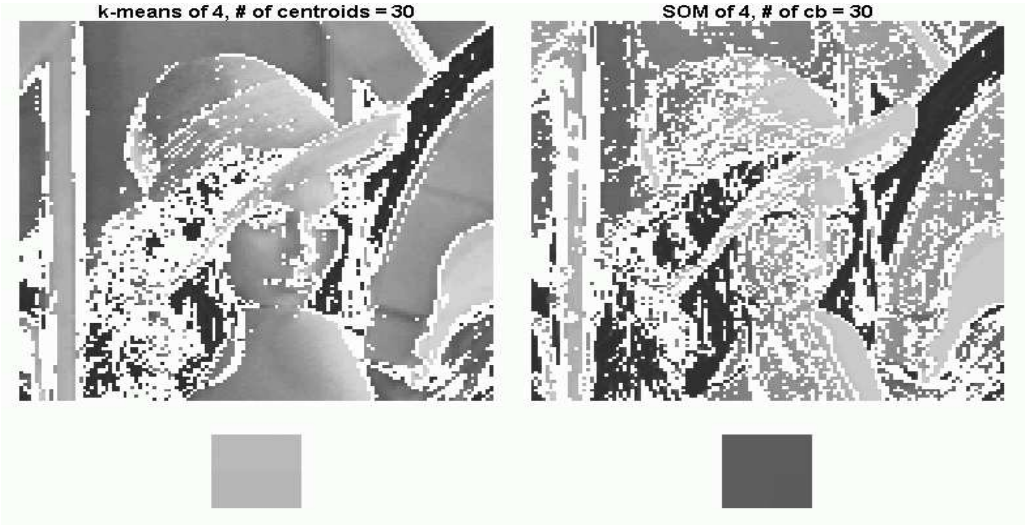

Figure 86.

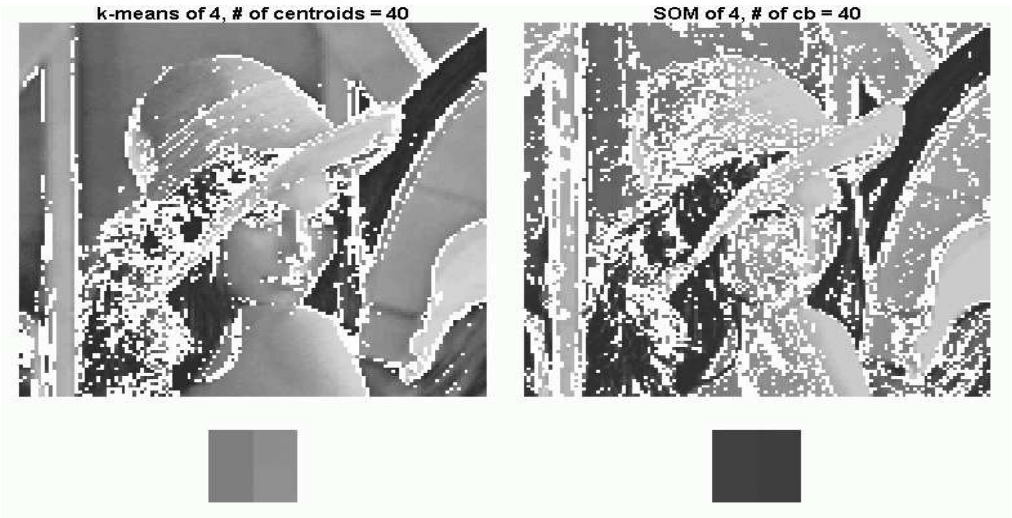

Figure 87.

