Yusuf Taha Sarı

# EEE 443 - Neural Networks

# Mini Project 1 – Digit Recognition

## Introduction

This mini project is aimed to create a small neural network dedicated to recognize 28x28 pixel handwritten digits. MNIST data base is used to train and test the neural network. Constructed neural network had 1 hidden layer as constant and had different versions with different neuron counts, learning coefficients and activation functions. All of these different cases are trained and their differences and errors are interpreted in this report. The code is also added to the report with comments on it.

## Setup

MINST data is downloaded and it is read in the program by special code. The all-individual data parts are flattened and put in a one matrix which has (batch size, 784) shape. Matrix had 784 columns because 28x28 pixel code had 784 individual pixels when it is flattened.

```
def load_mnist_data():

    mndata = MNIST('')
    mndata.gz = True

    images, labels = mndata.load_training()

    test_images, test_labels = mndata.load_testing()

    return np.array(images), np.array(labels), np.array(test_images), np.array(test_labels)
```

```
Train_input,Train_label,Test_input,Test_label= load_mnist_data()
```

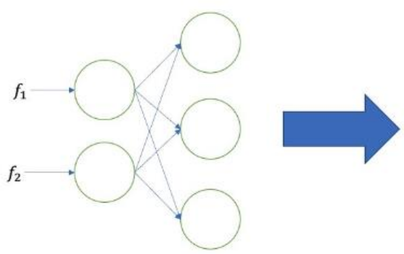
*Figure 1: Getting MNIST Data*

After that, I class called Neural Network is created. First function of this class was the initialize weights function that randomly assign weights to network between (-0.01, 0.01). This function also saved number of hidden layers and learning coefficient to class.

```python
def initiliaze_weights(self,neuron_number_hidden_layer,Learn_coef):
    self.learningCoef = Learn_coef
    self.hiddenLayerNum = neuron_number_hidden_layer
    self.Hidden_layer_weight = np.random.uniform(-0.01, 0.01, size=(neuron_number_hidden_layer, 784))
    self.Output_layer_weight = np.random.uniform(-0.01, 0.01, size=(10, neuron_number_hidden_layer))
    self.hiddenLayerNum = neuron_number_hidden_layer
    return
```

*Figure 2: Initializing Weights*

After getting the data and creating random weights, I finally started forward propagation. To forward propagate I had to use matrix multiplication of weights and input data. It is logic is given in Figure 3.




$$\begin{bmatrix} 23 & 2 \\ 41 & 19 \\ 39 & 14 \\ 26 & 4 \end{bmatrix} \times \begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \end{bmatrix}$$

*Figure 3: Forward Propagation Logic*

In forward propagation process, results are divided to 255 according to the assignment sheet. This process is called normalizing.

```python
def forward_pass_case1(self,image):
    z_1 = np.dot(self.Hidden_layer_weight,image)
    o_1 = self.Tanh_activation(z_1)

    z_2 = np.dot(self.Output_layer_weight,o_1)
    o_2 = self.Tanh_activation(z_2)

    return o_1,o_2

def forward_pass_case2(self,image):
    z_1 = np.dot(self.Hidden_layer_weight,image)
    o_1 = self.relu_activation(z_1)

    z_2 = np.dot(self.Output_layer_weight,o_1)
    o_2 = self.sigmoid_activation(z_2)

    return o_1,o_2
```

*Figure 4: Forward Propagation Code*

As it seen there is are 2 different forward pass functions due to 2 cases asked in assignment sheet. There are 3 different functions to use in activation part of the neurons. These are tanh, relu, sigmoid.

After this code bit, everything called Case 1 are related to tanh activation function case and everything called Case 2 are related to Relu and sigmoid activation functions.

```
def relu_activation(self,x):
    return np.maximum(x, 0)

def relu_activation_derivative(self,x):
    x[x>=0]=1
    x[x<0]=0
    return x

def sigmoid_activation(self,x):
    y = expit(x)
    return y

def sigmoid_activation_derivative(self,x):
    return x-x*x

def Tanh_activation(self,x):
    return np.tanh(x)

def Tanh_activation_derivative(self,x):
    return (1/2)*(1-x*x)
```

*Figure 5: Activation Functions*

These activation functions and their derivatives which will be useful in the back propagation part are written according to their definitions.

