In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from numpy import random
import time
from sklearn import *
```

In [2]:
```
from mnist import MNIST

mndata = MNIST(r"C:\Users\kalya\OneDrive - University of Illinois at Chicago\!UIC\!Semesters\2nd Sem\Courses
\CS 559 NN\Homeworks\HW2\Q2\data\t")

xtrain, ytrain = mndata.load_training()
xtest, ytest = mndata.load_testing()
xtrain = np.reshape(xtrain,(60000,784))
xtest = np.reshape(xtest,(10000,784))
ytrain = np.reshape(ytrain,(60000,1))
ytest = np.reshape(ytest,(10000,1))
xtrain = (xtrain)/255
xtest = (xtest)/255
xtrain.shape
```

Out[2]:
```
(60000, 784)
```

In [3]:
```
x_train = []
x_test = []
for i in range(len(xtrain)):
    x_train.append(np.append(xtrain[i],1))

for i in range(len(xtest)):
    x_test.append(np.append(xtest[i],1))
x_train = np.asarray(x_train)
x_test = np.asarray(x_test)
```

In [4]:
```
# x_test1 = pd.DataFrame(data = x_test.reshape(len(x_test),785))
# ytest1 = pd.DataFrame(data = ytest.reshape(len(ytest),1))
```

In [5]:
```
# x_test1.head()
```

In [6]:
```
# x=[1,1]
# np.linalg.norm(x)
```

In [7]:
```
ytrain
```

Out[7]:
```
array([[5],
       [0],
       [4],
       ...,
       [5],
       [6],
       [8]], dtype=uint8)
```

In [20]:
```
# x =[[0,1, 1],[1,0, 0],[1,1, 0], [0,0, 0]]
# y=[4,1,2,3]
# x = np.asarray(x)
# y = np.asarray(y)

d = np.zeros(shape = (len(xtrain),10))

for i in range(len(ytrain)):
    d[i][ytrain[i]] = 1

dt = np.zeros(shape = (len(xtest),10))
for i in range(len(ytest)):
    dt[i][ytest[i]] = 1

# d = np.zeros(shape = (len(x),10))
# for i in range(len(y)):
#     d[i][y[i]] = 1
# xtrain = x
# xtrain
d[0]
```

Out[20]:
```
array([0., 0., 0., 0., 0., 1., 0., 0., 0., 0.])
```

In [21]:
```
# # # d = np.zeros(shape = (len(xtrain),10))
# # # for i in range(len(ytrain)):
# # #     d[i][ytrain[i]] = 1

# # # dt = np.zeros(shape = (len(xtest),10))
# # # for i in range(len(ytest)):
# # #     dt[i][ytest[i]] = 1

# # #w = random.normal(size = (10,784))

# d = np.zeros(shape = (len(xtrain)))
# for i in range(len(ytrain)):
#     d[i] = ytrain[i][0]

# dt = np.zeros(shape = (len(xtest)))
# for i in range(len(ytest)):
#     dt[i] = ytest[i][0]
```

In [22]:
```
d
```

Out[22]:
```
array([[0., 0., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 1., 0.]])
```

In [23]:
```
a = [[1.5, 2.5], [2.6, 3.3]]
np.around(a)
```

Out[23]:
```
array([[2., 2.],
       [3., 3.]])
```

In [24]:
```
# weights = np.random.uniform(low =-2, high = 2, size = (784, 60000))
# np.matmul(weights, x_train)
```

In [25]:
```
d
```

Out[25]:
```
array([[0., 0., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 1., 0.]])
```

In [26]:
```
# t = np.asarray([1,2,3])*(np.asarray([4,5,6]))
# t
```

In [232]:
```
a=np.asarray([[1,2,3], [4,5,6]])
print(a[:-1])
c = np.asarray([[7,8,9]])
b =np.append(a,c, axis=0)
b
```

```
[[1 2 3]]
```

Out[232]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [427]:
```
class Neural_Network():

    def __init__(self, x=[[]], y=[], p=[[]], q=[], nHiddenLayers = 0, nHiddenNodes =0, numOutputs = 0, eta = 
1, iter = 0, prec = 0):
        self.data = x
        self.labels = y
        self.test = p
        self.testlabels = q
        self.nInputNodes = x.shape[1]
        self.nHiddenLayers = nHiddenLayers
        self.nHiddenNodes = nHiddenNodes
        self.numOutputs = numOutputs
        self.eta = eta
        self.maxIt = iter
        self.prec = prec
        self.weights=[np.random.uniform(low = -2, high = 2, size = (self.nHiddenNodes, self.nInputNodes))]
        for i in range(self.nHiddenLayers-1):
            self.weights.append(np.random.uniform(low =-2, high = 2, size =(self.nHiddenNodes, self.nHiddenNo
des+1)))
        self.weights.append(np.random.uniform(low =-2, high = 2, size = (self.numOutputs, self.nHiddenNodes+1
)))

    # Tanh activation function for all layers except the output layer
    def tanh(self, s):
        return np.tanh(s)

    # Derivative of tanh for all layers
    def tanhPrime(self, s):
        return (1 - s**2)

    # Activation function for all the input data
    def af_predict(self,t):
#         t = np.tanh(t)
        p =[]
        for i in range(len(t)):
            max = np.argmax(t[i])
            y = np.zeros(len(t[i]))
            y[max] = 1
            p.append(y)
        return np.asarray(p)

    # Activation function for the output layer of the feedforward graph
```

```
    def af(self,t):
#         t = np.tanh(t)
        max = np.argmax(t[0])
        y = np.zeros(self.numOutputs)
        y[max] = 1
#         temp = np.asarray(y).reshape(1,self.numOutputs)
        return np.asarray(y)

#     # Derivative of sigmoid for the output layer
#     def sigmoidPrime(self, s):
#         return s * (1 - s)

#     def decision_A(self,x):
#         out =[]
# #         print(x.shape)
# #         print(f)
#         for i in range(x.shape[0]):
#             s =np.zeros(x.shape[1])
#             max = np.argmax(x[i])
#             #print(max)
#             s[max] = 1
#             out.append(s)
#         out = np.asarray(out)
#         print(out.shape)
#         print(f)
#         return out

    def misclassifications(self, x,y):
        count = 0
        for i in range(y.shape[0]):
            if np.any(x[i]-y[i]):
                count += 1
        return count

    def predict(self, data=[]):
        prev = data.T

        for i in range(len(self.weights)-1):
            temp = (np.matmul(self.weights[i], prev))
            temp2 = self.tanh(temp)
            y=[np.ones(temp2.shape[1])]
            prev = np.append(y, temp2, axis = 0)
#             print(prev.shape)
```

