Deep Neural Networks

Neural Networks: ECE 5930

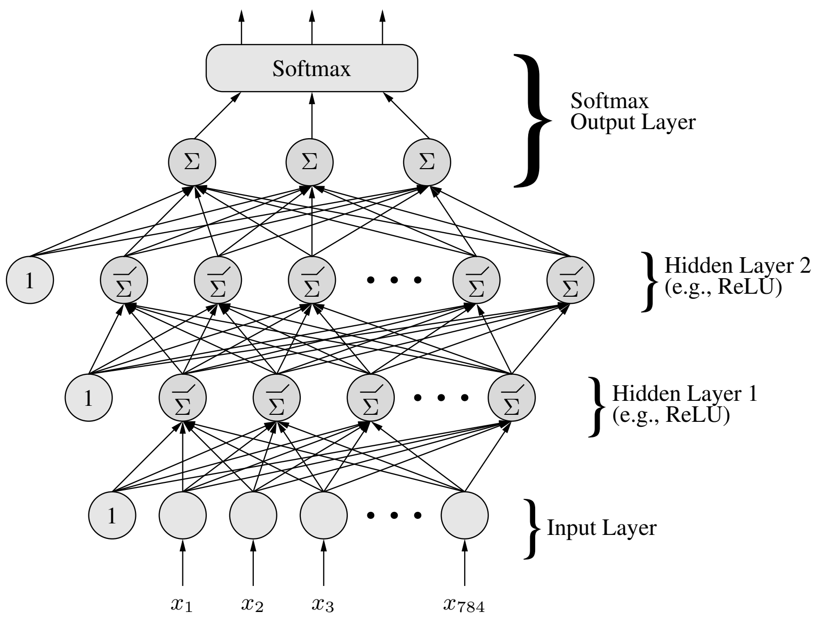


Figure: Two Hidden Layer Neural Network

Clint Ferrin
Utah State University

Mon Oct 23, 2017

Contents

List of Figures

List of Tables

1 Summary

Neural Networks have applications in image recognition, data compression, and even stock market prediction. The basic concept behind Neural Networks is simply depicted as seen on the main figure of title page. This paper presents the basic structure for machine learning on classified data using a randomly generated data-set (2-classes), and the MNIST data-set (10 classes).

The MNIST data-set consists of 70,000 small images of digits handwritten by high school students and employees of the US Census Bureau. Each image is 28 28 so that when it is vectorized it has dimension 784. The MNIST data-set is ideal for machine learning because of the variable nature of handwriting and the limited classes of numbers.

Ten numbers from the data-set can been seen in Figure 1.

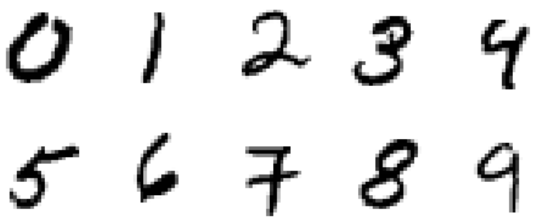


Figure 1: 10 Digits from the MNIST Data-set

The remainder of the paper will be dedicated to analyzing the effectiveness of Neural Networks in identifying correctly different classes of data such as the numbers seen in Figure 1.

2 Program Description

The generic neural network that I wrote can have any number of layers, neurons, and activation function passed to it for an `n` demensional input with `k` number of classes. The Network is initialized by specifying `num_inputs`, `num_outputs`, `batch_size`, and `epics`.

In testing the different classes in this paper, the ReLU (Rectified Linear Unit) was used for the majority of the classification problems.

The desired data is read in, and any activation functions are defined for the different layers. As seen in the heading `# input layer`, the layers are created by passing the number of inputs that they will receive, the number of desired neurons, and the type of activation function. The final layer seen in the code snippet below does not have an activation function because by default softmax is run on the output of the network. In this way, the output option can easily be changed between the softmax and sigmoid functions.

