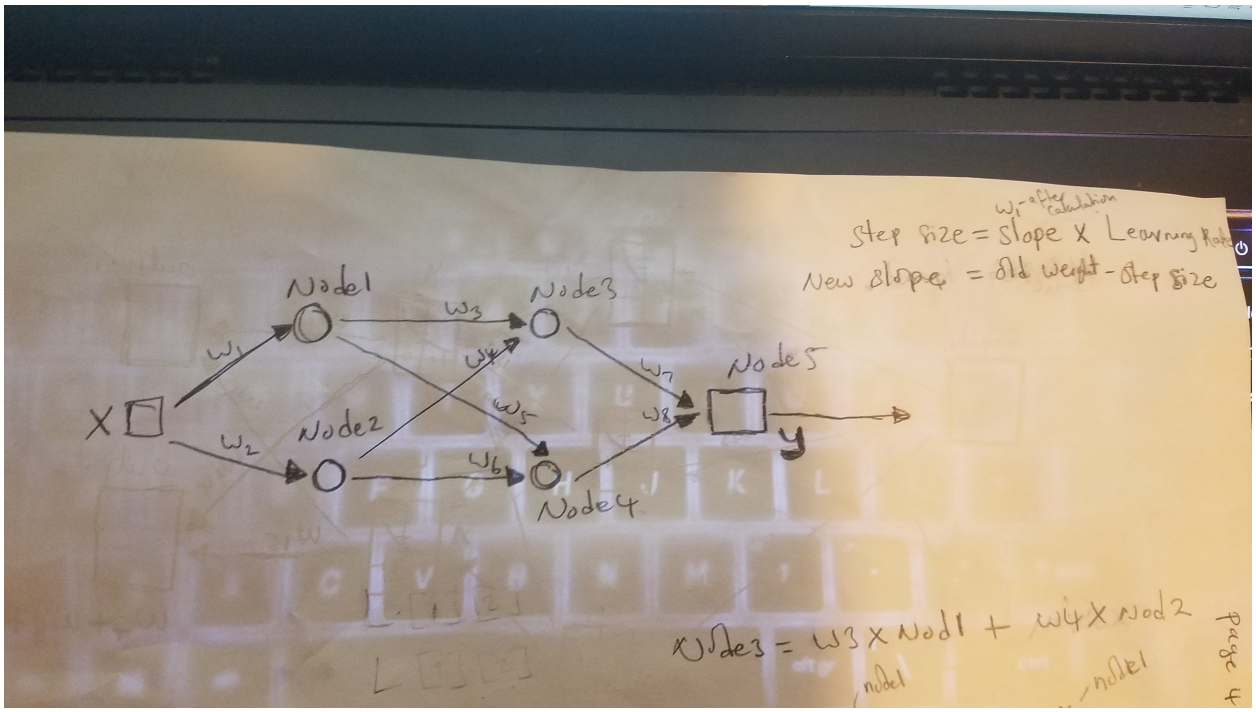
Name: Brandon Eyongherok



1.)
B. The The learning parameter of a fix value is slower than the learning parameter of a time decreasing value, but a fix value learning parameter gives a better approximation of the gradient of the weights and weight biases, so better prediction

C. The network does reasonably well generalizing data beyond its training set. A major reason is because the function is fairly simple.

D. The interpolation capability gives a better generalization of the data, but the extrapolation capability in this data set also gives a good approximation, although not as good as interpolation. This shows that for more complex functions, the farther out you extrapolate the more likely your error increases.

work:

```
import pandas as pd
from PIL import Image
from numpy import asarray
import numpy as np
import math
```

```
import matplotlib.pyplot as plt
import os
import sklearn

graph1 = [0 for i in range(100)]

############################################################
i = 0
X_values = [] * 500
while i <= 100:
      i += 0.2
      X_values.append(i)



count = 0
Y_values = [] * 500
while count <= 499:
    Y_values.append(1/X_values[count])
    count += 1

print('\n')
print('\n')
plt.figure(figsize=(20,10))
plt.xlabel('X_values')
plt.ylabel('Y_values')
#plt.plot(X_values, Y_values)
plt.scatter(X_values, Y_values)
plt.show()

count = 0
Test_Xvalues = [] * 100
Test_Yvalues = [] * 100
Training_Xvalues = [] * 400
Training_Yvalues = [] * 400

while count <= 499:
      if count % 5 == 0:
            Test_Xvalues.append(X_values[count])

            Test_Yvalues.append(Y_values[count])
      else:
        Training_Xvalues.append(X_values[count])
```

```
        Training_Yvalues.append(Y_values[count])

      count += 1

x1 = len(Test_Xvalues)
x2 = len(Test_Yvalues)
x3 = len(Training_Xvalues)
x4 = len(Training_Yvalues)
x5 = len(X_values)



#normalizing the dataset
count = 0
while count < 400:
    Training_Yvalues[count] = Training_Yvalues[count] / 100
    count += 1

count = 0
while count < 100:
    Test_Yvalues[count] = Test_Yvalues[count] / 100
    count += 1

print('\n')
print(x1)
print(x2)
print(x3)
print(x4)
print(x5)

print('\n')
print('\n')
#plt.figure(figsize=(20,10))
plt.xlabel('Training_Xvalues')
plt.ylabel('Training_Yvalues')
plt.scatter(Training_Xvalues, Training_Yvalues)
plt.show()

print('\n')
#plt.figure(figsize=(20,10))
plt.xlabel('Test_Xvalues')
plt.ylabel('Test_Yvalues')
plt.scatter(Test_Xvalues, Test_Yvalues)
plt.show()
```

```
############################################################


#initialize the actual Y value for the output
ob = [1, 2, 5, 8, 11]

# get the maximum value of the array
max_value = np.max(ob)

#normalize the data
count = 0
while count < len(ob):
    ob[count] = ob[count]/max_value
    count += 1

print("print out normalize observe")
print(ob)
#initialize the actual X value for the input
inp = [1, 2, 3, 4, 5]


#initializing random weights
w1 = 0.3
w2 = 0.4
w3 = 0.5
w4 = 0.2
w5 = 0.7
w6 = 0.6
w7 = 0.3
w8 = 0.1

#initialize the bias to be a constant 1
bias = 1

#intitialize the weights of the bias
bw1 = 0.2
bw2 = 0.5
bw3 = 0.7
bw4 = 0.3
bw5 = 0.4
```

```
#this will save the X values at each node
array1 = 1
array2 = 1
array3 = 1
array4 = 1
array5 = 1



#this will save the Y values at each node
Node1 = 1
Node2 = 1
Node3 = 1
Node4 = 1
Node5 = 1



#intialize the learning rate
learningrate = 0.03


#initialize the actual Y value for the output
observe = 1

# I know after you have to regularize the data of observe


#initialize the actual X value for the input/ you don't regularize the X input
arr = 1

#Graph the X and Y values
print('\n')
plt.xlabel('Xvalues')
plt.ylabel('Yvalues')
plt.scatter(inp, ob)
plt.show()


index = 0
# Train your data
```

```
epoch = 0
while epoch < 200:
    inp, ob = sklearn.utils.shuffle(Training_Xvalues, Training_Yvalues)
    index = 0

    while index < len(Training_Xvalues):
        arr = Training_Xvalues[index]
        observe = Training_Yvalues[index]


        # Weight1 multiplied by the values of the input array
        array1 = w1 * arr + bias * bw1

        # Weight2 multiplied by the vales of the input array
        array2 = w2 * arr + bias * bw2

        #print(array1)




        # Node 1: takes in the input X and passes it through the sigmoid function then stores it at
the corresponding Y
        Node1 = 1 / (1 + math.exp(-(array1)))

        # Node 2: takes in the input X and passes it through the sigmoid function then stores it at
the correspodning X
        Node2 = 1 / (1 + math.exp(-array2))




        array3 = w3 * Node1 + w4 * Node2 + bias * bw3

        array4 = w5 * Node1 + w6 * Node2 + bias * bw4

        count = 0

        # function
        # Node 3 and 4
        Node3 = 1 / (1 + math.exp(-array3))

        # Node 2
```

```
Node4 = 1 / (1 + math.exp(-array4))




array5 = w7 * Node3 + w8 * Node4 + bias * bw5

# Node 5
Node5 = 1 / (1 + math.exp(-array5))
# The predicted Y values are stores at node 5
predicted = Node5


error = observe - predicted

print(error)



# intiliazing a storage for each weight, which you will use to obtain the new weight in gradient descent
w1old = w1
w2old = w2
w3old = w3
w4old = w4
w5old = w5
w6old = w6
w7old = w7
w8old = w8


oldbw1 = bw1
oldbw2 = bw2
oldbw3 = bw3
oldbw4 = bw4
oldbw5 = bw5




#gradient descent
```