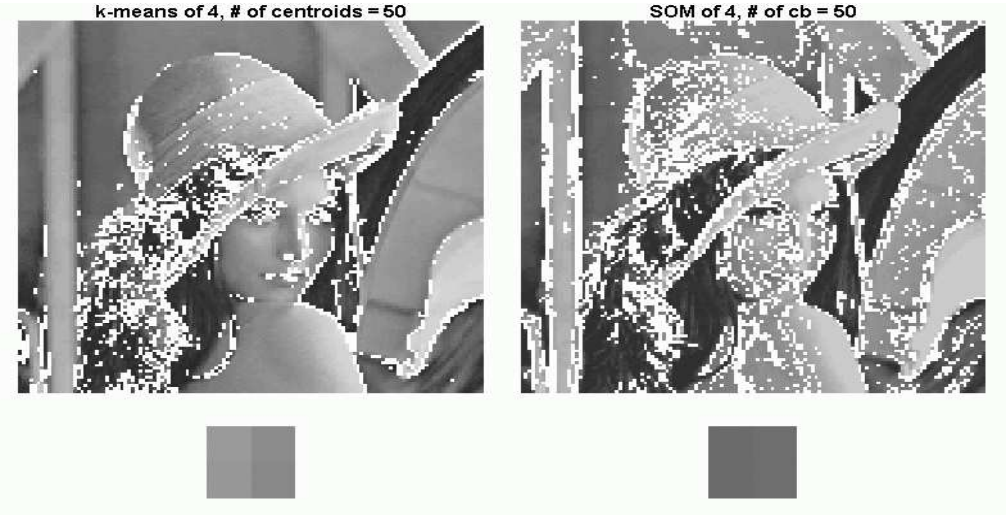

Figure 88.

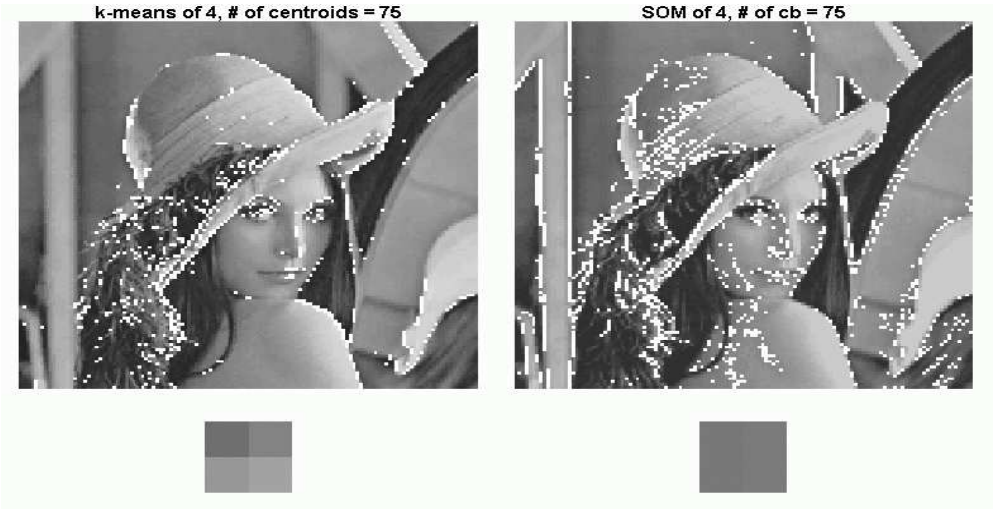

Figure 89.

Figure 90.

Figure 91.

Figure 92.

As before, we can look at the detail and the difference between the original and the quantized versions,