Additional parameters exist for the initialization of the network, but they are optional parameters. Such variables include the momentum β and step size η, as seen in the initialization of the Neural Network.

```
    num_outputs= 2
    batch_size = 200
    epics = 800

    # X,Y = pickle.load(open("./in_out.p","rb"))
    # X,Y,X_test,Y_test = get_classasgn_80_20()
    X,Y = get_moon_class_data()
```

```
    X_test,Y_test = get_moon_gendata()
    # X,Y = get_mnist_train("./data")

    relu = activation_function(relu_func,relu_der)
    sig  = activation_function(sigmoid_func,sigmoid_der)
    no_activation = activation_function(return_value,return_value)

    num_neurons = 5
    # input layer
    layers = [layer(num_inputs,num_neurons,sig)]
    layers.append(layer(num_neurons,num_outputs,sig))

    # create neural network
    network = NeuralNetwork(layers)

    # train network
    network.train_network(X,Y,batch_size,epics)

    # classify data
    Yhat = network.classify_data(X_test)
    network.validate_results(Yhat,Y_test)
```

For a full description of the different classes, such as the class `layer` and `activation_function`, see **??**.

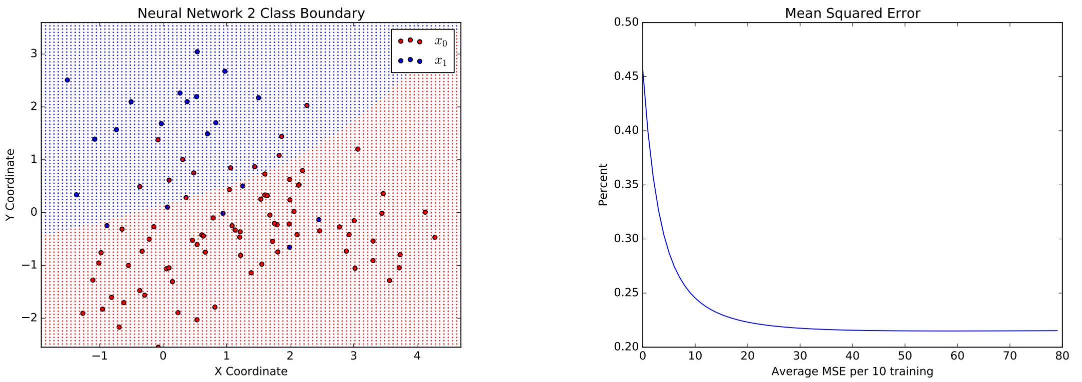
The Training of the system is done by back propagation, which will be discusses in later detail later on.

3 Two-class Classifier

The data set from `classasgntrain1.dat` is a grouping of data centered around 10 different points with a Gaussian Distribution for each class. I split the data into 80% training data and 20% testing data using the function seen in the listing below:

```
def get_classasgn_80_20():
    data = np.loadtxt("./data/classasgntrain1.dat",dtype=float)
    x0 = data[:,0:2]
    x1 = data[:,2:4]
    data = data_frame(x0,x1)
    return data.train_tot,data.train_class_tot,data.test_data,data.test_class_tot

class data_frame:
    def __init__(self, data0, data1):
        self.x0 = data0
        self.x1 = data1
        self.xtot = np.r_[self.x0,self.x1]
        self.N0 = self.x0.shape[0]
        self.N1 = self.x1.shape[0]
        self.N = self.N0 + self.N1
        self.xlim = [np.min(self.xtot[:,0]),np.max(self.xtot[:,0])]
        self.ylim = [np.min(self.xtot[:,1]),np.max(self.xtot[:,1])]
        class_x0 = np.c_[np.zeros([self.N0,1]),np.ones([self.N0,1])]
        class_x1 = np.c_[np.ones([self.N1,1]),np.zeros([self.N1,1])]
        self.class_tot = np.r_[class_x0,class_x1]
        self.y = np.r_[np.ones([self.N0,1]),np.zeros([self.N1,1])]

        # create a training set from the classasgntrain1.dat
        self.train_x0 = data0[0:80]
        self.train_x1 = data1[0:80]
        self.train_tot = np.r_[data0[0:80],data1[0:80]]
        self.train_class_tot = np.r_[self.class_tot[0:80],self.class_tot[100:180]]
        self.test_data = np.r_[data0[80:100],data1[80:100]]
        self.test_class_tot = np.r_[self.class_tot[80:100],self.class_tot[180:200]]
```

The network trained the data using sigmoid functions, and it produced the output seen in Figure 2:



(a) 80% Training Data, 20% Testing Data (b) Mean Squared Error for 80%, 20% Data

Figure 2: Trained Output for 80 Training Data, 20 Testing Data with 0.8 Momentum

The resulting listing from the program showed that it correctly classified the small batch of test data with only 3 mistakes with a 92.5% accuracy.