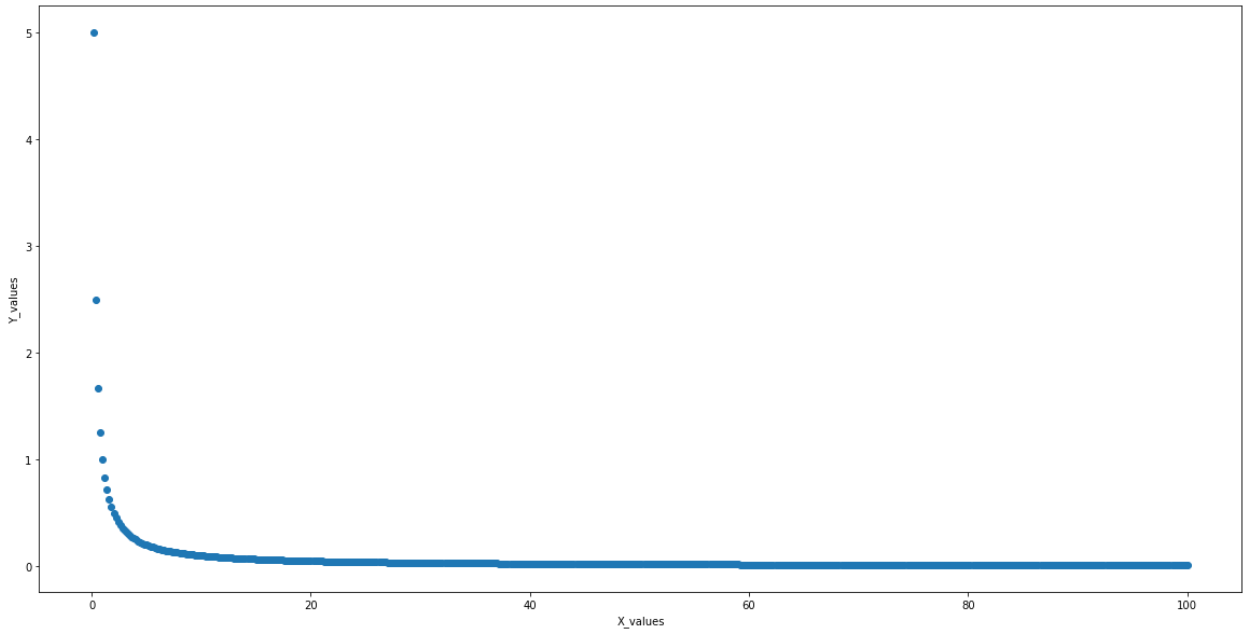
```
# Get the derivative for each node using it's correspoding X values
s   = 1/(1 + math.exp(-array5))
dernode5 = s*(1-s)
L5  = dernode5 * error
bw5 = learningrate * L5 * error
w7  = learningrate * L5 * Node3
w8  = learningrate * L5 * Node4


s   = 1/(1 + math.exp(-array3))
dernode3 = s*(1-s)
L3  = dernode3 * ( L5 * w7)
bw3 = learningrate * L3 * bias
w3  = learningrate * L3 * Node1
w4  = learningrate * L3 * Node2


s   = 1/(1 + math.exp(-array4))
dernode4 = s*(1-s)
L4  = dernode4 * (L5 * w8)
bw4 = learningrate * L4 * bias
w5  = learningrate * L4 * Node1
w6  = learningrate * L4 * Node2


s   = 1/(1 + math.exp(-array1))
dernode1 = s*(1-s)
L1  = dernode1 * (L3 * w3) * (L4 * w5)
bw1 = learningrate * L1 * bias
w1  = learningrate * L1 * arr


s   = 1/(1 + math.exp(-array2))
dernode2 = s*(1-s)
L2  = dernode2 * (L3 * w4) * (L4 * w6)
bw2 = learningrate * L2 * bias
w2  = learningrate * L2 * arr


#Using gradient descent to find the where the slope equals to zero fast for each weight.
w1 = w1 + w1old
w2 = w2 + w2old
w3 = w3 + w3old
w4 = w4 + w4old
```

```
        w5 = w5 + w5old
        w6 = w6 + w6old
        w7 = w7 + w7old
        w8 = w8 + w8old


        bw1 = bw1 + oldbw1
        bw2 = bw2 + oldbw2
        bw3 = bw3 + oldbw3
        bw4 = bw4 + oldbw4
        bw5 = bw5 + oldbw5

        index += 1



    epoch += 1


print("print Test_values length")
print(len(Test_Xvalues))

print("print graph1 length")
print(len(graph1))

index = 0
while index < len(Test_Xvalues):

    arr = Test_Xvalues[index]

    # Weight1 multiplied by the values of the input array
    array1 = w1 * arr + bias * bw1

    # Weight2 multiplied by the vales of the input array
    array2 = w2 * arr + bias * bw2

    #print(array1)




    # Node 1: takes in the input X and passes it through the sigmoid function then stores it at the
corresponding Y
    Node1 = 1 / (1 + math.exp(-(array1)))
```

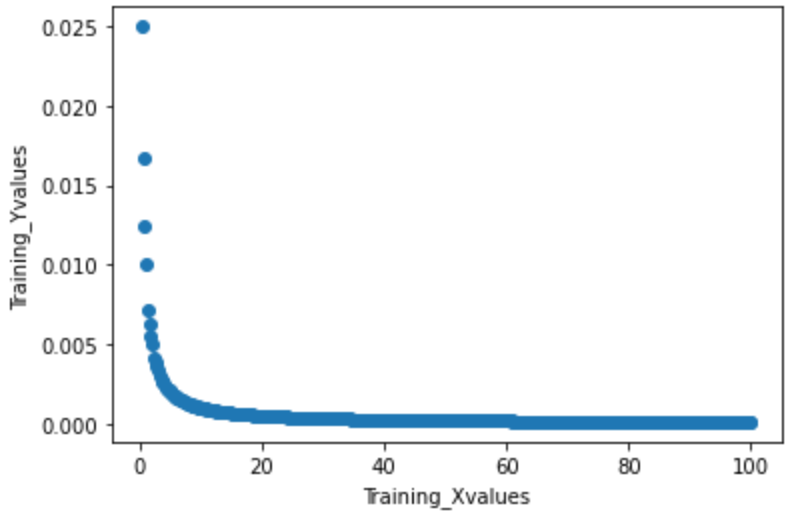
```
    # Node 2: takes in the input X and passes it through the sigmoid function then stores it at the
correspodning X
    Node2 = 1 / (1 + math.exp(-array2))




    array3 = w3 * Node1 + w4 * Node2 + bias * bw3

    array4 = w5 * Node1 + w6 * Node2 + bias * bw4

    count = 0

    # function
    # Node 3 and 4
    Node3 = 1 / (1 + math.exp(-array3))

    # Node 2
    Node4 = 1 / (1 + math.exp(-array4))




    array5 = w7 * Node3 + w8 * Node4 + bias * bw5

    # Node 5
    Node5 = 1 / (1 + math.exp(-array5))
    # The predicted Y values are stores at node 5

    graph1[index] = Node5
    index += 1


#count = 0
#while count < len(graph1):
#    graph1[count] = graph1[count] * max_value
#    count += 1

plt.xlabel('TestedXvalues')
plt.ylabel('TestedYvalues')
```

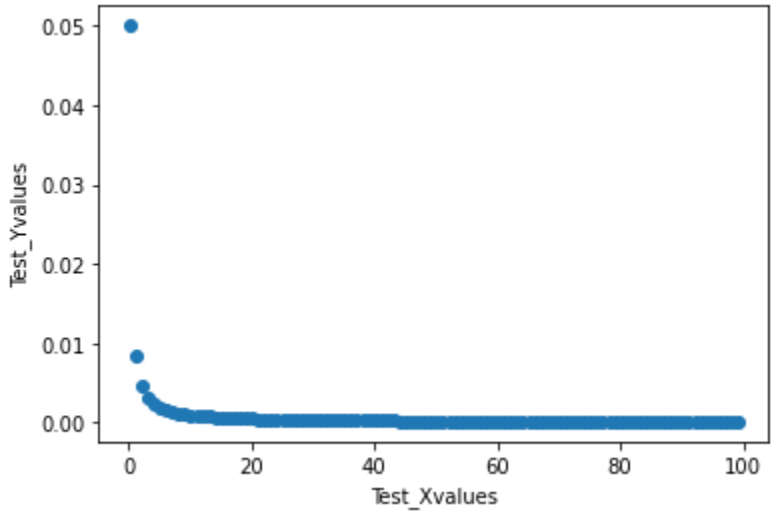
plt.scatter(Test_Xvalues, graph1)

Output:

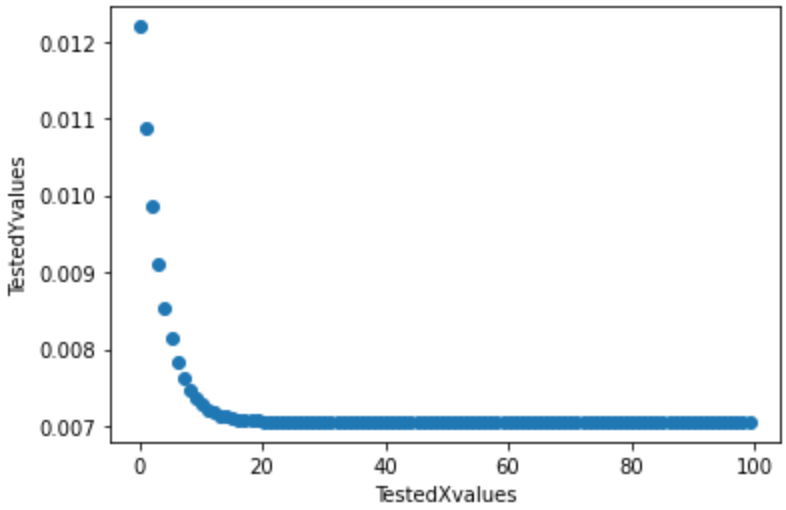


length of input testing
100
length of output testing
100
length of input training
400
length of output training
400
total length of training and testing
500

