HW07

Elliot Smith, Eugen Hruska, Varun Suriyanaranana

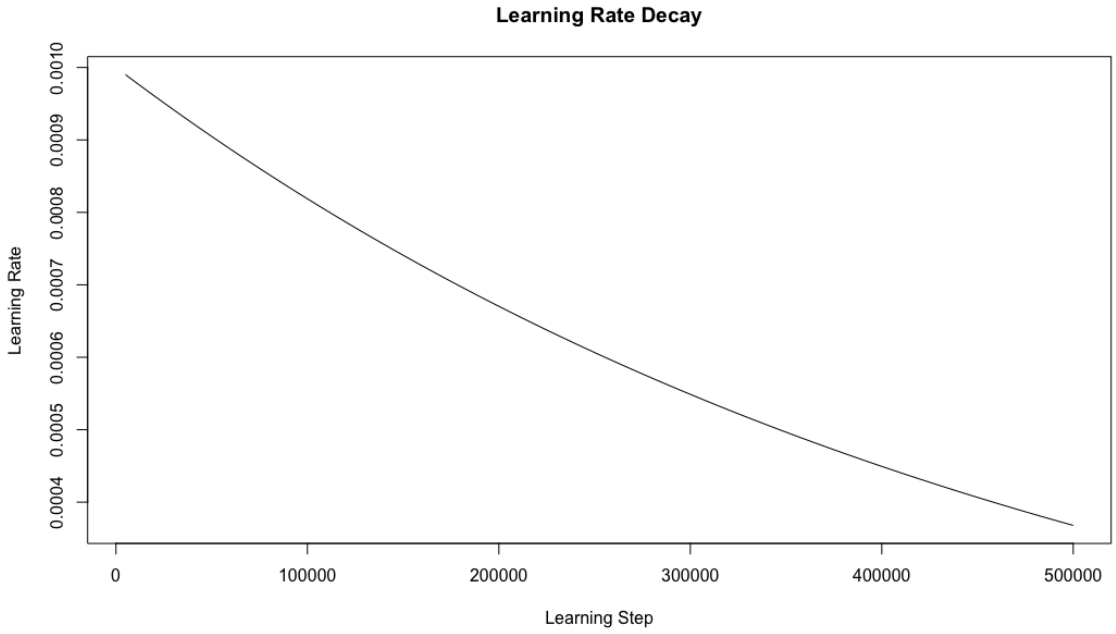
3/21/2018

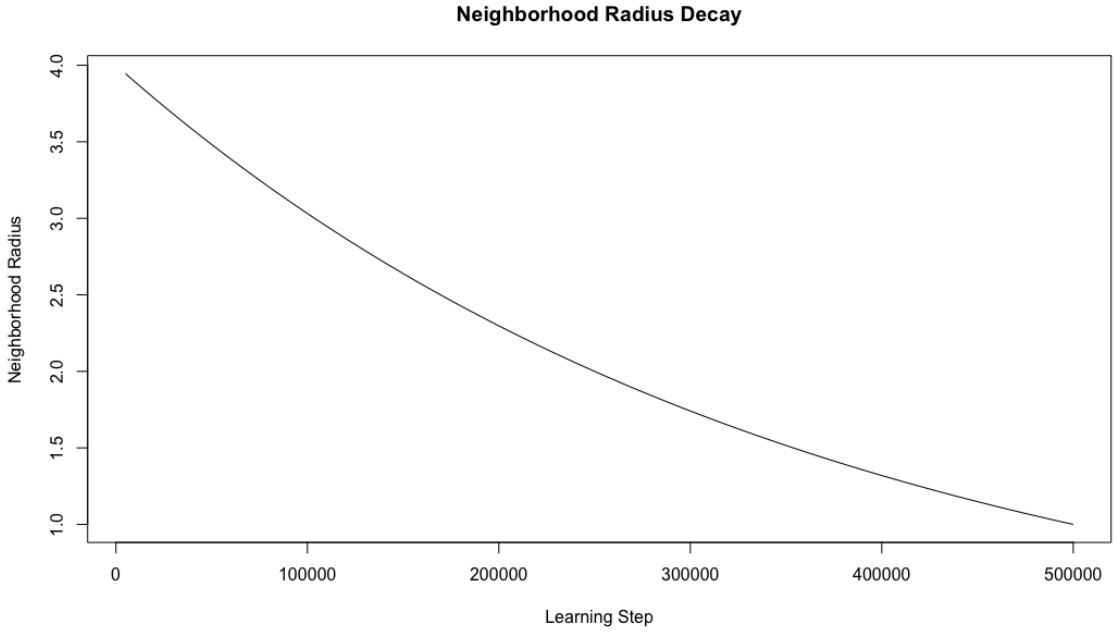
Problem 2a

Parameters

- Network Parameters
  - Topology: 64 PEs (8x8 2-dimensional square lattice)
  - Each PE is a 2-dimensional vector (see below for weight draws)
- Learning Parameters
  - Initial Weights: Drawn from U[0, 1]
  - Initial Learning Rate (α_{init}): 0.001
  - Learning Decay Rate: $\alpha_{init} * e^{\frac{-i}{n}}$
    - i = current learning step
    - n = total number of learning steps
  - Initial Radius (θ_{init}): 4
  - Radius Decay Rate: $\theta_{init} * e^{\frac{-i}{\mu}}$
    - i = current learning step
    - n = total number of learning steps
    - $\mu = \frac{n}{log(\theta_{init})}$
  - Momentum: None
  - Stopping Criteria: 500,000 learning steps
- Input Data
  - 1000 Samples drawn from each (4000 total):
    - Normal[(7,7), 0.1]
    - Normal[(0,7), 0.1]
    - Normal[(7,0), 0.1]
    - Normal[(0,0), 0.1]
- Error and Performance Measure
  - Learning Steps Performed: 500,000
  - Monitoring Frequency: Every 100,000 Learning Steps

