Datamining and Machine Learning
T-809-DATA
Exercise 3

Ólafur Jón Thoroddsen

October 12, 2015

Contents

1 Feed-forward Neural Network

1.1 Function implementation

A function for computing the output and hidden layer variables for a single layer feed-forward neural network is implemented in the program *ffnn.py* (see section B.1). The logistic sigmoid function is chosen as the non-linear activation function between layers.

$$\sigma(\gamma) = \frac{1}{1 + e^{-\gamma}} \tag{1}$$

1.2 Functionality testing

The *ffnn* function is tested with the following parameters using the script *tester.py* (see section B.2).

$$\boldsymbol{x} = \begin{bmatrix} 1 & 1 \end{bmatrix}$$

$$\boldsymbol{W}_h = \begin{bmatrix} -0.1 & 0.2 & 0.1 \\ 0.1 & 0.2 & -0.2 \end{bmatrix} \quad \boldsymbol{W}_o = \begin{bmatrix} 0.1 & 0.2 & 0.1 \end{bmatrix}$$

The results are shown in table 1

Network output (y)	0.56524
Hidden layer inputs ($\boldsymbol{a}_1$)	$[\ 0.2\ 0.1\ ]^T$
Hidden layer outputs ($\boldsymbol{z}_1$)	$[\ 0.54983\ 0.524979\ ]^T$

Table 1: Outputs of the network

This is a two layer network with two neurons in the hidden layer and one neuron in the output layer as shown in figure 1.

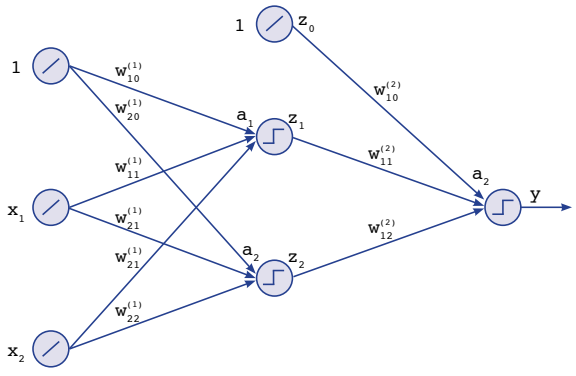

Figure 1: The two layer neural network

2 The Error Backpropogation

Equations referred to in this section can be found in appendix A where an overview of the backprop algorithm can be found.

2.1 Implementation

The backpropagation algorithm is implemented in Python using Numpy in the program *myBackprop.py* (see section B.3). The program is divided into three functions to simplify things.

2.1.1 `outputDeltas()`

`outputDeltas()` calculates the δ_k's using equation 10 for the network given the target vector t and the output y (which is calculated using `ffnn()`).

2.1.2 `networkDeltas()`

`networkDeltas()` calculates the δ_j's for the network using equation 11. Because the activation function is chosen to be a sigmoid, the derivative $h'(a_j)$ is easily evaluated

$$\sigma'(a_j) = \sigma(a_j)(1 - \sigma(a_j))$$

By defining a vector $\boldsymbol{\delta}^o$ where $\delta_k^o = \delta_k$, equation 11 can be simplified to a vector dot product

$$\delta_j = h'(a_j)\boldsymbol{w}_j^T\boldsymbol{\delta^o} \tag{2}$$

where $\boldsymbol{w}_j$ is the j^{th} column in the weight matrix W_2

2.1.3 `myBackprop()`

`myBackprop()` is the main function which distributes calculations to the other functions and does the final calculations to find the gradient matrices

$$\frac{\partial E_n}{w_{ji}^{(1)}} \qquad \frac{\partial E_n}{w_{kj}^{(2)}}$$

where $w_{ji}^{(1)}$ and $w_{kj}^{(2)}$ are the elements of the weight matrices W_1 and W_2 respectively.