After completing forward propagation, some calculations about backward propagation is done on paper to ensure that logic will be correct.

$$\frac{\partial C_0}{\partial w_{jk}^{(L)}} = \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial C_0}{\partial a_j^{(L)}}$$

*Figure 6: Activation Functions*

The backpropagation algorithm works on calculating how sensitive the change in output is to the change in input. To calculate that sensitivity, we have to calculate derivative at the output for input. In

Figure 6, this process is broken into 3 parts. After doing first part to do second part I had to know my output and desired output. To get desired output, functions called desired_tanh and desired_sigmoid are created.

```
def desired_tanh(self,label):
    d = -np.ones((10,1))
    d[label] = 1
    return d

def desired_sigmoid(self,label):
    d = np.zeros((10,1))
    d[label] = 1
    return d
```

*Figure 7: Desired Output*

These functions create (10,1) array. A simple error calculator function is created.

```
def error_cal(self,out,label):
    e = label - out
    return e
```

*Figure 8: Error Calculator*

After all the parts, finally I could calculate the gradient error. In gradient calculation, I have united functions and calculated gradient error. Again, there are two different functions to use it in two different cases.

```
def calculate_output_gradient_case1(self,error,o_2):
    derivative =  np.eye(10) * self.Tanh_activation_derivative(o_2)
    gradient = np.dot(derivative,error)

    return np.reshape(gradient,(10,1))

def calculate_output_gradient_case2(self,error,o_2):
    derivative =  np.eye(10) * self.sigmoid_activation_derivative(o_2)
    gradient = np.dot(derivative,error)

    return np.reshape(gradient,(10,1))
```

*Figure 9: First Gradient Calculation*

I also had another gradient calculation function due to neural network having to layers. One is hidden and two is output.

```python
def calculate_first_gradient_case1(self,o_1,output_gradient):
    derivative = np.eye(self.hiddenLayerNum) * self.Tanh_activation_derivative(o_1)

    gradient = np.dot( np.dot( derivative ,np.transpose(self.Output_layer_weight)), output_gradient)

    return np.reshape(gradient,(self.hiddenLayerNum,1))


def calculate_first_gradient_case2(self,o_1,output_gradient):
    derivative = np.eye(self.hiddenLayerNum) * self.relu_activation_derivative(o_1)

    gradient = np.dot( np.dot( derivative ,np.transpose(self.Output_layer_weight)), output_gradient)

    return np.reshape(gradient,(self.hiddenLayerNum,1))
```

*Figure 10: Second Gradient Calculation*

After these functions, I could finally get everything together and create a backpropagation function with two different versions.

```python
def backpropagation_case1(self,inputs, labels):
    o_1,o_2 = self.forward_pass_case1(inputs)
    desired = self.desired_tanh(labels)
    error = self.error_cal(o_2,desired)
    output_gradients = self.calculate_output_gradient_case1(error,o_2)
    self.Output_layer_weight = self.Output_layer_weight + self.learningCoef * np.dot(output_gradients, o_1.T)
    input_gradients = self.calculate_first_gradient_case1(o_1,output_gradients)
    self.Hidden_layer_weight = self.Hidden_layer_weight + self.learningCoef * np.dot(input_gradients, inputs.T)

    return error
```

*Figure 11: Backpropagation for Case 1*

```python
def backpropagation_case2(self,inputs, labels):
    o_1,o_2 = self.forward_pass_case2(inputs)
    desired = self.desired_sigmoid(labels)
    error = self.error_cal(o_2,desired)
    output_gradients = self.calculate_output_gradient_case2(error,o_2)
    self.Output_layer_weight = self.Output_layer_weight + self.learningCoef * np.dot(output_gradients, o_1.T)
    input_gradients = self.calculate_first_gradient_case2(o_1,output_gradients)
    self.Hidden_layer_weight = self.Hidden_layer_weight + self.learningCoef * np.dot(input_gradients, inputs.T)

    return error
```

*Figure 12: Backpropagation for Case 2*

After that I created two big functions which are train and test. Train does the training and test does the testing.

```
def train(self,images,labels,batch_size,epochs):
    startTime = time.time()

    train_Err = 0.0

    for epoch in range(epochs):
        cum_Err = 0.0
        for i in range(batch_size):
            data = np.reshape(images[i], (784, 1)) / 255
            err = self.backpropagation_case1(data, labels[i])
            cum_Err += np.sum(err**2 * 0.5)

        epoch_err_mean = cum_Err / batch_size
        train_Err += epoch_err_mean
        print("One Epoch Finished: ", epoch + 1,"\nMean Squares Err of Epoch: " , epoch_err_mean)

    train_Err_mean = train_Err / 50
    stopTime = time.time()
    print("Training Finished!\nCPU Time: ", stopTime-startTime, " seconds","\nOverall Err:",train_Err_mean)
```