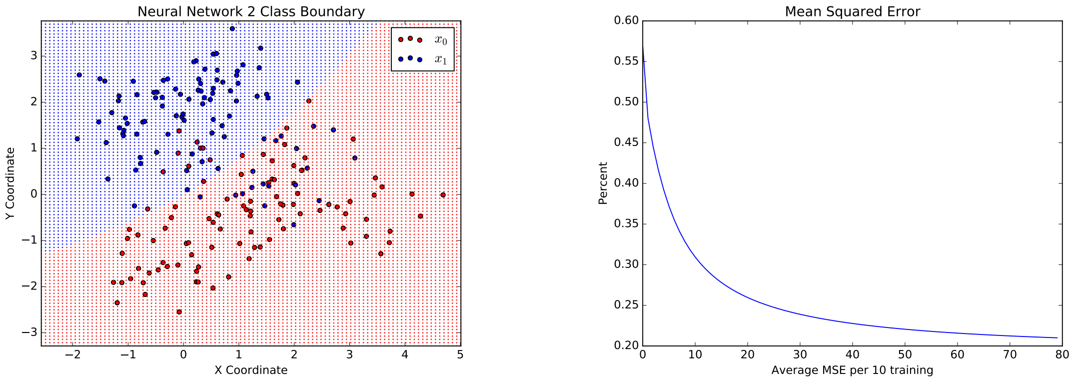
```
-------- Running --------
Training Data...
Epic 1 MSE: 0.484913
Epic 81 MSE: 0.263872
Epic 161 MSE: 0.229964
Epic 241 MSE: 0.220691
Epic 321 MSE: 0.217230
Epic 401 MSE: 0.215792
Epic 481 MSE: 0.215209
Epic 561 MSE: 0.215026
Epic 641 MSE: 0.215038
Epic 721 MSE: 0.215145
Final MSE: 0.215292
3 Mistakes. Training Accuracy: 92.50%


real: 58.670s

Press ENTER or type command to continue
```

In previous processing, I found that the classification methods in Table 1 performed with the following errors in percent. Note that the Bayes Optimal Classifier performed the best because it knew the true distribution of the data.

		Errors in %	
Method	Run-time	Training	Test
Linear Regression	1.23s	14.5	20.49
Quadratic Regression	1.70s	14.5	20.44
Linear Discriminant Analysis	2.49s	15.0	19.98
Quadratic Discriminant Analysis	3.26s	14.5	20.23
Logistic Regression	2.00s	14.0	20.00
1-Nearest Neighbor	35.02s	00.0	21.83
5-Nearest Neighbor	37.92s	12.0	20.29
15-Nearest Neighbor	36.47s	16.0	19.25
Bayes Naive	1.22s	14.0	20.04
Bayes Optimal Classifier	0.20s	14.0	19.14

Table 1: Binary Classifier Performance Comparison

To compare the Neural Network with the other classifiers, I trained on all data points from the `classasgntrain1.dat` data set and ran it on 20000 randomly generated data points. The results can be seen in Figure 3



