HW07

Elliot Smith, Eugen Hruska, Varun Suriyanaranana

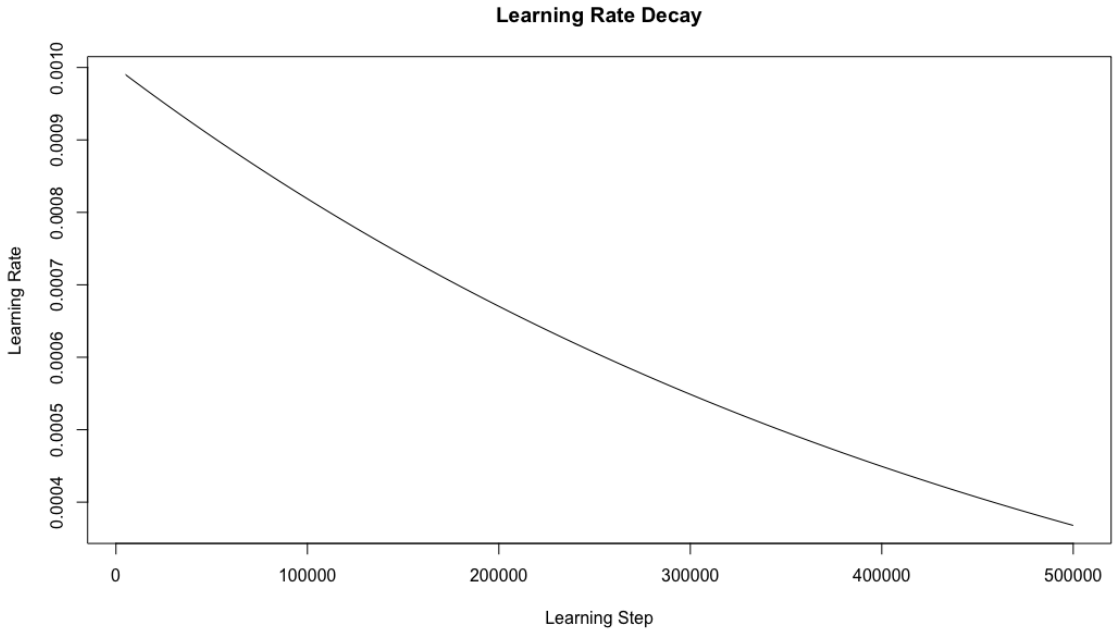
3/21/2018

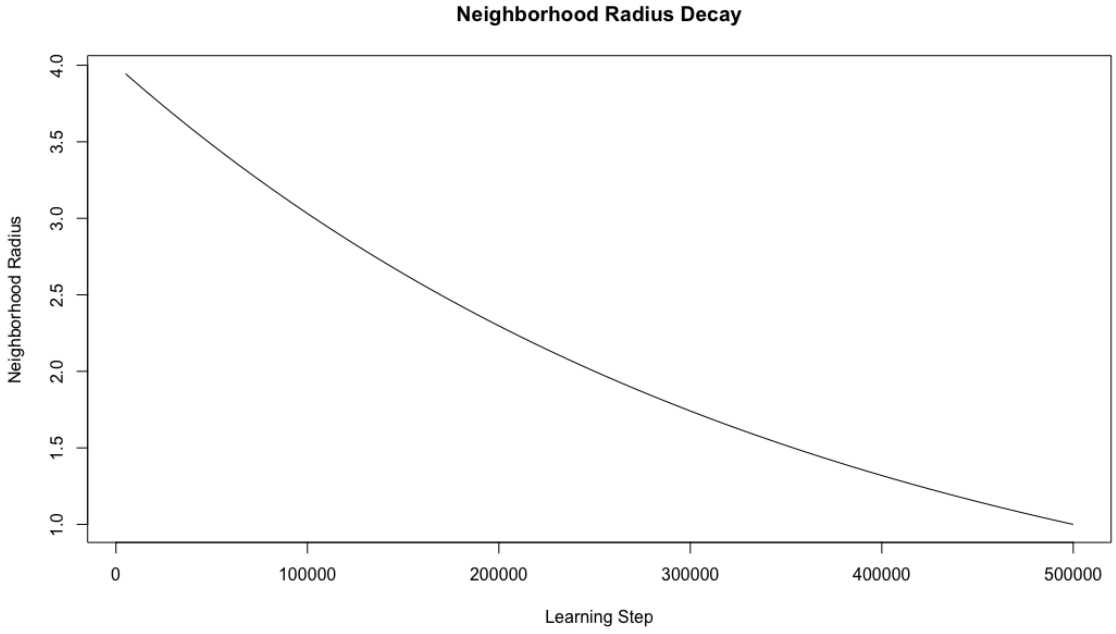
Problem 2a

Parameters

- Network Parameters
  - Topology: 64 PEs (8x8 2-dimensional square lattice)
- 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
    - $\mu = \frac{n}{log(\theta_{init})}$
  - 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]