*Figure 13: Training for Case 1*

```
def train_2(self,images,labels,batch_size,epochs):
    startTime = time.time()

    train_Err = 0.0

    for epoch in range(epochs):
        cum_Err = 0.0
        for i in range(batch_size):
            data = np.reshape(images[i], (784, 1)) / 255
            err = self.backpropagation_case2(data, labels[i])
            cum_Err += np.sum(err**2 * 0.5)

        epoch_err_mean = cum_Err / batch_size
        train_Err += epoch_err_mean
        print("One Epoch Finished: ", epoch + 1,"\nMean Squares Err of Epoch: " , epoch_err_mean)

    train_Err_mean = train_Err / 50
    stopTime = time.time()
    print("Training Finished!\nCPU Time: ", stopTime-startTime, " seconds","\nOverall Err:",train_Err_mean)
```

*Figure 14: Training for Case 2*

Then two testing functions are coded.

```
def evaluate_accuracy(self,images,labels):
    cum_err = 0.0
    miss_calc = 0

    for i in range(10000):
        testData = np.reshape(images[i], (784, 1)) / 255
        o1, output = self.forward_pass_case1(testData)

        d = self.desired_tanh(labels[i])
        err = self.error_cal(output,d)
        cum_err += np.sum(err**2 * 0.5)

        predicted = np.argmax(output)

        if labels[i] != predicted:
            miss_calc += 1

    misclasificationPercentage = miss_calc / 10000 * 100

    testSetErrorMean = cum_err / 10000

    print("Number Of Misclasification:",miss_calc,"\nError Percentage: %",misclasificationPercentage,"\nTest Data Mean Square
```

*Figure 15: Testing for Case 1*

```python
def evaluate_accuracy_2(self,images,labels):
    cum_err = 0.0
    miss_calc = 0

    for i in range(10000):
        testData = np.reshape(images[i], (784, 1)) / 255
        o1, output = self.forward_pass_case2(testData)

        d = self.desired_tanh(labels[i])
        err = self.error_cal(output,d)
        cum_err += np.sum(err**2 * 0.5)

        predicted = np.argmax(output)

        if labels[i] != predicted:
            miss_calc += 1

    misclasificationPercentage = miss_calc / 10000 * 100

    testSetErrorMean = cum_err / 10000

    print("Number Of Misclasification:",miss_calc,"\nError Percentage: %",misclasificationPercentage,"\nTest Data Mean Squar
```

*Figure 16: Testing for Case 2*

**Results**

General Function is created.

```python
def Train_and_Test(hidden_layer, learning_coef, batch_size, epochs, Case_selection):
    if Case_selection == 1:
        Case1.initiliaze_weights(hidden_layer, learning_coef)
        Case1.train(Train_input,Train_label,batch_size,epochs)
        Case1.evaluate_accuracy(Test_input,Test_label)
    if Case_selection == 2:
        Case2.initiliaze_weights(hidden_layer, learning_coef)
        Case2.train2(Train_input,Train_label,batch_size,epochs)
        Case2.evaluate_accuracy2(Test_input,Test_label)
```

*Figure 17: Train and Test Function*

18 different functions are written to get results.

```python
Train_and_Evaluate(300, 0.01, 1250, 5, 1)
```

```python
Train_and_Evaluate(500, 0.01, 1250, 5, 1)
```

```python
Train_and_Evaluate(1000, 0.01, 1250, 5, 1)
```

*Figure 18: First 3 of Functions*

Results are shown in Tables.

| Case 1 | Epoch | Training MSE | Test MSE | Error | Time |
|---|---|---|---|---|---|
| **N = 300 η = 0.01** | 50 | 0.267857858 | 0.434667889 | % 12.78 | 168 |
| **N = 300 η = 0.05** | 50 | 0.198373763 | 0.416735657 | % 12.13 | 150 |
| **N = 300 η = 0.09** | 50 | 0.185242354 | 0.457794567 | % 11.74 | 170 |
| **N = 500 η = 0.01** | 50 | 0.252355676 | 0.525676467 | % 13.8 | 320 |
| **N = 500 η = 0.05** | 50 | 0.204673563 | 0.412057694 | % 12.36 | 305 |
| **N = 500 η = 0.09** | 50 | 0.324556678 | 0.414668457 | % 11.82 | 326 |
| **N = 1000 η = 0.01** | 50 | 0.244574567 | 0.530475931 | % 13.47 | 890 |
| **N = 1000 η = 0.05** | 50 | 0.310483765 | 0.5656537493651452 | % 14.77 | 932 |
| **N = 1000 η = 0.09** | 50 | 0.753041303 | 0.6534311881593174 | % 15.71 | 989 |