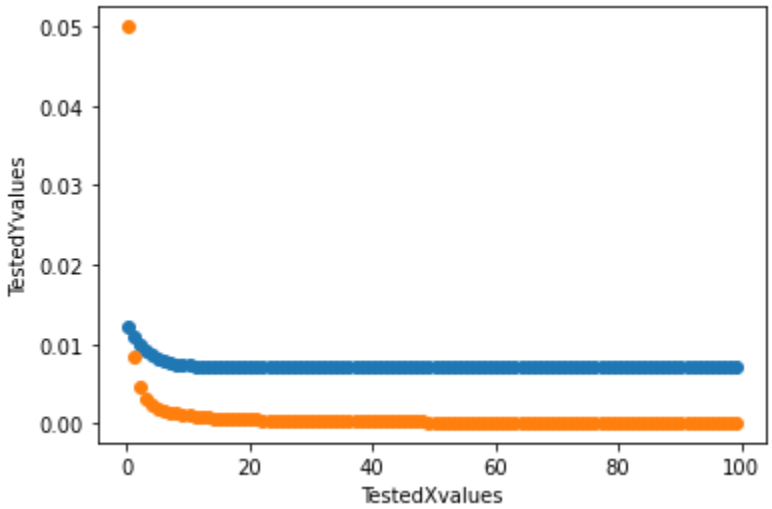




<matplotlib.collections.PathCollection at 0x1dd9f6c9490>



The plot of just the predicted values



The predicted in blue and the actual in orange

print Test_values length
100
print graph1 length
100

Error at the top
-0.6733206546264414
-0.6708089996572396

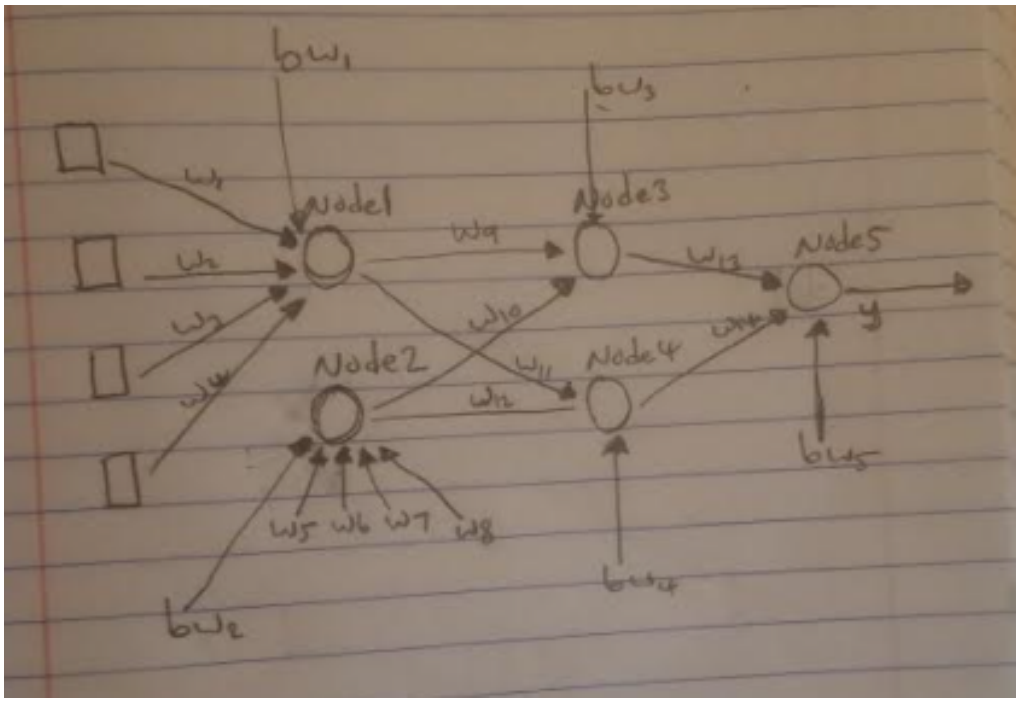
-0.6690108637981329
-0.666641953118757
-0.6646118632245519
-0.662402832268714
-0.6601142316527048
-0.6579235480860569
-0.655600065502836
-0.648501963873197
-0.6510215641879438
-0.6489926364526982
-0.6463206372873845
-0.6443413726180203
-0.6415429480019373
-0.6377636896919884
-0.6369334459924385
-0.6342691032099474
-0.6324606387821298
-0.6301962384801799
-0.6277726619360092
-0.6253467066389389
-0.6225828172436615
-0.6203387195347759
-0.6179816931658288
-0.6152993485168916
-0.6129325278978952
-0.6104075476108335
-0.6079210801587849
-0.6054899437766019
-0.6029019556221241
-0.600019461562999
-0.5974236105148828
-0.5953618106488586
-0.5928329774086407
-0.5902023209696429
-0.5875596101870477
-0.5849797431602638
-0.5821694896652316
-0.5798316164680106
-0.5771579909093824
-0.5745528063622647
-0.5716380848391027
-0.5694074701684381
-0.5667109496123096
-0.5641240921349788
-0.5615183132203864
-0.5588337005671007

-0.5558771209059588
-0.5533212536735271

Error at the bottom:

0.006468044522915264
-0.006775138772943122
-0.006936280333896422
-0.006920350802420777
-0.006941911642073204
-0.006914118363775817
-0.00667284391620771
-0.006851592539592299
-0.006925419290134784
-0.0069507604920109505
-0.006404074212781476
-0.006805519954245833
-0.006263596876429787
-0.006504771451201168
-0.006947837554417374
-0.006113802224176628
-0.006267242399841194
-0.006677984258802022
-0.006234804775869792
-0.006903164530636151
-0.0069414586629109615
-0.006919914466592977
-0.006809926279759139
-0.006947201126851942
-0.006913671350739052
-0.0069155932793998655
-0.006638056826885661
-0.0069272293045657
-0.0011015954503611342
-0.006923145485749968
-0.0062400697988948175

2.)



Learning rate = 0.01
Epoch = 200
3 K-fold
Four inputs and one output

C. It classified the first section the best. The graphs for the first section predicted are similar to the graph of the observed, with the classes alternating between each other. The second section took the weights from the first section and it classified the second the worst. The graphs for the second section are not as clearly alternating. Finally, the last section took the weights from the first and the second section and it classified the last section medium relatively to the other sections. I think overall the deep learning algorithm performs quite well taking into consideration it has only two hidden layers and two neurons in each hidden layer. For further research, I can try adding more layers and more neurons per layer. For example, I can add 5 hidden layers and 4 neurons per layer or I can try using a different activation function, tanh. The training sample was small, only 150, but using the k-fold helped in providing variety to the training so the sample does not memorize the dataset, overfit, but rather properly make good generalizations.

Code:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn import datasets
import numpy as np
import sklearn
```

```
from sklearn.decomposition import PCA

#open iris dataset
iris = datasets.load_iris()
#print(iris)

#print(len(iris))

#X = iris.data[:, :2]  # we only take the first two features.
#print(X)
#print(len(X))




array = iris.data[:, :4]



#print(len(array))

print(array)







#Setosa
group1 = [[0 for i in range(4)] for j in range(50)]

#Vesicolour
group2 = [[0 for i in range(4)] for j in range(50)]

#Virginia
group3 = [[0 for i in range(4)] for j in range(50)]

#Find the maximum value of each column
col0max = 0
col1max = 0
col2max = 0
col3max = 0
row = 0
```

```
col = 0
while row < 150:
    col = 0
    while col < 4:
        if col == 0:
            if col0max < array[row][col]:
                col0max = array[row][col]
        if col == 1:
            if col1max < array[row][col]:
                col1max = array[row][col]
        if col == 2:
            if col2max < array[row][col]:
                col2max = array[row][col]
        if col == 3:
            if col3max < array[row][col]:
                col3max = array[row][col]
        col += 1
    row += 1


print("print maximum value at column 0")
print(col0max)

print("print maximum value at column 1")
print(col1max)




row = 0
col = 0
while row < 150:
    col = 0
    while col < 4:
        if col == 0:

            array[row][col] = (array[row][col])/col0max

        if col == 1:

            array[row][col] = (array[row][col])/col1max

        if col == 2:
```

```
            array[row][col] = (array[row][col])/col2max

        if col == 3:

            array[row][col] = (array[row][col])/col3max

        col += 1
    row += 1

print("print the normalized matrix")
print(array)



# Seperate all the different classes

row1 = 0
row2 = 0

col = 0
row = 0
while row < 150:
    col = 0
    while col < 4:
        if row < 50:
            group1[row][col] = array[row][col]

        elif 50 <= row and row < 100:
            group2[row1][col] = array[row][col]

        else:
            group3[row2][col] = array[row][col]
        col += 1

    if row < 50:
        row1 = 0
    else:
        row1 += 1

    if row < 100:
        row2 = 0
    else:
        row2 += 1
```

```
    row += 1



#print("print out group1")
#print(group1)

#print("print out group2")
#print(group2)

#print("print out group3")
#print(group3)




# The shuffle of group1, Setosa
print("print out again group1")
print(group1)
np.random.shuffle(group1)

#The shuffle of group2, Vesicolour
np.random.shuffle(group2)
np.random.shuffle(group3)
#print(group1)

#The shuffle of group3, Virginia
print('\n')
print("print group 1 shuffle")
print(group1)


#initialize the array of targeted value
targetvalues = [0 for i in range(150)]


#initialize the groups which will be use for training and testing
mixgroup1 = group1
mixgroup2 = group2
mixgroup3 = group3
```

```
combine = [[0 for i in range(4)] for j in range(150)]


#combine the mix matrix for each group
col = 0
row = 0
while row < 150:
    col = 0
    while col < 4:
        if row < 50:
            combine[row][col] = group1[row][col]

        elif 50 <= row and row < 100:
            combine[row][col] = group2[row1][col]

        else:
            combine[row][col] = group3[row2][col]
        col += 1

    if row < 50:
        row1 = 0
    else:
        row1 += 1

    if row < 100:
        row2 = 0
    else:
        row2 += 1

    row += 1


print('\n')
print("print combine")
print(combine)
```

```
# The entire matrix is now mix and alternating between the classes
combinemix = [[0 for i in range(4)] for j in range(150)]


row1 = 50
row2 = 100
row3 = 0

count = 0
resetcount = 1
reset = 1
col = 0
row = 0
while row < 150:
    reset = resetcount % 3

    if reset == 1:
        targetvalues[count] = 0
        col = 0
        while col < 4:
            combinemix[row][col] = combine[row3][col]
            col += 1
        row3 += 1
    elif reset == 2:
        targetvalues[count] = 1
        col = 0
        while col < 4:
            combinemix[row][col] = combine[row1][col]
            col += 1
        row1 += 1
    else:
        targetvalues[count] = 2
        col = 0
        while col < 4:
            combinemix[row][col] = combine[row2][col]
            col += 1
        row2 += 1

    if reset == 0:
        resetcount = 1
    else:
        resetcount += 1
    count += 1
```