When the δ_js have been calculated, we define a vector $\boldsymbol{\delta^1}$ where $\delta_j^1 = \delta_j$ so the calculation of the gradient becomes a matrix multiplication

$$\frac{\partial E_n}{w^{(1)}} = \boldsymbol{\delta^1}\tilde{\boldsymbol{x}}^T \qquad \frac{\partial E_n}{w^{(2)}} = \boldsymbol{\delta^o}\tilde{\boldsymbol{z}}^T$$

where $\tilde{\boldsymbol{x}}$ and $\tilde{\boldsymbol{z}}$ are the inputs to the network and the outputs from the hidden layer respectively with an added column of ones to include the biases in the matrix calculations.

3 Training with gradient decent

The data used for training is a two class Gaussian distributed data with $N_1 = N_2 = 100$ points as shown in figure 2. The parameters of the data sets are

$$\boldsymbol{\mu}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \qquad \boldsymbol{\Sigma}_1 = \begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix}$$

$$\boldsymbol{\mu}_2 = \begin{bmatrix} -1 \\ -1 \end{bmatrix} \quad \boldsymbol{\Sigma}_2 = \begin{bmatrix} 1 & -0.5 \\ -0.5 & 1 \end{bmatrix}$$

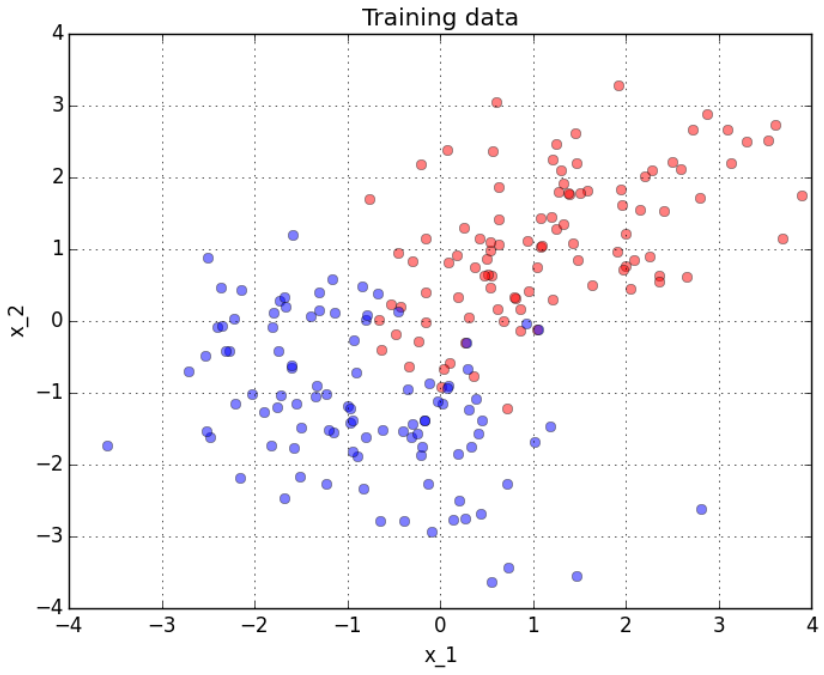

Figure 2: The data used to train our nerual network

Once we've got the gradient matrices from the backpropogation we're ready to start training our neural network. The gradient decent is an iterative approach to optimizing a function by taking steps towards the steepest gradient in each iteration. The way it's implemented here is with sequential

updating which means the weight matrices $W1$ and $W2$ are recalculated every time a new training point goes through the network and the gradients reevaluated as shown in equation 3. In each iteration the weight matrices are updated with

$$\boldsymbol{w}^{(\tau+1)} = \boldsymbol{w}^{(\tau)} - \eta \nabla E_n(\boldsymbol{w}^{(\tau)}) \tag{3}$$

where η is the *learning rate*. By always going in the direction of the negative gradient we will hopefully end up close to a point where $\nabla E_n \approx 0$ which will mean we've found a minimum in the error function.

3.1 Implementation

We initialize the weights with random matrices generated by a uniform distribution between 0 and 1 and sequentially update them using backprop to evaluate the gradient each time and moving through the weight space according to equation 3. Each iteration is one run through all the training data.