```
#             print(f)
        temp_f = np.matmul(self.weights[self.nHiddenLayers], prev)
        temp4 = self.af_predict(np.transpose(temp_f))
        return temp4

    def feedforward(self, x):
#         print(x)
        self.r = []
        self.r.append(x)
#         print(self.r)
#         print(f)
        prev = x.T
        for i in range(len(self.weights)-1):
#             print(self.weights[i])
#             print(prev)
            temp = np.matmul(self.weights[i], prev)
#             print(temp)
            temp2 = self.tanh(temp)
            prev = np.append([[1]], temp2).reshape(temp2.shape[0]+1,1)
#             print(prev.shape)
#             print(f)
            self.r.append(prev)
#             print(self.r)
#             print('\n')

#         print(self.weights[self.nHiddenLayers])
#         print(prev)
        temp_f = np.matmul(self.weights[self.nHiddenLayers], prev)
        temp4 =self.af(temp_f.T)
        return temp4

    def backward(self, x, y):
        self.gradient=[]
#         print(x)
#         print(y)
#         print(f)
        o = self.feedforward(x)
#         print(o)
        self.out_error = y - o
#         print(self.out_error)
#         print(f)

        for i in reversed(range(len(self.weights))):
```

```
            if i == 0 and i != len(self.weights)-1:
                temp2 = temp.T.dot(self.weights[i+1]).T
#                 print(self.sigmoidPrime(self.r[i+1]))
                temp3 = temp2*self.tanhPrime(self.r[i+1])
                temp4 = (temp3.dot(self.r[i]))*2/len(self.data)
#                 print('w')
#                 print(self.weights[i])
#                 print((self.eta*temp4*2)/len(self.data))
                self.gradient.append(self.eta*2*temp4[1:])

            elif i > 0 and i<len(self.weights)-1:
                temp2 = temp.T.dot(self.weights[i+1]).T

#                 print(self.sigmoidPrime(self.r[i+1]))
                temp3 = temp2*self.tanhPrime(self.r[i+1])
#                 print(temp3.shape)
                temp4 = (temp3.dot(self.r[i].T))*2/len(self.data)
#                 print('w')
#                 print(self.weights[i])
#                 print((self.eta*temp4*2)/len(self.data))
                self.gradient.append(self.eta*2*temp4[1:])
                temp = temp3[1:]

            elif i == len(self.weights)-1:
#                 print(self.weights)
#                 print(self.out_error)
#                 print(o)
#                 print(self.af_derivative(o).T)
                temp = self.out_error.reshape(self.numOutputs,1)
#                 temp = (self.out_error*self.tanhPrime(o)).reshape(self.numOutputs,1)
#                 print(temp.shape)
#                 print(f)
#                 print(self.weights[i])
                temp2 = self.r[i].dot(temp.T)*2/len(self.data)
                self.gradient.append(self.eta*temp2.T)
#                 print(self.weights[i])
#                 print('\n')

        self.gradient = self.gradient[::-1]

        for i in range(len(self.weights)):
            self.weights[i] += self.gradient[i]
#         print(self.weights)
```

```
#         print(f)

    def train (self):
        #print(self.eta)
#         print(self.weights[0].shape)
#         print(f)
        e = 0
        obj_training =[]
        obj_testing = []
        epoch = []
        mis_training = []
        mis_testing =[]
        epoch.append(e)
        pred_train = self.predict(self.data)
        pred_test = self.predict(self.test)
        obj_training.append(((np.linalg.norm(self.labels - pred_train))**2)/len(self.data))
        obj_testing.append(((np.linalg.norm(self.testlabels - pred_test))**2)/len(self.test))
        mis_training.append(self.misclassifications(self.labels, pred_train))
        mis_testing.append(self.misclassifications(self.testlabels, pred_test))
        mse = 100000000000
#         while e <= self.maxIt and mse >= self.prec:
        while e <= self.maxIt:
            prev = mse
#             print(prev)
#             print(f)
            e += 1
            for i in range(len(self.data)):
                self.backward(self.data[i].reshape(1,self.nInputNodes), self.labels[i])
#                 print(self.delta)
#             print(self.weights)
#             print(f)
#             print('after')
#             print(self.weights)
            pred_train = self.predict(self.data)
            pred_test = self.predict(self.test)
            mse = ((np.linalg.norm(self.labels - pred_train))**2)/len(self.data)
            mse_test = ((np.linalg.norm(self.testlabels - pred_test))**2)/len(self.test)
#             print(mse)
#             print(f)
            obj_training.append(mse)
            obj_testing.append(mse_test)
            epoch.append(e)
            mis_training.append(self.misclassifications(self.labels, pred_train))
```

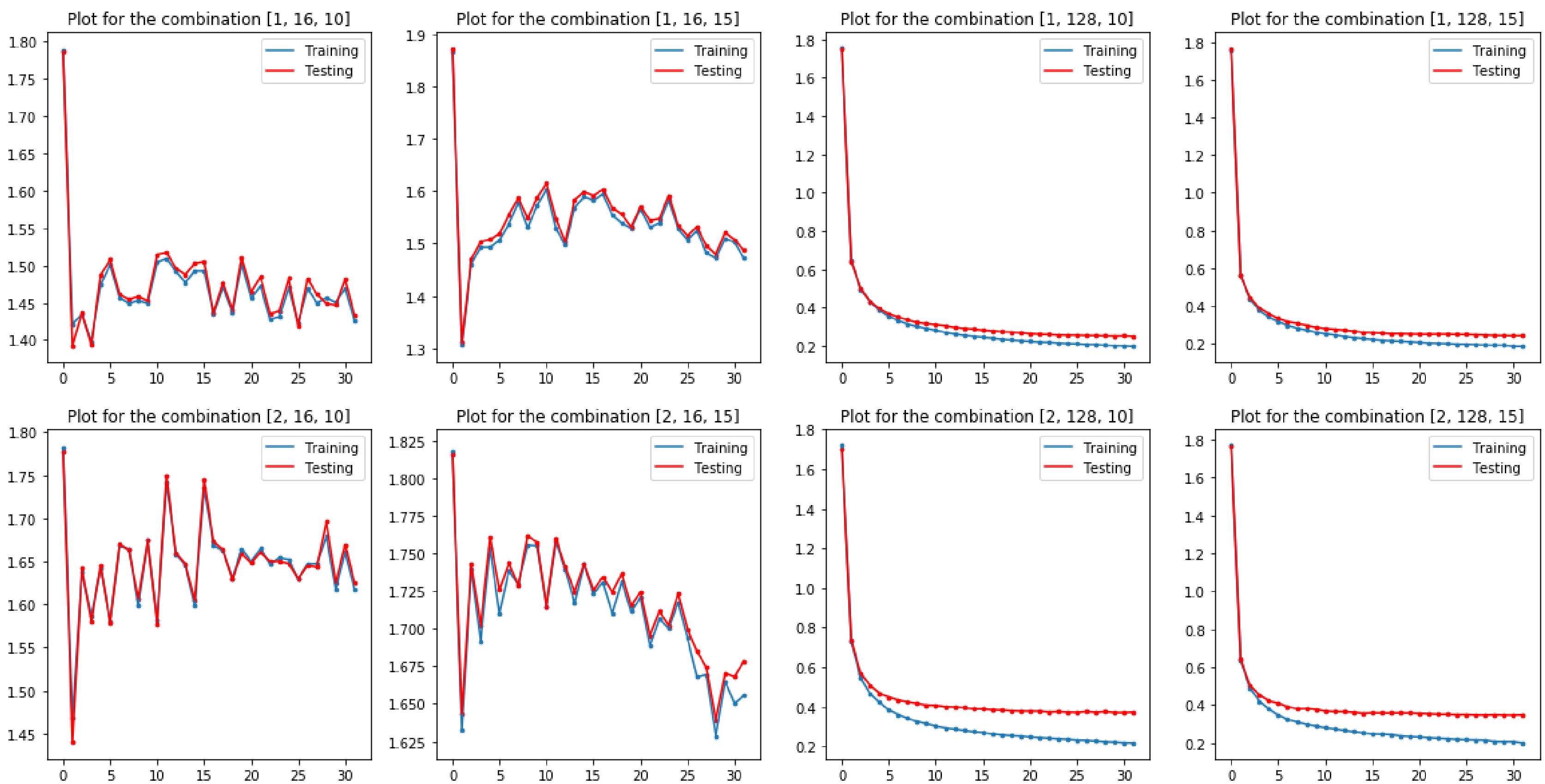
```
            mis_testing.append(self.misclassifications(self.testlabels, pred_test))
            if mse >= prev:
                self.eta = 0.7*self.eta
            prev = mse
        return self.weights, epoch, obj_training, mis_training, obj_testing, mis_testing
```

Hyperparameter Selection

In [417]:
```
# nHiddenLayers = 1
# nHiddenNodes =2
# numOutputs = 10
# eta = 1
# iter = 100
# prec = 1.6
#print(xtrain.shape)