```
    row += 1

print('\n')
print("print the combine matrix")
print(combinemix)



target1 = [0 for i in range(50)]
target2 = [0 for i in range(50)]
target3 = [0 for i in range(50)]


print("print the values of the target array")
print(targetvalues)


# Seperate the mixgroups respective targeted values
count1 = 0
count2 = 0
count = 0
while count < 150:
    if count < 50:
        target1[count] = targetvalues[count]
    elif count < 100:
        target2[count1] = targetvalues[count]
        count1 += 1
    else:
        target3[count2] = targetvalues[count]
        count2 += 1
    count += 1

print("print target1 values")
print(target1)

print("print target2 values")
print(target2)

print("print target3 values")
print(target3)

#normalize the targeted values
```

```
count = 0
while count < 50:
    target1[count] = target1[count] / 2
    target2[count] = target2[count] / 2
    target3[count] = target3[count] / 2
    count += 1
# Now save the combine matrix into their respective groups
row1 = 0
row2 = 0

col = 0
row = 0
while row < 150:
    col = 0
    while col < 4:
        if row < 50:
            mixgroup1[row][col] = combinemix[row][col]

        elif 50 <= row and row < 100:
            mixgroup2[row1][col] = combinemix[row][col]

        else:
            mixgroup3[row2][col] = combinemix[row][col]
        col += 1

    if row < 50:
        row1 = 0
    else:
        row1 += 1

    if row < 100:
        row2 = 0
    else:
        row2 += 1

    row += 1

print('\n')
print("print out the mix group 1")
print(mixgroup1)


sett = [0 for i in range(50)]
```

```
count = 0
while count < 50:
    sett[count] = count
    count += 1

print("print sections testing, which will be later use to compare the predicted values")

print("print group 3, section 1")

plt.xlabel('input')
plt.ylabel('grp 3 pred')
plt.scatter(sett, target3)
plt.show()




print("print group 1, section 2")

plt.xlabel('input')
plt.ylabel('grp 1 pred')
plt.scatter(sett, target1)
plt.show()


print("print group 2, section 3")

plt.xlabel('input')
plt.ylabel('grp 2 pred')
plt.scatter(sett, target2)
plt.show()



##########################################################################
#####

wrong = 0

#initializing random weights
w1 = 0.3
w2 = 0.4
w3 = 0.5
```

```
w4 = 0.2
w5 = 0.7
w6 = 0.6
w7 = 0.3
w8 = 0.1
w9 = 0.61
w10 = 0.24
w11 = 0.9
w12 = 0.11
w13 = 0.41
w14 = 0.32

#initialize the bias to be a constant 1
bias = 1

#intitialize the weights of the bias
bw1 = 0.2
bw2 = 0.5
bw3 = 0.7
bw4 = 0.3
bw5 = 0.4



#this will save the X values at each node
array1 = 1
array2 = 1
array3 = 1
array4 = 1
array5 = 1



#this will save the Y values at each node
Node1 = 1
Node2 = 1
Node3 = 1
Node4 = 1
Node5 = 1



#intialize the learning rate
learningrate = 0.01
```

```
#initialize the actual Y value for the output
observe = 1

# I know after you have to regularize the data of observe


#initialize the actual X value for the input/ you don't regularize the X input
arr1 = 1
arr2 = 1
arr3 = 1
arr4 = 1

#Graph the X and Y values
#print('\n')
#plt.xlabel('Xvalues')
#plt.ylabel('Yvalues')
#plt.scatter(inp, ob)
#plt.show()

er= [0 for i in range(50)]

bob= [0 for i in range(50)]

count = 0
while count < 50:
    bob[count] = count
    count += 1




index = 0
# Train your data
epoch = 0
while epoch < 200:
    mixgroup1, target1  = sklearn.utils.shuffle(mixgroup1, target1)
    row = 0

    while row < 50:

        observe = target1[row]
```

```
    col = 0

    while col < 4:

        if col == 0:
            arr1 = mixgroup1[row][col]
        if col == 1:
            arr2 = mixgroup1[row][col]
        if col == 2:
            arr3 = mixgroup1[row][col]
        if col == 3:
            arr4 = mixgroup1[row][col]

        col += 1




    # Weight1 multiplied by the values of the input array
    array1 = w1 * arr1 + w2 * arr2 + w3 * arr3 + w4 * arr4 + bias * bw1

    # Weight2 multiplied by the vales of the input array
    array2 = w5 * arr1 + w6 * arr2 + w7 * arr3 + w8 * arr4 + bias * bw2

    #print(array1)




    # Node 1: takes in the input X and passes it through the sigmoid function then stores it at
the corresponding Y
    Node1 = 1 / (1 + math.exp(-(array1)))

    # Node 2: takes in the input X and passes it through the sigmoid function then stores it at
the correspodning X
    Node2 = 1 / (1 + math.exp(-array2))




    array3 = w9 * Node1 + w10 * Node2 + bias * bw3

    array4 = w11 * Node1 + w12 * Node2 + bias * bw4
```

```
count = 0

# function
# Node 3 and 4
Node3 = 1 / (1 + math.exp(-array3))

# Node 2
Node4 = 1 / (1 + math.exp(-array4))




array5 = w13 * Node3 + w14 * Node4 + bias * bw5

# Node 5
Node5 = 1 / (1 + math.exp(-array5))
# The predicted Y values are stores at node 5
predicted = Node5


error = observe - predicted

#print(error)



# intiliazing a storage for each weight, which you will use to obtain the new weight in gradient descent
w1old = w1
w2old = w2
w3old = w3
w4old = w4
w5old = w5
w6old = w6
w7old = w7
w8old = w8
w9old = w9
w10old = w10
w11old = w11
w12old = w12
w13old = w13
```

```
w14old = w14


oldbw1 = bw1
oldbw2 = bw2
oldbw3 = bw3
oldbw4 = bw4
oldbw5 = bw5




#gradient descent

# Get the derivative for each node using it's correspoding X values
s   = 1/(1 + math.exp(-array5))
dernode5 = s*(1-s)
L5  = dernode5 * error
bw5 = learningrate * L5 * error
w13  = learningrate * L5 * Node3
w14  = learningrate * L5 * Node4


s   = 1/(1 + math.exp(-array3))
dernode3 = s*(1-s)
L3  = dernode3 * ( L5 * w13)
bw3 = learningrate * L3 * bias
w9  = learningrate * L3 * Node1
w10  = learningrate * L3 * Node2


s   = 1/(1 + math.exp(-array4))
dernode4 = s*(1-s)
L4  = dernode4 * (L5 * w14)
bw4 = learningrate * L4 * bias
w11  = learningrate * L4 * Node1
w12  = learningrate * L4 * Node2


s   = 1/(1 + math.exp(-array1))
dernode1 = s*(1-s)
L1  = dernode1 * (L3 * w9) * (L4 * w11)
bw1 = learningrate * L1 * bias
w1  = learningrate * L1 * arr1
```

```
        w2  = learningrate * L1 * arr2
        w3  = learningrate * L1 * arr3
        w4  = learningrate * L1 * arr4

        s   = 1/(1 + math.exp(-array2))
        dernode2 = s*(1-s)
        L2  = dernode2 * (L3 * w10) * (L4 * w12)
        bw2 = learningrate * L2 * bias
        w5  = learningrate * L2 * arr1
        w6  = learningrate * L2 * arr2
        w7  = learningrate * L2 * arr3
        w8  = learningrate * L2 * arr4

        #Using gradient descent to find the where the slope equals to zero fast for each weight.
        w1 = w1 + w1old
        w2 = w2 + w2old
        w3 = w3 + w3old
        w4 = w4 + w4old
        w5 = w5 + w5old
        w6 = w6 + w6old
        w7 = w7 + w7old
        w8 = w8 + w8old
        w9 = w9 + w9old
        w10 = w10 + w10old
        w11 = w11 + w11old
        w12 = w12 + w12old
        w13 = w13 + w13old
        w14 = w14 + w14old



        bw1 = bw1 + oldbw1
        bw2 = bw2 + oldbw2
        bw3 = bw3 + oldbw3
        bw4 = bw4 + oldbw4
        bw5 = bw5 + oldbw5

        row += 1



    epoch += 1

index = 0
```

```
# Train your data
epoch = 0
while epoch < 200:
    mixgroup2, target2  = sklearn.utils.shuffle(mixgroup2, target2)
    row = 0

    while row < 50:

        observe = target2[row]
        col = 0

        while col < 4:

            if col == 0:
                arr1 = mixgroup2[row][col]
            if col == 1:
                arr2 = mixgroup2[row][col]
            if col == 2:
                arr3 = mixgroup2[row][col]
            if col == 3:
                arr4 = mixgroup2[row][col]

            col += 1




        # Weight1 multiplied by the values of the input array
        array1 = w1 * arr1 + w2 * arr2 + w3 * arr3 + w4 * arr4 + bias * bw1

        # Weight2 multiplied by the vales of the input array
        array2 = w5 * arr1 + w6 * arr2 + w7 * arr3 + w8 * arr4 + bias * bw2

        #print(array1)




        # Node 1: takes in the input X and passes it through the sigmoid function then stores it at
the corresponding Y
        Node1 = 1 / (1 + math.exp(-(array1)))

