Project 4 Neural Net
Allison Bolen
Winter 2019
WolffeCIS 678

Goal:

Neural nets are a great way to classify messy data that doesn't have a clear pattern. Neural nets are based on the biological functionality of brain neurons. Where neurons will fire off signals and the stronger the signal the better. The goal of this project was to detect what number was handwritten based on the pixels in an image. We also worked with a fishing data set to determine if it was a good day to fish or not.

Preprocessing:

To preprocess the data for the fish data set which had categorical data. I searched for the columns with only two values and encoded them to binary numbers of 1 or 0. In this case that mean that the column Wind and Air which previously held attributes like {Strong, Weak} or {Warm, cool} were now encoded as {1,0}, {1,0}, respectively. Since the class value of {yes, no} is also binary that was encoded by {1,0} as well. For the columns like Water and Forecast that had more than two possible descriptive options one hot encoding is used to the values. Using one hot encoding you correlate each option with a row from the identity matrix. In this case Water which held the options {Warm, Cool, Moderate} was encoded as {{1,0,0},{0,1,0},{0,0,1}}. The values for Forecast were encoded similarly.

To preprocess the data for the digit set with discrete attribute values between 0 to 16 inclusive, I decided to normalize those values into attributes between 0 and 1 inclusive. I used the normalization equation for feature scaling shown in Figure 1. Additionally since this data set had a multi class target I decided to one hot encode the target values as well.

$$X' = \frac{X - X_{\min}}{X_{\max} - X_{\mathrm{mir}}}$$

Figure 1: Feature Scaling Normalization

Algorithm:

This implementation is a three layer neural net. One layer for the input nodes, one layer for the hidden nodes and onlayer for the output nodes. An visual representation is shown in Figure 2.

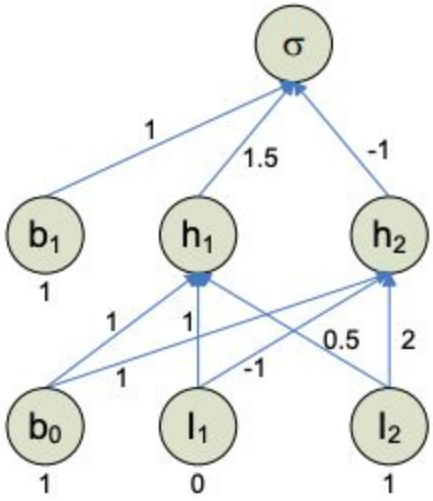

Figure 2: Layers of a neural net.

Every neural net has edge connections from each node in a layer to of nodes of the next layer. Additionally, each layer has a bias node though the bias nodes do not get pointed to by any nodes, see Figure 2. Each edge has its own weight values and these weights are used to process input data to output data.

The basic structure of the algorithm is:

- Loop for batch number/instance/condition
  - Feed input through the network
  - Back Propagate the error
  - Update the weights for the next feed forward step

Your loop condition can be a hardcoded value of how many times you would like to process the data. It's worth keeping in mind if you would like to update the weights after processing each instance or after processing the whole data set. This is known as online vs batch training. Ideally you would use a combination of both. You could also loop until a certain threshold of error is met. We will discuss this in a future step.

The feed forward step is the step where you process the inputs through the nodes at each level. In Order to get the value of the hidden nodes we need to take the sum of the weights multiplied by the inputs + the value of the bias node. In mathematical terms: $\sigma = \sum_i wi\, xi \; + \; bias$ this is the value held at each node. For the hidden node 1 in Figure 2 the value would be (0*1+.5*1)+1 = 1.5. Now we need to pass this value through an activation function. This will force the value at the node to be between 1 and 0 inclusive. This shows us technically how much the 'neuron' fired off, or how strong the signal is. You do this for each node in the hidden layer. We will use the sigmoid function for this calculation, see Figure 3.

$$\text{output} = \frac{1}{1+e^{-\sigma}}$$

Figure 3: Where the output is the value of the activation at this node.

Once you have your hidden node activations you will feed forward those results to the output node in the same way that we did to get the inputs to the hidden nodes. There is one catch, the activation function we use to process the result at the output node can be either the sigmoid function or the function known as softmax. If you have only one output node you can use the sigmoid function. However if you have more than one output node you will have to use the softmax function. Softmax reduces the sum of the output activations to numbers between 0 and 1, giving us a probability spread for the output value where the highest probability is the prediction. This is different than sigmoid because with a multi class output we need to process a list of activations, where with a single output we just need to measure the strength at which it fired.