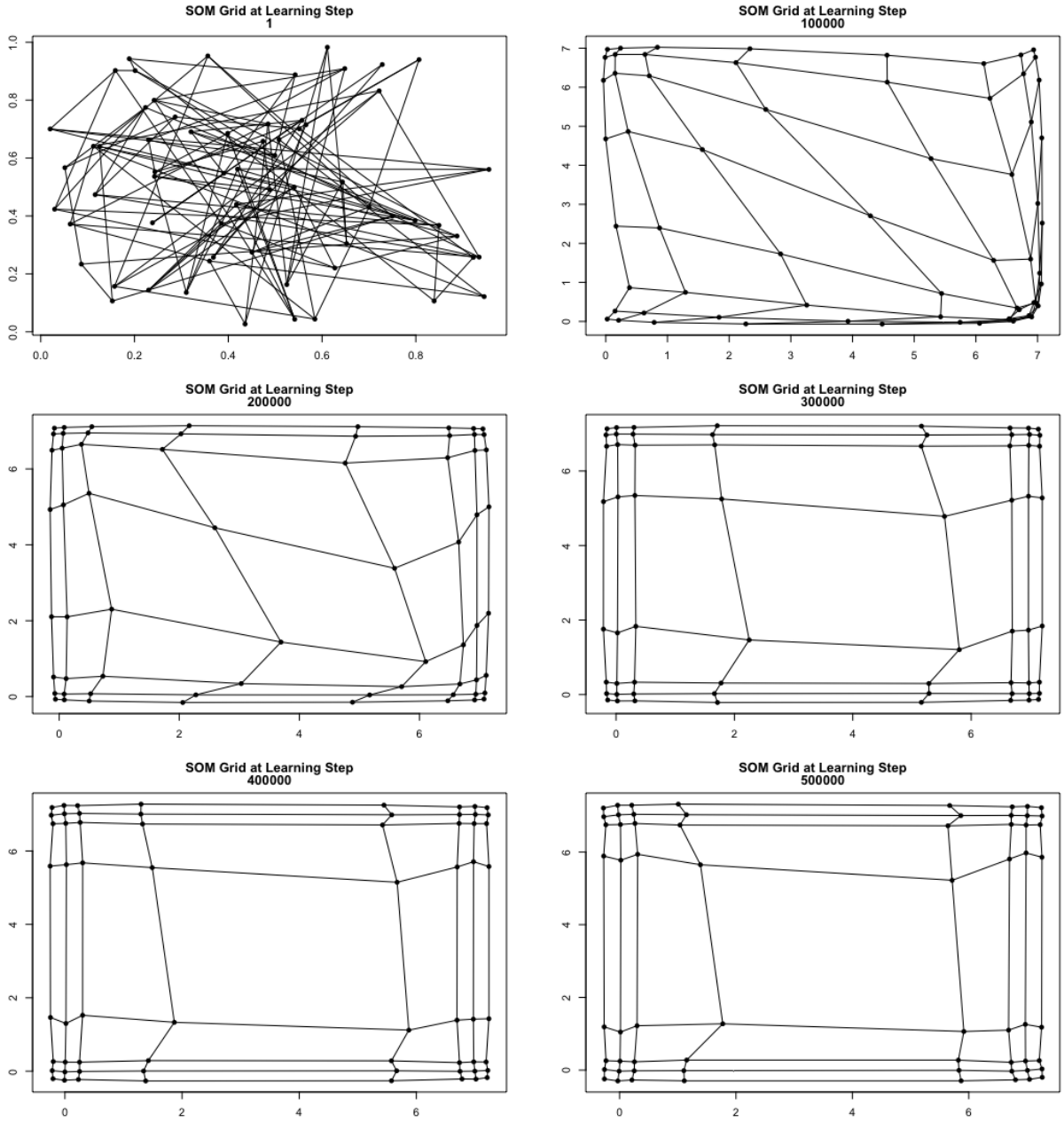
Decay Graphs





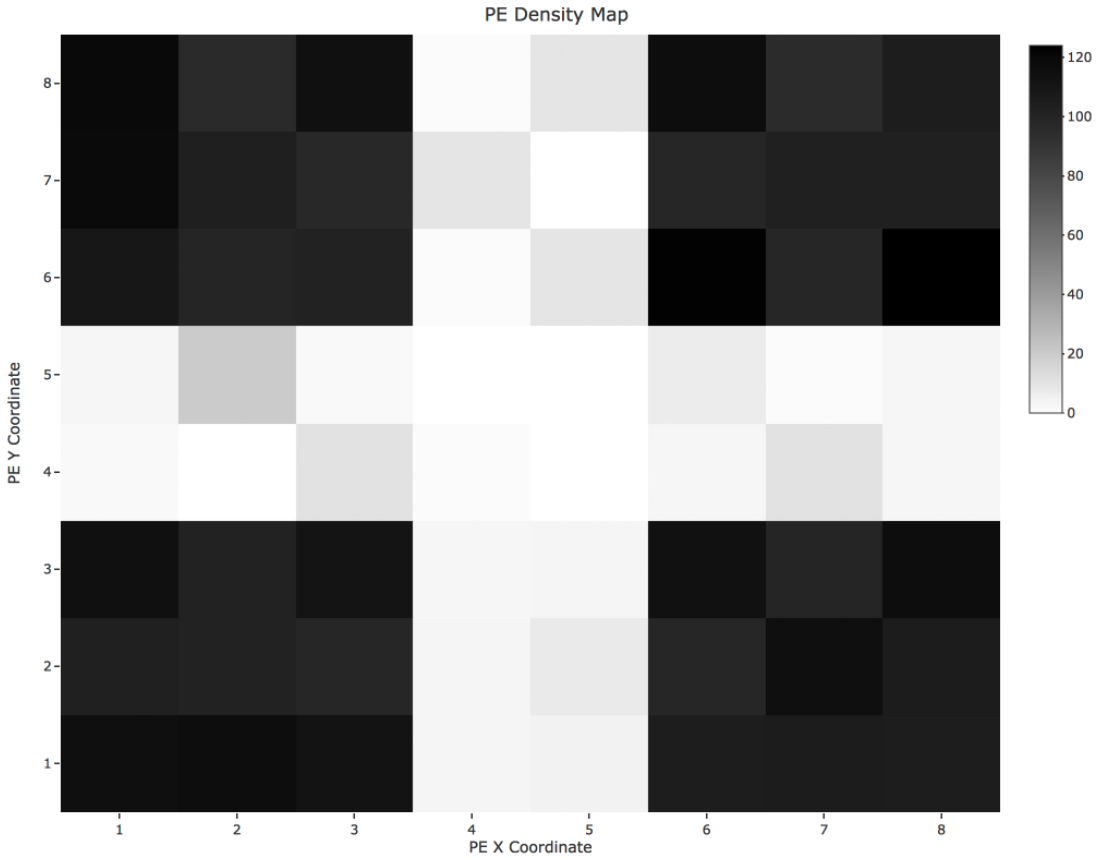
As we can see, we used an exponential decay function to decrease the neighborhood radius and learning rate over time as a function of the numer of learning steps completed; this helped us to obtain very strong network results.

Lattice Learning Graphs



As we can see, we achieved very good results on our data; we can see very clearly at the end of our learning that the prototypes have gravitated to the points (0,0), (0,7), (7,0) and (7,7). We can also see that the "density" of the data at those locations are very similar and almost equal as the prototypes form a symmetry at the four quadrants (this makes sense as our data is equally distributed around those four points). The learning in the first 100,000 steps causes a massive change to the graph showing the prototypes mapped back to the input space, while as the learning continues it changes less drastically, both as a result of our parameter decay and the convergence of the network to our stable solution.

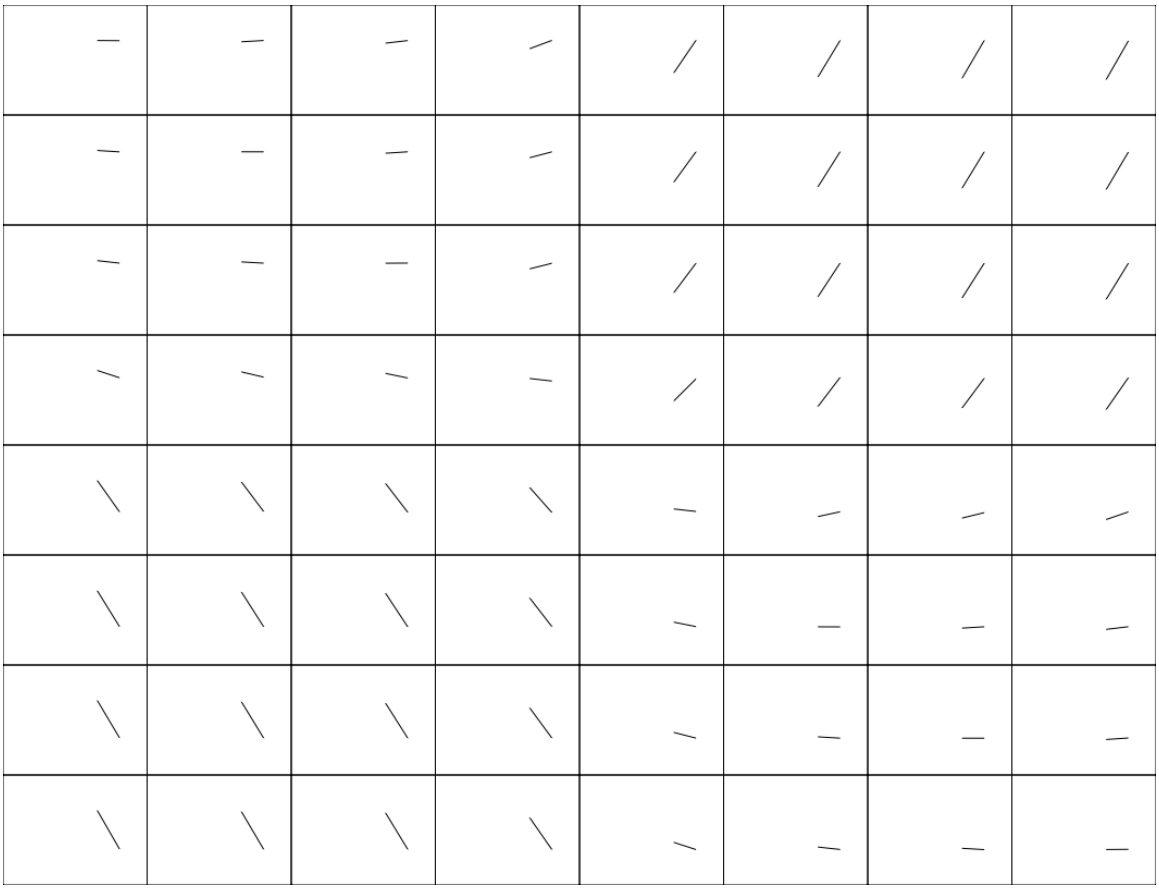
Recalled PE Mapping



Our recalled density mapping shows great results as well. As we can clearly see in this grey-scale heatmap representation of the amount of data points that map to each of our 64 PEs, the clear majority of the data points map to one of the PEs in the corners represented by the prototypes that map to our data epicenters. We have very clear definition in the intermediary PEs showing that very few data points map to them; this is our expected result, since the learning process would have mapped most of our points to the prototypes in the four corners.