        # Node 2: takes in the input X and passes it through the sigmoid function then stores it at
the correspodning X
```

```
    Node2 = 1 / (1 + math.exp(-array2))



    array3 = w9 * Node1 + w10 * Node2 + bias * bw3

    array4 = w11 * Node1 + w12 * Node2 + bias * bw4

    count = 0

    # function
    # Node 3 and 4
    Node3 = 1 / (1 + math.exp(-array3))

    # Node 2
    Node4 = 1 / (1 + math.exp(-array4))



    array5 = w13 * Node3 + w14 * Node4 + bias * bw5

    # Node 5
    Node5 = 1 / (1 + math.exp(-array5))
    # The predicted Y values are stores at node 5
    predicted = Node5


    error = observe - predicted

    #print(error)



    # intiliazing a storage for each weight, which you will use to obtain the new weight in
gradient descent
    w1old = w1
    w2old = w2
    w3old = w3
    w4old = w4
```

```
w5old = w5
w6old = w6
w7old = w7
w8old = w8
w9old = w9
w10old = w10
w11old = w11
w12old = w12
w13old = w13
w14old = w14


oldbw1 = bw1
oldbw2 = bw2
oldbw3 = bw3
oldbw4 = bw4
oldbw5 = bw5




#gradient descent

# Get the derivative for each node using it's correspoding X values
s   = 1/(1 + math.exp(-array5))
dernode5 = s*(1-s)
L5  = dernode5 * error
bw5 = learningrate * L5 * error
w13  = learningrate * L5 * Node3
w14  = learningrate * L5 * Node4


s   = 1/(1 + math.exp(-array3))
dernode3 = s*(1-s)
L3  = dernode3 * ( L5 * w13)
bw3 = learningrate * L3 * bias
w9  = learningrate * L3 * Node1
w10  = learningrate * L3 * Node2


s   = 1/(1 + math.exp(-array4))
dernode4 = s*(1-s)
L4  = dernode4 * (L5 * w14)
bw4 = learningrate * L4 * bias
```

```
w11  = learningrate * L4 * Node1
w12  = learningrate * L4 * Node2


s   = 1/(1 + math.exp(-array1))
dernode1 = s*(1-s)
L1  = dernode1 * (L3 * w9) * (L4 * w11)
bw1 = learningrate * L1 * bias
w1  = learningrate * L1 * arr1
w2  = learningrate * L1 * arr2
w3  = learningrate * L1 * arr3
w4  = learningrate * L1 * arr4

s   = 1/(1 + math.exp(-array2))
dernode2 = s*(1-s)
L2  = dernode2 * (L3 * w10) * (L4 * w12)
bw2 = learningrate * L2 * bias
w5  = learningrate * L2 * arr1
w6  = learningrate * L2 * arr2
w7  = learningrate * L2 * arr3
w8  = learningrate * L2 * arr4

#Using gradient descent to find the where the slope equals to zero fast for each weight.
w1 = w1 + w1old
w2 = w2 + w2old
w3 = w3 + w3old
w4 = w4 + w4old
w5 = w5 + w5old
w6 = w6 + w6old
w7 = w7 + w7old
w8 = w8 + w8old
w9 = w9 + w9old
w10 = w10 + w10old
w11 = w11 + w11old
w12 = w12 + w12old
w13 = w13 + w13old
w14 = w14 + w14old


bw1 = bw1 + oldbw1
bw2 = bw2 + oldbw2
bw3 = bw3 + oldbw3
bw4 = bw4 + oldbw4
```

```
        bw5 = bw5 + oldbw5

        row += 1



    epoch += 1

print("The error for the first section of testing")
print("The first section of testing use the first and second group for training, and the third group for testing")
row = 0
while row < 50:

        observe = target3[row]
        col = 0

        while col < 4:

            if col == 0:
                arr1 = mixgroup3[row][col]
            if col == 1:
                arr2 = mixgroup3[row][col]
            if col == 2:
                arr3 = mixgroup3[row][col]
            if col == 3:
                arr4 = mixgroup3[row][col]

            col += 1




        # Weight1 multiplied by the values of the input array
        array1 = w1 * arr1 + w2 * arr2 + w3 * arr3 + w4 * arr4 + bias * bw1

        # Weight2 multiplied by the vales of the input array
        array2 = w5 * arr1 + w6 * arr2 + w7 * arr3 + w8 * arr4 + bias * bw2

        #print(array1)
```

```
    # Node 1: takes in the input X and passes it through the sigmoid function then stores it at the corresponding Y
    Node1 = 1 / (1 + math.exp(-(array1)))

    # Node 2: takes in the input X and passes it through the sigmoid function then stores it at the correspodning X
    Node2 = 1 / (1 + math.exp(-array2))




    array3 = w9 * Node1 + w10 * Node2 + bias * bw3

    array4 = w11 * Node1 + w12 * Node2 + bias * bw4

    count = 0

    # function
    # Node 3 and 4
    Node3 = 1 / (1 + math.exp(-array3))

    # Node 2
    Node4 = 1 / (1 + math.exp(-array4))




    array5 = w13 * Node3 + w14 * Node4 + bias * bw5

    # Node 5
    Node5 = 1 / (1 + math.exp(-array5))
    # The predicted Y values are stores at node 5
    predicted = Node5


    error = observe - predicted

    er[row] = error
    print(error)
```

```
        row += 1


plt.xlabel('error')
plt.ylabel('Tested values')
plt.scatter(bob,er)
plt.show()
#plt.scatter(Test_Xvalues, Test_Yvalues)

print("The error for the second section of testing")

print("In this section, will use the second and third for training and use the first group for testing")
index = 0
# Train your data
epoch = 0
while epoch < 200:
    mixgroup2, target2  = sklearn.utils.shuffle(mixgroup2, target2)
    row = 0

    while row < 50:

        observe = target2[row]
        col = 0

        while col < 4:

            if col == 0:
                arr1 = mixgroup2[row][col]
            if col == 1:
                arr2 = mixgroup2[row][col]
            if col == 2:
                arr3 = mixgroup2[row][col]
            if col == 3:
                arr4 = mixgroup2[row][col]

            col += 1




        # Weight1 multiplied by the values of the input array
        array1 = w1 * arr1 + w2 * arr2 + w3 * arr3 + w4 * arr4 + bias * bw1

        # Weight2 multiplied by the vales of the input array
```

```
    array2 = w5 * arr1 + w6 * arr2 + w7 * arr3 + w8 * arr4 + bias * bw2

    #print(array1)



    # Node 1: takes in the input X and passes it through the sigmoid function then stores it at the corresponding Y
    Node1 = 1 / (1 + math.exp(-(array1)))

    # Node 2: takes in the input X and passes it through the sigmoid function then stores it at the correspodning X
    Node2 = 1 / (1 + math.exp(-array2))



    array3 = w9 * Node1 + w10 * Node2 + bias * bw3

    array4 = w11 * Node1 + w12 * Node2 + bias * bw4

    count = 0

    # function
    # Node 3 and 4
    Node3 = 1 / (1 + math.exp(-array3))

    # Node 2
    Node4 = 1 / (1 + math.exp(-array4))



    array5 = w13 * Node3 + w14 * Node4 + bias * bw5

    # Node 5
    Node5 = 1 / (1 + math.exp(-array5))
    # The predicted Y values are stores at node 5
    predicted = Node5
```

```
error = observe - predicted

#print(error)



# intiliazing a storage for each weight, which you will use to obtain the new weight in
gradient descent
w1old = w1
w2old = w2
w3old = w3
w4old = w4
w5old = w5
w6old = w6
w7old = w7
w8old = w8
w9old = w9
w10old = w10
w11old = w11
w12old = w12
w13old = w13
w14old = w14


oldbw1 = bw1
oldbw2 = bw2
oldbw3 = bw3
oldbw4 = bw4
oldbw5 = bw5




#gradient descent

# Get the derivative for each node using it's correspoding X values
s   = 1/(1 + math.exp(-array5))
dernode5 = s*(1-s)
L5  = dernode5 * error
bw5 = learningrate * L5 * error
w13  = learningrate * L5 * Node3
w14  = learningrate * L5 * Node4
```

```
s   = 1/(1 + math.exp(-array3))
dernode3 = s*(1-s)
L3  = dernode3 * ( L5 * w13)
bw3 = learningrate * L3 * bias
w9  = learningrate * L3 * Node1
w10  = learningrate * L3 * Node2


s   = 1/(1 + math.exp(-array4))
dernode4 = s*(1-s)
L4  = dernode4 * (L5 * w14)
bw4 = learningrate * L4 * bias
w11  = learningrate * L4 * Node1
w12  = learningrate * L4 * Node2


s   = 1/(1 + math.exp(-array1))
dernode1 = s*(1-s)
L1  = dernode1 * (L3 * w9) * (L4 * w11)
bw1 = learningrate * L1 * bias
w1  = learningrate * L1 * arr1
w2  = learningrate * L1 * arr2
w3  = learningrate * L1 * arr3
w4  = learningrate * L1 * arr4

s   = 1/(1 + math.exp(-array2))
dernode2 = s*(1-s)
L2  = dernode2 * (L3 * w10) * (L4 * w12)
bw2 = learningrate * L2 * bias
w5  = learningrate * L2 * arr1
w6  = learningrate * L2 * arr2
w7  = learningrate * L2 * arr3
w8  = learningrate * L2 * arr4

#Using gradient descent to find the where the slope equals to zero fast for each weight.
w1 = w1 + w1old
w2 = w2 + w2old
w3 = w3 + w3old
w4 = w4 + w4old
w5 = w5 + w5old
w6 = w6 + w6old
w7 = w7 + w7old
w8 = w8 + w8old
```

```
        w9 = w9 + w9old
        w10 = w10 + w10old
        w11 = w11 + w11old
        w12 = w12 + w12old
        w13 = w13 + w13old
        w14 = w14 + w14old



        bw1 = bw1 + oldbw1
        bw2 = bw2 + oldbw2
        bw3 = bw3 + oldbw3
        bw4 = bw4 + oldbw4
        bw5 = bw5 + oldbw5

        row += 1



    epoch += 1




index = 0
# Train your data
epoch = 0
while epoch < 200:
    mixgroup3, target3  = sklearn.utils.shuffle(mixgroup3, target3)
    row = 0

    while row < 50:

        observe = target3[row]
        col = 0

        while col < 4:

            if col == 0:
                arr1 = mixgroup3[row][col]
            if col == 1:
                arr2 = mixgroup3[row][col]
```