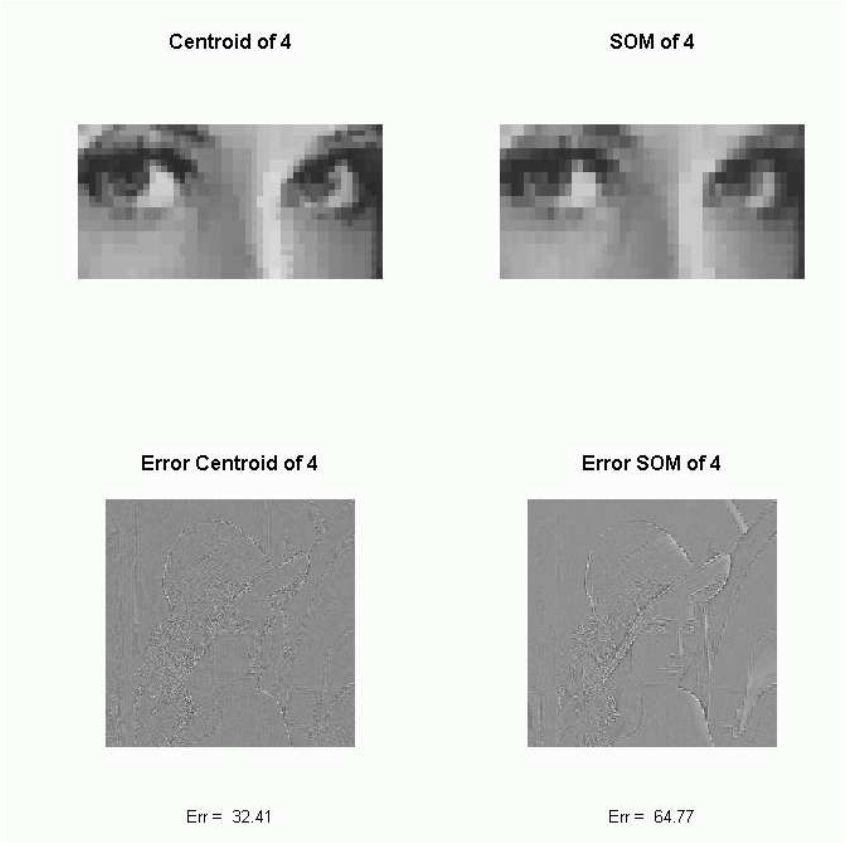

Figure 93.

It appears that the *k*-means (centroid) method has produced a better result in this case.

Local PCA

While PCA can be used to reduce the dimension of high dimensional data, it is often the situation that the data will be of lower dimension in only *part* of the space. One method that has been suggested to deal with this is a local PCA.

The concept is that of using clustering to identify local structure and then do a PCA on that structure. As a simple example consider the universe. Looked at from the PCA point of view, the stars, planets etc. will be 3 dimensional although the galaxies are individually fairly flat (2-dimensional).

First read in some files that contain code that we need.

```
> drive <- "D:"
> code.dir <- paste(drive, "DATA/Data Mining R-Code", sep="/")
> data.dir <- paste(drive, "DATA/Data Mining Data", sep="/")
> source(paste(code.dir, "EllipseCloud.r", sep="/"))
> source(paste(code.dir, "3DRotations.r", sep="/"))
> source(paste(code.dir, "MakeStereo.r", sep="/"))
> source(paste(code.dir, "VQ_helpers.r", sep="/"))
> source(paste(code.dir, "array2arraydist.r", sep="/"))
```

We will use

```
> library(scatterplot3d)
```

As an example, suppose *we put six Gaussian clouds on the faces of a cube*.

```
> numb <- 500      # data set with 500 points
> a <- 15         # semi-major axis
> b <- 10         # semi-minor axis
```

```
> c <- 0.0001      # minimal thickness,
> data <- ellipse.cloud(numb, a, b, c, xc=0, yc=0, zc=0)
> data.z.1 <- Rot.translate(data, 0, 0, pi/4, 0, 0, 15)
> data.z.2 <- Rot.translate(data, 0, 0, -pi/4, 0, 0, -15)
> Z.1 <- rbind(data.z.1, data.z.2)
> Z.2 <- R.y(Z.1, pi/2)
> Z.3 <- R.x(Z.1, pi/2)
> Z <- rbind(Z.1, Z.2, Z.3)
> Z <- rbind(Z.1, Z.2, Z.3)
> Z.R <- R.z(Z, pi/8)
> Z.R <- R.x(Z.R, -pi/3)
> Z.R <- R.y(Z.R, +pi/4)
> pairs(Z.R, asp=1)
> make.Stereo(Z.R, c(rep("1",dim(Z)[1])), Main="Ellipses on Cube")
```

