NNTut2

May 16, 2021

STUDENTS: -Yaseen Haffejee 1827555 -Ziyaad Ballim 1828251 -Jeremy Crouch 1598024 -Fatima Daya 1620146

[20]:
```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
```

0.1 Question 1

Consider a neural network with three input nodes, one hidden layer with two nodes, and one output node. All activation functions are sigmoids (·). The initial weights are as follows (including bias nodes which take the form of x0 = 1):

Theta1 = [1 -1 0.5 1]
[2 -2 1 -1]

theta2 = [-1 2 1]

0.1.1 (a) What is the network output for the following input data?

i. $(0, 3 - 1)$
ii. $(1, 2, 1)$
iii. $(-1, 1, 2)$

[21]:
```python
def sigmoid(x):
    s = 1/(1+np.exp(-x))

    return s

def network(data):
    θ1 = np.array([[1,-1,0.5,1],[2,-2,1,-1]])
    bias1 =θ1[:,0]
    θ2 = np.array([-1,2,1])
    z2 = np.dot(θ1,np.transpose(np.insert(data,0,[1],axis=1)))
    a2init = sigmoid(z2)
    a2 = np.insert(a2init,0,[1],axis=0)
    z3 = np.dot(θ2,a2)
    h = sigmoid(z3)
    return [a2,h]
```

```
i = np.array([[0,3,-1]])
ii = np.array([[1,2,1]])
iii = np.array([[-1,1,2]])
print("Output for i: ",network(i)[1][0])
print("Output for ii: ",network(ii)[1][0])
print("Output for iii: ",network(iii)[1][0])
```

```
Output for i:  0.8365359392688426
Output for ii:  0.8164760997195802
Output for iii:  0.8733158426540304
```

0.1.2 (b)

Update the weights of the network using the backpropagation algorithm when trained on the point (0, 3, −1) with target (class) 0. Use learning rate = 0.1. Repeat the process for point (1, 2, 1) with target 1, and point (−1, 1, 2) with target 0.

[22]:
```
def backpropogate(data,targets,learningRate):
    θ1 = np.array([[1,-1,0.5,1],[2,-2,1,-1]])
    θ2 = np.array([-1,2,1])
    avals = network(data)
    avals.insert(0,np.insert(data,0,[1],axis=1))
    δ3 = avals[2]-targets
    #print(avals[1]*(1-avals[1]))
    #print(np.transpose( 2)* 3)
    #print(avals[1])
    δ2 =np.transpose( 2)* 3*(np.transpose(avals[1]*(1-avals[1])))
    errors = [ 2, 3]
    #print(errors)
    Δ = []
    for i in range(len(avals)-1):
        ans = np.transpose(avals[i])*errors[i]
        #print(ans)
        Δ.append(ans)

    θ2 =θ2 - learningRate*Δ[1]
    θ1 =  np.subtract(θ1,learningRate*np.transpose(np.delete(Δ[0],0,axis=1)))

    return [θ1,θ2[0]]

print("The updated weights are:\n",backpropogate(i,0,0.
↪1)[0],"\n",backpropogate(i,0,0.1)[1])
print("The updated weights are:\n",backpropogate(ii,1,0.
↪1)[0],'\n',backpropogate(ii,0,0.1)[1])
print("The updated weights are:\n",backpropogate(iii,0,0.
↪1)[0],'\n',backpropogate(iii,0,0.1)[1])
```

```
The updated weights are:
 [[ 0.97504673 -1.          0.42514018  1.02495327]
 [ 1.99979367 -2.          0.999381   -0.99979367]]
 [-1.08365359  1.93160696  0.91655325]
The updated weights are:
 [[ 1.00385377 -0.99614623  0.50770753  1.00385377]
 [ 2.0036083  -1.9963917   1.0072166  -0.9963917 ]]
 [-1.08164761  1.92808502  0.94031081]
The updated weights are:
 [[ 0.99810207 -0.99810207  0.49810207  0.99620414]
 [ 1.99605465 -1.99605465  0.99605465 -1.0078907 ]]
 [-1.08733158  1.91362792  0.91681019]
```

0.2 Question 2

As in the logistic regression tut, we will generate our own data to train a neural network. This time, we will generate data in the region [0, 1]^2 with a complicated decision boundary. In your favourite language, perform the following tasks:

(a) Define the function f (x) = x 2 sin(2 x) + 0.7.