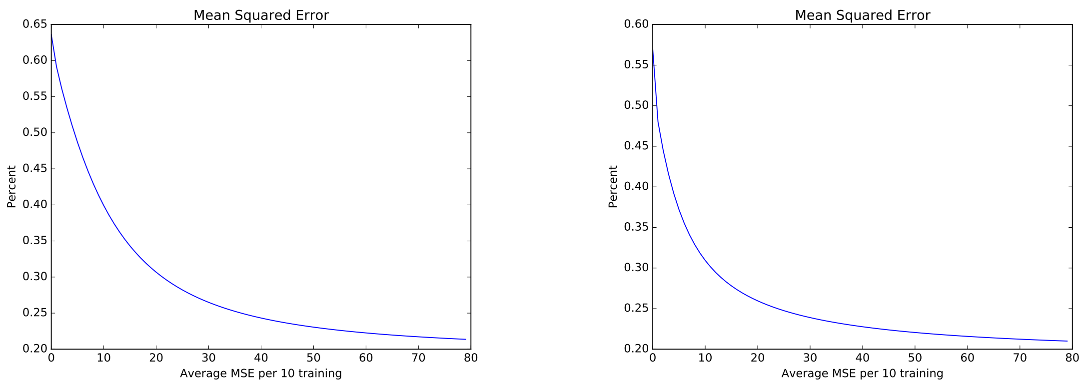
(a) Training on All 200 data points

(b) MSE for the Training Routine

Figure 3: Trained Output for 200 Data Points in `classasgntrain1.dat` with 0.8 Momentum and Step Size of 0.1

```
-------- Running --------
Training Data...
Epic 1 MSE: 0.668210
Epic 81 MSE: 0.339729
Epic 161 MSE: 0.277568
Epic 241 MSE: 0.251570
Epic 321 MSE: 0.237392
Epic 401 MSE: 0.228466
Epic 481 MSE: 0.222349
Epic 561 MSE: 0.217920
Epic 641 MSE: 0.214591
Epic 721 MSE: 0.212018
Final MSE: 0.210010
3888 Mistakes. Training Accuracy: 80.56%
```

The Neural Network returned an error of 19.44%, which puts its results just behind the Bayes Optimal Classifier and the k-nearest neighbor approach. It is also important to note that the momentum term has a significant effect on the speed at which the Mean Squared Error drops. Figure 4 shows the dramatic speed difference that the momentum has on the convergence of the Mean Squared Error. Increasing the momentum to 0.8 did not have a significant effect on the percent of errors.