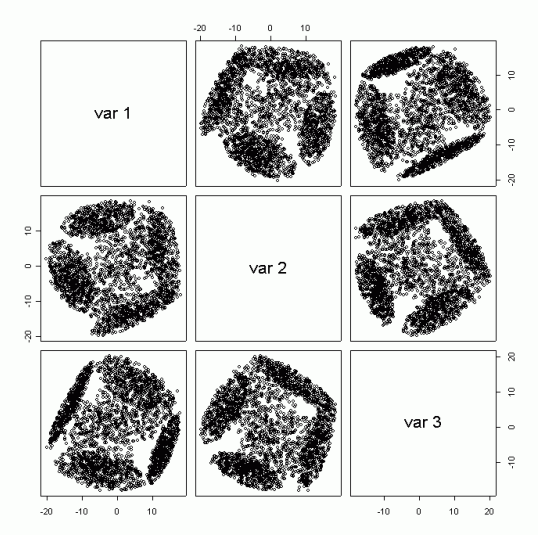

Figure 94.

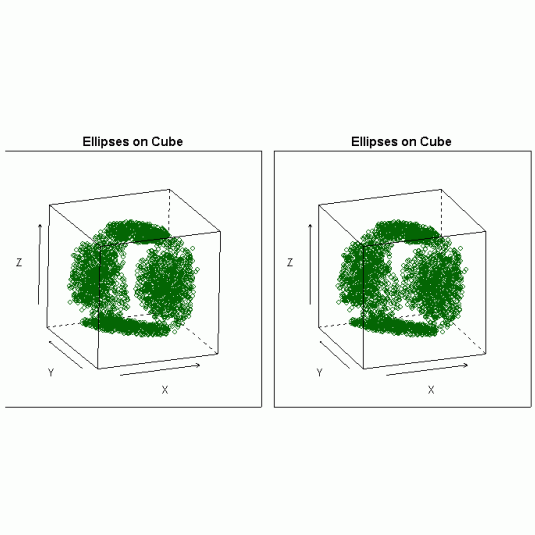

Figure 95.

If we do a PCA on this data we find that the variance is the same in all 3 directions so it is unclear which direction we could ignore with respect to the amount of variation.

```
> prcomp(Z.R)
Standard deviations:
[1] 9.847832 9.842122 9.839066
Rotation:
           PC1         PC2        PC3
[1,] -0.91170190  0.02271629  0.4102239
[2,] -0.08061754  0.96916871 -0.2328365
[3,] -0.40286530 -0.24534871 -0.8817616
```

We will try vector quantization starting with *two* centers that are positioned just slightly off the centre of mass. We use *k*-means to move the centers to the 'best' position and split the center whose 'cluster' has the largest variance. (See notes on Vector Quantization and the figures below.)

```
> Col <- rainbow(10)
> Data <- Z.R
> n.x <- dim(Data)[1]
> k <- 3
> ep <- 0.01
> N <- 1
>    # Put the vector at the mean
```

```
> pv <- rbind(apply(Data, 2, mean))
> s3d <- scatterplot3d(Data)
> s3d$points3d(pv, col="red", pch=16, cex=1.5)
> pv <- rbind(pv, (1+ep)*pv[1,])
> pv[1,] <- (1-ep)*pv[1,]
> N <- 2
> for (loop in (1:5)) {
>    res <- move.centroids(pv, Data, Dim=3)
>    pv <- res[[1]]
>    belongs.to <- res[[2]]
>    for (bt in unique(belongs.to)) {
>       cat(prcomp(Z.R[(belongs.to==bt),])$sdev, "\n")
>    }
>    the.Distortions <- compute.distortion(pv, Data, belongs.to)
>    print(the.Distortions)
>    print("================")
>       # Find the largest
>    worst.d <- order(-the.Distortions)[1]
>    pv <- rbind(pv, (1+ep)*pv[worst.d,])
>    pv[worst.d,] <- (1-ep)*pv[worst.d,]
>    N <- N+1
> }
```

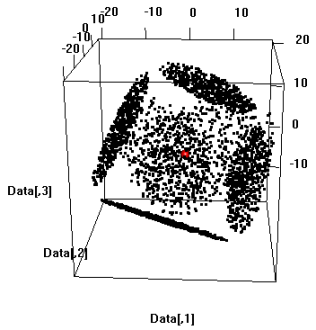

Figure 96. The original center.