For Best Case Scenario I used 60 000 data.

| Best Case Tanh | Epoch | Training MSE | Test MSE | Error | Time (s) |
|---|---|---|---|---|---|
| **N = 300 η = 0.01** | 50 | 0.051723562 | 0.078754225 | % 2.21 | 7276 |
| **N = 300 η = 0.05** | 50 | 0.096372167 | 0.110105726 | % 2.37 | 7157 |
| **N = 300 η = 0.09** | 50 | 0.216583786 | 0.181366838 | % 3.97 | 7029 |

Case 2 Results

| Case 2 | Epoch | Training MSE | Test MSE | Error | Time |
|---|---|---|---|---|---|
| **N = 300 η = 0.01** | 50 | 0.112453453 | 0.156824573 | % 12.12 | 141 |
| **N = 300 η = 0.05** | 50 | 0.041252146 | 0.112434656 | % 11.5 | 134 |
| **N = 300 η = 0.09** | 50 | 0.038252151 | 0.113534646 | % 14.6 | 127 |
| **N = 500 η = 0.01** | 50 | 0.111246432 | 0.144536342 | % 12.7 | 291 |
| **N = 500 η = 0.05** | 50 | 0.047124564 | 0.125656342 | % 11.5 | 277 |
| **N = 500 η = 0.09** | 50 | 0.035714612 | 0.094353462 | % 12.5 | 304 |
| **N = 1000 η = 0.01** | 50 | 0.121254634 | 0.11344253 | % 12.6 | 982 |
| **N = 1000 η = 0.05** | 50 | 0.041245546 | 0.134634462 | % 11.7 | 917 |
| **N = 1000 η = 0.09** | 50 | 0.033463136 | 0.112435466 | % 12.9 | 975 |

For Best Case Scenario (Sigmoid) I couldn't get data because there was not enough time.

Mini Batch Results.

| Mini Batch | Epoch | Training MSE | Test MSE | Error | Time |
|---|---|---|---|---|---|
| **N=10** | 50 | 0.567486587 | 0.482637744894622 | % 84.78 | 8 |
| **N = 50** | 50 | 0.442466846 | 0.4340819066870133 | % 78.15 | 35 |
| **N = 100** | 50 | 0.335146776 | 0.29650959100868635 | % 38.53 | 76 |

**Conclusion**

In this project, I have created a neural network and trained it with data to get right weights. After that I have tested my results and see results. This project helped me understand the basics of neural network construction. Also increased my speed which will be helpful in Term Project.