Problem 3

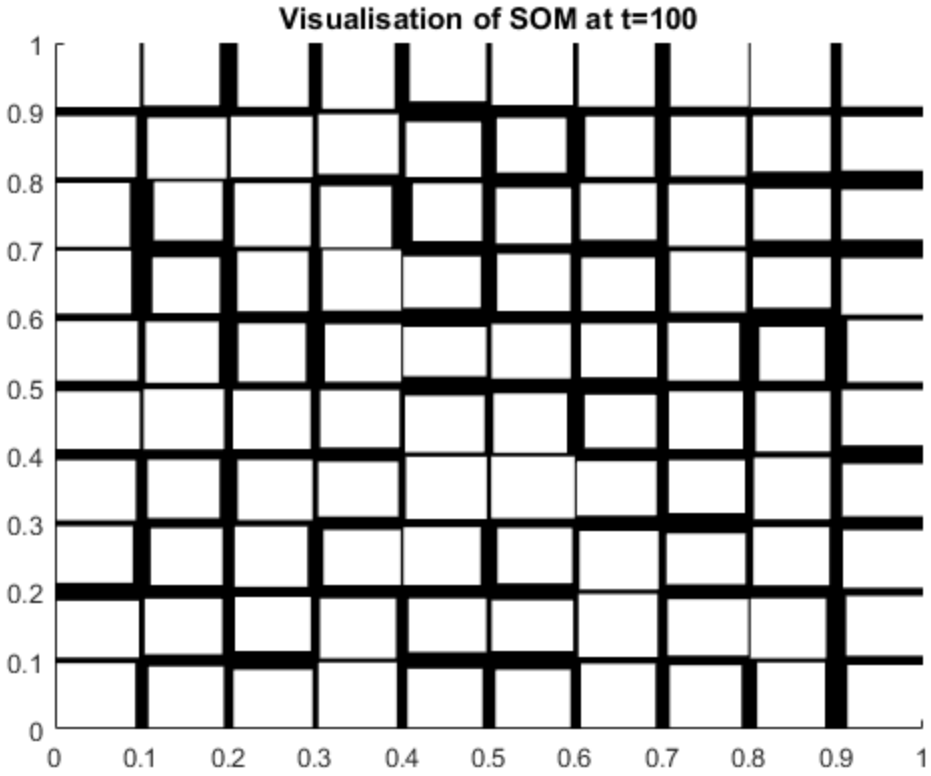
2D Weight Vectors Plot

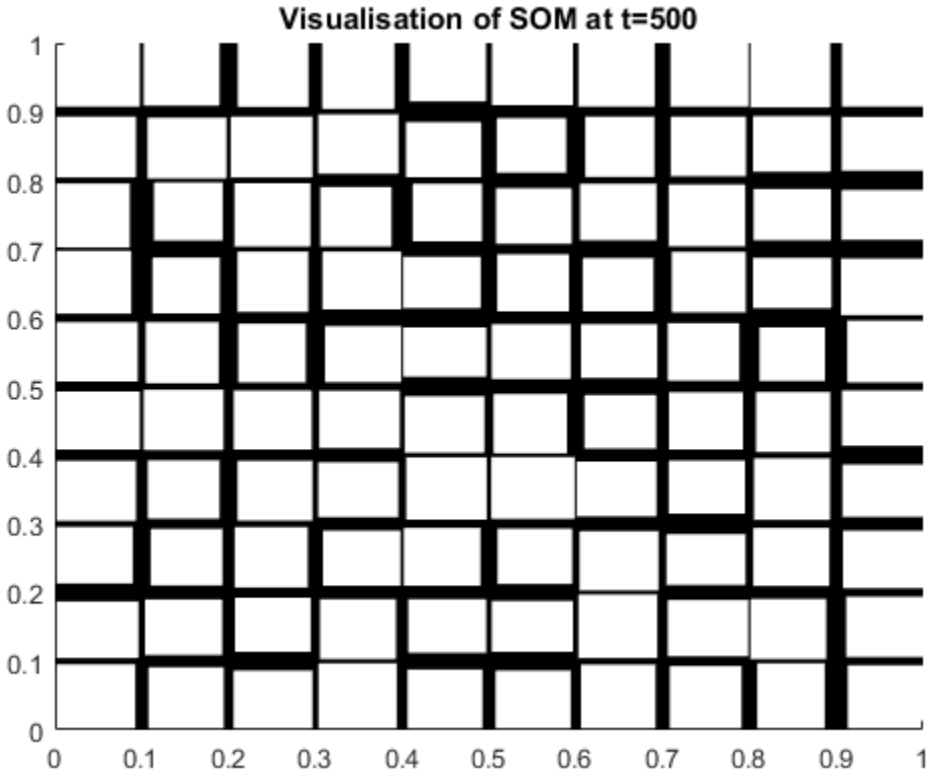


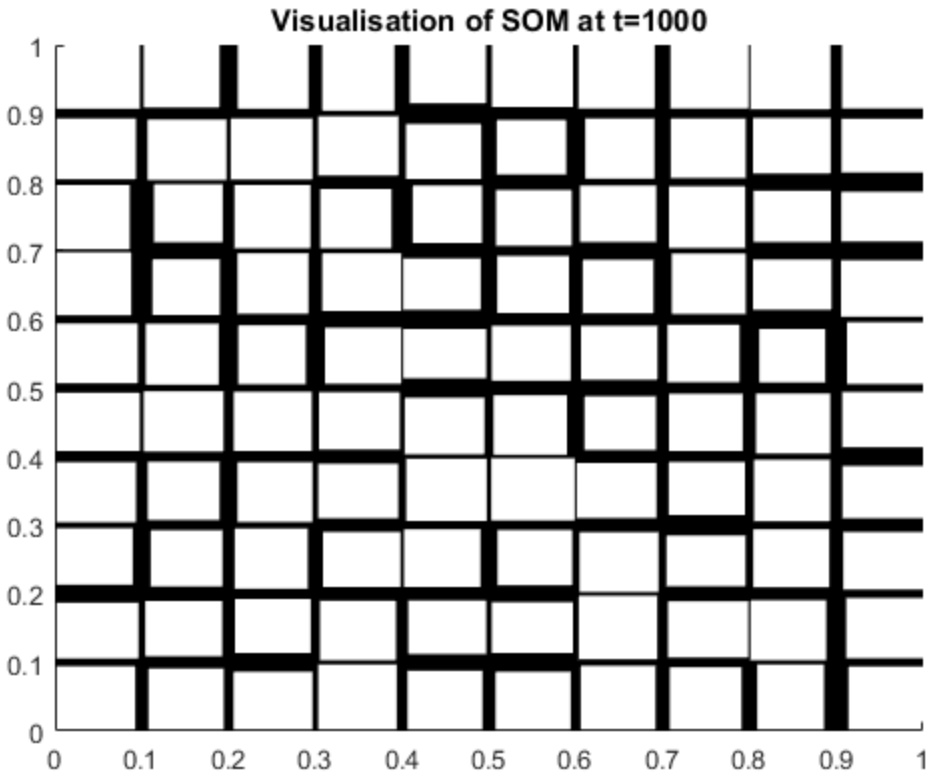
Our 2-dimension weight vector plots does an excellent job of showing how well our clustering relations are visualized. Each cell in this 8x8 visualization represents the corresponding PE in our lattice, where the x-axis is the sequential number of dimensions of that PE in in the input space (in this case, two dimension) and the y-axis represents the value at those dimensions. Even without the dead zones marked, we can see very clear and profound delineations between our four clusters.

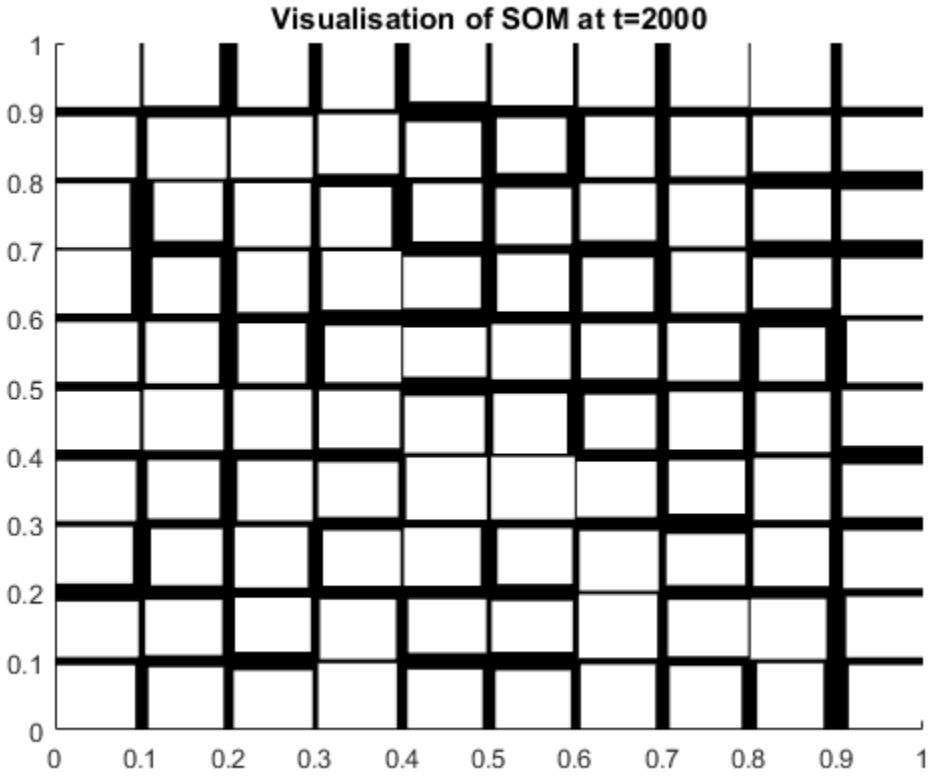
Problem 4

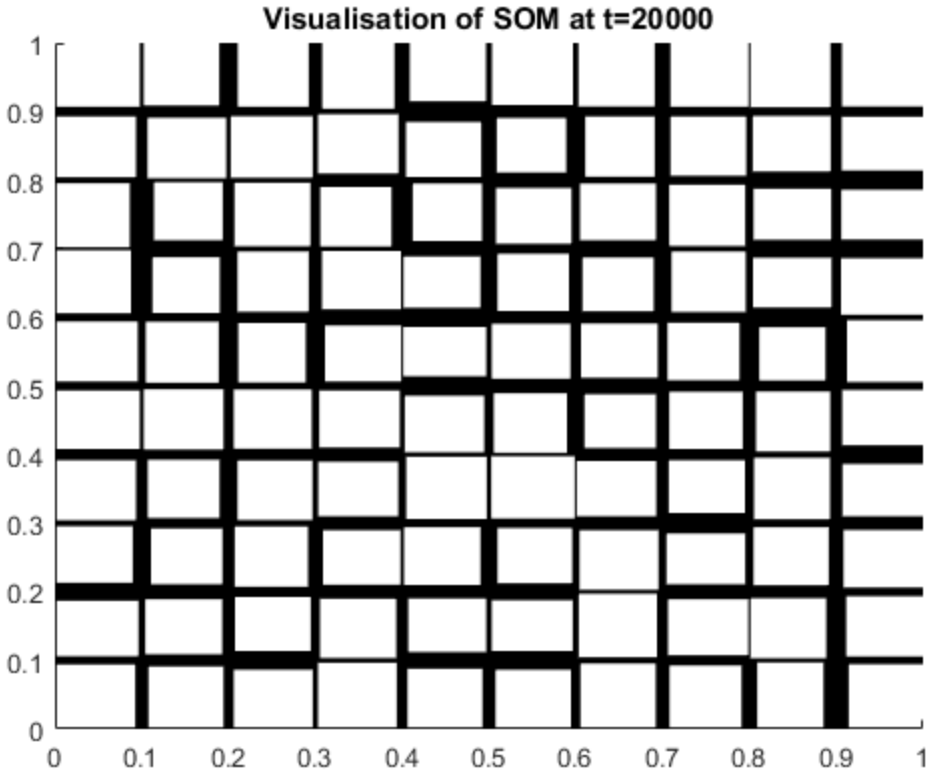
See the provided plots below. The first 3 plots display the mU-matrix associated with the network training for identifying the distinct plant species. The last 3 plots show for each plant species, which PEs were activated how often. It is clear that the PEs that activate are distinct and hence the SOM is "classifying well" but it hasn't converged because the regions activated by each species aren't contiguous. Hence, to conclude on why the couple of samples earlier were misinterpreted is difficult. Nevertheless, we can notice that a couple of species 3 samples activate the same PEs as several species 2 (bottom left corner), this shows that these species have data points almost indistinguishable from species 2 which would explain why some classification errors are there.