Once you have your predicted result you can calculate error (also known as cost) between your prediction and your target. There are a few options you can choose from: Sum Squared Error, Mean Squared Error, and Cross Entropy. I chose cross entropy (because I was curious to see how it worked since we discussed SSE in class). Cross entropy measures the performance of a model whose output is a probability between 0 and 1. You can see the function in Figure 4. As the value decreases the closer your prediction is to your target value.

$$H(y, \hat{y}) = -\sum_{i} y_i \log \hat{y}_i$$

Figure 4: Cross Entropy. Where y is the target and y hat is the prediction.

The next step is Backpropagation. The goal here is to minimize the cost function (find the minima of the gradient of the function). This requires us to first take the derivative of the cost function, which in this case is cross entropy. This calculation is done for the output nodes and the hidden layer. The result is the error for each node at that layer and the weights for the edges pointing to that layer will change in the direction to decrease that error.

The final step of this iterative algorithm is to update the weights. To do this you will update the weights for each layer by multiplying the learning rate (learning rate is how fast the weights change at a time, ie the strength of the update) by the error calculations in the previous step and subtract the current weights values at this layer.

Then you repeat these steps until you have a desirable next error value for the net based on the cross entropy function.

My Implementation:

I used linear algebra to create my neural net. I chose to use cross entropy as my cost function and I chose to have a final learning rate of 0.1. My weights were randomly initialized in the range of +/-(1/sqrt(n)) where n is the number of input nodes. For the fish set I had 8 input nodes, 6 hidden nodes and 1 output node, though 9 input nodes including bias and 8 hidden

nodes including bias. For the digit set I had 64 input nodes, 49 hidden nodes and 10 output nodes, though 65 input nodes including bias and 50 hidden nodes including bias. I hard coded my batch count to 55 for both, I realize something smaller would have worked for the fishing set as well but 55 was sufficient for both. I updated the weights based on the online implementation. The learning rate for the fish set is 0.5 and the rate for the digit set is 0.1.

Results:

I get 96.5% accuracy with the digit set and I correctly identify the fish test instance as 'yes'. I have an error graph for both that shows the average error for each batch. See Figure 5 for the fish average error over batch runs.

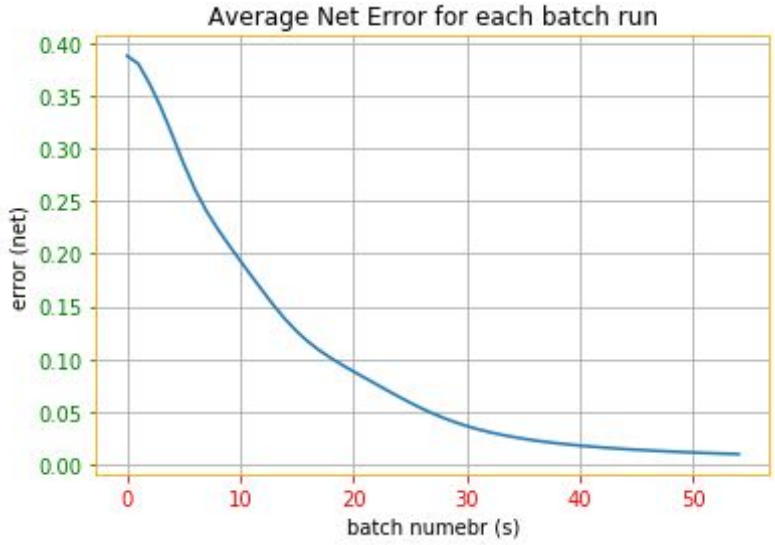

Figure 5: Fishing error.

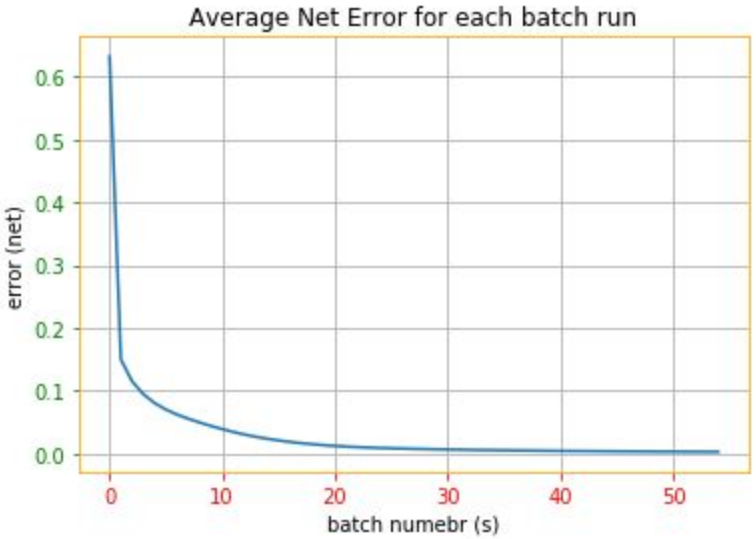

Figure 6: Digit Error.