[23]:
```python
def f(x):
    y = (x**2)*np.sin(2*np.pi*x)+0.7
    return y
```

(b) Generate a uniform random point (x 1 , x 2) = [0, 1]^2 . Associate with this point the class 0 if f (x 1) > x 2 , and class 1 otherwise.

(c) Generate 100 points in this way. Plot them with different symbols for the two classes.

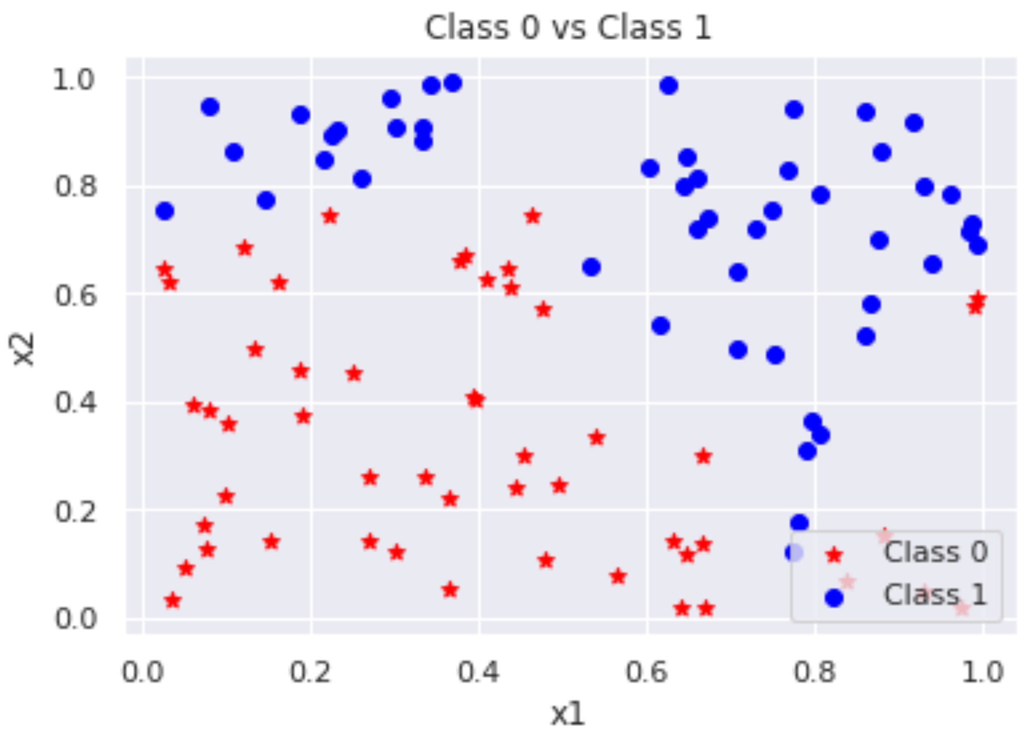
[24]:
```python
def generateData(size):
    class0 = []
    class1 = []
    x0vals = []
    y0vals = []
    x1vals = []
    y1vals = []

    for i in range(size):

        x = np.random.uniform(low=0,high=1,size=(2,))
        if(f(x[0])>x[1]):
            x0vals.append(x[0])
            y0vals.append(x[1])
        else:
            x1vals.append(x[0])
            y1vals.append(x[1])
    class0.append([x0vals,y0vals])
    class1.append([x1vals,y1vals])
    return class0,class1
```

```
target0,target1 = generateData(100)

plt.scatter(target0[0][0],target0[0][1],marker="*",color="red",label="Class 0")
plt.scatter(target1[0][0],target1[0][1],color="blue",label="Class 1")
plt.title("Class 0 vs Class 1")
plt.legend(loc = "lower right")
plt.xlabel("x1")
plt.ylabel("x2")
plt.show()
plt.close()
```