(a) MSE with No Momentum

(b) MSE with 0.8 Momentum

Figure 4: Comparison of Two-class Classifier with and without Momentum

4 Ten-class Classifier

5 Appendix

```
# Clint Ferrin
# Oct 12, 2017
# Neural Network Classifier

import matplotlib.pyplot as plt
import numpy as np
import pickle
from   tensorflow.examples.tutorials.mnist import input_data

def main():
    num_inputs = 2
    num_outputs= 2
    batch_size = 200
    epics = 800

    # X,Y = pickle.load(open("./in_out.p","rb"))
    # X,Y,X_test,Y_test = get_classasgn_80_20()
    X,Y = get_moon_class_data()
    X_test,Y_test = get_moon_gendata()
    # X,Y = get_mnist_train("./data")

    relu = activation_function(relu_func,relu_der)
    sig  = activation_function(sigmoid_func,sigmoid_der)
    no_activation = activation_function(return_value,return_value)

    num_neurons = 5
    # input layer
    layers = [layer(num_inputs,num_neurons,sig)]
    layers.append(layer(num_neurons,num_outputs,sig))

    # create neural network
    network = NeuralNetwork(layers)

    # train network
    network.train_network(X,Y,batch_size,epics)

    # classify data
    Yhat = network.classify_data(X_test)
    network.validate_results(Yhat,Y_test)

    plot_data(X[0:100],X[100:200])
    xtot = np.r_[X,X_test]
    xlim = [np.min(xtot[:,0]),np.max(xtot[:,0])]
    ylim = [np.min(xtot[:,1]),np.max(xtot[:,1])]
    plot_boundaries(xlim,ylim,network.classify_data)
    plt.show()

    # plot error
    network.plot_error()
    plt.show()

def get_mnist_train(file_path):
    mnist = input_data.read_data_sets(file_path)
    X = mnist.train.images
    y = mnist.train.labels.astype("int")
    Y = (np.arange(np.max(y) + 1) == y[:, None]).astype(float)
    return X,Y

def get_2_class_data():
    X = np.array([[0.05, 0.1],
                  [0.07, 0.1],
                  [0.05, 0.1],
                  [0.05, 0.1],
                  [0.05, 0.1]])

    Y = np.array([[0.01, 0.99],
```

```
                  [0.01, 0.99],
                  [0.01, 0.99],
                  [0.01, 0.99],
                  [0.01, 0.99]])
    return X,Y

def get_3_class_data():
    X = np.array([[0.05, 0.1],
                  [0.07, 0.3],
                  [0.09, 0.5],
                  [0.05, 0.1]])

    Y = np.array([[1, 0, 0],
                  [0, 1, 0],
                  [0, 0, 1],
                  [1, 0, 0]])
    return X,Y

def get_sprial_class_data():
    np.random.seed(0)
    N = 100 # number of points per class
    D = 2 # dimensionality
    K = 3 # number of classes
    X = np.zeros((N*K,D))
    y = np.zeros(N*K, dtype='uint8')
    for j in xrange(K):
        ix = range(N*j,N*(j+1))
        r = np.linspace(0.0,1,N) # radius
        t = np.linspace(j*4,(j+1)*4,N) + np.random.randn(N)*0.2 # theta
        X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
        y[ix] = j
    # fig = plt.figure()
    # plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral)
    # plt.xlim([-1,1])
    # plt.ylim([-1,1])
    Y = (np.arange(np.max(y) + 1) == y[:, None]).astype(float)
    return X,Y

class NeuralNetwork:
    def __init__(self, layers, softmax=True, momentum=0, eta=0.1, MSE_freq=10):
        self.softmax=softmax
        self.num_layers = len(layers)
        self.num_outputs = layers[self.num_layers-1].num_neurons
        self.layers = layers
        self.momentum = momentum
        self.eta = eta
        self.softmax = softmax
        self.error_plot = []
        self.error_array = []
        self.MSE_freq = MSE_freq
        self.__set_GRV_starting_weights()

    def __set_GRV_starting_weights(self):
        for i in range(self.num_layers-2):
            self.layers[i].num_outputs = self.layers[i+1].num_neurons
        self.layers[-1].num_outputs = self.num_outputs

        for layer in self.layers:
            sigma = np.sqrt(float(2) / (layer.num_inputs + layer.num_inputs))
            layer.W = np.random.normal(0,sigma,layer.W.shape)

    def forward_prop(self, X):
        prev_out = X
        for layer in self.layers:
            prev_out = np.c_[prev_out,np.ones([prev_out.shape[0],1])]
            prev_out = layer.forward(prev_out)

        if self.softmax is True:
            self.layers[-1].output = self.stable_softmax(self.layers[-1].net)

        return self.layers[-1].output

    def classify_data(self, X):
```

```
        Yhat = self.forward_prop(X)
        class_type = np.argmax(Yhat,axis=1)
        return class_type