hid = [1,2]
hidnodes= [16,128]
lrate = [10, 15]
a =[]
b =[]
c =[]
q =[]

weights =[]
u =[]
iter = 30
numOutputs = 10
prec = 0.2
for i in range(len(hid)):
    for j in range(len(hidnodes)):
        for k in range(len(lrate)):
            u.append([hid[i],hidnodes[j],lrate[k]])
            print([hid[i],hidnodes[j],lrate[k]])
            NN = Neural_Network(x_train, d, x_test, dt, hid[i], hidnodes[j], numOutputs, lrate[k], iter, prec
)
            w, epoch, obj_training, mis_training, obj_testing, mis_testing = NN.train()
            weights.append(w)
            a.append(obj_training)
            b.append(mis_training)
            c.append(obj_testing)
            q.append(mis_testing)
```

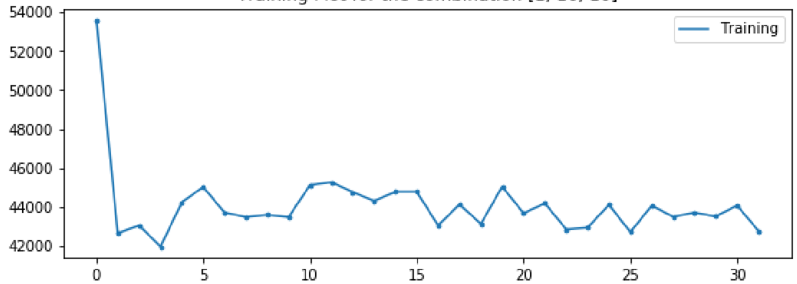
```
[1, 16, 10]
[1, 16, 15]
[1, 128, 10]
[1, 128, 15]
[2, 16, 10]
[2, 16, 15]
[2, 128, 10]
[2, 128, 15]
```

In [418]:
```
fig, ax = plt.subplots(2,4,figsize = (20,10))

# for i in range(len(hid)):
#     for j in range(len(hidnodes)):
#         for k in range(len(lta)):
k = 0
for i in range(2):
    for j in range(4):
        ax[i,j].scatter(epoch,a[k], s  = 7)
        ax[i,j].plot(epoch,a[k], label='Training')
        ax[i,j].scatter(epoch,c[k], s  = 7, c ='r')
        ax[i,j].plot(epoch, c[k], c='r',label='Testing')
        ax[i,j].legend()
        ax[i,j].title.set_text('Plot for the combination ' + str(u[k]))
        k += 1
```

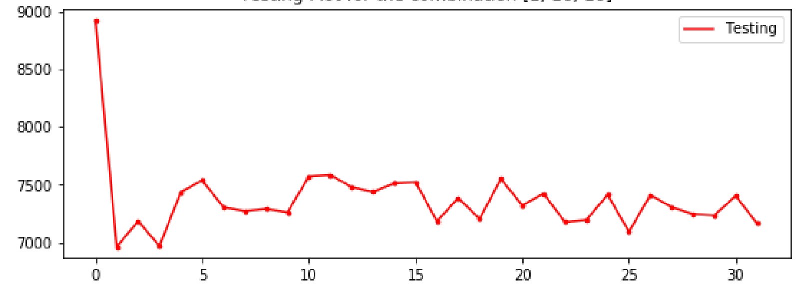


In [419]:
```
fig, ax = plt.subplots(8,2,figsize = (20,30))
k = 0
i = 0
p=0
while i < 8:
    ax[i,0].scatter(epoch,b[k], s  = 7)
    ax[i,0].plot(epoch,b[k], label='Training')
    ax[i,0].legend()
    ax[i,0].title.set_text('Training Plot for the combination ' + str(u[k]))

