AI Methods – Artificial Neural Network Coursework

Implementation of ANN

The ANN is designed with a flexible number of inputs and hidden nodes in mind but assumes there is only one output node.

It is developed using Spyder IDE (Python), which includes additional libraries that are installed by default. The only additional library used is pandas, which allows it to read excel spreadsheets

The code was developed with guidance from https://machinelearningmastery.com/implement-backpropagation-algorithm-scratch-python/

The code is also provided in a separate text file.

The program generates a network as a twice nested list of weights. For example:

[[[weights:x,y,z],[weights:x,y,z]],[weights,x,y,z]]. Where the first nested list is the hidden layer and the second is the output layer.

The data is also read as a nested list.

When the algorithm calculates the outputs it initially uses the data entry as an input. It then iterates through each node in a layer and appends the output (activation function) to the node. It then moves to the next layer, using the outputs of the previous nodes as inputs.

[[[weights:x,y,z output: a],[weights:x,y,z output: a]],[weights,x,y,z output:a]].

The algorithm then works backwards through the network, calculating the value of delta for a node and appending it to the list.

[[[weights:x,y,z output: a delta :b],[weights:x,y,z output: a delta: b]],[weights,x,y,z output:a delta: b]]

The weights are then updated for each node in the network and the algorithm iterates through the next data entry. After each data entry the error squared of that iteration is calculated and added to a total. At the end of each epoch the Root Mean Squared Error is calculated to use as a performance indicator.

The ANN is designed so that it can train, validate and test the ANN.

It can also train the network using momentum , bold driver and simulated annealing, although I think there is an error in the bold driver code.

There are two sections highlighted by a string of #:

> The backpropagation algorithm, split into forward pass, backwards pass and updating the weight
>
> The improvements to the algorithm