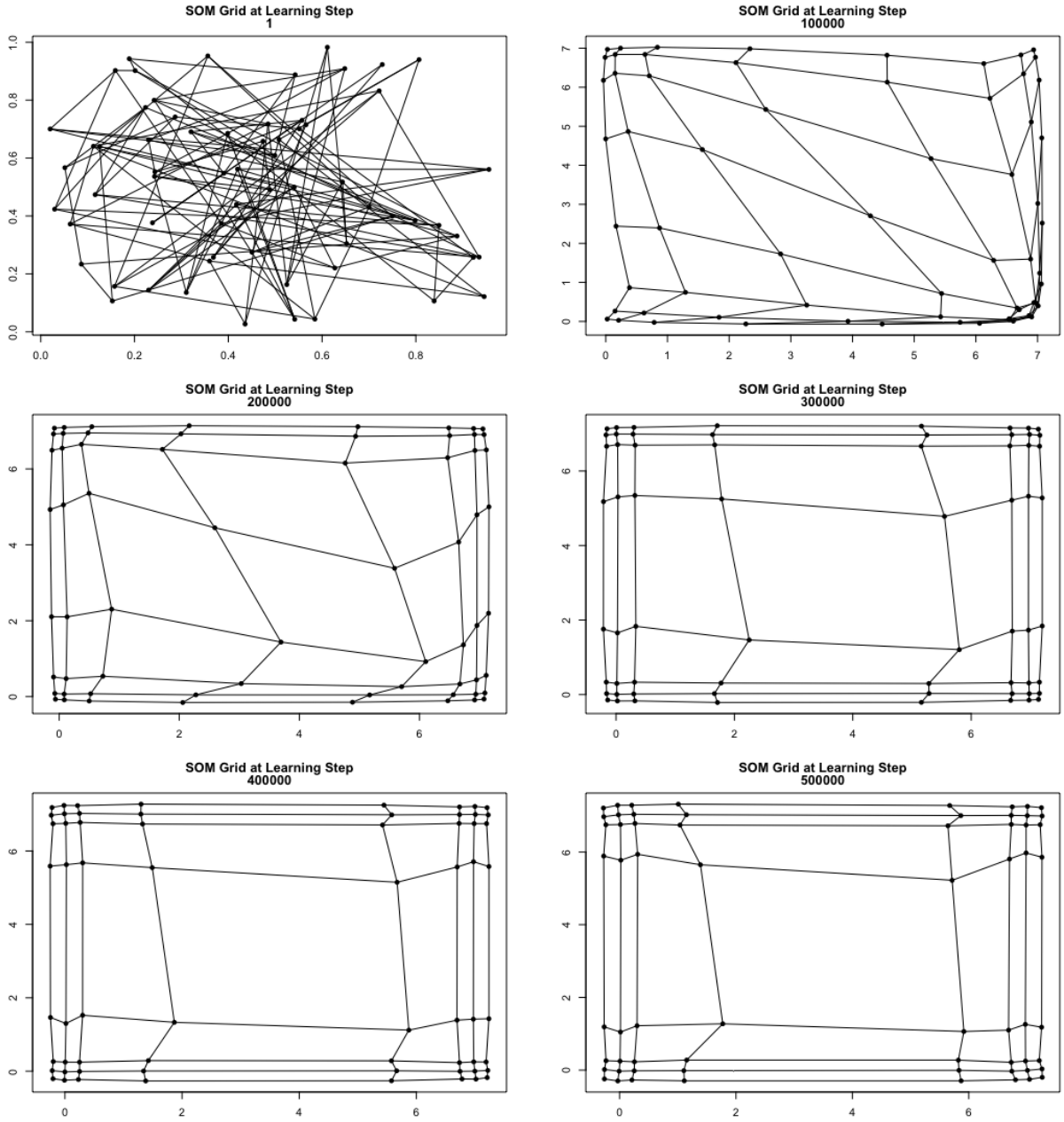
Decay Graphs





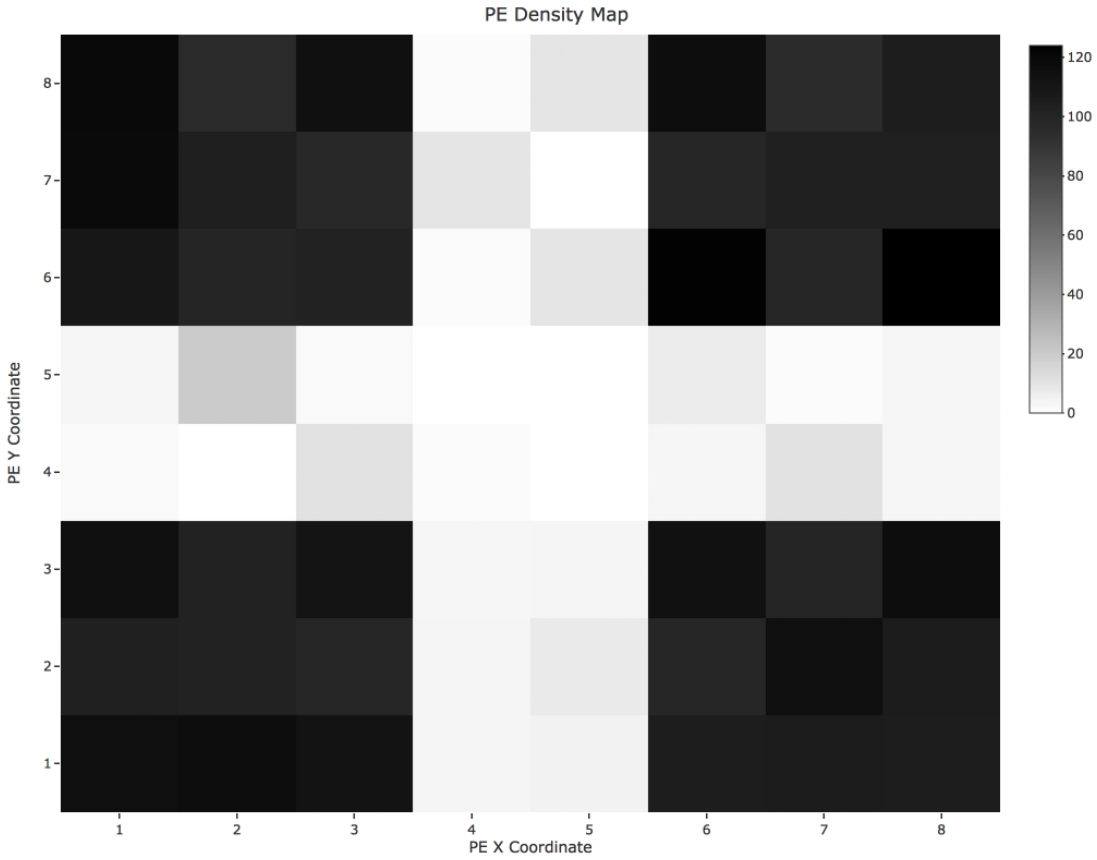
As we can see, we used an exponential decay function to decrease the neighborhood radius and learning rate over time as a function of the numer of learning steps completed; this helped us to obtain very strong network results.