**Appendix:**

```
from mnist import MNIST
import numpy as np
from scipy.special import expit
import time


def load_mnist_data():

    mndata = MNIST('')
    mndata.gz = True

    images, labels = mndata.load_training()

    test_images, test_labels = mndata.load_testing()

    return np.array(images), np.array(labels), np.array(test_images), np.array(test_labels)


Train_input,Train_label,Test_input,Test_label= load_mnist_data()
```

```
class DigitRecognition:



    def initiliaze_weights(self,hiddenLayerNum,learningCoef):
        self.Hidden_layer_weight = np.random.uniform(-0.01, 0.01, size=(hiddenLayerNum, 784))
        self.Output_layer_weight = np.random.uniform(-0.01, 0.01, size=(10, hiddenLayerNum))
        self.learningCoef = learningCoef
        self.hiddenLayerNum = hiddenLayerNum
        return

    def train(self,images,labels,batch_size,epochs):
        startTime = time.time()

        trainingError = 0.0

        for epoch in range(epochs):
            cumulativeError = 0.0
            for i in range(batch_size):
                data = np.reshape(images[i], (784, 1)) / 255
                error = self.backpropagation_case1(data, labels[i])
                cumulativeError += np.sum(error**2 * 0.5)

            epochErrorMean = cumulativeError / batch_size
            trainingError += epochErrorMean
            print("Epoch Completed: ", epoch + 1,"\nMean Squares Error of Epoch: " , epochErrorMean)
```

```
        trainingErrorMean = trainingError / 50
        stopTime = time.time()
        print("Training Completed!\nCPU Time: ", stopTime-startTime, " seconds","\nOverall 
Error:",trainingErrorMean)

    def train2(self,images,labels,batch_size,epochs):
        startTime = time.time()

        trainingError = 0.0

        for epoch in range(epochs):
            cumulativeError = 0.0
            for i in range(batch_size):
                data = np.reshape(images[i], (784, 1)) / 255
                error = self.backpropagation_case2(data, labels[i])
                cumulativeError += np.sum(error**2 * 0.5)

            epochErrorMean = cumulativeError / batch_size
            trainingError += epochErrorMean
            print("Epoch Completed: ", epoch + 1,"\nMean Squares Error of Epoch: " , epochErrorMean)

        trainingErrorMean = trainingError / 50
        stopTime = time.time()
        print("Training Completed!\nCPU Time: ", stopTime-startTime, " seconds","\nOverall 
Error:",trainingErrorMean)


    def forward_pass_case1(self,image):
        # Forward Hidden Layer
```

```
    z_1 = np.dot(self.Hidden_layer_weight,image)
    o_1 = self.Tanh_activation(z_1) # For Case 1
    #o1 = self.activation(v1,self.Relu) # For Case 2

    # Forward Output Layer
    z_2 = np.dot(self.Output_layer_weight,o_1)
    o_2 = self.Tanh_activation(z_2) # For Case 1
    #o2 = self.activation(v2,self.Relu) # For Case 2

    return o_1,o_2

def forward_pass_case2(self,image):
    # Forward Hidden Layer
    z_1 = np.dot(self.Hidden_layer_weight,image)
    o_1 = self.relu_activation(z_1) # For Case 1
    #o1 = self.activation(v1,self.Relu) # For Case 2

    # Forward Output Layer
    z_2 = np.dot(self.Output_layer_weight,o_1)
    o_2 = self.sigmoid_activation(z_2) # For Case 1
    #o2 = self.activation(v2,self.Relu) # For Case 2

    return o_1,o_2



def relu_activation(self,x):

    return np.maximum(x, 0)
```

```
def relu_activation_derivative(self,x):
    x[x>=0]=1
    x[x<0]=0
    return x



def sigmoid_activation(self,x):
    y = expit(x)
    return y


def sigmoid_activation_derivative(self,x):
    return x-x*x


def Tanh_activation(self,x):
    return np.tanh(x)


def Tanh_activation_derivative(self,x):
    return (1/2)*(1-x*x)



def desired_tanh(self,label):
    d = -np.ones((10,1))
    d[label] = 1
    return d


def desired_sigmoid(self,label):
    d = np.zeros((10,1))
    d[label] = 1
    return d
```

```
def error_cal(self,prediction,desired):
    e = desired - prediction
    return e


def calculate_output_gradient_case1(self,error,o_2):
    derivative =  np.eye(10) * self.Tanh_activation_derivative(o_2)
    gradient = np.dot(derivative,error)

    return np.reshape(gradient,(10,1))


def calculate_output_gradient_case2(self,error,o_2):
    derivative =  np.eye(10) * self.sigmoid_activation_derivative(o_2)
    gradient = np.dot(derivative,error)

    return np.reshape(gradient,(10,1))



def calculate_first_gradient_case1(self,o_1,output_gradient):
    derivative = np.eye(self.hiddenLayerNum) * self.Tanh_activation_derivative(o_1)

    gradient = np.dot( np.dot( derivative ,np.transpose(self.Output_layer_weight)), output_gradient)

    return np.reshape(gradient,(self.hiddenLayerNum,1))



def calculate_first_gradient_case2(self,o_1,output_gradient):