4 Evaluation

The test data is generated using exactly the same parameters as the training data but the number of data points is $N = 1000$

By defining the optimal values for the number of iterations `Niter` and the learning rate η, as those that generate the smallest sum of misclassified points in our test set, we loop through `Niter` $= 5, 10, \ldots, 35$ and $\eta = 0.0001, 0.0005, 0.0010, \ldots, 0.0030$, and find that `Niter` $= 20$ and $\eta = 0.003$ yield (see appendices B.4 and B.5)

$$\text{Sum of misclassified points} = 144 \tag{4}$$

which is the best result we could find with our two layer network. The resulting weight matrices are

$$W1 = \begin{bmatrix} -0.11711 & 1.30896 & 1.42017 \\ 0.39298 & 0.89084 & 0.46073 \end{bmatrix} \tag{5}$$

$$W2 = \begin{bmatrix} -0.72771 & 1.38562 & 0.47514 \end{bmatrix} \tag{6}$$

The misclassification rates (MCR) for the test and training sets are listed in tables 2 and 3.

	Classified as	
	Class 1	Class 2
Class 1	912	88
Class 2	939	61

Table 2: Confusion table for the test error

	Classified as	
	Class 1	Class 2
Class 1	90	10
Class 2	91	9

Table 3: Confusion table for the training error

References

[1] C. M. Bishop, *Pattern Recogintion and Machine Learning*. Springer, 2006.

Appendices

A The Backpropagation algorithm [1]

The backpropagation algorithm is essentially a smart way to compute partial derivatives of an error function with respect to every weight and bias parameter in our neural network, that is to evaluate

$$\frac{\partial E_n}{\partial w_{ij}^l} \tag{7}$$

for all l's, i's and j's in the network.

The algorithm is as follows:

1. Apply an input vector $\boldsymbol{x}_n$ to the network and forward propagate through the network using
$$a_j = \sum_i w_{ji} z_i \tag{8}$$
and
$$z_j = h(a_j) \tag{9}$$
to find the activations of all the hidden and output units.

2. Evaluate the δ_k's for all the outputs using
$$\delta_k = y_k - t_k \tag{10}$$

3. Backpropagate the δ's using
$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k \tag{11}$$
to obtain the δ_j's for each hidden unit in the network.

4. Use
$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i \tag{12}$$
to evaluate the elements of the gradient matrix.

B Python code

B.1 ffnn.py

```
#!/usr/local/bin/python3
#
# This program calculates the output of a two layer feed-forward
# neural network given the input pattern x, and weight matrices
    W1 and W2
#
# Author: Olafur Jon Thoroddsen (olafurjt13@ru.is)


# ----- Begin Importing modules ----- #

import numpy as np

# ----- End Importing modules ----- #


# ----- Begin Function Definitions ----- #

def sigmoid(a):
        try:
                tmp = np.matrix([])
                i = 0
                for b in a:
                        tmp = np.insert(tmp,i,(1/(1+np.exp(-b)))
                            )
                        i += 1
                return tmp
        except ValueError:
                return 1/(1+np.exp(-a))

def addOnes(X):
        try:
                return np.insert(X,0,1,axis=0)
        except IndexError:
                return np.insert(X,0,1)

def ffnn(x,W1,W2):
        # Adding a row of ones to the input to simplify
            calculations:
        x = addOnes(x)

        # Calculating the hidden layer inputs a1:
        a1 = np.dot(W1,x)

        # Calculating the hidden layer outputs z1:
        z1 = sigmoid(a1).T


        # Calculating the output layer inputs a2:
        a2 = np.dot(W2,addOnes(z1))

        # Calculating the output of the network y:
        y = sigmoid(a2)
```

```
        return [y,z1,a1]

# ----- End Function Definitions ----- #
```