```
        if col == 2:
            arr3 = mixgroup3[row][col]
        if col == 3:
            arr4 = mixgroup3[row][col]

        col += 1




    # Weight1 multiplied by the values of the input array
    array1 = w1 * arr1 + w2 * arr2 + w3 * arr3 + w4 * arr4 + bias * bw1

    # Weight2 multiplied by the vales of the input array
    array2 = w5 * arr1 + w6 * arr2 + w7 * arr3 + w8 * arr4 + bias * bw2

    #print(array1)




    # Node 1: takes in the input X and passes it through the sigmoid function then stores it at
the corresponding Y
    Node1 = 1 / (1 + math.exp(-(array1)))

    # Node 2: takes in the input X and passes it through the sigmoid function then stores it at
the correspodning X
    Node2 = 1 / (1 + math.exp(-array2))




    array3 = w9 * Node1 + w10 * Node2 + bias * bw3

    array4 = w11 * Node1 + w12 * Node2 + bias * bw4

    count = 0

    # function
    # Node 3 and 4
    Node3 = 1 / (1 + math.exp(-array3))

    # Node 2
```

```
Node4 = 1 / (1 + math.exp(-array4))




array5 = w13 * Node3 + w14 * Node4 + bias * bw5

# Node 5
Node5 = 1 / (1 + math.exp(-array5))
# The predicted Y values are stores at node 5
predicted = Node5


error = observe - predicted

#print(error)



# intiliazing a storage for each weight, which you will use to obtain the new weight in gradient descent
w1old = w1
w2old = w2
w3old = w3
w4old = w4
w5old = w5
w6old = w6
w7old = w7
w8old = w8
w9old = w9
w10old = w10
w11old = w11
w12old = w12
w13old = w13
w14old = w14


oldbw1 = bw1
oldbw2 = bw2
oldbw3 = bw3
oldbw4 = bw4
oldbw5 = bw5
```

```
#gradient descent

# Get the derivative for each node using it's correspoding X values
s   = 1/(1 + math.exp(-array5))
dernode5 = s*(1-s)
L5  = dernode5 * error
bw5 = learningrate * L5 * error
w13  = learningrate * L5 * Node3
w14  = learningrate * L5 * Node4


s   = 1/(1 + math.exp(-array3))
dernode3 = s*(1-s)
L3  = dernode3 * ( L5 * w13)
bw3 = learningrate * L3 * bias
w9  = learningrate * L3 * Node1
w10  = learningrate * L3 * Node2


s   = 1/(1 + math.exp(-array4))
dernode4 = s*(1-s)
L4  = dernode4 * (L5 * w14)
bw4 = learningrate * L4 * bias
w11  = learningrate * L4 * Node1
w12  = learningrate * L4 * Node2


s   = 1/(1 + math.exp(-array1))
dernode1 = s*(1-s)
L1  = dernode1 * (L3 * w9) * (L4 * w11)
bw1 = learningrate * L1 * bias
w1  = learningrate * L1 * arr1
w2  = learningrate * L1 * arr2
w3  = learningrate * L1 * arr3
w4  = learningrate * L1 * arr4

s   = 1/(1 + math.exp(-array2))
dernode2 = s*(1-s)
L2  = dernode2 * (L3 * w10) * (L4 * w12)
bw2 = learningrate * L2 * bias
```

```
        w5  = learningrate * L2 * arr1
        w6  = learningrate * L2 * arr2
        w7  = learningrate * L2 * arr3
        w8  = learningrate * L2 * arr4

        #Using gradient descent to find the where the slope equals to zero fast for each weight.
        w1 = w1 + w1old
        w2 = w2 + w2old
        w3 = w3 + w3old
        w4 = w4 + w4old
        w5 = w5 + w5old
        w6 = w6 + w6old
        w7 = w7 + w7old
        w8 = w8 + w8old
        w9 = w9 + w9old
        w10 = w10 + w10old
        w11 = w11 + w11old
        w12 = w12 + w12old
        w13 = w13 + w13old
        w14 = w14 + w14old



        bw1 = bw1 + oldbw1
        bw2 = bw2 + oldbw2
        bw3 = bw3 + oldbw3
        bw4 = bw4 + oldbw4
        bw5 = bw5 + oldbw5

        row += 1



    epoch += 1


row = 0
while row < 50:

        observe = target1[row]
        col = 0

        while col < 4:
```

```
        if col == 0:
            arr1 = mixgroup1[row][col]
        if col == 1:
            arr2 = mixgroup1[row][col]
        if col == 2:
            arr3 = mixgroup1[row][col]
        if col == 3:
            arr4 = mixgroup1[row][col]

        col += 1




    # Weight1 multiplied by the values of the input array
    array1 = w1 * arr1 + w2 * arr2 + w3 * arr3 + w4 * arr4 + bias * bw1

    # Weight2 multiplied by the vales of the input array
    array2 = w5 * arr1 + w6 * arr2 + w7 * arr3 + w8 * arr4 + bias * bw2

    #print(array1)



    # Node 1: takes in the input X and passes it through the sigmoid function then stores it at the corresponding Y
    Node1 = 1 / (1 + math.exp(-(array1)))

    # Node 2: takes in the input X and passes it through the sigmoid function then stores it at the correspodning X
    Node2 = 1 / (1 + math.exp(-array2))




    array3 = w9 * Node1 + w10 * Node2 + bias * bw3

    array4 = w11 * Node1 + w12 * Node2 + bias * bw4

    count = 0

    # function
```

```
        # Node 3 and 4
        Node3 = 1 / (1 + math.exp(-array3))

        # Node 2
        Node4 = 1 / (1 + math.exp(-array4))




        array5 = w13 * Node3 + w14 * Node4 + bias * bw5

        # Node 5
        Node5 = 1 / (1 + math.exp(-array5))
        # The predicted Y values are stores at node 5
        predicted = Node5


        error = observe - predicted

        er[row] = error
        print(error)


        row += 1

plt.xlabel('error')
plt.ylabel('Tested values')
plt.scatter(bob, er)
plt.show()
#plt.scatter(Test_Xvalues, Test_Yvalues)


print("The third section uses the first and the third group for training and the second group for
testing")
print("print the error for the third section")
index = 0
# Train your data
epoch = 0
while epoch < 200:
    mixgroup3, target3  = sklearn.utils.shuffle(mixgroup3, target3)
    row = 0
```

```
    while row < 50:

        observe = target3[row]
        col = 0

        while col < 4:

            if col == 0:
                arr1 = mixgroup3[row][col]
            if col == 1:
                arr2 = mixgroup3[row][col]
            if col == 2:
                arr3 = mixgroup3[row][col]
            if col == 3:
                arr4 = mixgroup3[row][col]

            col += 1




        # Weight1 multiplied by the values of the input array
        array1 = w1 * arr1 + w2 * arr2 + w3 * arr3 + w4 * arr4 + bias * bw1

        # Weight2 multiplied by the vales of the input array
        array2 = w5 * arr1 + w6 * arr2 + w7 * arr3 + w8 * arr4 + bias * bw2

        #print(array1)




        # Node 1: takes in the input X and passes it through the sigmoid function then stores it at
the corresponding Y
        Node1 = 1 / (1 + math.exp(-(array1)))

        # Node 2: takes in the input X and passes it through the sigmoid function then stores it at
the correspodning X
        Node2 = 1 / (1 + math.exp(-array2))
```

```
array3 = w9 * Node1 + w10 * Node2 + bias * bw3

array4 = w11 * Node1 + w12 * Node2 + bias * bw4

count = 0

# function
# Node 3 and 4
Node3 = 1 / (1 + math.exp(-array3))

# Node 2
Node4 = 1 / (1 + math.exp(-array4))




array5 = w13 * Node3 + w14 * Node4 + bias * bw5

# Node 5
Node5 = 1 / (1 + math.exp(-array5))
# The predicted Y values are stores at node 5
predicted = Node5


error = observe - predicted

#print(error)



# intiliazing a storage for each weight, which you will use to obtain the new weight in gradient descent
w1old = w1
w2old = w2
w3old = w3
w4old = w4
w5old = w5
w6old = w6
w7old = w7
w8old = w8
w9old = w9
w10old = w10
```

```
w11old = w11
w12old = w12
w13old = w13
w14old = w14


oldbw1 = bw1
oldbw2 = bw2
oldbw3 = bw3
oldbw4 = bw4
oldbw5 = bw5




#gradient descent

# Get the derivative for each node using it's correspoding X values
s   = 1/(1 + math.exp(-array5))
dernode5 = s*(1-s)
L5  = dernode5 * error
bw5 = learningrate * L5 * error
w13  = learningrate * L5 * Node3
w14  = learningrate * L5 * Node4


s   = 1/(1 + math.exp(-array3))
dernode3 = s*(1-s)
L3  = dernode3 * ( L5 * w13)
bw3 = learningrate * L3 * bias
w9  = learningrate * L3 * Node1
w10  = learningrate * L3 * Node2


s   = 1/(1 + math.exp(-array4))
dernode4 = s*(1-s)
L4  = dernode4 * (L5 * w14)
bw4 = learningrate * L4 * bias
w11  = learningrate * L4 * Node1
w12  = learningrate * L4 * Node2


s   = 1/(1 + math.exp(-array1))
dernode1 = s*(1-s)
```

```
L1  = dernode1 * (L3 * w9) * (L4 * w11)
bw1 = learningrate * L1 * bias
w1  = learningrate * L1 * arr1
w2  = learningrate * L1 * arr2
w3  = learningrate * L1 * arr3
w4  = learningrate * L1 * arr4

s   = 1/(1 + math.exp(-array2))
dernode2 = s*(1-s)
L2  = dernode2 * (L3 * w10) * (L4 * w12)
bw2 = learningrate * L2 * bias
w5  = learningrate * L2 * arr1
w6  = learningrate * L2 * arr2
w7  = learningrate * L2 * arr3
w8  = learningrate * L2 * arr4