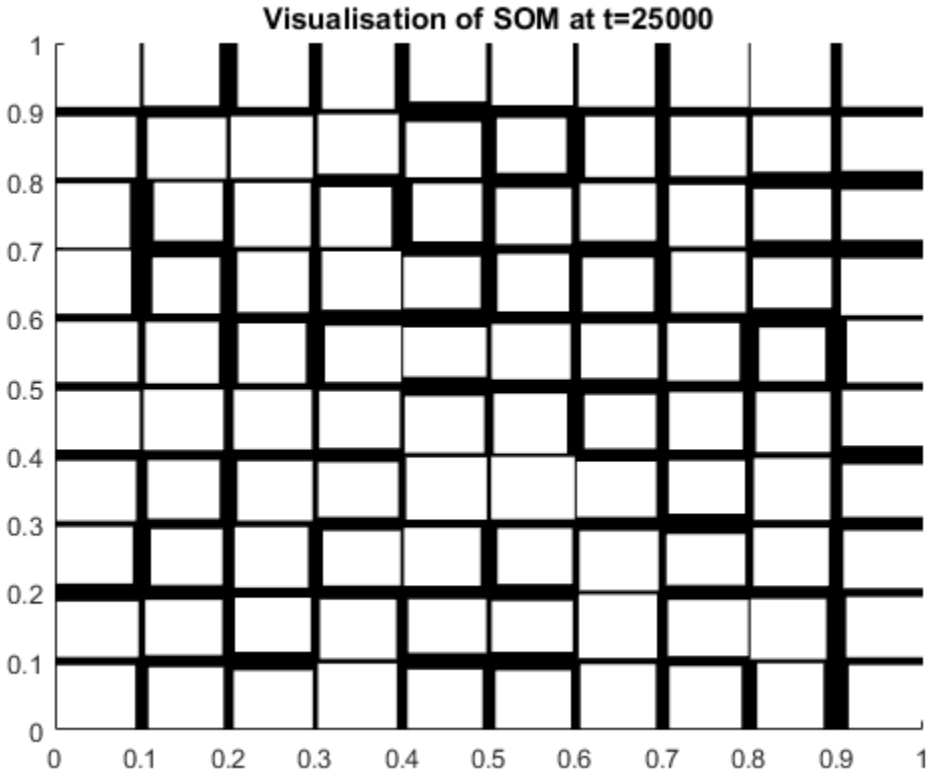

Visualisation of SOM at t=100
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1

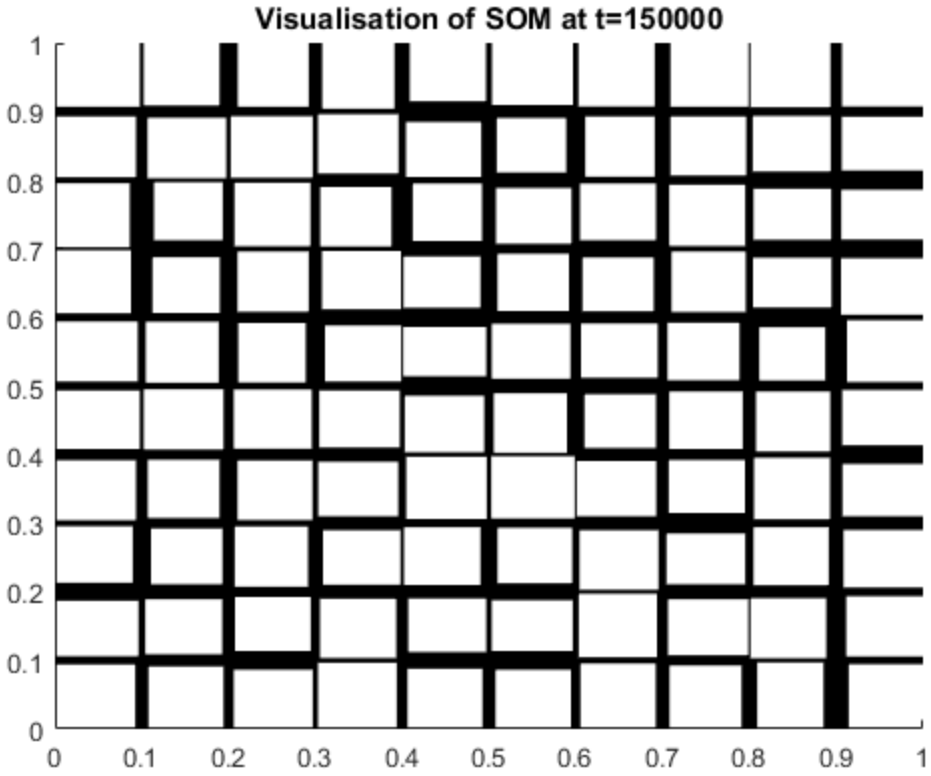

Visualisation of SOM at t=500
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1

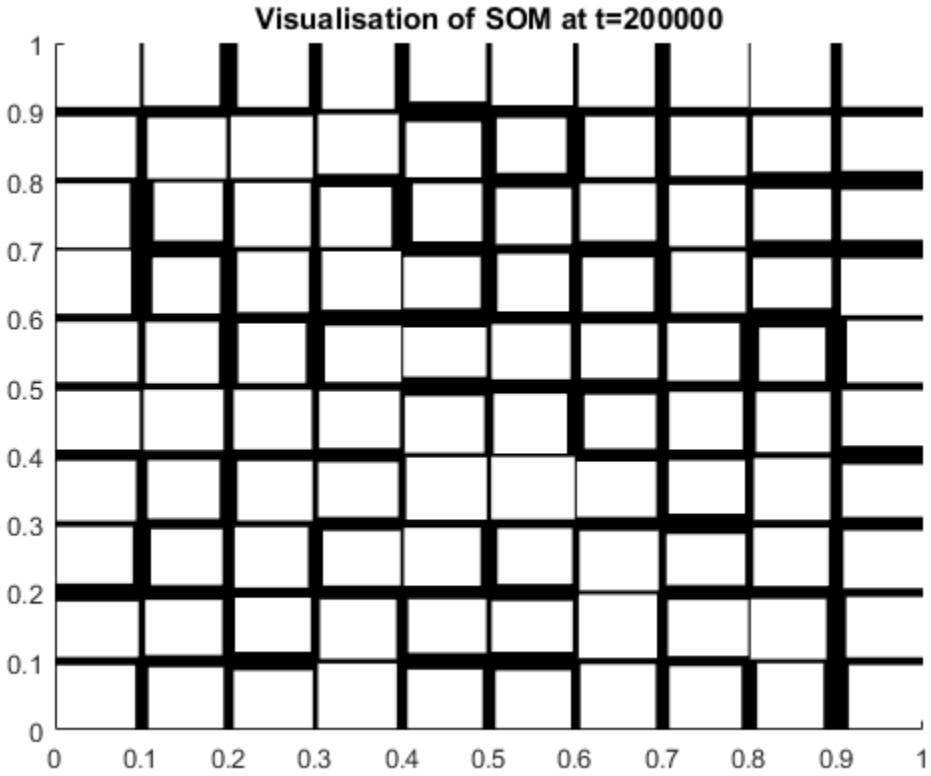

Visualisation of SOM at t=1000
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1

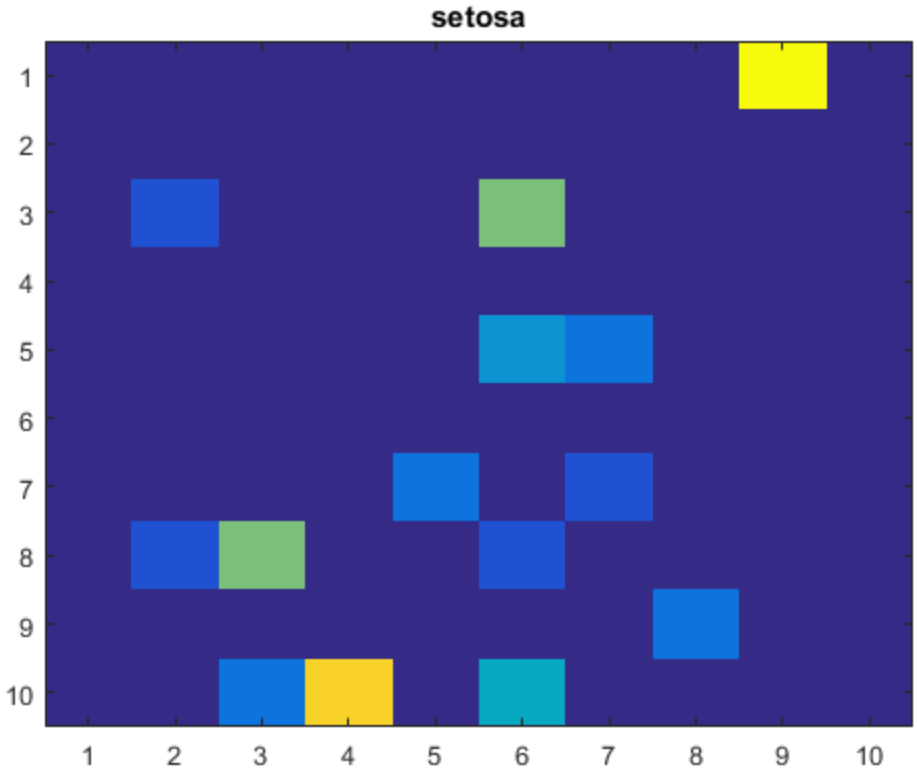

Visualisation of SOM at t=2000
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1

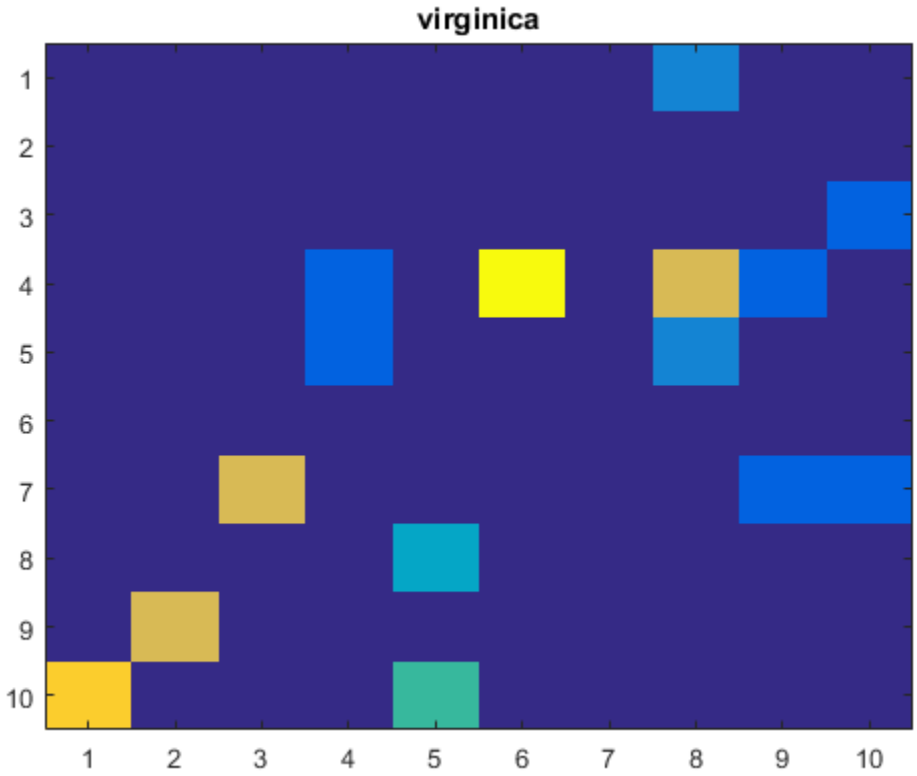

Visualisation of SOM at t=20000
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1