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.

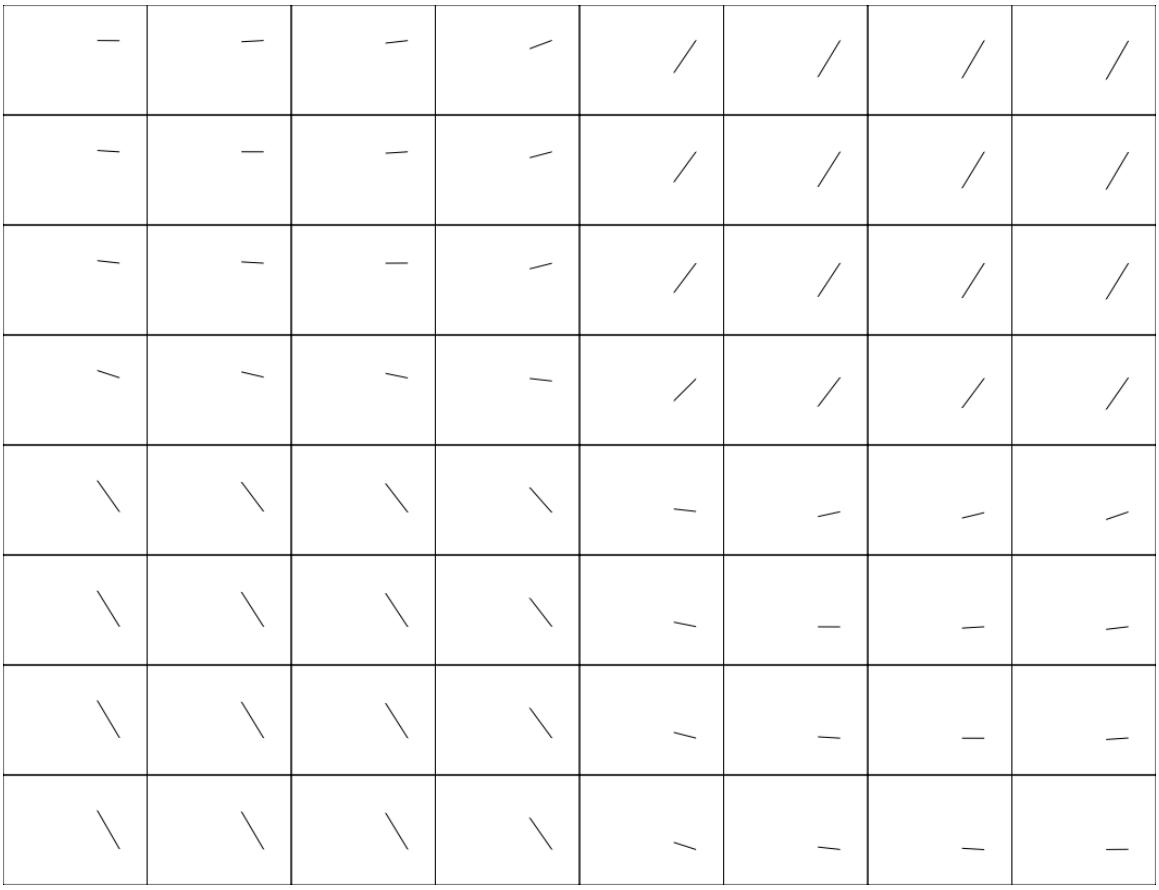
Recalled PE Mapping



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

Blah blah.

Problem 5

Blah blah.

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])

            }

        }

    }

}
```