B.2 tester.py

```
#!/usr/local/bin/python3
#
# ----- Begin Importing modules ----- #

import numpy as np
import sys
from ffnn import ffnn
from myBackprop import myBackprop

# ----- End Importing modules ----- #


def openData(Nx2_datafile):
        array = ([item.strip('\n').split(' ') for item in
            Nx2_datafile])
        for item in array:
                item[0] = float(item[0])
                item[1] = float(item[1])
        return np.matrix(array)


USAGE = """USAGE: tester.py "input pattern" "weight matrix 1" "
    weight matrix 2"
        Example: tester.py x W1 W2
        Where x is a Dx1 dimensional vector
        W1 is Dx(Mh+1) dimensional matrix
        W2 is (Mh+1)xMo dimensional matrix
"""


if len(sys.argv) != 4:
        #print(USAGE)
        exit(1)

script, x_input, W1_input, W2_input = sys.argv

x = np.matrix(np.loadtxt(x_input))
W1 = np.matrix(np.loadtxt(W1_input))
W2 = np.matrix(np.loadtxt(W2_input))

y,z,a = ffnn(x.T,W1,W2)

print("Network output: " + str(y) +'\n'+ "Hidden layer input: "
    + str(a) + '\n'+"Hidden layer output: " + str(z))
print("This configuration has " + str(np.shape(W2)[1]) + "
    neurons in the hidden layer")

t = 1   # Synthesizing the target value so that we get some
```

```
    error when running the script :)

dEn_w1, dEn_w2, y = myBackprop(x.T,t,W1,W2)
```

B.3 myBackprop.py

```
#!/usr/local/bin/python3
#
# This program calculates the gradient of the cross-entropy
# error function for a two layer nerual network given an
# input pattern x, target vector t and weight matrices W1 and W2
#
# Author: Olafur Jon Thoroddsen


# ----- Begin Importing modules ----- #

import numpy as np
from ffnn import ffnn, sigmoid

# ----- End Importing modules ----- #


# ----- Begin Function Definitions ----- #

def outputDeltas(t,y):
        return np.array(y - t)

def networkDeltas(deltaks,ays,W):

        a = np.matrix([])
        i = 0
        for values in ays:
                a = np.insert(a,i,sigmoid(values))
                i += 1
        a = a.T

        W = np.delete(W,0)
        deltajs = np.matrix([])
        i = 0
        for value in a:
                deltajs = np.insert(deltajs,i,(sigmoid(value) *
                    (1 - sigmoid(value)) * np.dot(W.T[i],deltaks)
                    ))
                i += 1
        deltajs = deltajs.T
        return np.matrix(deltajs)

def myBackprop(x,t,W1,W2):

        y,z,a = ffnn(x,W1,W2)
        # y: Network output
        # z: hidden layer output
        # a: hidden layer input
```

```
        # Compute output Deltas
        deltak = outputDeltas(t,y)

        # Computing network Deltas
        deltaj = networkDeltas(deltak,a,W2)

        # Computing the
        dEn_dw1 = deltaj * np.insert(x,0,1)
        dEn_dw2 = deltak * np.insert(z,0,1)

        return [dEn_dw1, dEn_dw2, y]
# ----- End Function Definitions ----- #
```