Code:

```
'''
Allison Bolen
W 19
Wolffe 678
Project 4

'''
# imports:
import numpy as np
import math, os, pickle
from numpy import genfromtxt
import matplotlib
import matplotlib.pyplot as plt

import statistics

class color:
    '''Printing pretty'''
    PURPLE = '\033[95m'
    CYAN = '\033[96m'
    DARKCYAN = '\033[36m'
    BLUE = '\033[94m'
    GREEN = '\033[92m'
    YELLOW = '\033[93m'
    RED = '\033[91m'
    BOLD = '\033[1m'
    UNDERLINE = '\033[4m'
    END = '\033[0m'

def netPlot(instance, error):
    ''' Plot the error '''
    instance = list(range(0, instance))
    with plt.rc_context({'axes.edgecolor':'orange', 'xtick.color':'red',
'ytick.color':'green', 'figure.facecolor':'white'}):
        fig, ax = plt.subplots()
        ax.plot(instance, error)
        ax.set(xlabel='batch numebr (s)', ylabel='error (net)',
               title='Average Net Error for each batch run')
        ax.grid()
```

```
    plt.show()

def sigmoid(x):
    '''Hidden layer activation always, single output activation function'''
    return 1/(1+np.exp(-x))

def sigmoid_der(x):
    '''back prop on activation hidden'''
    return sigmoid(x) *(1-sigmoid (x))

def softmax(A):
    ''' activation of output for multi class'''
    expA = np.exp(A)
    return expA / expA.sum()

def load_objects(file):
    with open(file, 'rb') as input:
        return pickle.load(input)

def save_it_all(obj, filename):
    os.makedirs(os.path.dirname(filename), exist_ok=True)
    with open(filename, 'wb') as output:  # Overwrites any existing file.
        pickle.dump(obj, output, pickle.HIGHEST_PROTOCOL)

def saveNet(wh, bh, wo, bo, fileName):
    weights = {"wh":wh, "bh":bh, "wo":wo, "bo":bo}
    save_it_all(weights, fileName)

def getArrayFromFile(name):
    array = genfromtxt(name, delimiter=',')
    return array

def mapTargetsToEncoded(targets, tMap):
    '''for multi class we need to replace the single target eith ethe
encoded target'''
    newTargets = []
    for item in targets.tolist():
        newTargets.append(tMap[int(item[0])])
    return newTargets

def nn(inputInstance, targetInstance, weightHidden, biasHidden,
weightOutput, biasOutput, ohe):
```

```
    '''neural net'''
        # feed forward
        # Phase 1 inputs fed through to hidden
        zetaHidden = np.dot(inputInstance, weightHidden) + biasHidden
        activationHidden = sigmoid(zetaHidden)

        # Phase 2 hidden fed through to the output
        zetaOutput = np.dot(activationHidden, weightOutput) + biasOutput
        if ohe == 1 :
            activationOutput = softmax(zetaOutput)
        else:
            activationOutput = sigmoid(zetaOutput)

        #### backpropigate with the cross entropy cost function
        # phase one
        derivativecost_zetaOutput = activationOutput - targetInstance

        derivativeZetaOutput_derivatieWeightOutput = activationHidden

        derivativecost_weightOutput =
np.dot(derivativeZetaOutput_derivatieWeightOutput.T,
derivativecost_zetaOutput)

        derivativecost_biasOutput = derivativecost_zetaOutput

        #     # phase two
        derivativeZetaOutput_derivativeActivationHidden = weightOutput

        derivativeCost_derivativeActivationHidden =
np.dot(derivativecost_zetaOutput ,
derivativeZetaOutput_derivativeActivationHidden.T)

        derivativeActivationHidden_derivativeZetaHidden =
sigmoid_der(zetaHidden)

        derivativeZetaHidden_derivativeWeightHidden = inputInstance

        derivativeCost_weightHidden =
np.dot(derivativeZetaHidden_derivativeWeightHidden.T,
derivativeActivationHidden_derivativeZetaHidden *
derivativeCost_derivativeActivationHidden)
```

```
        derivativeCost_biasHidden =
derivativeCost_derivativeActivationHidden *
derivativeActivationHidden_derivativeZetaHidden

        #  update Weights

        weightHidden -= learningRate * derivativeCost_weightHidden

        biasHidden -= learningRate * derivativeCost_biasHidden.sum(axis=0)

        weightOutput -= learningRate * derivativecost_weightOutput

        biasOutput -= learningRate * derivativecost_biasOutput.sum(axis=0)

        loss = np.sum(-targetInstance * np.log(activationOutput))

        return (loss, weightHidden, biasHidden, weightOutput, biasOutput)

def main():
    # load in normalized data set
    # dataSetFile, saveFile, testFile = ("normalizeFish.csv",
"./fishNet.pkl", "normalizeFishTest.csv")
    dataSetFile, saveFile, testFile = ("normDigit.csv", "./digitNet.pkl",