    ax[i,1].scatter(epoch,q[k], s  = 7, c ='r')
    ax[i,1].plot(epoch, q[k], c='r',label='Testing')
    ax[i,1].legend()
    ax[i,1].title.set_text('Testing Plot for the combination ' + str(u[k]))
    i += 1
    k += 1
```

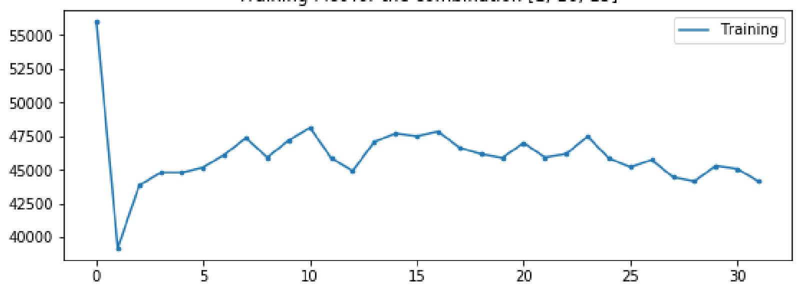
Training Plot for the combination [1, 16, 10]



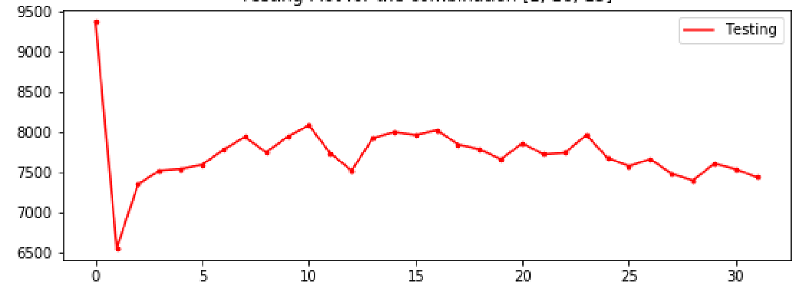
Testing Plot for the combination [1, 16, 10]



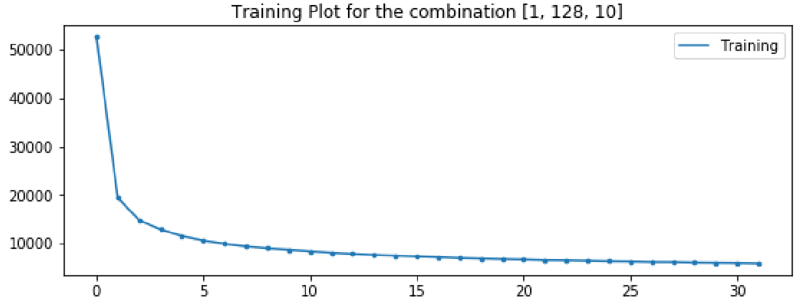
Training Plot for the combination [1, 16, 15]



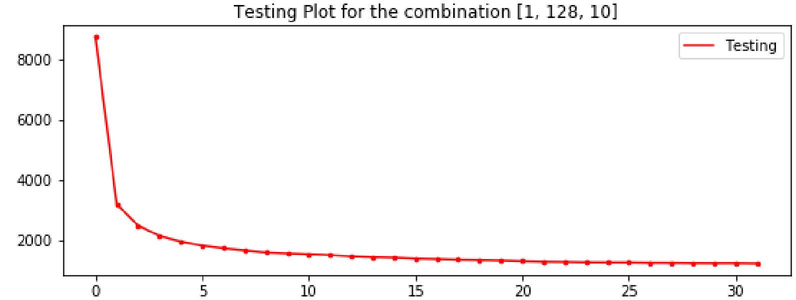
Testing Plot for the combination [1, 16, 15]



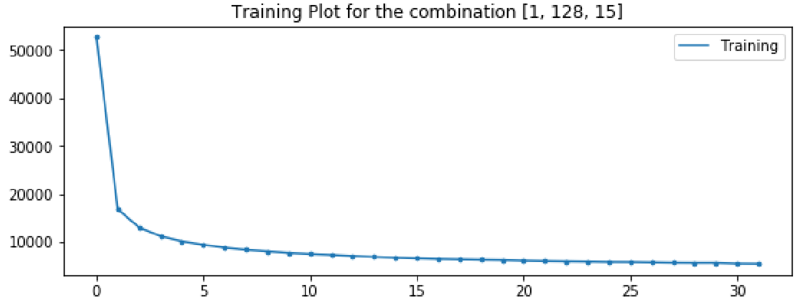
Training Plot for the combination [1, 128, 10]



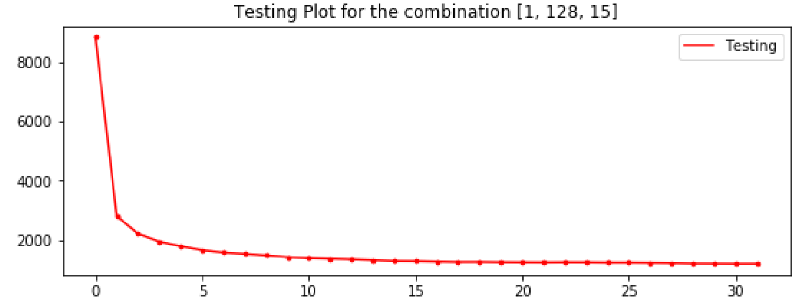
Testing Plot for the combination [1, 128, 10]



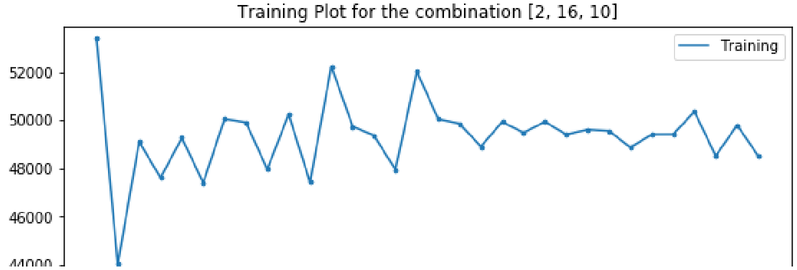
Training Plot for the combination [1, 128, 15]



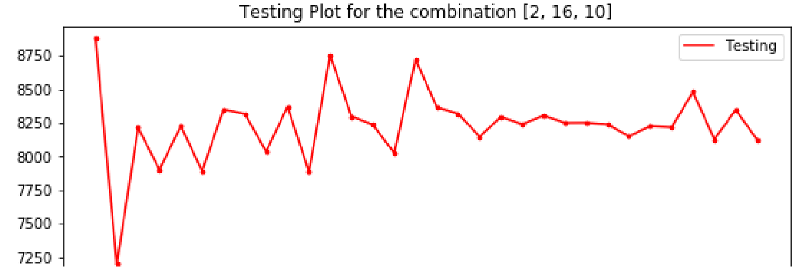