0.3 Question 3

For this question, use the dataset you generated for question 2 above. We are now going to train a neural network model to classify this data. (a) First we need to choose an architecture for the network. This data is 2D, in x 1 and x 2 . We therefore need two input parameters for the model, and one output variable. It is a classification problem, so we can choose the final activation function to be a sigmoid. We know the decision boundary is non-linear because we made the data – otherwise we may need to visualise some of it to figure this out, and so we need at least one hidden layer. Let's use three nodes on the hidden layer, and sigmoids for all activation functions.

```
[25]: def init_layer(x,y):

          weight_matrix = np.random.rand(x,y)
          bias_vector  = np.ones((y,1))

          return [weight_matrix,bias_vector]

      def init_network(neurons):
          length  = len(neurons)-1
          layers = []
          for i in range(length):
              x = neurons[i]
              y = neurons[i+1]
              layers.append(init_layer(x,y))

          return layers

      my_network = init_network([3,4,1])
      nn = np.copy(my_network)
      print(my_network)
```

```
[[array([[0.831251  , 0.77718497, 0.79542914, 0.99822356],
       [0.86055135, 0.5796654 , 0.19181062, 0.75786527],
       [0.72928863, 0.04948695, 0.65879109, 0.05652167]]), array([[1.],
       [1.],
       [1.],
       [1.]])], [array([[0.43708584],
       [0.14296384],
       [0.82135015],
       [0.33465545]]), array([[1.]])]]
```

(b) Implement forward propagation for this network. Do this is a vectorised way, so we can generalise to different architectures. For any input vector, we need a vector of activations at every layer.

```
[26]: def arrangeData(data):
          x1 = data[0][0]
          x2 = data[0][1]
          allData = []
          for i in range(len(x1)):
              point = (x1[i],x2[i])
              allData.append(point)
          return allData

      ClassZero = arrangeData(target0)
      ClassOne = arrangeData(target1)
      TrainingData = ClassZero+ClassOne
      np.random.shuffle(TrainingData)
```

```
def forwardPropogation(data,NN):

    layers = len(NN)  #The Number of layers will be given by len(NN)
    avalues = []
    for i in range(layers):
        if i ==0:
            l = NN[i]
            θ= l[0]
            bias = l[1]
            avalues.append(np.insert(data,0,np.ones((1,np.
 ↪shape(data)[0])),axis=1))
            z = np.dot(avalues[i], )
            a = sigmoid(z)
            avalues.append(a)

        else:
            l = NN[i]
            θ= l[0]
            bias = l[1]
            z = np.dot(avalues[i], )
            a = sigmoid(z)
            avalues.append(a)


    return avalues

ans = forwardPropogation([TrainingData[0]],my_network)
print(ans)
```

```
[array([[1.        , 0.43826174, 0.61273225]]), array([[0.8396002 , 0.7429871 ,
0.78298634, 0.79657106]]), array([[0.79945321]])]
```

(c) Now compute the error   at the final layer as the difference between the final activation and the target.

[27]:
```
def findTarget(data):
    ##Takes in array and checks what the target value is and returns array of␣
 ↪target values for each point
    x = data
    if(f(x[0])>x[1]):
        return 0
    else:
        return 1
```

[28]:
```
def error(output,data):
    ##Error in the final layer for each data point
    actualClass = findTarget(data)
    predictedClass = output.flatten()
    error = np.subtract(predictedClass,np.transpose(actualClass))

    return error

finalErrors = error(ans[2],TrainingData[0])
print(finalErrors)
```

```
[0.79945321]
```

(d) Moving backwards through the layers, compute the of each layer (in this case this would just be for the hidden layer).

```
[29]: def errorLayer(activations,network,errors):

    length = len(network)
    layererrors = []
    finalErrors = errors
    for i in range(length-1,0,-1):
        a = finalErrors*np.transpose(network[i][0])
        b = activations[i]*(1-activations[i])
        δ = a*b
        layererrors.append(δ)
    layererrors.append(finalErrors)
    return layererrors

LayerError = errorLayer(ans,my_network,finalErrors)
print(LayerError)
```

```
[array([[0.04705829, 0.02182506, 0.11157391, 0.0433539 ]]), array([0.79945321])]
```

(e) Given the activations and s, compute the gradients for each parameter.

```
[30]: def gradient(activations,LayerError):

    gradients = []
    for i in range (len(activations)-1):

        Δ  = np.transpose(activations[i])*LayerError[i]
        gradients.append(Δ)

    return gradients
gradients  = gradient(ans,LayerError)
print(gradients)
```

```
[array([[0.04705829, 0.02182506, 0.11157391, 0.0433539 ],
       [0.02062385, 0.00956509, 0.04889858, 0.01900036],
       [0.02883413, 0.01337292, 0.06836493, 0.02656434]]), array([[0.67122108],
       [0.59398342],
       [0.62596094],
       [0.63682129]])]
```