    def train_network(self, X, Y, batch_size, epics):
        print("Training Data...")

        if epics > 5000:
            print_frequency = epics/100
            print(print_frequency)
        else:
            print_frequency = epics/10

        for i in range(epics):
            batch = np.random.randint(0,X.shape[0],batch_size)
            # self.train_data(X[batch],Y[batch])
            self.train_data(X,Y)
            if i%print_frequency is 0:
                print("Epic %d MSE: %f"%(i+1, np.mean(self.error_array[-self.MSE_freq:])))

        # create error plot
        print("Final MSE: %f"%(np.mean(self.error_array[-self.MSE_freq:])))
        plot = self.error_array[::-1]
        for i in range(0,len(plot),self.MSE_freq):
            self.error_plot.append(np.mean(plot[i:i+self.MSE_freq]))
        self.error_plot = self.error_plot[::-1]

    def train_data(self, X, Y):
        Yhat = self.forward_prop(X)
        dE_dH = (Yhat-Y).T
        iterlayers = iter(self.layers[::-1])

        # back propagation
        if self.softmax is True:
            dE_dWeight = -np.dot((Y-Yhat).T,self.layers[-1].weight_der) / \
                        self.layers[-1].weight_der.shape[0]

            self.layers[-1].W += -self.eta*(dE_dWeight + self.momentum*self.layers[-1].
                momentum_matrix)
            self.layers[-1].momentum_matrix = dE_dWeight
            dE_dH = (Yhat-(Y==1).astype(int)).T[0,:]/Yhat.shape[0]
            next(iterlayers)

        for layer in iterlayers:
            dE_dNet = layer.der(layer.output).T*dE_dH
            dE_dWeight = (np.dot(dE_dNet,layer.weight_der))/layer.weight_der.shape[0]
            dE_dH = np.dot(layer.W[:,0].T,dE_dNet)

            layer.momentum_matrix = \
                    self.momentum * layer.momentum_matrix + \
                    self.eta * dE_dWeight
            layer.W += - layer.momentum_matrix

        # self.error_array.append(-np.mean(np.sum(np.log(Yhat)*Y)))
        self.error_array.append(np.mean(sum((Yhat-Y).T*(Yhat-Y).T)))

    def stable_softmax(self, X):
        exp_norm = np.exp(X - np.max(X))
        return exp_norm / np.sum(exp_norm, axis=1).reshape((-1,1))

    def plot_error(self):
        plt.plot(range(len(self.error_plot)), self.error_plot)
        plt.title("Mean Squared Error")
        plt.xlabel("Average MSE per %d training"%(self.MSE_freq))
        plt.ylabel("Percent")
        plt.show()

    def write_network_values(self, filename):
        pickle.dump(self, open(filename, "we"))
        print("Network written to: %s" %(filename))

    def validate_results(self, Yhat, Y):
        Yhat_enc = (np.arange(Y.shape[1]) == Yhat[:, None]).astype(float)
```

```
        num_err = np.sum(abs(Yhat_enc - Y))/2
        print("%d Mistakes. Training Accuracy: %.2f%%"%(int(num_err),
            (len(Yhat)-num_err)/len(Yhat)*100))

    def set_initial_conditions(self):
        # self.layers[0].W[0,:] = [0.15,0.2,0.35]
        # self.layers[0].W[1,:] = [0.25,0.3,0.35]
        # self.layers[0].W[2,:] = [0.25,0.3,0.35]

        self.layers[0].W[0,:] = [0.1,0.1,0.01]
        self.layers[0].W[1,:] = [0.2,0.2,0.1 ]
        self.layers[0].W[2,:] = [0.3,0.3,0.1 ]

class layer:
    def __init__(self,num_inputs,num_neurons, activation):
        self.num_neurons = num_neurons
        self.num_inputs = num_inputs
        self.num_outputs = None
        self.weight_der = None
        self.activation = activation
        self.net = None
        self.W = np.random.uniform(0,1,[num_neurons,num_inputs+1])
        self.momentum_matrix = np.zeros([num_neurons,num_inputs+1])
        self.output = None


    def forward(self, X):
        self.weight_der = X
        self.net = np.dot(X, self.W.T)
        self.output = self.activation.function(self.net)
        return self.output

    def der(self, X):
        return self.activation.derivative(X)

    def set_initial_conditions(self):
        print("test")

class activation_function:
    def __init__(self,function,derivative):
        self.function = function
        self.derivative = derivative

    def function(self,x):
        return self.function(x)

    def derivative(self,x):
        return self.derivative(x)

def get_moon_class_data():
    data = np.loadtxt("./data/classasgntrain1.dat",dtype=float)
    x0 = data[:,0:2]
    x1 = data[:,2:4]
    data = data_frame(x0,x1)
    return data.xtot,data.class_tot

def get_moon_gendata():
    x0 = gendata2(0,10000)