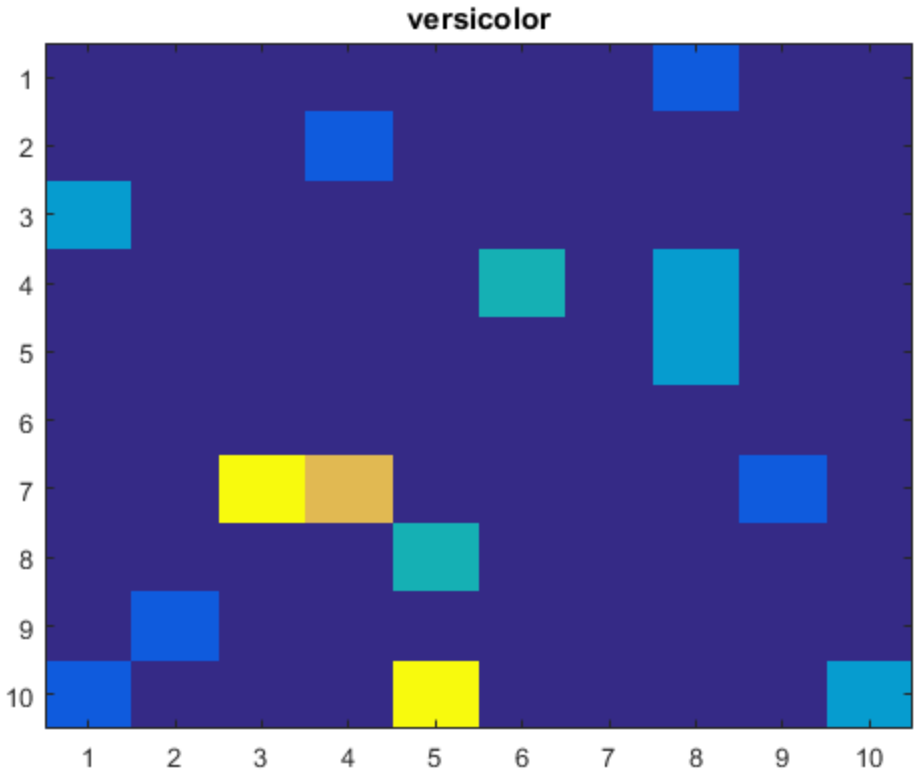

Visualisation of SOM at t=25000
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1


Visualisation of SOM at t=150000
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1


Visualisation of SOM at t=200000
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1


setosa
1
2
3
4
5
6
7
8
9
10
1
2
3
4
5
6
7
8
9
10


virginica
1
2
3
4
5
6
7
8
9
10
1
2
3
4
5
6
7
8
9
10



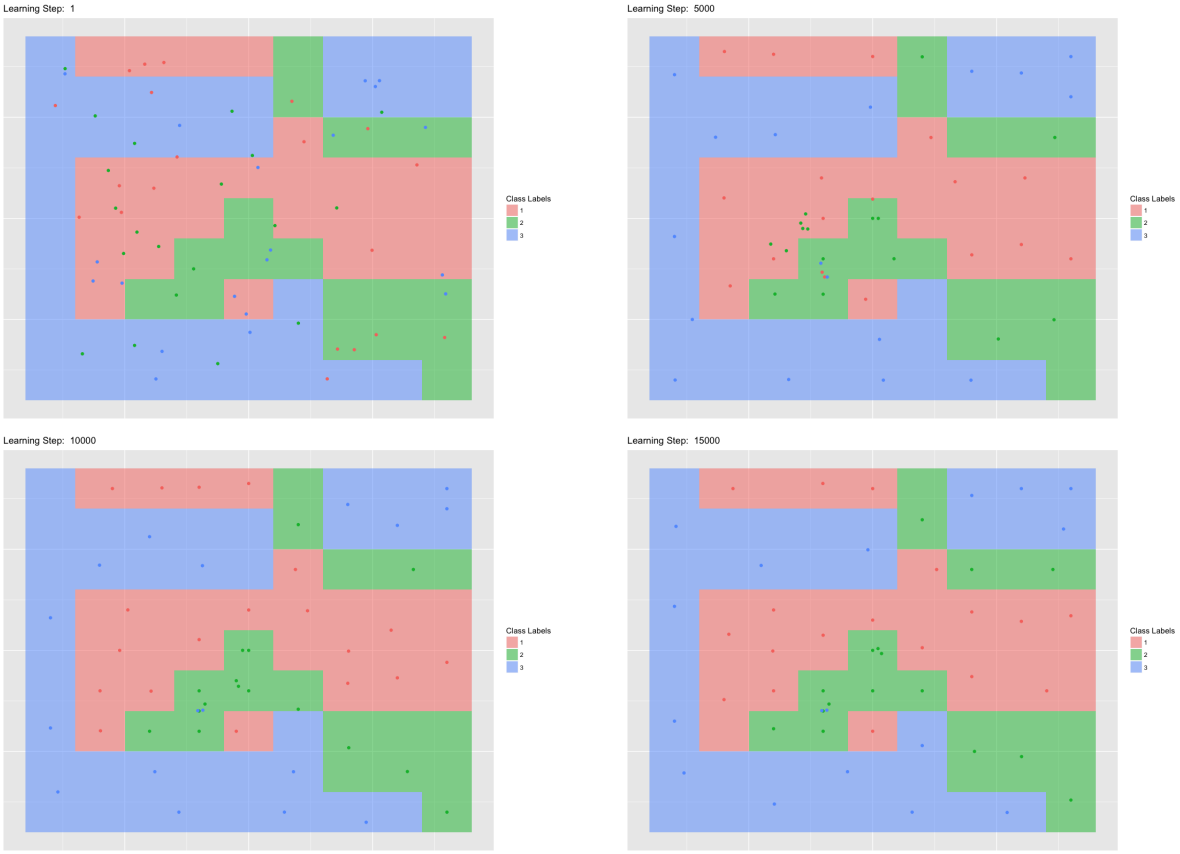
Published with MATLAB® R2016a

Problem 5

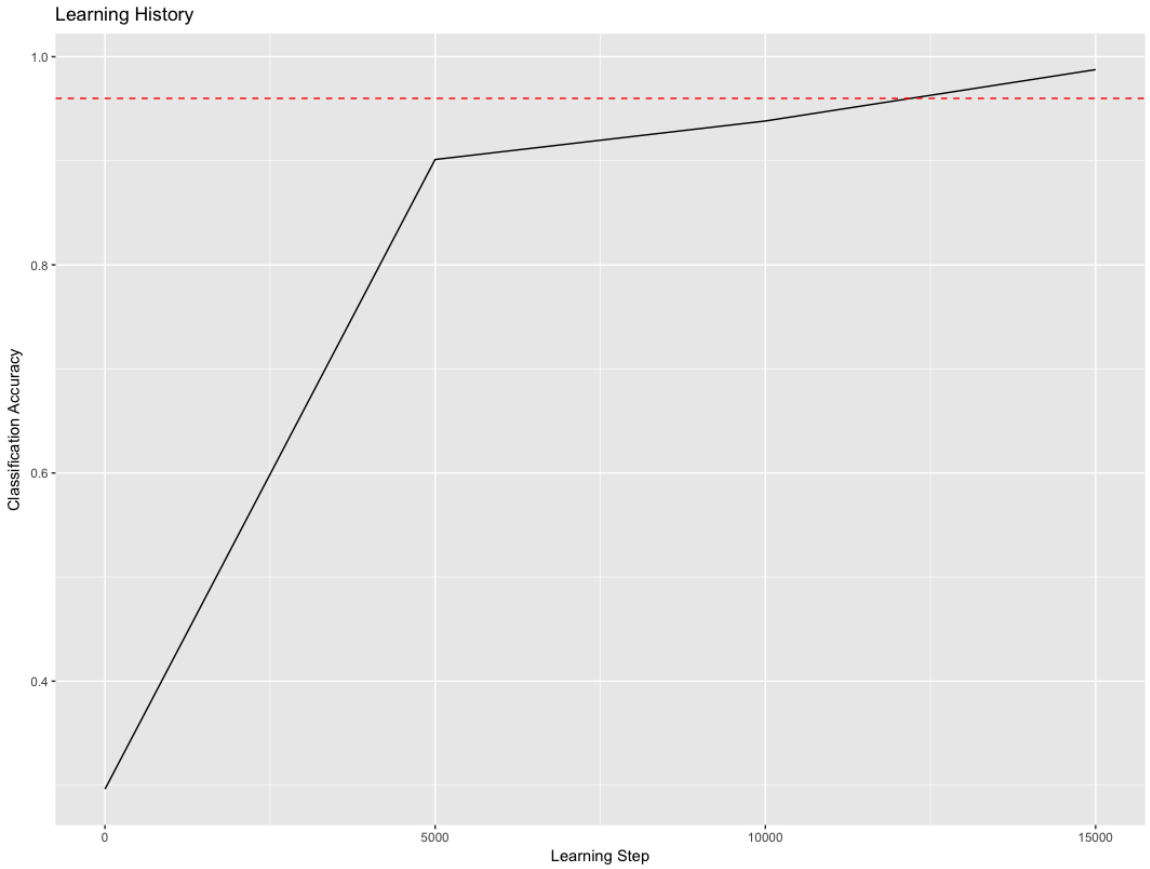
Parameters

- Network Parameters
  - Topology: 60 PEs (prototypes)
  - Each PE is a 2-dimensional vector (see below for weight draws)
- Learning Parameters
  - Initial Weights: Drawn from U[0.5, 9]
  - Initial Learning Rate (α_{init}): 0.5
  - Learning Decay Rate:
    - Learning Steps 1-15000: 0.5 Learning Rate
    - Learning Steps 15000-30000: 0.25 Learning Rate
    - Learning Steps 30000-45000: 0.125 Learning Rate
    - Learning Steps 45000-100000: 0.05 Learning Rate
  - Momentum: None
  - Stopping Criteria #1: 100,000 learning steps
  - Stopping Criteria #2: 96% or greater training classification accuracy
- Input Data
  - 3-class classifaction problem as defined in HW07
- Error and Performance Measure
  - Learning Steps Performed: 15,000
  - Monitoring Frequency: Every 5,000 Learning Steps
  - Accuracy Measure: Total correct classification / Total points
  - Accuracy at end of Training: 98.8%