#Using gradient descent to find the where the slope equals to zero fast for each weight.
w1 = w1 + w1old
w2 = w2 + w2old
w3 = w3 + w3old
w4 = w4 + w4old
w5 = w5 + w5old
w6 = w6 + w6old
w7 = w7 + w7old
w8 = w8 + w8old
w9 = w9 + w9old
w10 = w10 + w10old
w11 = w11 + w11old
w12 = w12 + w12old
w13 = w13 + w13old
w14 = w14 + w14old



bw1 = bw1 + oldbw1
bw2 = bw2 + oldbw2
bw3 = bw3 + oldbw3
bw4 = bw4 + oldbw4
bw5 = bw5 + oldbw5

row += 1
```

```
    epoch += 1

index = 0
# Train your data
epoch = 0
while epoch < 200:
    mixgroup1, target1  = sklearn.utils.shuffle(mixgroup1, target1)
    row = 0

    while row < 50:

        observe = target1[row]
        col = 0

        while col < 4:

            if col == 0:
                arr1 = mixgroup1[row][col]
            if col == 1:
                arr2 = mixgroup1[row][col]
            if col == 2:
                arr3 = mixgroup1[row][col]
            if col == 3:
                arr4 = mixgroup1[row][col]

            col += 1




        # Weight1 multiplied by the values of the input array
        array1 = w1 * arr1 + w2 * arr2 + w3 * arr3 + w4 * arr4 + bias * bw1

        # Weight2 multiplied by the vales of the input array
        array2 = w5 * arr1 + w6 * arr2 + w7 * arr3 + w8 * arr4 + bias * bw2

        #print(array1)




        # Node 1: takes in the input X and passes it through the sigmoid function then stores it at
the corresponding Y
        Node1 = 1 / (1 + math.exp(-(array1)))
```

```
    # Node 2: takes in the input X and passes it through the sigmoid function then stores it at the correspodning X
    Node2 = 1 / (1 + math.exp(-array2))




    array3 = w9 * Node1 + w10 * Node2 + bias * bw3

    array4 = w11 * Node1 + w12 * Node2 + bias * bw4

    count = 0

    # function
    # Node 3 and 4
    Node3 = 1 / (1 + math.exp(-array3))

    # Node 2
    Node4 = 1 / (1 + math.exp(-array4))




    array5 = w13 * Node3 + w14 * Node4 + bias * bw5

    # Node 5
    Node5 = 1 / (1 + math.exp(-array5))
    # The predicted Y values are stores at node 5
    predicted = Node5


    error = observe - predicted

    #print(error)



    # intiliazing a storage for each weight, which you will use to obtain the new weight in gradient descent
    w1old = w1
```

```
w2old = w2
w3old = w3
w4old = w4
w5old = w5
w6old = w6
w7old = w7
w8old = w8
w9old = w9
w10old = w10
w11old = w11
w12old = w12
w13old = w13
w14old = w14


oldbw1 = bw1
oldbw2 = bw2
oldbw3 = bw3
oldbw4 = bw4
oldbw5 = bw5



#gradient descent

# Get the derivative for each node using it's correspoding X values
s   = 1/(1 + math.exp(-array5))
dernode5 = s*(1-s)
L5  = dernode5 * error
bw5 = learningrate * L5 * error
w13  = learningrate * L5 * Node3
w14  = learningrate * L5 * Node4


s   = 1/(1 + math.exp(-array3))
dernode3 = s*(1-s)
L3  = dernode3 * ( L5 * w13)
bw3 = learningrate * L3 * bias
w9  = learningrate * L3 * Node1
w10  = learningrate * L3 * Node2


s   = 1/(1 + math.exp(-array4))
```

```
dernode4 = s*(1-s)
L4  = dernode4 * (L5 * w14)
bw4 = learningrate * L4 * bias
w11  = learningrate * L4 * Node1
w12  = learningrate * L4 * Node2


s   = 1/(1 + math.exp(-array1))
dernode1 = s*(1-s)
L1  = dernode1 * (L3 * w9) * (L4 * w11)
bw1 = learningrate * L1 * bias
w1  = learningrate * L1 * arr1
w2  = learningrate * L1 * arr2
w3  = learningrate * L1 * arr3
w4  = learningrate * L1 * arr4

s   = 1/(1 + math.exp(-array2))
dernode2 = s*(1-s)
L2  = dernode2 * (L3 * w10) * (L4 * w12)
bw2 = learningrate * L2 * bias
w5  = learningrate * L2 * arr1
w6  = learningrate * L2 * arr2
w7  = learningrate * L2 * arr3
w8  = learningrate * L2 * arr4

#Using gradient descent to find the where the slope equals to zero fast for each weight.
w1 = w1 + w1old
w2 = w2 + w2old
w3 = w3 + w3old
w4 = w4 + w4old
w5 = w5 + w5old
w6 = w6 + w6old
w7 = w7 + w7old
w8 = w8 + w8old
w9 = w9 + w9old
w10 = w10 + w10old
w11 = w11 + w11old
w12 = w12 + w12old
w13 = w13 + w13old
w14 = w14 + w14old



bw1 = bw1 + oldbw1
```

```
        bw2 = bw2 + oldbw2
        bw3 = bw3 + oldbw3
        bw4 = bw4 + oldbw4
        bw5 = bw5 + oldbw5

        row += 1



    epoch += 1



row = 0
while row < 50:

        observe = target1[row]
        col = 0

        while col < 4:

            if col == 0:
                arr1 = mixgroup1[row][col]
            if col == 1:
                arr2 = mixgroup1[row][col]
            if col == 2:
                arr3 = mixgroup1[row][col]
            if col == 3:
                arr4 = mixgroup1[row][col]

            col += 1




        # Weight1 multiplied by the values of the input array
        array1 = w1 * arr1 + w2 * arr2 + w3 * arr3 + w4 * arr4 + bias * bw1

        # Weight2 multiplied by the vales of the input array
        array2 = w5 * arr1 + w6 * arr2 + w7 * arr3 + w8 * arr4 + bias * bw2

        #print(array1)
```

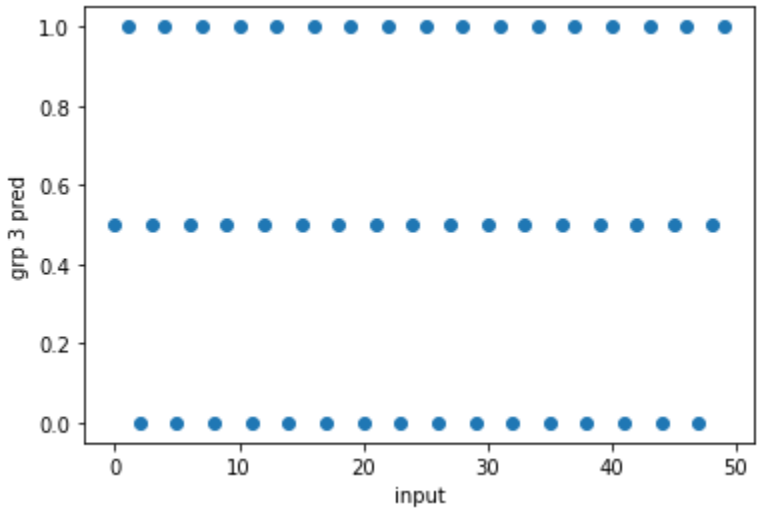
```
    # Node 1: takes in the input X and passes it through the sigmoid function then stores it at the corresponding Y
    Node1 = 1 / (1 + math.exp(-(array1)))

    # Node 2: takes in the input X and passes it through the sigmoid function then stores it at the correspodning X
    Node2 = 1 / (1 + math.exp(-array2))




    array3 = w9 * Node1 + w10 * Node2 + bias * bw3

    array4 = w11 * Node1 + w12 * Node2 + bias * bw4

    count = 0

    # function
    # Node 3 and 4
    Node3 = 1 / (1 + math.exp(-array3))

    # Node 2
    Node4 = 1 / (1 + math.exp(-array4))




    array5 = w13 * Node3 + w14 * Node4 + bias * bw5

    # Node 5
    Node5 = 1 / (1 + math.exp(-array5))
    # The predicted Y values are stores at node 5
    predicted = Node5


    error = observe - predicted
```

```
        print(error)
        er[row] = error

        row += 1

plt.xlabel('error')
plt.ylabel('Tested values')
plt.scatter(bob, er)
plt.show()
#plt.scatter(Test_Xvalues, Test_Yvalues)

#array [samples: setosa, vesicolour, and Virginia] [ Sepal length, sepal width, petal length, petal
width]
#print(array)

#So should I begin by seperating each sample(row) from the stacks of continous rows
#Before that, how do I find the target values first for each row