(f) Finally, perform a weight update. Use the learning rate of $\alpha = 0.1$.

```
[31]: def weightUpdate(network,alpha,gradients):
          weights = []
          for i in range(len(network)):
              layer = network[i]
              θ= np.copy(layer[0])
              grad = gradients[i]
              step = alpha*grad
              #print( )
              θ = np.subtract(θ, step)
              #print( )
              weights.append([layer[0],θ])
              layer[0] = θ
          return weights

      weights = weightUpdate(my_network,0.1,gradients)
      #print(weights)
      for i in range (len(weights)):
          print("The weights were:\n",weights[i][0],"\n")
          print("Updated weights are: \n",weights[i][1],"\n")
```

```
The weights were:
 [[0.831251   0.77718497 0.79542914 0.99822356]
 [0.86055135 0.5796654  0.19181062 0.75786527]
 [0.72928863 0.04948695 0.65879109 0.05652167]]

Updated weights are:
 [[0.82654518 0.77500247 0.78427175 0.99388817]
 [0.85848897 0.5787089  0.18692076 0.75596523]
 [0.72640522 0.04814965 0.6519546  0.05386524]]

The weights were:
 [[0.43708584]
 [0.14296384]
 [0.82135015]
 [0.33465545]]

Updated weights are:
 [[0.36996373]
 [0.0835655 ]
 [0.75875405]
 [0.27097332]]
```

(g) Repeat the update in a loop. Technically there are two nested loops: for each epoch (iteration of the outer loop) we run through all our data points and update the weights. We then usually use two terminating conditions on the outer loop: the first is to look at the normed difference between the parameter vector between two successive iterations and we terminate when this is small, i.e. || new − old || < Epsilon, with alpha = 0.05. We also usually set a maximum number of epochs, e.g. 1000, just in case. Run the learning until convergence. What is the error on the training data?

[32]:
```python
def loopUpdate(network,data,α,epochs,ε):

    control = 1
    control2 =1
    while(epochs>0 and (control>  or control2> ) ):

        gradients = 0

        for d in data:
            x = np.reshape(d,(1,len(d)))
            avals = forwardPropogation(x,network)
            finalErrors = error(avals[2][0],d)
            LayerErrors = errorLayer(avals,network,finalErrors)
            Gradients = gradient(avals,LayerErrors)
            w = weightUpdate(network, ,Gradients)
        control = np.linalg.norm(w[0][1]-w[0][0])
        control2 = np.linalg.norm(w[1][1]-w[1][0])
        epochs-=1
    return w
newWeights = loopUpdate(my_network,TrainingData,0.1,1000,0.05)
```

[44]:
```python
def confusionMatrix(Data,Network):
    predicted = []
    actual = []
    for d in Data:

        predicted.append(forwardPropogation([d],Network)[2][0][0])
        actual.append(findTarget(d))

    confusion = np.zeros((2,2))
    PredictedOne = np.where(np.asarray(predicted)>0.5)[0]
    PredictedZero = np.where(np.asarray(predicted)<=0.5)[0]

    #print(predicted)
    #print(actual)
    for p in PredictedOne:

        if(actual[p]==1):
            confusion[1][1]+=1
        elif(actual[p]==0):
            confusion[1][0]+=1
    for z in PredictedZero:
        if(actual[z]==0):
            confusion[0][0]+=1
        elif(actual[z]==1):
            confusion[0][1]+=1

    accuracy = (np.sum(np.diag(confusion))/np.sum(confusion))*100
    return [confusion,accuracy]

c = confusionMatrix(TrainingData,my_network)
print("The Confusion Matrix is:\n",c[0],"\n The Accuracy is thus:",c[1],"%")
```

```
The Confusion Matrix is:
 [[30.  2.]
 [21. 47.]]
 The Accuracy is thus: 77.0 %
```

(h) Generate 100 more datapoints from the procedure in question 2. This will be our validation data. Classify them using your trained model and tabulate the results in a confusion matrix. Compare the error on the training data to the error on the validation data. What do you notice?