Training Results

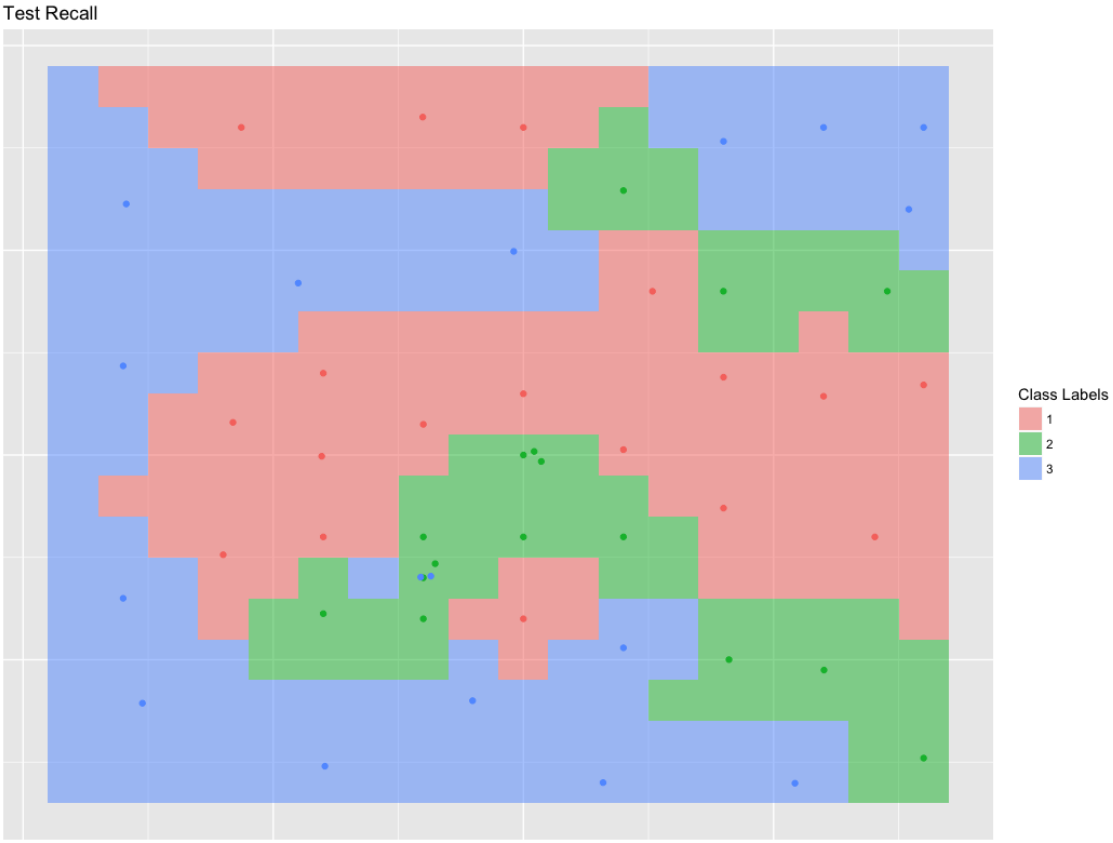


As we can see by our training results, our network converged quite quickly during this (our best) iteration; of the maximum allowed 100,000 learning steps, training was stopped at 15,000 with the great result of 98.8% classification accuracy. The training at the 15,000th step looks great except for the collection of Class 3 points that have congregated in the central Class 2 region.



Our learning history plot shows how successful this run was for us. Our initial accuracy was 30% at Step 1, followed by 90% at Step 5,000, then 94% at Step 10,000 and finally almost 99% at Step 15,000. We were able to reach our pre-determined tolerance level of 96% very quickly on this run.

Test Recall Results



On the fine grid using the network we had already trained, we again achieved fairly strong results with a classification accuracy of 83%; this was the best result we had obtained and we felt very strongly about this classification measure. There is very good representation for all of the original classification regions following this test recall.

Training and Test Comparison

Looking at a side-by-side view of our test data and our training data, I think our results are very encouraging. Aside from our 83% accuracy measure (which we feel is very strong) a cursory look at the training and test data plot will show that there is very good representation for the original classification regions. An interesting anomaly (again) is the blue region that has manifested in the certain of our plot; but aside from that, we are extremely happy with these results!