    derivative = np.eye(self.hiddenLayerNum) * self.relu_activation_derivative(o_1)
```

```
        gradient = np.dot( np.dot( derivative ,np.transpose(self.Output_layer_weight)), output_gradient)

        return np.reshape(gradient,(self.hiddenLayerNum,1))




    def evaluate_accuracy(self,images,labels):
        cumulativeError = 0.0
        numOfMissClassification = 0

        for i in range(10000):
            testData = np.reshape(images[i], (784, 1)) / 255
            o1, output = self.forward_pass_case1(testData)

            d = self.desired_tanh(labels[i])
            error = self.error_cal(output,d)
            cumulativeError += np.sum(error**2 * 0.5)

            predicted = np.argmax(output)

            if labels[i] != predicted:
                numOfMissClassification += 1

        misclasificationPercentage = numOfMissClassification / 10000 * 100

        testSetErrorMean = cumulativeError / 10000

        print("Number Of Misclasification:",numOfMissClassification,"\nError Percentage: 
%",misclasificationPercentage,"\nTest Data Mean Square Error:",testSetErrorMean)
```

```
    def evaluate_accuracy2(self,images,labels):
        cumulativeError = 0.0
        numOfMissClassification = 0

        for i in range(10000):
            testData = np.reshape(images[i], (784, 1)) / 255
            o1, output = self.forward_pass_case2(testData)

            d = self.desired_sigmoid(labels[i])
            error = self.error_cal(output,d)
            cumulativeError += np.sum(error**2 * 0.5)

            predicted = np.argmax(output)

            if labels[i] != predicted:
                numOfMissClassification += 1

        misclasificationPercentage = numOfMissClassification / 10000 * 100

        testSetErrorMean = cumulativeError / 10000

        print("Number Of Misclasification:",numOfMissClassification,"\nError Percentage:
%",misclasificationPercentage,"\nTest Data Mean Square Error:",testSetErrorMean)




    def backpropagation_case1(self,inputs, labels):
```

```
        o_1,o_2 = self.forward_pass_case1(inputs)
        desired = self.desired_tanh(labels)
        error = self.error_cal(o_2,desired)
        output_gradients = self.calculate_output_gradient_case1(error,o_2)
        self.Output_layer_weight = self.Output_layer_weight + self.learningCoef * np.dot(output_gradients,
o_1.T)
        input_gradients = self.calculate_first_gradient_case1(o_1,output_gradients)
        self.Hidden_layer_weight = self.Hidden_layer_weight + self.learningCoef * np.dot(input_gradients,
inputs.T)

        return error

    def backpropagation_case2(self,inputs, labels):
        o_1,o_2 = self.forward_pass_case2(inputs)
        desired = self.desired_sigmoid(labels)
        error = self.error_cal(o_2,desired)
        output_gradients = self.calculate_output_gradient_case2(error,o_2)
        self.Output_layer_weight = self.Output_layer_weight + self.learningCoef * np.dot(output_gradients,
o_1.T)
        input_gradients = self.calculate_first_gradient_case2(o_1,output_gradients)
        self.Hidden_layer_weight = self.Hidden_layer_weight + self.learningCoef * np.dot(input_gradients,
inputs.T)

        return error


def Train_and_Test(hidden_layer, learning_coef, batch_size, epochs, Case_selection):
    Case2=DigitRecognition()
    Case1=DigitRecognition()
```

```
    if Case_selection == 1:
        Case1.initiliaze_weights(hidden_layer, learning_coef)
        Case1.train(Train_input,Train_label,batch_size,epochs)
        Case1.evaluate_accuracy(Test_input,Test_label)
    if Case_selection == 2:
        Case2.initiliaze_weights(hidden_layer, learning_coef)
        Case2.train2(Train_input,Train_label,batch_size,epochs)
        Case2.evaluate_accuracy2(Test_input,Test_label)


# In[72]:


Train_and_Evaluate(300, 0.01, 1250, 5, 1)


# In[74]:


Train_and_Test(300, 0.01, 1250, 50, 1)


# In[75]:


Train_and_Test(500, 0.01, 1250, 50, 1)
```

```
# In[76]:

Train_and_Test(1000, 0.01, 1250, 50, 1)

# In[77]:

Train_and_Test(300, 0.05, 1250, 50, 1)

# In[78]:

Train_and_Test(500, 0.05, 1250, 50, 1)

# In[79]:

Train_and_Test(1000, 0.05, 1250, 50, 1)

# In[80]:

Train_and_Test(300, 0.09, 1250, 50, 1)
```

```
# In[81]:

Train_and_Test(500, 0.09, 1250, 50, 1)

# In[82]:

Train_and_Test(1000, 0.09, 1250, 50, 1)

# In[83]:

Train_and_Test(300, 0.01, 1250, 50, 2)

# In[84]:

Train_and_Test(500, 0.01, 1250, 50, 2)

# In[85]:
```

Train_and_Test(1000, 0.01, 1250, 50, 2)

# In[86]:

Train_and_Test(300, 0.05, 1250, 50, 2)

# In[87]:

Train_and_Test(500, 0.05, 1250, 50, 2)

# In[88]:

Train_and_Test(1000, 0.05, 1250, 50, 2)

# In[89]:

Train_and_Test(300, 0.09, 1250, 50, 2)

# In[90]:

```
Train_and_Test(500, 0.09, 1250, 50, 2)


# In[91]:


Train_and_Test(1000, 0.09, 1250, 50, 2)


# In[92]:


print('done')
```