[45]:
```
c0,c1 = generateData(100)
ValidationData = arrangeData(c0)+arrangeData(c1)
np.random.shuffle(ValidationData)

vc = confusionMatrix(ValidationData,my_network)
print("The Confusion Matrix is:\n",vc[0],"\n The Accuracy is thus:",vc[1],"%")
```

```
The Confusion Matrix is:
 [[34.  0.]
 [21. 45.]]
 The Accuracy is thus: 79.0 %
```

The error in the validation data and training data is very similar

(i) Change the hyperparameters. Before, we considered changing the learning rate alpha and termination threshold epsilon. Now try add another node or two to the hidden layer, and even add in a second hidden layer. For each different setting of the hyperparameters, retrain your model on the training data and evaluate it on the validation data

[46]:
```
##Neural Network With additional Node in the hidden layer
new_network = init_network([3,5,1])
loopUpdate(new_network,TrainingData,0.000000000000001,1000,0.0000000005)
nc = confusionMatrix(TrainingData,new_network)
print("Training Data:\n")
print("The Confusion Matrix is:\n",nc[0],"\n The Accuracy is thus:",nc[1],"%\n")
```

```
nv = confusionMatrix(ValidationData,new_network)
print("Validation Data: \n")
print("The Confusion Matrix is:\n",nv[0],"\n The Accuracy is thus:",nv[1],"%")
```

```
Training Data:

The Confusion Matrix is:
 [[ 0.  0.]
 [51. 49.]]
 The Accuracy is thus: 49.0 %

Validation Data:

The Confusion Matrix is:
 [[ 0.  0.]
 [55. 45.]]
 The Accuracy is thus: 45.0 %
```

[47]:
```
#Neural Network with 2 additional Nodes in hidden layer
new2_network = init_network([3,6,1])
loopUpdate(new2_network,TrainingData,0.1,1000,0.000005)
n2c = confusionMatrix(TrainingData,new2_network)
print("Training Data:\n")
print("The Confusion Matrix is:\n",n2c[0],"\n The Accuracy is thus:
 ↪",n2c[1],"%\n")

n2v = confusionMatrix(ValidationData,new2_network)
print("Validation Data: \n")
print("The Confusion Matrix is:\n",n2v[0],"\n The Accuracy is thus:",n2v[1],"%␣
 ↪")
```

```
Training Data:

The Confusion Matrix is:
 [[51.  2.]
 [ 0. 47.]]
 The Accuracy is thus: 98.0 %

Validation Data:

The Confusion Matrix is:
 [[55.  2.]
 [ 0. 43.]]
 The Accuracy is thus: 98.0 %
```

[48]:
```
new3_network = init_network([3,4,1])
loopUpdate(new3_network,TrainingData,0.00000001,1000,0.0000005)
```

```
n3c = confusionMatrix(TrainingData,new3_network)
print("Training Data:\n")
print("The Confusion Matrix is:\n",n3c[0],"\n The Accuracy is thus:
 ↪",n3c[1],"%\n")

n3v = confusionMatrix(ValidationData,new3_network)
print("Validation Data: \n")
print("The Confusion Matrix is:\n",n3v[0],"\n The Accuracy is thus:",n3v[1],"%␣
 ↪")
```

```
Training Data:

The Confusion Matrix is:
 [[ 0.  0.]
 [51. 49.]]
 The Accuracy is thus: 49.0 %

Validation Data:

The Confusion Matrix is:
 [[ 0.  0.]
 [55. 45.]]
 The Accuracy is thus: 45.0 %
```

It can be seen that they best model is the model with with two additional nodes in the hidden layer and an alpha of 0.00000001 and epsilon of 0.0000005

(j) Keep the best values of your hyperparameters. Now generate 100 more datapoints. This will be our testing data. Classify them using your trained model and tabulate the results in a confusion matrix. This is the final performance of the classifier with optimised hyperparameters!

[49]:
```
t0,t1 = generateData(100)
TestData = arrangeData(t0)+arrangeData(t1)
np.random.shuffle(TestData)

tc = confusionMatrix(TestData,new2_network)
print("The Confusion Matrix is:\n",tc[0],"\n The Accuracy is thus:",tc[1],"%")
```

```
The Confusion Matrix is:
 [[53.  4.]
 [ 1. 42.]]
 The Accuracy is thus: 95.0 %
```

(k) Why is it important to have the three data sets: training, validation, and testing?

-Training Data enables us to traing our model on as large as a data set as possible which enables our model to possibly view all variations of data which is important. -Validation Data enables us to find the optimal values for our hyperparamaters which ensures we learn as effeciently and accurately as possible. -Test Data enables us to see how good our model performs on data it has never seen before. This is vital since it gives us a barometer of success of our models. All three data sets are pivotal to machine learning.