Code

```python
"""

@author: Alex Vong B827861

Developed using spyder IDE
some guidance from https://machinelearningmastery.com/implement-backpropagation-algorithm-scratch-python/
"""


'''''imports'''
import math
import random
```

```python
#reads from excel
import pandas as pd

'''''initialise the network'''
def initialise(inputn,hiddenn,outputn): #takes number of inputs, hidden nodes (assuming 1 hidden layer) and outputs as parameters
    network = list() #nested list of weights/biases for each node in the network
    #generate weights and bias for each node in the hidden layer
    hiddenlayer = [{"weights":[random.random() for i in range(inputn+1)]}for j in range(hiddenn)] #{"n"}: splits the list into sections
    network.append(hiddenlayer)
    #generate weights and bias for output layer
    outputlayer = [{"weights":[random.random() for i in range(hiddenn+1)]}]
    network.append(outputlayer)
    return network
    #print(network)

'''''calculates weighted sum'''
def weightedsum(weights, inputs):
    wsum = weights[-1] #assume bias to be last value in list
    for i in range(len(weights)-1): #for each weight (-1 to exclude bias)
        wsum += weights[i] * inputs[i] #sum of weight x input
    return wsum

'''''activation/transfer function'''
def activationf(wsum):
    activation = 1 / (1 + math.exp(-wsum))
    return activation


'''''derivative of the transfer function'''
def transferderiv(output):
    return output * (1.0 - output)

'''''
Backpropagation algorithm
#########################################################################################################################
'''

'''''does the forward propagation'''
def forwardprop(network, inputv): #takes the network and input values as parameters
    inputs = inputv
    for layer in network: #for each layer in the network
        newinputs =[] #inputs for next layer
        for node in layer: #for each node in the layer
            s = weightedsum(node["weights"],inputs) #find weighted sum at node
            node["output"] = activationf(s) #return transfer function of node
            newinputs.append(node["output"])
        inputs = newinputs
    return inputs #returns the outputs for each node


'''''does the backpropagation'''
def backprop(network,expected):
    for i in reversed(range(len(network))): #work backwards through the network
        layer = network[i] #current layer
        #if not at output layer (at hidden layer)
        if i != len(network)-1:
            for j in range(len(layer)):
                nodeh = layer[j] #get node at layer
                #get weight and delta of output layer
                for node in network[i+1]: #for loop seems to be the only way to get the values
                    outweight = node['weights'][j]
                    outdelta = node["delta"]
                #calculate delta at node (weight of node to output x delta at output x (output * (1 - output)))
                nodeh["delta"] = (outweight* outdelta * transferderiv(node["output"]))

        #at output layer
        else:
            #find delta of output node
            node = layer[0] #only 1 output node so will be the first node in the layer
            #get error (expected output - actual output)
            errorout = expected - node["output"]
            outdelta = errorout * transferderiv(node["output"]) #get delta
            node["delta"] = outdelta #set delta for node


'''''update weight of nodes'''
def updateweights(network, data, lr):
    for i in range (len(network)): #for each layer in the network
        inputs = data[:-1] #all data except for the last
        if i !=0: #if the output layer
            inputs = [node["output"]for node in network[i - 1]] #inputs are the outputs of prevous layer

        for node in network[i]: #for each node in the layer
            for j in range(len(inputs)): #for each weight
                node['weights'][j] += lr * node['delta'] * inputs[j] #add learning rate x delta x output
            node['weights'][-1] += lr * node['delta'] #for bias output is always 1





        '''''
        #########################################################################################################################
        '''


        '''''train the network'''
        def trainnetwork(network, data, lr, epochs, outputs,):
            for epoch in range(epochs): #iterate through epochs
                    sumerror = 0
                    for row in data: #for each data entry
                        outputs = forwardprop(network,row) #forward pass
                        expected = row[-1]
                        sumerror += (expected-outputs[0])**2 #sum of the errors squared
                        backprop(network,expected) #backwards pass
                        updateweights(network,row,lr) #update the weights
                    rmse = math.sqrt(sumerror/len(data)) #calculate root mean squared error for epoch
                    #print('>epoch=%d, lrate=%.3f, RMSE=%.3f' % (epoch+1, lr, rmse)) #print the epoch number, learning rate and RMSE for that epoch
            print("RMSE=%.3f" % (rmse))



        '''''
        Improvements
        #########################################################################################################################
        '''


        '''''updates weights with momentum added'''
        def updatewithmomentum(network, data, lr, mfunction):