Testing Plot for the combination [1, 128, 15]

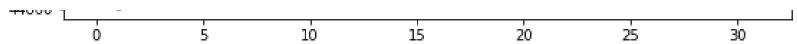


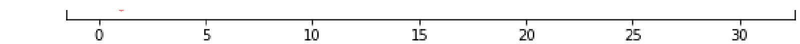
Training Plot for the combination [2, 16, 10]



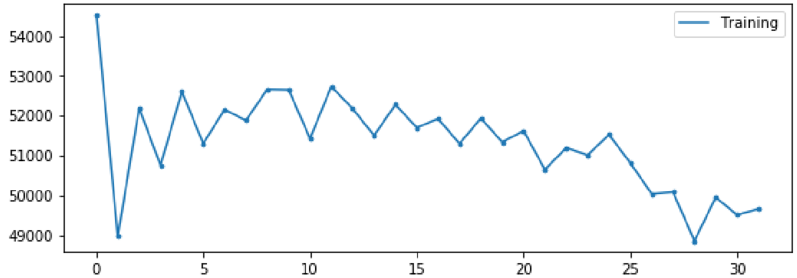
Testing Plot for the combination [2, 16, 10]



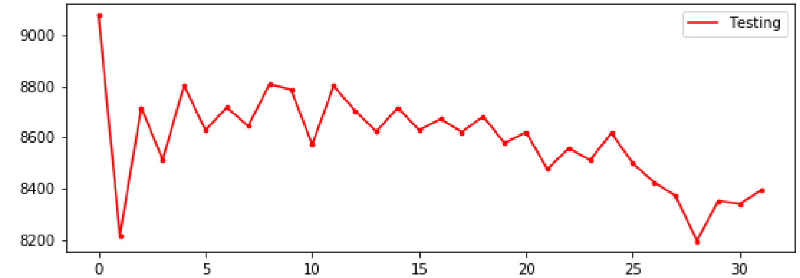




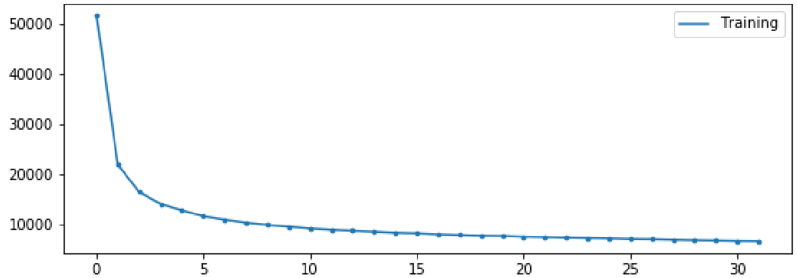
Training Plot for the combination [2, 16, 15]



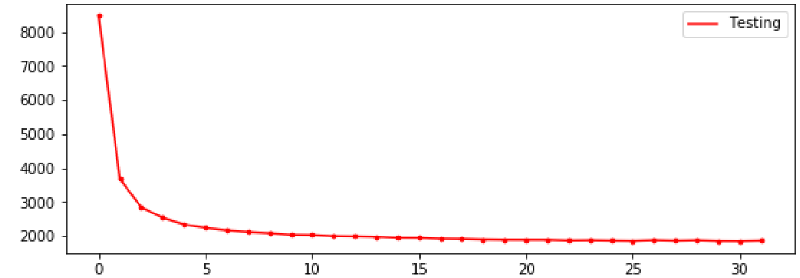
Testing Plot for the combination [2, 16, 15]



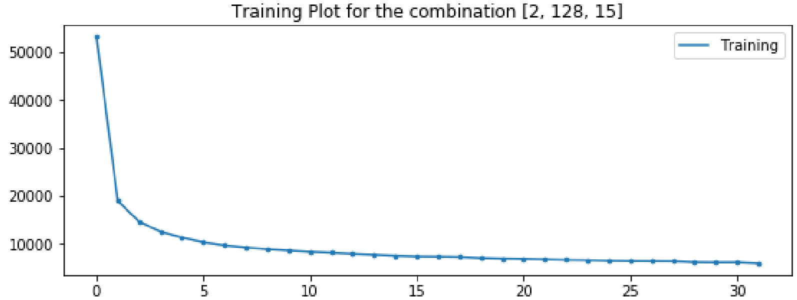
Training Plot for the combination [2, 128, 10]



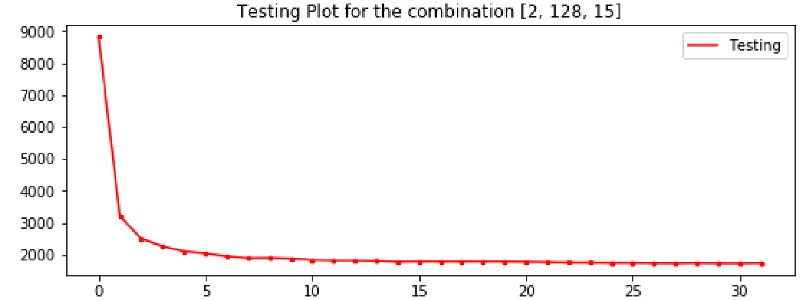
Testing Plot for the combination [2, 128, 10]



Training Plot for the combination [2, 128, 15]



Testing Plot for the combination [2, 128, 15]



In [420]:
```
# Tanh activation function for all layers except the output layer
def tanh(s):
    return np.tanh(s)