Code Appendix

```
## No scientific notation

options(scipen = 999)

## Load in the necessary libraries

library(plotly)

## Read in the necessary functions

source("~/Documents/Rice_University/Spring_2018/NML502/HW07/build_SOM.R")
source("~/Documents/Rice_University/Spring_2018/NML502/HW07/learn_map.R")
source("~/Documents/Rice_University/Spring_2018/NML502/HW07/plot_SOM.R")
source("~/Documents/Rice_University/Spring_2018/NML502/HW07/recall_PE.R")



##### Problem 2 #####



## Set the SOM lattice dimensions

matrix_dim <- 8

## Set the initial radius of influence on neighbors

radius <- matrix_dim / 2

## Set the number of iterations

num_iter <- 500000

## Set the initial learning rate

ler_rate <- 0.001

## Load in the input data

x_1 <- rnorm(n = 1000, mean = 7, sd = sqrt(0.1))
x_2 <- rnorm(n = 1000, mean = 7, sd = sqrt(0.1))

x_3 <- rnorm(n = 1000, mean = 0, sd = sqrt(0.1))
x_4 <- rnorm(n = 1000, mean = 7, sd = sqrt(0.1))
```

```
x_5 <- rnorm(n = 1000, mean = 7, sd = sqrt(0.1))
x_6 <- rnorm(n = 1000, mean = 0, sd = sqrt(0.1))

x_7 <- rnorm(n = 1000, mean = 0, sd = sqrt(0.1))
x_8 <- rnorm(n = 1000, mean = 0, sd = sqrt(0.1))

X_1 <- rbind(x_1, x_2)
X_2 <- rbind(x_3, x_4)
X_3 <- rbind(x_5, x_6)
X_4 <- rbind(x_7, x_8)

X <- cbind(X_1, X_2, X_3, X_4)

rownames(X) <- NULL

## Get the size of a typical input

input_size <- length(X[, 1])

## Build the SOM matrix

SOM_lattice <- build_SOM(input_size, matrix_dim)

## Perform the learning process

learn_results <- learn_map(SOM_lattice, X, num_iter, radius, ler_rate)

## Plot the results

par(mfrow = c(3, 2))
par(mar = c(3, 3, 3, 3))
plot_SOM(learn_results)

## Plot the decay of the radius and learning rate

plot(x = param_container[[1]], y = param_container[[2]], type = "l",
     xlab = "Learning Step",
     ylab = "Neighborhood Radius",
     main = "Neighborhood Radius Decay")

plot(x = param_container[[1]], y = param_container[[3]], type = "l",
     xlab = "Learning Step",
     ylab = "Learning Rate",
     main = "Learning Rate Decay")

## Perform the recall

recall_PE(learn_results, X)



##### Problem 3 #####
```

```
## Unwrap the final PE prototypes

x_lattice <- learn_results[[length(learn_results)]][[2]][[1]]
y_lattice <- learn_results[[length(learn_results)]][[2]][[2]]

## Scale the PEs down and vectorize for plotting purposes

scale_x <- scale(c(t(x_lattice)))
scale_y <- scale(c(t(y_lattice)))

## Generate the frame and add the plots

par(mfrow = c(matrix_dim, matrix_dim))
par(mar = c(0, 0, 0, 0))

for (i in 1:length(x_lattice)) {

    plot(x = 1:input_size,
         y = c(scale_x[i], scale_y[i]),
         type = "l",
         xaxt = "n",
         yaxt = "n",
         ann = FALSE,
         xlim = c(-3, 3),
         ylim = c(-3, 3))

}

## Build the function that learns the mapping

learn_map <- function(SOM_lattice, X, num_iter, radius, ler_rate) {

    ## Container for the current prototype lattice

    SOM_container <- list()

    ## Container for learning step, radius and learning rate

    param_container <- list()
    ler_steps <- numeric()
    radii <- numeric()
    ler_rates <- numeric()

    ## Save the initial parameters

    init_radius <- radius
    decay_constant <- num_iter / log(init_radius)

    init_ler_rate <- ler_rate

    ## Perform the iterated learning
```

```
for (i in 1:num_iter) {

    ## Decay the learning rate

    ler_rate <- init_ler_rate * exp(-i / num_iter)

    ## Decay the radius

    radius <- init_radius * exp(-i / decay_constant)

    ## Add the parameters states to the container

    if (i %% 5000 == 0) {

        ler_steps[length(ler_steps) + 1] <- i
        radii[length(radii) + 1] <- radius
        ler_rates[length(ler_rates) + 1] <- ler_rate

    }

    ## Randomly select an input vector

    rand_ind <- sample(x = 1:dim(X)[2], size = 1)
    x <- as.matrix(X[, rand_ind], ncol = 1)

    ## Calculate the matrix of differences

    matrix_diffs <- list()
    matrix_diffs[[1]] <- x[1, ] - SOM_lattice[[2]][[1]]
    matrix_diffs[[2]] <- x[2, ] - SOM_lattice[[2]][[2]]

    ## Get the Euclidean distance

    mat_a <- matrix_diffs[[1]]^2
    mat_b <- matrix_diffs[[2]]^2
    matrix_dist <- sqrt(mat_a + mat_b)

    ## Find the winning PE

    min_neuron <- min(matrix_dist)
    min_loc <- as.vector(which(matrix_dist == min(matrix_dist), arr.ind = TRUE))

    ## Calculate the Manhattan distance

    mat_c <- abs(SOM_lattice[[1]][[1]] - min_loc[1])
    mat_d <- abs(SOM_lattice[[1]][[2]] - min_loc[2])
    man_dist <- mat_c + mat_d
    neighbor_func <- exp(-((man_dist)/(radius))^2)

    ## Update the weights

    SOM_lattice[[2]][[1]] <- SOM_lattice[[2]][[1]] + ler_rate *
        (neighbor_func * matrix_diffs[[1]])
```

```
        SOM_lattice[[2]][[2]] <- SOM_lattice[[2]][[2]] + ler_rate *
            (neighbor_func * matrix_diffs[[2]])

        ## Add to the container

        if ((i == 1) || (i %in% seq(from = 0, to = num_iter, length.out = 6))) {

            SOM_container[[length(SOM_container) + 1]] <- list(i, SOM_lattice[[2]])

        }

    }

    param_container[[1]] <- ler_steps
    param_container[[2]] <- radii
    param_container[[3]] <- ler_rates

    param_container <<- param_container

    return(SOM_container)

}

## Build the function to recall each input

recall_PE <- function(learn_results, X) {

    ## Unwrap the final SOM lattice

    x_lattice <- learn_results[[length(learn_results)]][[2]][[1]]
    y_lattice <- learn_results[[length(learn_results)]][[2]][[2]]

    ## Container for which PE the input maps to

    neuron_map <- matrix(0, ncol = dim(x_lattice)[1], nrow = dim(x_lattice)[1])
    colnames(neuron_map) <- 1:dim(x_lattice)[1]
    rownames(neuron_map) <- 1:dim(x_lattice)[1]

    ## Loop through the input data for recall

    for (i in 1:dim(X)[2]) {

        ## Get the current input vector

        x <- matrix(X[, i], ncol = 1)

        ## Calculate the matrix of differences

        matrix_diffs <- list()
        matrix_diffs[[1]] <- x[1, ] - x_lattice
        matrix_diffs[[2]] <- x[2, ] - y_lattice

        ## Get the Euclidean distance
```

```
        mat_a <- matrix_diffs[[1]]^2
        mat_b <- matrix_diffs[[2]]^2
        matrix_dist <- sqrt(mat_a + mat_b)

        ## Find the winning PE

        min_neuron <- min(matrix_dist)
        min_loc <- as.vector(which(matrix_dist == min(matrix_dist), arr.ind = TRUE))

        ## Add the location to the neuron map

        if (neuron_map[min_loc[1], min_loc[2]] == 0) {

            neuron_map[min_loc[1], min_loc[2]] <- 1

        } else {

            neuron_map[min_loc[1], min_loc[2]] <- neuron_map[min_loc[1], min_loc[2]] + 1

        }

    }

    s <- as.character(1:matrix_dim)

    temp_vec <- as.vector(neuron_map)
    temp_mat <- matrix(temp_vec, nrow = matrix_dim, ncol = matrix_dim)
    temp_mat <- apply(temp_mat, 2, rev)


    plot_ly(z = temp_mat, x = ~s, y = ~s, colors = colorRamp(c("white", "black")),
            type = "heatmap") %>%
        layout(title = "PE Density Map", xaxis = list(title = "PE X Coordinate"),
               yaxis = list(title = "PE Y Coordinate"))

}

## Function to build the SOM matrix

build_SOM <- function(input_size, matrix_dim) {

    ## Container for the matrices of weights

    SOM_weights <- list()

    ## Container for the indices

    SOM_indices <- list()

    ## Build the list of weight indices

    SOM_indices[[1]] <- matrix(rep(1:matrix_dim, times = matrix_dim),
                               nrow = matrix_dim, ncol = matrix_dim)
```

```
    SOM_indices[[2]] <- matrix(rep(1:matrix_dim, times = matrix_dim),
                               byrow = T, nrow = matrix_dim, ncol = matrix_dim)

    ## Build the list of weights

    for (i in 1:input_size) {

        SOM_weights[[i]] <- matrix(runif(n = matrix_dim^2, min = 0, max = 1),
                                   ncol = matrix_dim, nrow = matrix_dim)

    }

    ## Return the results

    return(list(SOM_indices, SOM_weights))

}

## Build the plot_SOM function

plot_SOM <- function(learn_results) {

    ## For loop to iterate over the results

    for (i in 1:length(learn_results)) {

        ## Unwrap the learning step and lattice at each step

        ler_step <- learn_results[[i]][[1]]
        x_lattice <- as.vector(learn_results[[i]][[2]][[1]])
        y_lattice <- as.vector(learn_results[[i]][[2]][[2]])

        ## Build the lattice

        plot_lattice <- cbind(x_lattice, y_lattice)

        ## Plot the lattice

        plot(plot_lattice, pch = 16, main = c("SOM Grid at Learning Step", ler_step)
             xlab = "", ylab = "")

        ## Add the horizontal neighbor lines

        for (i in 1:dim(plot_lattice)[1]) {

            if (i %% matrix_dim != 0) {

                segments(plot_lattice[i, 1], plot_lattice[i, 2], plot_lattice[i + 1, 1],
                         plot_lattice[i + 1, 2])

            }

        }
```

```
        ## Add the vertical neighbor lines

        for (i in 1:dim(plot_lattice)[1]) {

            if (((dim(plot_lattice)[1] - i) - matrix_dim) >= 0) {

                segments(plot_lattice[i, 1], plot_lattice[i, 2], plot_lattice[i + matrix_dim, 1],
                         plot_lattice[i + matrix_dim, 2])

            }

        }

    }

}




########## Problem 5 ##########




## Load in the necessary packages

library(ggplot2)

## Build the recall function

recall_func <- function(X, Cx, W, Cw) {

    Cy <- apply(X, 2, function(x) {

        d <- apply(x - W, 2, norm, type = "2")
        min_d <- min(d)
        min_ind <- which(d == min_d)

        if (length(min_ind) != 1) {

            min_ind <- min_ind[1]

        }

        Cw[min_ind]

    })

    accuracy <- sum((Cy - Cx) == 0) / length(Cx)
```

```
    return(list(accuracy, Cy))

}

## Build the function for testing recall

recall_test_func <- function(X, W, Cw) {

    Cy <- apply(X, 2, function(x) {

        d <- apply(x - W, 2, norm, type = "2")
        min_d <- min(d)
        min_ind <- which(d == min_d)

        if (length(min_ind) != 1) {

            min_ind <- min_ind[1]

        }

        Cw[min_ind]

    })

    accuracy <- NA

    return(list(accuracy, Cy))


}

## Build the plotting function

plot_func <- function(A, nrows, ncols, W, Cw) {

    data <- data.frame(

        factor = factor(A),
        row = rep(1:nrows, each = ncols),
        col = rep(1:ncols, times = nrows)

    )

    plot <- ggplot() +
        geom_rect(aes(xmin = col - 0.5, xmax = col + 0.5, ymin = -row - 0.5, ymax = 0.5 - row,
                      fill = factor),
                  data = data, alpha = 0.5) +
        xlim(0.5, ncols + 0.5) +
        ylim(-0.5 - nrows, -0.5) +
        theme(axis.text = element_blank(), axis.ticks = element_blank()) +
        labs(fill = "Class Labels") +
        xlab("") +
        ylab("")
```

```
    proto_data <- data.frame(

        factor = Cw,
        row = t(W)[, 1],
        col = t(W)[, 2]

    )

    proto_data$factor <- as.factor(proto_data$factor)

    plot = plot + geom_point(aes(x = col, y = -row, color = factor), data = proto_data) +
        guides(color = FALSE)

    return(plot)

}



##### Part a #####



A = c(3, 1, 1, 1, 1, 2, 3, 3, 3,
      3, 3, 3, 3, 3, 2, 3, 3, 3,
      3, 3, 3, 3, 3, 1, 2, 2, 2,
      3, 1, 1, 1, 1, 1, 1, 1, 1,
      3, 1, 1, 1, 2, 1, 1, 1, 1,
      3, 1, 1, 2, 2, 2, 1, 1, 1,
      3, 1, 2, 2, 1, 3, 2, 2, 2,
      3, 3, 3, 3, 3, 3, 2, 2, 2,
      3, 3, 3, 3, 3, 3, 3, 3, 2)

nrows <- 9
ncols <- 9
ndim <- 2

# A <- matrix(A, nrow = nrows, ncol = ncols, byrow = FALSE)

N <- nrows * ncols
nP <- 60
nC <- 3
maxsteps <- 100000
tol <- 0.96
mu <- 0.5
LRsched <- matrix(
    c(0, 0.5, 15000, 0.25, 30000, 0.125, 45000, 0.05),
    ncol = 2, byrow = TRUE
)
Mfrsched <- c(1, seq(from = 5000, to = maxsteps, by = 5000))
prots_per_class <- round(nP / nC)
Cw <- rep(1:nC, each = prots_per_class)
```

```
X <- rbind(rep(1:nrows, each = ncols), rep(1:ncols, nrows))
Cx <- A

W <- matrix(runif(ndim * nP, min(X), max(X)), nrow = ndim, ncol = nP)

ler_step <- numeric()
step_acc <- numeric()
plot_container <- list()

for (i in 1:maxsteps) {

    mu <- tail(LRsched[i > LRsched[, 1], 2], 1)

    rand_ind <- sample(1:N, 1)

    d <- apply(W - X[, rand_ind], 2, norm, type = "2")
    min_d <- min(d)
    min_ind <- which(d == min_d)

    if (Cw[min_ind] == Cx[rand_ind]) {

        W[, min_ind] <- W[, min_ind] + mu * (X[, rand_ind] - W[, min_ind])

    } else {

        W[, min_ind] <- W[, min_ind] - mu * (X[, rand_ind] - W[, min_ind])

    }

    if (i %in% Mfrsched) {

        ler_step[length(ler_step) + 1] <- i

        recall <- recall_func(X, Cx, W, Cw)
        accuracy <- recall[[1]]
        step_acc[length(step_acc) + 1] <- accuracy
        plot <- plot_func(A = A, nrows = nrows, ncols = ncols, W = W, Cw = Cw)
        plot <- plot + labs(title = paste("Learning Step: ", i))

        plot_container[[length(plot_container) + 1]] <- plot

        if (accuracy > tol) {

            break

        }

    }

}

len <- length(plot_container)
```

```
plot_container[[len]]

ggplot() +
    geom_line(aes(x = ler_step, y = step_acc)) +
    geom_hline(aes(yintercept = tol), color = "red", linetype = "dashed") +
    labs(x = "Learning Step", y = "Classification Accuracy") +
    ggtitle("Learning History")



##### Part b #####



nrows <- 18
ncols <- 18
ndim <- 2

N <- nrows * ncols

X <- rbind(rep(1:nrows, each = ncols), rep(1:ncols, times = nrows)) / 2

recall <- recall_test_func(X, W, Cw)
Cy <- recall[[2]]

plot <- plot_func(Cy, nrows, ncols, W = W * 2, Cw = Cw)

plot + ggtitle("Test Recall")

data_1 <- data.frame(

    factor = factor(rep(factor(A), times = 4)),
    row = rep(c(1,3,5,7,9,11,13,15,17,2,4,6,8,10,12,14,16,18), times = 2, each = (nrows / 2)),
    col = c(rep(c(1,3,5,7,9,11,13,15,17), times = 18), rep(c(2,4,6,8,10,12,14,16,18), times = 18))

)

data_2 <- data.frame(

    factor = factor(Cy),
    row = rep(1:nrows, each = ncols),
    col = rep(1:ncols, times = nrows)

)

join_data <- inner_join(data_1, data_2, by = c('row', 'col'))
join_data$diff <- as.numeric(join_data$factor.x) - as.numeric(join_data$factor.y)

count_correct <- sum(join_data$diff == 0)
count_total <- dim(join_data)[1]
count_correct / count_total
```

```
ggplot() +
    geom_rect(aes(xmin = col - 0.5, xmax = col + 0.5, ymin = -row - 0.5, ymax = 0.5 - row,
                  fill = factor.x),
              data = join_data, alpha = 0.5) +
    theme(axis.text = element_blank(), axis.ticks = element_blank()) +
    xlab("") +
    ylab("") +
    labs(fill = "Class Labels") +
    ggtitle("Final Training Results")

ggplot() +
    geom_rect(aes(xmin = col - 0.5, xmax = col + 0.5, ymin = -row - 0.5, ymax = 0.5 - row,
                  fill = factor.y),
              data = join_data, alpha = 0.5) +
    theme(axis.text = element_blank(), axis.ticks = element_blank()) +
    xlab("") +
    ylab("") +
    labs(fill = "Class Labels") +
    ggtitle("Final Test Results")
```