    x1 = gendata2(1,10000)
    data = data_frame(x0,x1)
    return data.xtot, data.class_tot

def get_classasgn_80_20():
    data = np.loadtxt("./data/classasgntrain1.dat",dtype=float)
    x0 = data[:,0:2]
    x1 = data[:,2:4]
    data = data_frame(x0,x1)
    return data.train_tot,data.train_class_tot,data.test_data,data.test_class_tot

class data_frame:
    def __init__(self, data0, data1):
        self.x0 = data0
        self.x1 = data1
```

```
        self.xtot = np.r_[self.x0,self.x1]
        self.N0 = self.x0.shape[0]
        self.N1 = self.x1.shape[0]
        self.N = self.N0 + self.N1
        self.xlim = [np.min(self.xtot[:,0]),np.max(self.xtot[:,0])]
        self.ylim = [np.min(self.xtot[:,1]),np.max(self.xtot[:,1])]
        class_x0 = np.c_[np.zeros([self.N0,1]),np.ones([self.N0,1])]
        class_x1 = np.c_[np.ones([self.N1,1]),np.zeros([self.N1,1])]
        self.class_tot = np.r_[class_x0,class_x1]
        self.y = np.r_[np.ones([self.N0,1]),np.zeros([self.N1,1])]

        # create a training set from the classasgntrain1.dat
        self.train_x0 = data0[0:80]
        self.train_x1 = data1[0:80]
        self.train_tot = np.r_[data0[0:80],data1[0:80]]
        self.train_class_tot = np.r_[self.class_tot[0:80],self.class_tot[100:180]]
        self.test_data = np.r_[data0[80:100],data1[80:100]]
        self.test_class_tot = np.r_[self.class_tot[80:100],self.class_tot[180:200]]

def plot_data(x0,x1):
    xtot = np.r_[x0,x1]
    xlim = [np.min(xtot[:,0]),np.max(xtot[:,0])]
    ylim = [np.min(xtot[:,1]),np.max(xtot[:,1])]

    fig = plt.figure() # make handle to save plot
    plt.scatter(x0[:,0],x0[:,1],c='red',label='$x_0$')
    plt.scatter(x1[:,0],x1[:,1],c='blue',label='$x_1$')
    plt.xlabel('X Coordinate')
    plt.ylabel('Y Coordinate')
    plt.title("Neural Network 2 Class Boundary")
    plt.legend()

def plot_boundaries(xlim, ylim, equation):
    xp1 = np.linspace(xlim[0],xlim[1], num=100)
    yp1 = np.linspace(ylim[0],ylim[1], num=100)

    red_pts = np.array([[],[]])
    blue_pts= np.array([[],[]])
    for x in xp1:
        for y in yp1:
            point = np.array([x,y]).reshape(1,2)
            prob = equation(point)
            if prob == 0:
                blue_pts = np.c_[blue_pts,[x,y]]
            else:
                red_pts = np.c_[red_pts,[x,y]]

    plt.scatter(blue_pts[0,:],blue_pts[1,:],color='blue',s=0.25)
    plt.scatter(red_pts[0,:],red_pts[1,:],color='red',s=0.25)
    plt.xlim(xlim)
    plt.ylim(ylim)
    plt.show()

def sigmoid_func(x):
    return 1/(1+np.exp(-x))

def sigmoid_der(x):
    return (x*(1-x))

def return_value(X):
    return X

def relu_func(X):
    return np.maximum(0,X)

def relu_der(X):
    X[X<0]=0
    return X

def stable_softmax_func(x):
    shiftx = x - np.max(x)
    exps = np.exp(shiftx)
    return exps / np.sum(exps)
```

```python

def softmax_func(x):
    exps = np.exp(x)
    return exps / np.sum(exps)

def gendata2(class_type,N):
    m0 = np.array(
         [[-0.132,0.320,1.672,2.230,1.217,-0.819,3.629,0.8210,1.808, 0.1700],
          [-0.711,-1.726,0.139,1.151,-0.373,-1.573,-0.243,-0.5220,-0.511,0.5330]])

    m1 = np.array(
          [[-1.169,0.813,-0.859,-0.608,-0.832,2.015,0.173,1.432,0.743,1.0328],
          [ 2.065,2.441,0.247,1.806,1.286,0.928,1.923,0.1299,1.847,-0.052]])

    x = np.array([[],[]])
    for i in range(N):
        idx = np.random.randint(10)
        if class_type == 0:
            m = m0[:,idx]
        elif class_type == 1:
            m = m1[:,idx]
        else:
            print("not a proper classifier")
            return 0
        x = np.c_[x, [[m[0]],[m[1]]] + np.random.randn(2,1)/np.sqrt(5)]
    return x.T

def get_ordered_digits(X_train):
    ordered = [
            X_train[7] , # 0
            X_train[4] , # 1
            X_train[16], # 2
            X_train[1] , # 3
            X_train[2] , # 4
            X_train[27], # 5
            X_train[3] , # 6
            X_train[14], # 7
            X_train[5] , # 8
            X_train[8] , # 9
            ]
    return ordered

def print_digits(X,ordered,m,n):
    f, ax = plt.subplots(m,n)
    ordered = get_ordered(X);
    for i in range(m):
        for j in range(n):
            ordered[i*n+j] = ordered[i*n+j].reshape(28,28)
            ax[i][j].imshow(ordered[i*n+j], cmap = plt.cm.binary, interpolation="nearest")
            ax[i][j].axis("off")

    plt.show()

if __name__ == '__main__':
  main()
```