# Derivative of tanh for all layers except the output layer
def tanhPrime(s):
    return (1 - s**2)

# Activation function for all the input data
def af_predict(t):
#     t = np.tanh(t)
    p =[]
    for i in range(len(t)):
        max = np.argmax(t[i])
        y = np.zeros(len(t[i]))
        y[max] = 1
        p.append(y)
    return np.asarray(p)

# Activation function for the output layer of the feedforward graph
def af(t):
#     t = np.tanh(t)

    max = np.argmax(t[0])
    y = np.zeros(numOutputs)
    y[max] = 1
#         temp = np.asarray(y).reshape(1,self.numOutputs)
    return np.asarray(y)

def misclassifications(x,y):
    count = 0
    for i in range(y.shape[0]):
        if np.any(x[i]-y[i]):
            count += 1
    return count

def predict(data, weights):
    prev = data.T

    for i in range(len(weights)-1):
        temp = (np.matmul(weights[i], prev))
        temp2 = tanh(temp)
```

```
        y=[np.ones(temp2.shape[1])]
        prev = np.append(y, temp2, axis = 0)
#           print(prev.shape)
#           print(f)
    temp_f = np.matmul(weights[nHiddenLayers], prev)
    temp4 = af_predict(np.transpose(temp_f))
    return temp4
```

In [425]:
```
nHiddenLayers = 1
nHiddenNodes =128
numOutputs = 10
# eta = 10
# iter =50

y = predict(x_test, weights[3])
y
misclassifications(dt, y)
```

Out[425]: 1206

In [435]:
```
nHiddenLayers = 1
nHiddenNodes =128
numOutputs = 10
eta = 15
iter =50


NN = Neural_Network(x_train, d, x_test, dt, nHiddenLayers, nHiddenNodes, numOutputs, eta, iter, prec)
w, epoch, obj_training, mis_training, obj_testing, mis_testing = NN.train()
```

In [436]:
```
y = predict(x_test, w)
y
misclassifications(dt, y)
```

Out[436]: 1097

HW5

1. Network Topology:
   - For simplicity, I am using same number of neurons for all the HIDDEN Layers
   - There are 784 neurons in the input layer
   - For the number of hidden layers and number of hidden nodes per layer, please check the HYPERPARAMETER Selection section below
   - I am using 10 output neurons. For example, the value 1 is represented by [010000000]
   - I am using sigmoid activation function for all the neurons except the output neuron
   - I am converting the output neuron with highest local field value as digit 1 and rest as 0 using “argmax”
   - I am using an energy function of form **summation (di – yi)/n**
   - I am using learning rates of 10 and 15 because with high learning rate, the algorithm is oscillating around the minima.
   - And in the feedback graph, I am increasing the learning rate by 1.05 times (self.eta*1.05) after every layer to tackle the VANISHING gradient problem.

2. I standardized the data.
3. Design Process:
   a. I am using a generalizable algorithm.
   b. First, I built an algorithm without matrices. It worked for homework 4 but it was not scalable.
   c. So, next I developed an algorithm using numpy arrays. But with this algorithm, the MSE is continuously increasing with every epoch.
   d. After carefully crosschecking the intermediate values with values I calculated myself, I found that instead of temporarily storing the gradient vectors of each layer, I was updating the weights in the same iteration. So, since we need the original weights for the next layer not the updated ones, the algorithm was not giving optimal weights at each epoch.
   e. Then, the algorithm was working well in terms of MSE but not misclassifications. I was just using sigmoid function at the output neurons. Since, we might get more than one 1’s in the output neurons, the misclassifications remained same even though the MSE was decreasing with every epoch.
   f. So, for the next algorithm, instead of using the sigmoid activation functions at the output neurons, I just used the “argmax” function which gives the index of maximum value. I am changing the value at the output of argmax, in an numpy array of zeros of size 10, to 1.
   g. Then, I forgot to include the learning rate (self.eta) parameter in the algorithm. I included it now.
   h. This is my final algorithm.

4. **Tanh activation Function:** (for all layers except the output layer)

```
def tanh(self, s):
    return np.tanh(s)
```

5. **Derivative of tanh:** (for all layers except the output layer)

```
def sigmoidPrime(self, s):
    return s * (1 - s)
```

6. **Activation Function for the output layer in the feedforward graph**

```
def af(self,s):
    t = 1/(1+np.exp(-s))
    max = np.argmax(t[0])
    y = np.zeros(self.numOutputs)
    y[max] = 1
    return np.asarray(y)
```

7. **Derivative of sigmoid for the output layer**

```
def sigmoidPrime(self, s):
    return s * (1 - s)
```

8. **Activation function for the whole input data**

```
def af_predict(self,s):
    t = 1/(1+np.exp(-s))
    p =[]
    for i in range(len(t)):
        max = np.argmax(t[i])
        y = np.zeros(len(t[i]))
        y[max] = 1
        p.append(y)
    return np.asarray(p)
```

9. **Misclassifications Function:**

This is used to calculate the mean squared error corresponding to the given weights.

Algorithm:

```
def misclassifications(self, x,y):
    count = 0
    for i in range(y.shape[0]):
        if np.any(x[i]-y[i]):
            count += 1
    return count
```

10. I created a **NeuralNetworks** class which has predict, feedforward, backward and train functions.
11. The NeuralNetworks class takes the data, labels, number of hidden layers, number of nodes per hidden layer, learning rate, and maximum iterations as input parameters.
12. **Function call:**

    Ex:

    ```
    nLayers = 1
    nNodes = 2
    nOut = 1
    eta = 10
    iter = 10
    ep = 0.2
    w, epoch, obj, mis = NeuralNetwork(x, d, nLayers, nNodes, nOut, eta, iter, ep).train()
    ```
13. I am using a list with name "**weights**" to store weights (including biases). I am uniformly choosing weights between -2 and 2. For each layer, I am using a list of size = (number of nodes in the current layer x number of nodes in the previous layer) in the list "weights".

```
self.weights=[np.random.uniform(low = -2, high = 2, size = (self.nHiddenNodes, self.nInputNodes))]
for i in range(self.nHiddenLayers-1):
self.weights.append(np.random.uniform(low =-2, high = 2, size =(self.nHiddenNodes, self.nHiddenNodes+1)))        # +1 is for biases
self.weights.append(np.random.uniform(low =-2, high = 2, size = (self.numOutputs, self.nHiddenNodes+1)))        # +1 is for biases
```

Ex: For number of hidden layers = 1 and num of nodes per hidden layer = 2.