Problem 4 Code (MATLAB Code)

```
function elec502hw7()
global number_of_steps
global size_of_dataset
global dim
global SOMD1
global SOMD2
global min_learn_rate
global spatial_decay_rate
global initial_learn_rate
global learn_decay_rate
global min_spatial_decay_rate
global spatial_decay_power
%global smallest_neighbourhood
number_of_steps=200000;
size_of_dataset=150;
dim=2;
SOMD1=10; % Number of Rows
SOMD2=10; % Number of Columns
epsilon=eps;
initial_learn_rate=1/sqrt(SOMD1*SOMD2);
min_learn_rate=0.0006;
learn_decay_rate=0.0003;
spatial_decay_rate=(6)/(SOMD1*SOMD2); % The functions I use:
%alpha=initial_learn_rate*e^(-t*learn_decay_rate) but,
%if this is lessthan the min learn rate, the min learn rate is used as
 alpha instead
spatial_decay_power=0.01;
min_spatial_decay_rate=0.1;
% h(horizontal distance,vertical distance, t)=
%e^(-(1+t*spatial_decay_rate)*(1-
max(h_dist,v_dist)*spatial_decay_power))
% IF YOU'LL FEEL THESE FUNCTIONS AREN'T OPTIMAL, FEEL FREE TO CHANGE
 THEM
steps_to_see=[100 500 1000 2000 5000 10000 20000 25000 50000 75000
 100000 125000 150000 200000]; % Steps at which plots are produced
mdif=15;
mmin=0.4; %mdif and mmin relate to the marker width when plotting the
 SOM visualisation.
% mmin is the min thickness, mdif is the scaling with distance between
% weight vectors of consecutive PEs.
%smallest_neighbourhood=sqrt(dim)*sqrt(SOMD1*SOMD2);
% Generate the data
%X=rand(dim,size_of_dataset);
load fisheriris.mat
X=meas';
X=X/(max(max(X)));
y=species;
% Initialise the weights
W=rand(SOMD1,SOMD2,dim);
dist1=0*W;
% Perform the learning
for learning_step=1:number_of_steps
```

```
    % Choose the sample
    sample=round(rand(1)*size_of_dataset+0.5-epsilon);
    % vect is the corresponding vector
    vect=X(:,sample);
    for i=1:SOMD1
        for j=1:SOMD2
            %disp(vect)
            %disp(dist1(i,j,:))
            for k=1:dim
                dist1(i,j,k)=W(i,j,k)-vect(k);
            %dist1(i,j,2)=W(i,j,2)-vect(2);
            end
        end
    end
    % Having constructed a matrix of positions with respect to the
 chosen
    % vector we find distances.
    dist=dist1.^2;
    dist=sum(dist,3);
    [Y,I]=min(dist,[],1);
    [~,I2]=min(Y);
    W=weight_update(W,I2,I(I2),learning_step,dist1);
    if ismember(learning_step,steps_to_see)
        figure
        hold on
        %for r=0:SOMD1
        %    plot([0 1],[r/SOMD1 r/SOMD1],'k')
        %end
        %for r=0:SOMD2
        %    plot([r/SOMD2 r/SOMD2],[0 1],'k')
        %end
        % The loops below plot the SOM image
        for r=1:SOMD1
            for r2=1:SOMD2-1
                vectval=W(r,r2,:)-W(r,r2+1,:);
                vectval=sqrt(sum(vectval.^2));
                plot([r2/SOMD2 r2/SOMD2],[1-(r-1)/SOMD1 1-r/
SOMD1],'k','LineWidth',(mdif*vectval+mmin))
            end
        end
        for r2=1:SOMD2
            for r=1:SOMD1-1
                vectval=W(r,r2,:)-W(r+1,r2,:);
                vectval=sqrt(sum(vectval.^2));
                plot([1-(r2-1)/SOMD2 1-r2/SOMD2],[1-r/SOMD1 1-r/
SOMD1],'k','LineWidth',(mdif*vectval+mmin))
            end
        end
        title(['Visualisation of SOM at t=',num2str(learning_step)])
        %K=W(:,:,1)
        %K2=W(:,:,2)

        %K=reshape(K,1,100);
        %K2=reshape(K2,1,100);
```

```
            %scatter(K,K2)
            %hold on
            %for r=1:SOMD1
            %    vect1=K(r,:);
            %    vect2=K2(r,:);
            %    plot(K(r,:),K2(r,:))
            %end
        end
        %if mod(learning_step,5000)==0
        %    disp(learning_step)
        %end
    end
    i1=1:50;
    i2=51:100;
    i3=101:150;
    m1=meas(i1,:);
    m2=meas(i2,:);
    m3=meas(i3,:);
    Y1=zeros(10);
    Y2=Y1;
    Y3=Y2;
    for sample=1:150
        vect=X(:,sample);
        for i=1:SOMD1
            for j=1:SOMD2
                %disp(vect)
                %disp(dist1(i,j,:))
                for k=1:dim
                    dist1(i,j,k)=W(i,j,k)-vect(k);
                %dist1(i,j,2)=W(i,j,2)-vect(2);
                end
            end
        end
        % Having constructed a matrix of positions with respect to the
     chosen
        % vector we find distances.
        dist=dist1.^2;
        dist=sum(dist,3);
        [Y,I]=min(dist,[],1);
        [~,I2]=min(Y);
        if sample<=50
            Y1(I2,I(I2))=Y1(I2,I(I2))+1;
        end
        if sample>50 && sample <=100
            Y2(I2,I(I2))=Y2(I2,I(I2))+1;
        end
        if sample>100
            Y3(I2,I(I2))=Y3(I2,I(I2))+1;
        end
    end
    figure
    imagesc(Y1)
    title('setosa')
    figure
```

```
imagesc(Y2)
title('virginica')
figure
imagesc(Y3)
title('versicolor')
K=W(:,:,1);
K2=W(:,:,2);
K=reshape(K,1,100);
K2=reshape(K2,1,100);
%disp(size(K))
%disp(size(K2))
%scatter(K,K2)
disp(W)
end


function W=weight_update(W,hpos,vpos,t,dist1)
W=W+activation_func(hpos,vpos,t).*(-dist1);
end


function val=activation_func(hpos,vpos,t)
global number_of_steps
global size_of_dataset
global dim
global SOMD1
global SOMD2
global min_learn_rate
global spatial_decay_rate
global initial_learn_rate
global learn_decay_rate
global min_spatial_decay_rate
global spatial_decay_power
alpha=max(initial_learn_rate*exp(-t*learn_decay_rate),min_learn_rate);
for i=1:SOMD1
    for j=1:SOMD2
        for k=1:dim
            val(i,j,k)=exp((1-spatial_decay_power*max(abs(j-
hpos),abs(i-vpos)))*-(1+t*spatial_decay_rate));
        end
    end
end
val(vpos,hpos,1:2)=[1 1];
val=alpha*val;
val=min(val,1);
end


(:,:,1) =

  Columns 1 through 7

    0.4621    0.4193    0.8394    0.5311    0.7781    0.6615    0.7735
    0.4693    0.5976    0.6541    0.6674    0.5271    0.6240    0.6829
```

```
    0.8232    0.4386    0.4524    0.5410    0.4166    0.7369    0.8674
    0.3927    0.8472    0.5139    0.6342    0.6335    0.7818    0.9507
    0.4952    0.3860    0.7300    0.3769    0.3611    0.3486    0.5823
    0.5082    0.4668    0.6225    0.7658    0.6877    0.7901    0.3940
    0.4535    0.3974    0.8038    0.7026    0.6835    0.7113    0.7209
    0.7676    0.5108    0.3849    0.7328    0.7922    0.5357    0.4002
    0.6139    0.4929    0.6816    0.6593    0.4022    0.4984    0.6202
    0.5889    0.4103    0.6441    0.6067    0.8464    0.8424    0.6805

  Columns 8 through 10

    0.4795    0.4797    0.7152
    0.5682    0.7016    0.5610
    0.5634    0.6708    0.6772
    0.5498    0.6094    0.6440
    0.7942    0.4264    0.8157
    0.6956    0.3893    0.6456
    0.4341    0.5942    0.3956
    0.5685    0.7278    0.4056
    0.4024    0.3893    0.8277
    0.7583    0.3945    0.9623


(:,:,2) =

  Columns 1 through 7

    0.5816    0.5228    0.4134    0.4443    0.6959    0.6829    0.7000
    0.3566    0.5843    0.5181    0.6199    0.5354    0.1889    0.6370
    0.6440    0.2114    0.4829    0.6801    0.1923    0.2627    0.3899
    0.5443    0.3155    0.5713    0.2892    0.2540    0.6212    0.3633
    0.4998    0.3675    0.6740    0.3851    0.3645    0.4730    0.4432
    0.6672    0.6056    0.4433    0.3723    0.4345    0.6125    0.4902
    0.2298    0.4931    0.6138    0.6646    0.4937    0.5792    0.5578
    0.2785    0.6213    0.3683    0.3404    0.3184    0.6206    0.5435
    0.3912    0.3199    0.5463    0.3414    0.2516    0.3153    0.3100
    0.5580    0.3117    0.3184    0.2178    0.2103    0.5087    0.3789

  Columns 8 through 10

    0.4821    0.4789    0.3672
    0.2904    0.3114    0.5164
    0.3884    0.2419    0.4684
    0.4911    0.6389    0.4317
    0.4194    0.7034    0.3667
    0.5320    0.5745    0.4778
    0.5698    0.6874    0.4783
    0.5838    0.4939    0.6025
    0.4282    0.4012    0.5872
    0.6879    0.6830    0.4727
```