            for i in range (len(network)): #for each layer in the network
                inputs = data[:-1] #all data except for the last
                if i !=0: #if the output layer
                    inputs = [node["output"]for node in network[i - 1]] #inputs are the outputs of prevous layer

                for node in network[i]: #for each node in the layer
                    for j in range(len(inputs)): #for each weight
                        wchange =lr * node['delta'] * inputs[j] #calculate change in weight
                        node['weights'][j] += wchange + (mfunction * wchange)  #add weight + momentum
                        wchange = lr * node['delta']#for bias output is always 1
                        node['weights'][-1] += wchange + (mfunction * wchange)


        '''''train the network with momentum'''
```

```python
    def trainwithmomentum(network, data, lr, epochs, outputs,):
        for epoch in range(epochs): #iterate through epochs
                sumerror = 0
                for row in data: #for each data entry
                    outputs = forwardprop(network,row) #forward pass
                    expected = row[-1]
                    sumerror += (expected-outputs[0])**2
                    backprop(network,expected) #backwards pass
                    updatewithmomentum(network,row,lr,0.9) #update the weights
                rmse = math.sqrt(sumerror/len(data)) #calculate root mean squared error for epoch
                #print('>epoch=%d, lrate=%.3f, RMSE=%.3f' % (epoch+1, lr, rmse)) #print the epoch number, learning rate and RMSE for that epoch
        print("RMSE=%.3f" % (rmse))

    '''''train the network with bold driver'''
    def trainwithbdriver(network, data, lr, epochs, outputs,):
        rmse = 0
        learning_rate = lr
        for epoch in range(epochs): #iterate through epochs
            errorincrease = True
            preverror = rmse
            networkcopy = network #make a backup of the network
            if (epoch+1) % 1000 == 0: #every 1000 epochs try bold driver
                print("implementing bold driver")
                while errorincrease == True:
                    sumerror = 0
                    for row in data: #for each data entry
                        outputs = forwardprop(network,row) #forward pass
                        expected = row[-1]
                        sumerror += (expected-outputs[0])**2
                        backprop(network,expected) #backwards pass
                        updateweights(network,row,learning_rate) #update the weights
                    rmse = math.sqrt(sumerror/len(data)) #calculate root mean squared error for epoch
                    if rmse - preverror > 0: #if the error function increased and it isnt the first epoch
                        print("error increased")
                        network = networkcopy #restore network to previous iteration
                        if learning_rate * 0.5 >= 0.01: #keep within the boundary
                            print("learning rate decreased")
                            learning_rate = learning_rate *0.5 #half the learning rate
                        else:
                            print("learning rate unchanged")
                            errorincrease = False
                    else:
                        print("error decreased")
                        if learning_rate * 1.1 <= 0.5 :
                            print("learning rate increased")
                            learning_rate = learning_rate *1.1
                        errorincrease = False

                    print("learning rate: "+str(learning_rate))
            else:
                sumerror = 0
                for row in data: #for each data entry
                    outputs = forwardprop(network,row) #forward pass
                    expected = row[2]
                    sumerror += (expected-outputs[0])**2
                    backprop(network,expected) #backwards pass
                    updateweights(network,row,learning_rate) #update the weights
                rmse = math.sqrt(sumerror/len(data)) #calculate root mean squared error for epoch
            #print('>epoch=%d, lrate=%.3f, RMSE=%.3f' % (epoch+1, lr, rmse)) #print the epoch number, learning rate and RMSE for that epoch
        print("RMSE=%.3f" % (rmse))

    '''''calculate learning rate through annealing'''
    def calculatelr(epochn,maxepoch, startlr, endlr):
        lr = endlr + ((startlr - endlr)*(1 - (1/ (1+ math.exp(10 - ((20 * epochn)/maxepoch))))))
        return lr

    '''''train the network with annealing'''
    def trainwithanneal(network, data, lr, epochs, outputs):
        for epoch in range(epochs): #iterate through epochs
            learning_rate = calculatelr(epoch, epochs, 0.1, 0.01) #calculate leraning rate for this epoch
            sumerror = 0
            for row in data: #for each data entry
                outputs = forwardprop(network,row) #forward pass
                expected = row[-1]
                sumerror += (expected-outputs[0])**2 #sum of the errors squared
                backprop(network,expected) #backwards pass
                updateweights(network,row,learning_rate) #update the weights
            rmse = math.sqrt(sumerror/len(data)) #calculate root mean squared error for epoch
            #print('>epoch=%d, lrate=%.3f, RMSE=%.3f' % (epoch+1, lr, rmse)) #print the epoch number, learning rate and RMSE for that epoch