```
[array([[0.78888993, 0.36114814, 0.84949542],
```

[0.05266805, 0.89136156, 0.4386164]]), array([[0.22397066, 0.34915997, 0.12250444]])]

14. Train function:

- This function calls the backward function which updates the weights every epoch until maximum epochs are reached.
- And this finds the mse and number of misclassifications for both training and testing datasets using updated weights after each epoch.
- If after an epoch, the mse is more than that of the mse of previous epoch, I am reducing the learning rate to 0.9 * learning rate

Algorithm:

```
def train (self):
    e = 0
    obj_training =[]
    obj_testing = []
    epoch = []
    mis_training = []
    mis_testing =[]
    epoch.append(e)
    pred_train = self.predict(self.data)
    pred_test = self.predict(self.test)
    obj_training.append(((np.linalg.norm(self.labels - pred_train))**2)/len(self.data))
    obj_testing.append(((np.linalg.norm(self.testlabels - pred_test))**2)/len(self.test))
    mis_training.append(self.misclassifications(self.labels, pred_train))
    mis_testing.append(self.misclassifications(self.testlabels, pred_test))
    mse = 100000000000
    while e <= self.maxIt:
        prev = mse
        e += 1
        for i in range(len(self.data)):
            self.backward(self.data[i].reshape(1,self.nInputNodes), self.labels[i])
        pred_train = self.predict(self.data)
        pred_test = self.predict(self.test)
        mse = ((np.linalg.norm(self.labels - pred_train))**2)/len(self.data)
        mse_test = ((np.linalg.norm(self.testlabels - pred_test))**2)/len(self.test)
```

```
        obj_training.append(mse)
        obj_testing.append(mse_test)
        epoch.append(e)
        mis_training.append(self.misclassifications(self.labels, pred_train))
        mis_testing.append(self.misclassifications(self.testlabels, pred_test))
        if mse >= prev:
            self.eta = 0.9*self.eta
        prev = mse
    return self.weights, epoch, obj_training, mis_training, obj_testing, mis_testing
```

15. Predict Function: This finds the output labels for the whole input data

```
def predict (self, data=[]):
    prev = data.T
    for i in range(len(self.weights)-1):
        temp = (np.matmul(self.weights[i], prev))
        temp2 = self.sigmoid(temp)
        prev = temp2
    temp_f = np.matmul(self.weights[self.nHiddenLayers], prev)
    temp4 = self.af_predict(np.transpose(temp_f))
    return temp4
```

16. Feedforward function: This finds the output label for the given input datapoint

For the given input, this records the local field values and values after the activation function of every neuron in each layer in the feedforward graph.

Algorithm:

```
def feedforward (self, x):
    self.r = []
    self.r.append(x)
    prev = x.T
    for i in range(len(self.weights)-1):
        temp = np.matmul(self.weights[i], prev)
```

```
        temp2 = self.sigmoid(temp)
        self.r.append(temp2)
        prev = temp2
    temp_f = np.matmul(self.weights[self.nHiddenLayers], prev)
    temp4 =self.af(temp_f.T)
    return temp4
```

17. Backward function:

- For the given input, this records the values before the weights of all the layers in the backward (feedback) graph and the corresponding gradient descent vector.
- Then it updates the weights using the gradient descent vector.

Algorithm:

```
def backward (self, x, y):
    self.delta = []
    self.gradient=[]
    o = self.feedforward(x)
    self.out_error = y – o
    for i in reversed(range(len(self.weights))):
        if i == 0 and i != len(self.weights)-1:
            temp2 = self.weights[i+1].T.dot(temp)
            temp3 = temp2*self.sigmoidPrime(self.r[i+1])
            temp4 = (temp3.dot(self.r[i]))*2/len(self.data)
            self.gradient.append(self.eta*1.05*temp4)
            temp = temp3
            self.delta.append(temp3)

        elif i > 0 and i<len(self.weights)-1:
            temp2 = self.weights[i+1].T.dot(temp)
            temp3 = temp2*self.sigmoidPrime(self.r[i+1])
            temp4 = (temp3.dot(self.r[i].T))*2/len(self.data)
            self.gradient.append(self.eta*1.05*temp4)
```

```
            temp = temp3
            self.delta.append(temp3)

          elif i == len(self.weights)-1:
            temp = self.out_error.reshape(self.numOutputs,1)
            temp2 = self.r[i].dot(temp.T)*2/len(self.data)
            self.delta.append(temp)
            self.gradient.append(self.eta*temp2.T)

        self.gradient = self.gradient[::-1]
        for i in range(len(self.weights)):
          self.weights[i] += self.gradient[i]
```

18. Hyperparameter Selection:

- I am using three lists each for number of hidden layers, number of nodes per hidden layer and learning rates
- I am building 8 Neural network models using the above
- I am running the algorithm for 50 epochs for now.

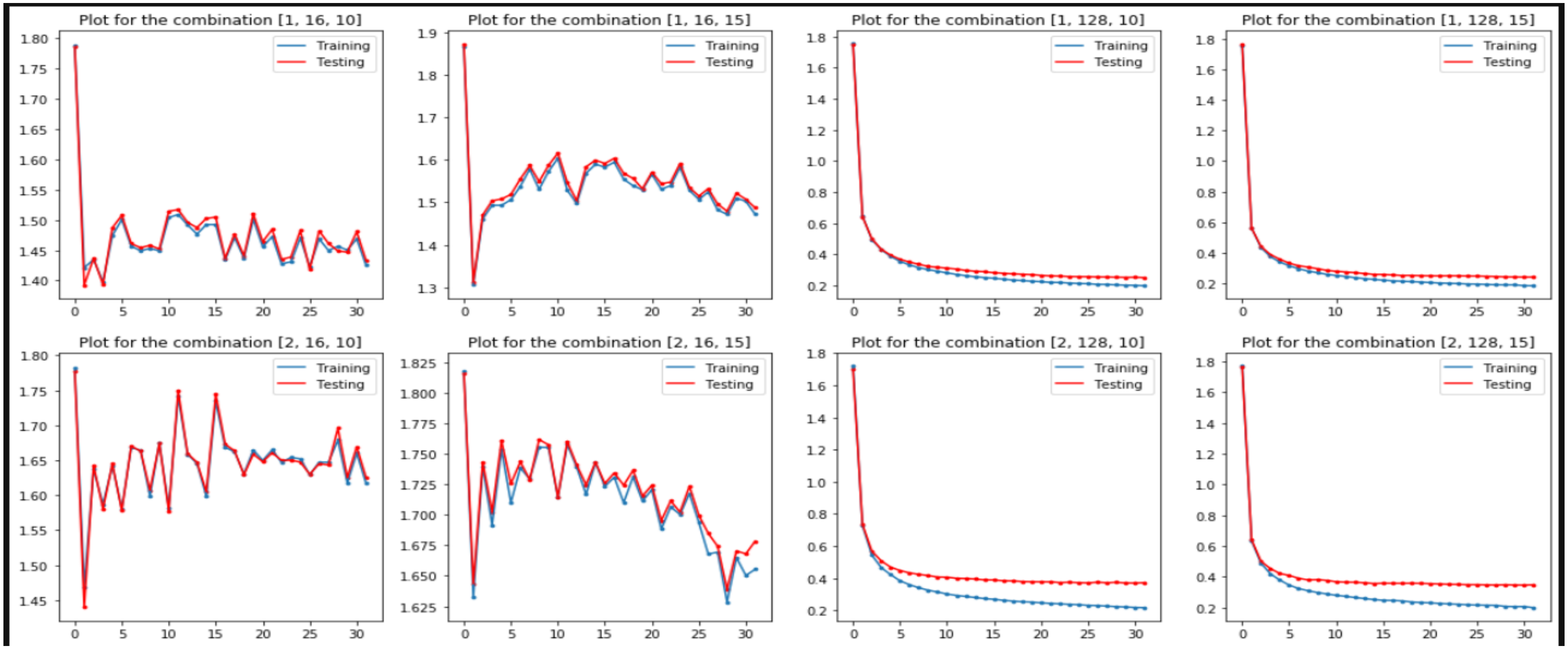
```
hid = [1,2]
hidnodes= [64,128]