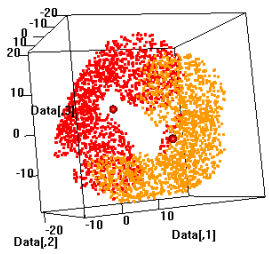

Figure 97. Two 'clusters' just before the next split

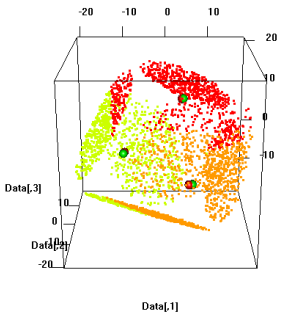

Figure 98. Three centers.

The numbers below give the standard deviations for the principal components of each cluster, followed by the total distortion (variance) of each cluster.

Two clusters:

```
9.604449 9.508693 4.561806
10.29491 10.03966 4.511364
[1] 226.9872 203.3290
[1] "================"
```

Three clusters:

```
10.25674 7.108831 3.757628
9.991916 6.970491 3.558182
9.465971 7.573968 3.176691
[1] 160.9172 169.6924 156.9058
[1] "================"
```

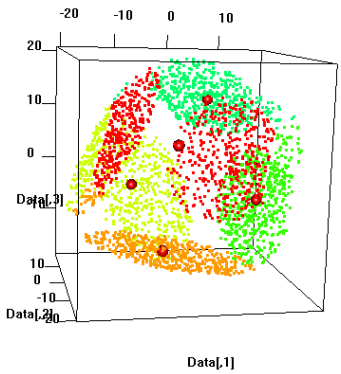

Figure 99. Five centers.

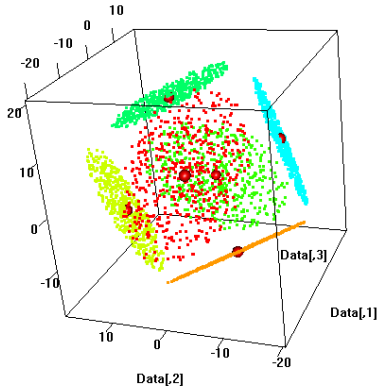

Figure 100. The final configuration

Four clusters:

```
8.239774 6.628257 2.128199
8.210401 8.172371 3.397154
8.205408 7.10965 2.370092
7.582223 6.894833 2.368372
[1] 145.5835 123.3249 110.4681 116.1853
[1] "=================="
```

Five clusters:

```
7.531088 5.595137 1.426861
9.471026 5.388701 3.007637
6.849992 5.413957 0.9876877
8.36528 5.465606 2.076307
7.291604 4.755411 0.9587378
[1] 127.61186 76.55588 103.99595 89.90106 77.06317
[1] "=================="
```

Six clusters:

```
6.685769 4.644615 4.506728e-05
6.685769 4.644615 4.506728e-05
6.685769 4.644615 4.506728e-05
6.685769 4.644615 4.506728e-05
6.685769 4.644615 4.506728e-05
6.685769 4.644615 4.506728e-05
[1] 66.13941 66.13941 66.13941 66.13941 66.13941 66.13941
[1] "=================="
```

We see that the final configuration has 6 centers - and the centers are on the cube.

The standard deviations for each of the principal components of the 6 clusters found are given above and, as can be seen from the following, are the same as the standard deviations for the original cloud.

The principal components of the original cloud are -

```
> prcomp(data)
Standard deviations:
[1] 6.685769e+00 4.644615e+00 4.506728e-05
Rotation:
            PC1           PC2           PC3
x -9.989196e-01 -4.647161e-02  1.994049e-07
y -4.647161e-02  9.989196e-01 -2.243986e-07
```

```
z -1.887613e-07 -2.334228e-07 -1.000000e+00
```

As can be seen, the clustering has isolated the regions in which we could do a local dimension reduction.

Note: A concern with this method is to know when to stop. There are indices (such as the Davies-Bouldin) that penalize the number of clusters by using a formula involving the intra-cluster variance vs. the inter-cluster spacing. A different approach might be to be guided by a reduction of the variance in one of the directions.