        print("RMSE=%.3f" % (rmse))


    '''''
    #########################################################################################################################
    '''

    '''''validate the network'''
    def validatenetwork(network, data, lr, epochs, outputs):
        preverror = 0
        rmse = 10 # set the initial rmse to a high value so that it doesnt stop early
        stop = False
        for epoch in range(epochs): #iterate through epochs
            sumerror = 0
            preverror = rmse #get error of previous epoch
            if stop == False: #as long as error isnt increasing
                for row in data: #for each data entry
                    outputs = forwardprop(network,row) #forward pass
                    expected = row[-1]
                    sumerror += (expected-outputs[0])**2
                    backprop(network,expected) #backwards pass
                    updateweights(network,row,lr) #update the weights
                rmse = math.sqrt(sumerror/len(data)) #calculate root mean squared error for epoch
                #print('>epoch=%d, lrate=%.3f, RMSE=%.3f' % (epoch+1, lr, rmse)) #print the epoch number, learning rate and RMSE for that epoch
                if preverror - rmse < 0: #if error increases
                    print("STOPPED EARLY") #notify if stopped early
                    stop = True
                    break
            else:
                break
        print("RMSE=%.3f" % (rmse))
        return epoch

    '''''calculate outputs using the network and find average error'''
    '''''doesn't backpropagate'''
    def testnetwork(network, data):
        sumerror = 0
        for row in data: #for each data entry
            outputs = forwardprop(network,row) #forward pass
            expected = row[-1]
            sumerror += (expected-outputs[0])**2 #sum of the errors squared
        rmse = math.sqrt(sumerror/len(data)) #calculate root mean squared error for epoch
        print('RMSE=%.3f' % (rmse)) #print the epoch number, learning rate and RMSE for that epoch



    '''''create training dataset from excel'''
    def createtrainingdata():
        #read excel spreadsheet
        print("function started")
        df = pd.read_excel('TestData.xlsx',sheet_name="Training")
        print("Spreadsheet opened")
        #get inputs and output from fields in spreadsheet
        input1 = df['sT'].tolist() #input 1
    input2 = df['sW'].tolist() #input 2
    input3 = df['sSr'].tolist() #input 3
    input4 = df['sDSP'].tolist() #input 4
    input5 = df['sDRH'].tolist() #input 5
    output = df['sPanE'].tolist() #predictand
    print("Pulled data")
    dataset = [[input1[i],input2[i],input3[i],input4[i],input5[i],output[i]] for i in range(len(input1))] #form data entries as nested list
    print("data formed into list")
    return dataset

'''''create validation dataset from excel'''
def createvalidationdata():
    #read excel spreadsheet
    print("function started")
    df = pd.read_excel('TestData.xlsx',sheet_name="Validation")
    print("Spreadsheet opened")
    #get inputs and output from fields in spreadsheet
    input1 = df['sT'].tolist() #input 1
    input2 = df['sW'].tolist() #input 2
    input3 = df['sSr'].tolist() #input 3
    input4 = df['sDSP'].tolist() #input 4
    input5 = df['sDRH'].tolist() #input 5
    output = df['sPanE'].tolist() #predictand
    print("Pulled data")
    dataset = [[input1[i],input2[i],input3[i],input4[i],input5[i],output[i]] for i in range(len(input1))] #form data entries as nested list
    print("data formed into list")
    return dataset


'''''create test dataset from excel'''
def createtestdata():
    #read excel spreadsheet
    print("function started")
    df = pd.read_excel('TestData.xlsx',sheet_name="Testing")
    print("Spreadsheet opened")
    #get inputs and output from fields in spreadsheet
    input1 = df['sT'].tolist() #input 1
    input2 = df['sW'].tolist() #input 2
    input3 = df['sSr'].tolist() #input 3
    input4 = df['sDSP'].tolist() #input 4
    input5 = df['sDRH'].tolist() #input 5
    output = df['sPanE'].tolist() #predictand
    print("Pulled data")
    dataset = [[input1[i],input2[i],input3[i],input4[i],input5[i],output[i]] for i in range(len(input1))] #form data entries as nested list
    print("data formed into list")
    return dataset

'''''test code'''
def test():
    print("PROGRAM START")
    trainingdata = createtrainingdata()
    validationdata = createvalidationdata()
    testdata = createtestdata()
    #sample data for quick testing
    sampledata = [[0.32,0.36,0.19,0.68,0.71,0.14],[0.38,0.64,0.22,0.35,0.69,0.18],[0.36,0.51,0.19,0.51,0.72,0.17]]
    sampledata2 = [[0.3,0.7,0],[0.7,0.3,0],[0.7,0.2,1.0],[0.8,0.3,1.0],[0.6,0.3,1.0],[0.7,0.4,1.0],[0.3,0.6,1.0]]
    #initialise network
    print("DATA PROCESSED")
    #set number of inputs and hidden nodes here
    network = initialise(5, 2, 1)
    print("NETWORK INITIALISED")
    #train network
    trainwithanneal(network, trainingdata, 0.1, 1000, 1)
    print("NETWORK TRAINED")
    #validate network
    validatenetwork(network, validationdata, 0.1, 1000, 1)
    print("NETWORK VALIDATED")
    #test the network
```

```
            testnetwork(network, testdata)
            print("NETWORK TESTED")
            #end of program
            print("PROGRAM COMPLETE")

        test()
```