#y = iris.target
```

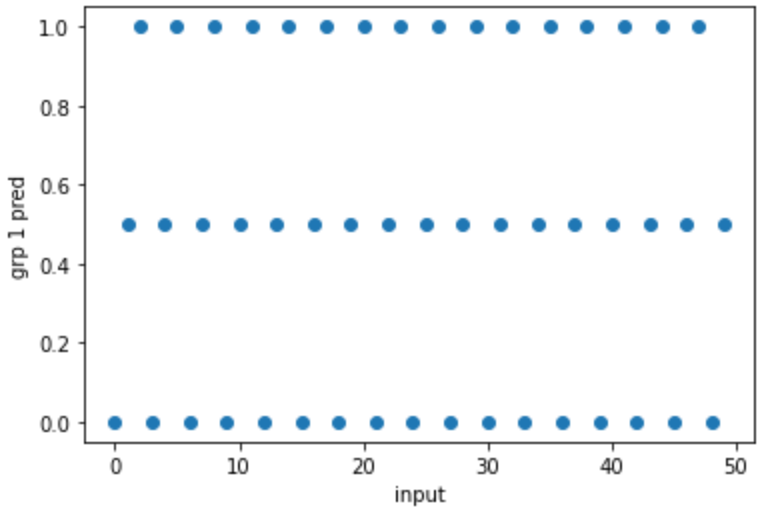
Output:

print sections testing, which will be later use to compare the predicted values

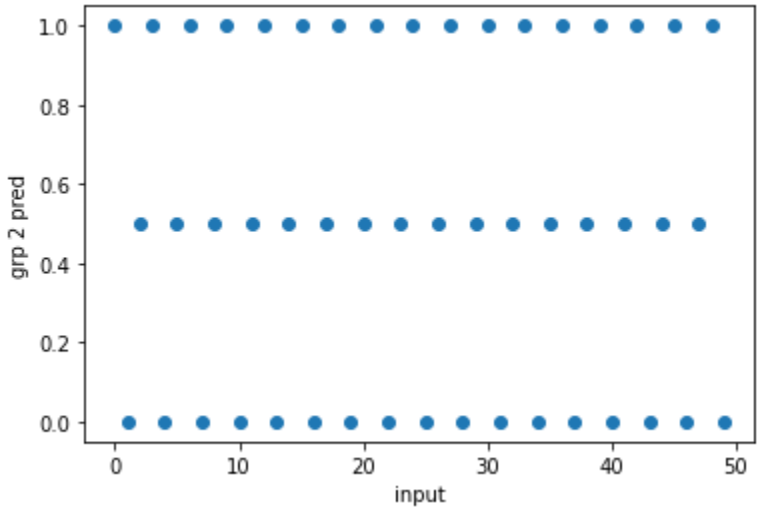
print group 3, section 1



print group 1, section 2

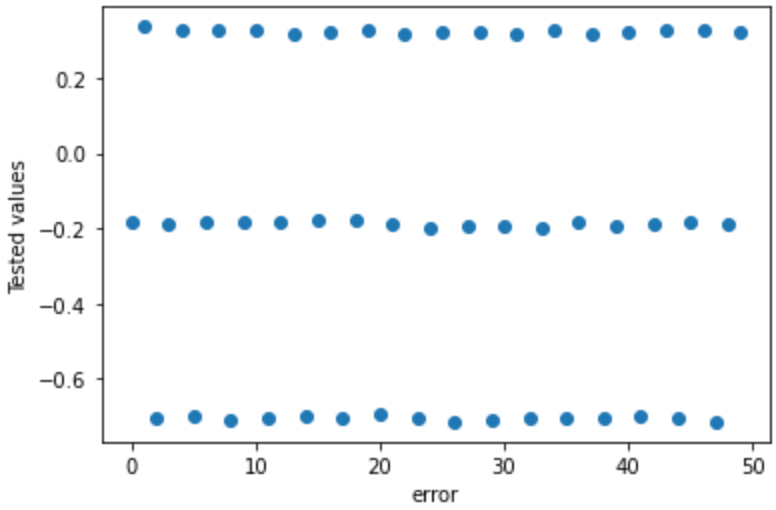


print group 2, section 3



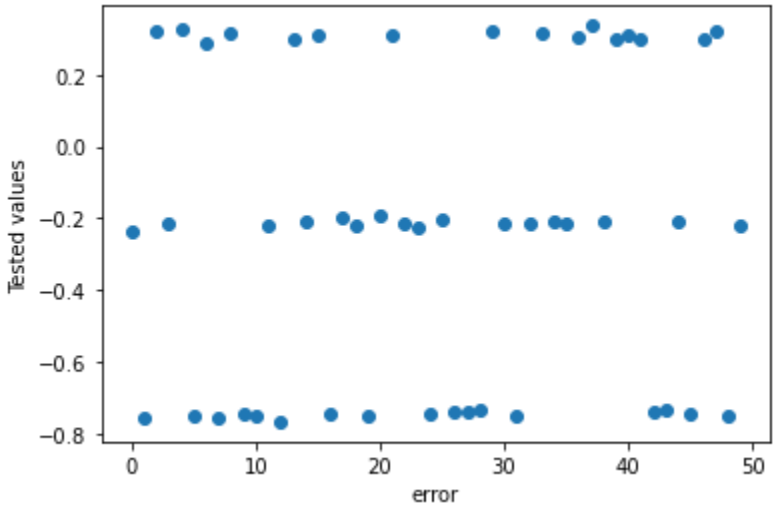
The error for the first section of testing
The first section of testing use the first and second group for training, and the third group for testing
-0.18403548603555775
0.33765491027094363
-0.7042322672613793
-0.18950573503633072
0.327564731374008
-0.6995209989070915
-0.18178359110504583
0.32867074965843834
-0.7102391333153932
-0.1824874203689576
0.3263334127978278
-0.7053376458808089
-0.18496275925170902

0.3178874552807459
-0.70114384829219
-0.18004090939745665
0.3219365026665114
-0.7053208938613037
-0.17914164962012036
0.3278065765115784
-0.6959616530604241
-0.185684333083743
0.3193272993440359
-0.7056213199204058
-0.2006135902736097
0.32058345559672485
-0.7151000105009904
-0.19413193507730786
0.3228034604913693
-0.709326565342849
-0.19272050279831432
0.31631100419889857
-0.7043929549865887
-0.19654632753121826
0.3297660704787031
-0.7050893849441465
-0.18367489168038575
0.3175164101158666
-0.7030354778146358
-0.1913725186716827
0.32360825168496166
-0.7002296067922305
-0.18759141290519066
0.3284728940148799
-0.7029090521919844
-0.18299958062437616
0.3282241176800864
-0.7132808533320417
-0.18601048161704825
0.3255969809623023



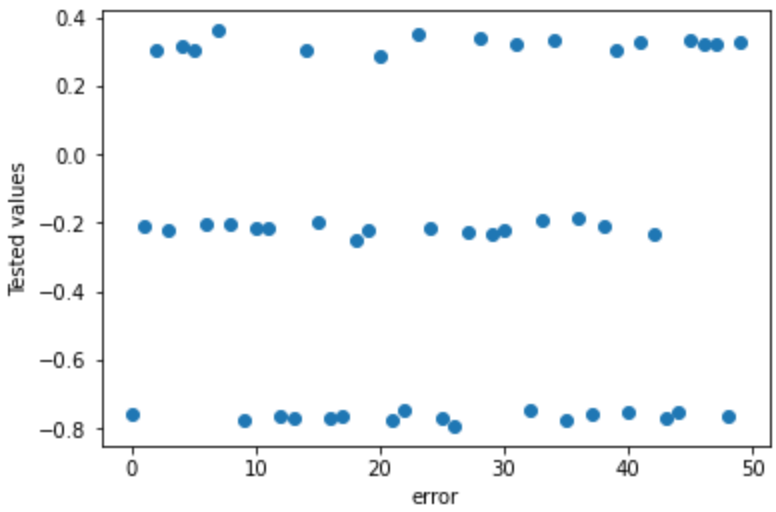
The error for the second section of testing
In this section, will use the second and third for training and use the first group for testing
-0.23460335945768263
-0.7537931480871178
0.3229465859187354
-0.21516639746448163
0.3290304591349741
-0.7518103134678946
0.29012688258988806
-0.7537794628629144
0.31548426154771503
-0.7423788477342264
-0.7507206716517568
-0.21946850833026732
-0.7667324558922373
0.3006690648153836
-0.20639891703530022
0.31145636973874724
-0.7468394253112335
-0.19636022273146492
-0.22275068211264648
-0.751476357150755
-0.19339480584653035
0.3114471351205079
-0.21627293897856592
-0.22570342167039814
-0.7462868143378751
-0.20265815865098347
-0.7408080069480993
-0.7402842970118212
-0.7338596829924928
0.3195175779099012

-0.2142940015194661
-0.7528864649207461
-0.21188391729902012
0.3162457334112022
-0.207616867334384
-0.21210346598143248
0.30643682528644345
0.337038099626639
-0.2079007031819139
0.29983317171638224
0.31239034313180425
0.30052257779430336
-0.7372239936719218
-0.7336023726203186
-0.20713800281806405
-0.7451199230014328
0.30031003946609025
0.3203979855516095
-0.7515769020519997
-0.21816316377985923



The third section uses the first and the third group for training and the second group for testing
print the error for the third section
-0.7587737557758
-0.207543620325496
0.3049583391689673
-0.22303614921540882
0.3143436955972315
0.3042249388379642
-0.20527180282015678
0.36192159377335575
-0.20638961311217074

-0.7739028400155615
-0.21721853752410625
-0.21358072163899688
-0.7652261341982136
-0.7723671742366043
0.30528294782173127
-0.19956838124639265
-0.7720340677769093
-0.7644307017827842
-0.2474507135247238
-0.2202030421468969
0.28945021201227483
-0.7751801494654229
-0.7463216146850821
0.3493795720775794
-0.2139119110813431
-0.7718915865655638
-0.7934210390419774
-0.22500044430540356
0.339899805243813
-0.2343014416476361
-0.218535361476217
0.32352598144733846
-0.7459387410115759
-0.18994010853643994
0.3345750440602726
-0.7751986051004337
-0.18537856175003098
-0.7564876995446841
-0.2071240870794192
0.3055094696665491
-0.7512560372732892
0.3283256292885489
-0.2299022975700964
-0.7708103752483598
-0.7557351299538319
0.3359405738774216
0.3220701527416756
0.322090580522051
-0.7627443436177018
0.3295009883882287


0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
Tested values
0
10
20
30
40
50
error