# I read some where that the number of hidden nodes in the powers of 2 would result in better performance

lrate = [10,15]
a =[]
b =[]
c=[]
q =[]
iter = 50
numOutputs = 10
prec = 1.6
for i in hid:
```

```
for j in hidnodes:
    for k in lrate:
        #print([i,j,k])
        NN = Neural_Network(xtrain, d, xtest, dt, i, j, numOutputs, k, iter, prec)
        w, epoch, obj_training, mis_training, obj_testing, mis_testing = NN.train()
        a.append(obj_training)
        b.append(mis_training)
        c.append(obj_testing)
        q.append(mis_testing)
```

I generated the plots using matplotlib. Please check my Jupyter notebook.

19. Results of Hyperparameter Selection:

From the plots, I got the continuous decrease in the MSE with number of hidden nodes = 128

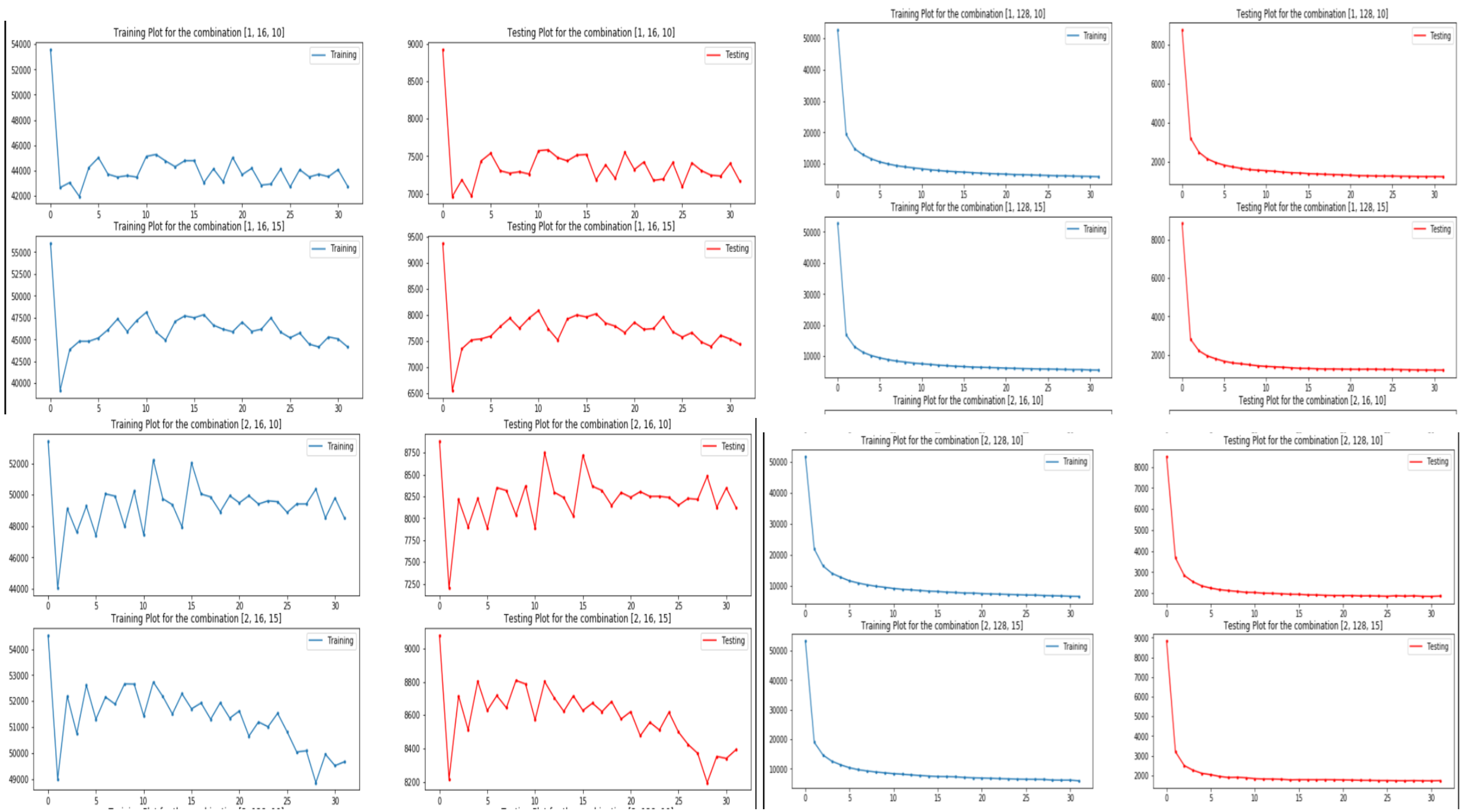


and less testing misclassifications with the following combination:

Number of hidden layers: 1

Number of nodes per hidden layer: 128

Learning Rate = 10



20. Conclusion:

Now to further improve the accuracy, I am running the algorithm for 30 epochs with

Number of hidden layers: 1

Number of nodes per hidden layer: 128

Learning Rate = 10

And I got the accuracy of 89 %

```
nHiddenLayers = 1
nHiddenNodes =128
numOutputs = 10
eta = 15
iter =50


NN = Neural_Network(x_train, d, x_test, dt, nHiddenLayers, nHiddenNodes, numOutputs, eta, iter, prec)
w, epoch, obj_training, mis_training, obj_testing, mis_testing = NN.train()
```

```
y = predict(x_test, w)
y
misclassifications(dt, y)
```

```
1097
```