Data Pre-Processing

To remove extreme values the interquartile range of each predictor was calculated and upper and lower bounds set using:

Lower bound = 1st quartile – 1.5 * interquartile range

Upper bound = 3rd quartile + 1.5* interquartile range

The data was then filtered and any entries that exceeded the limits were removed

For some predictors a value below 0 would also be invalid (such as humidity in %) so entries where those predictors were below 0 were also removed.

The data was then standardised into values between 0.1 and 0.9

To split the data into subsets a random value between 0 and 1 was generated for each entry and the data split where the values ranged from 0 – 0.6, 0.61 – 0.8, 0.81 – 1. This meant that the data should be split into 60%, 20% and 20% of the total data.

The data set used in this ANN is also available in a separate file.

Training the ANN

The performance of the ANN is measured using the Root Mean Squared Error (RMSE) returned after the network has performed a given number of epochs. A lower RMSE would suggest that the ANN is more accurate.

Using the basic backpropagation algorithm, the number of epochs was first decided.

Using the training and validation data, 2 hidden nodes and a learning rate of 0.1:

Hidden Nodes = 2	Learning Rate = 0.1	
Epochs	Training RMSE	Validation RMSE
1000	0.02	0.019
2000	0.013	0.012
3000	0.013	0.013
5000	0.014	0.013
10000	Returned math range error	

There is a substantial improvement between 1000 and 2000 epochs but afterwards the change in error becomes quite small and increases at 5000, which shows that the ANN may be over trained at that point.

At 10000 epochs the algorithm stops working, not sure whether there is a coding error or if the numbers are too extreme at that point.

I will go with 2000 epochs from this point as it seems to be the point where the performance starts to stabilise.

Now the number of hidden nodes will be decided. As there are 5 inputs 2-10 hidden nodes should be tested.

Using the same data and learning rate and 2000 epochs:

Epochs = 2000	Learning rate = 0.1	
Hidden Nodes	Training RMSE	Validation RMSE
2	0.013	0.012
4	0.014	0.013

5	0.013	0.013
6	0.018	0.016
8	0.014	0.013
10	0.012	0.012

From the results it appears that the ANN performs best with 10 hidden nodes, but the error does increase around 6-8 nodes. I will go with 5 hidden nodes as it seems to be a stable number to go with.

These are the weights on each node of a network with 5 hidden nodes after 2000 epochs:

RMSE: 0.014

Hidden nodes:

[weight, weight, weight, weight, weight, bias]

[0.33377520635076835, -0.3204135151685644, 0.11547621750966765, 0.7844796875635062, 0.21683714969746468, 0.2579279923298947]

[-1.4820388140101834, -0.09758342367361017, -1.5811144016053778, 1.6344336460103595, 3.524542352716634, 6.914915089333097]

[0.5811922788585482, -0.1789040505145315, 0.2368482662292241, -0.021010791527175002, -0.13555971985423085, -0.8750008127892623]

[1.533083810726213, 1.079226842042484, 1.197231280924445, -0.46145901416188456, -1.3293110088183417, -2.457424844909402]

[0.17801120537111279, 0.25649785177962997, 0.22926397493096665, 0.7260662866789053, 0.23399925842776875, -1.5284544558286868]

Output node:

[0.5595430909635305, -5.126740337702883, 3.984951225966446, 6.150019474751223, 2.347237087712582, 0.530169532365318]

<u>Evaluation of final model</u>

A new network will be initialised and its performance tested on the test data. It will also be trained using the improvements on the backpropagation algorithm and their performance compared as well.

Epochs = 2000	Learning Rate = 0.1	Hidden Nodes = 5	
Algorithm	Training RMSE	Validation RMSE	Test RMSE
Basic Backpropagation	0.014	0.013	0.016
With momentum	0.014	0.014	0.016
With bold driver	0.09	0.014	0.016
With annealing	0.013	0.013	0.014

The error for bold driver is a lot higher than expected, which I think may be due to an error in the code, so the results won’t be look at.

The errors from the training set and the validation set are almost the same with each test, which shows that the ANN is performing as it should.

The basic backpropagation algorithm and the algorithm with momentum added seem to have a similar performance, although the difference in error may be very small given the size of the numbers, which cannot be seen due to the rounding of the numbers.

Adding annealing into the algorithm improved performance to a measurable extent, which may be due to the learning rate decreasing every epoch, which reduces the likelihood of overtraining the ANN.

Comparison with another data model

Using LINEST in excel I generated weights for each predictor and the bias of the equation. I then formed it into a single layer network and found the RMSE of the data model.

Weights: T, W, Sr, DSP, DRH, bias

0.53864, 0.247723, 0.393995,0.033318,-0.15136,-0.16677

When comparing it with a newly generated ANN:

Model	Test RMSE
ANN	0.015
LINEST	0.256

From the results the ANN is far more accurate than the LINEST model.