"testDigit.csv")
    data = getArrayFromFile(dataSetFile)
    inputs = data[:,0:data.shape[1]-1] # get the input values
    targets = data[:, data.shape[1]-1:data.shape[1]] # get the class values
    ohe = 0

    if np.unique(targets).shape[0] > 2:
        targetMap = load_objects("./TestingdigitTargetCleanDict.pkl")
        oneHotTargets = np.asarray(mapTargetsToEncoded(targets, targetMap),
dtype=np.float32)
        ohe = 1

    instances = inputs.shape[0]
    attributes = inputs.shape[1]

    numInputNodes = attributes
    numOutputNodes = 1 if np.unique(targets).shape[0] == 2 else
np.unique(targets).shape[0]
    numHiddenNodes = int((2/3)*(numInputNodes+numOutputNodes))
```

```
    print("Input node num: "+ str(numInputNodes))
    print("Hidden node num: "+ str(numHiddenNodes))
    print("Output node num: " + str(numOutputNodes))

    lowRange = (-1/math.sqrt(numInputNodes))
    highRange = math.fabs(lowRange)

    weightHidden = np.random.uniform(low=lowRange, high=highRange,
size=(numInputNodes, numHiddenNodes, ))
    biasHidden = np.random.uniform(low=lowRange, high=highRange,
size=(numHiddenNodes))

    weightOutput = np.random.uniform(low=lowRange, high=highRange,
size=(numHiddenNodes, numOutputNodes))
    biasOutput = np.random.uniform(low=lowRange, high=highRange,
size=(numOutputNodes))

    print()
    print(str(weightHidden.shape))
    print(str(biasHidden.shape))
    print(str(weightOutput.shape))
    print(str(biasOutput.shape))

    print()
    print(str(inputs.shape))
    print()
    learningRate = .01

    errorCost = []
    batchError = []

    for epoch in range(55):
        trackedNetError= []
        print("------------Epoch: "+str(epoch)+"--")
        for instanceRow in range(0, instances):

            inputInstance = np.array([inputs[instanceRow]])
            if ohe == 1:
                targetInstance = np.array([oneHotTargets[instanceRow]])
            else:
                targetInstance = targets[instanceRow]
```

```
            loss, weightHidden, biasHidden, weightOutput, biasOutput =
nn(inputInstance, targetInstance, weightHidden, biasHidden, weightOutput,
biasOutput, ohe)

            trackedNetError.append(loss)
        avg_err = statistics.mean(trackedNetError)
        print("Loss: "+str(avg_err))
        batchError.append(avg_err)
        print()

    netPlot(55, batchError)
    saveNet(weightHidden, biasHidden, weightOutput, biasOutput, saveFile)
    Classify(testFile, saveFile)

def Classify(fileName, NetName):
    data = getArrayFromFile(fileName)
    #data = np.array([getArrayFromFile("normalizeFishTest.csv")])
    inputs = data[:,0:data.shape[1]-1] # get the input values
    targets = data[:, data.shape[1]-1:data.shape[1]] # get the class values

    ohe = 0

    if np.unique(targets).shape[0] > 2:
        ohe = 1

    instances = inputs.shape[0]
    attributes = inputs.shape[1]

    numInputNodes = attributes
    numOutputNodes = 1 if np.unique(targets).shape[0] == 2 else
np.unique(targets).shape[0]
    numHiddenNodes = int((2/3)*(numInputNodes+numOutputNodes))

    nnw = load_objects(netName)

    weightHidden = nnw["wh"]
    biasHidden = nnw["bh"]
    weightOutput = nnw["wo"]
    biasOutput = nnw["bo"]

    correct = 0
```

```
    for instanceRow in range(0, instances):

        inputInstance = np.array([inputs[instanceRow]])
        target = int(targets[instanceRow][0])

        zetaHidden = np.dot(inputInstance, weightHidden) + biasHidden
        activationHidden = sigmoid(zetaHidden)

        zetaOutput = np.dot(activationHidden, weightOutput) + biasOutput
        if ohe == 1:
            activationOutput = softmax(zetaOutput)
        else:
            activationOutput = sigmoid(zetaOutput)

        pred = np.where(activationOutput==np.max(activationOutput))[1][0]
if ohe == 1 else int(round(activationOutput[0][0]))

        if(pred == target):
            correct = correct + 1

    totalPercentRight = (correct/instances)*100
    print("Correct for: "+str(totalPercentRight))

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

References:

I had quite a bit of trouble with the linear algebra part of this but I found some good tutorials on that helped me track my dimensions.

- https://stackabuse.com/creating-a-neural-network-from-scratch-in-python-multi-class-classification/
- https://medium.com/coinmonks/representing-neural-network-with-vectors-and-matrices-c6b0e64db9fb