B.4 NN_eval.py

```
#!/usr/local/bin/python3
#
# ----- Begin Importing modules ----- #

import numpy as np
from ffnn import ffnn
from trainNN import trainNN
import matplotlib.pyplot as plt

# ----- End Importing modules ----- #


def openData(Nx2_datafile):
        array = ([item.strip('\n').split(' ') for item in
            Nx2_datafile])
        for item in array:
                item[0] = float(item[0])
                item[1] = float(item[1])
        return np.matrix(array)


x1_train = open('x1_data.txt')
x2_train = open('x2_data.txt')
x1 = openData(x1_train)
x2 = openData(x2_train)

X = np.matrix(np.concatenate((x1,x2),axis=0))
T = np.matrix(np.concatenate((np.ones(np.size(x1,axis=0)),np.
    zeros(np.size(x2,axis=0))),axis=0)).T

x1_1000_data = open('x1_1000.txt')
x2_1000_data = open('x2_1000.txt')

x1_1000 = openData(x1_1000_data)
x2_1000 = openData(x2_1000_data)

#W1_file = open('W1_trained.txt')
#W2_file = open('W2_trained.txt')
```

```
W1i = np.matrix(np.random.rand(2,3))
W2i = np.matrix(np.random.rand(1,3))



#W1_trained, W2_trained = trainNN(X,T,W1i,W2i,Niter,eta)


#W1_trained = np.matrix
    ([[-0.11711,1.30896,1.52017],[0.39298,0.89084,0.46073]])
#W2_trained = np.matrix([-0.72771,1.38562,0.47514])


# y_training_class1=[]
# for exes in x1:
#       y,z,a = ffnn(exes.T,W1_trained,W2_trained)
#       y_training_class1.append(y[0,0])

# y_training_class2=[]
# for exes in x2:
#       y,z,a = ffnn(exes.T,W1_trained,W2_trained)
#       y_training_class2.append(y[0,0])

# MCR_1 = 0
# for value in y_training_class1:
#       if value < 0.5:
#               MCR_1 += 1

# MCR_2 = 0
# for value in y_training_class2:
#       if value > 0.5:
#               MCR_2 += 1

# print("Testing MCR:")
# print("Class 1: " + str(MCR_1) + '\n' + "Class 2: " + str(
    MCR_2))


# # Evaluating test error:
# x1_1000_data = open('x1_1000.txt')
# x2_1000_data = open('x2_1000.txt')

# x1_1000 = openData(x1_1000_data)
# x2_1000 = openData(x2_1000_data)


# classX1eval = []
# classX2eval = []
# for exes in x1_1000:
#       y,z,a = ffnn(exes.T,W1_trained,W2_trained)
#       classX1eval.append(y[0,0])
```

```
# for exes in x2_1000:
#       y,z,a = ffnn(exes.T,W1_trained,W2_trained)
#       classX2eval.append(y[0,0])

# misclassification_1 = 0
# for value in classX1eval:
#       if value < 0.5:
#               misclassification_1 += 1

# misclassification_2 = 0
# for value in classX2eval:
#       if value > 0.5:
#               misclassification_2 += 1

# print("Training MCR:")
# print("Class 1: " + str(misclassification_1) + '\n' + "Class
    2: " + str(misclassification_2))


# plt.plot(x1[:,0],x1[:,1],'ro',alpha=0.5)
# plt.plot(x2[:,0],x2[:,1],'bo',alpha=0.5)
# plt.grid()
# plt.title('Training data')
# plt.xlabel('x_1')
# plt.ylabel('x_2')
# plt.show()

outputfile = open('Training optimization data.txt','w')

outputfile.write(str(W1i))
outputfile.write(str(W2i))
outputfile.write('Niter eta Training_MCR_C1 Training_MCR_C2
    Test_MCR_C1 Test_MCR_2\n')
for Niter in [5,10,15,20,25,30,35]:
        for eta in
            [0.0001,0.0005,0.001,0.0015,0.002,0.0025,0.0030]:

                W1_trained, W2_trained = trainNN(X,T,W1i,W2i,
                    Niter,eta)

                # Evaluating the training error
                y_training_class1=[]
                for exes in x1:
                        y,z,a = ffnn(exes.T,W1_trained,
                            W2_trained)
                        y_training_class1.append(y[0,0])

                y_training_class2=[]
                for exes in x2:
                        y,z,a = ffnn(exes.T,W1_trained,
                            W2_trained)
                        y_training_class2.append(y[0,0])

                MCR_1 = 0
```

```
for value in y_training_class1:
        if value < 0.5:
                MCR_1 += 1

MCR_2 = 0
for value in y_training_class2:
        if value > 0.5:
                MCR_2 += 1

print("Testing␣MCR:")
print("Class␣1:␣" + str(MCR_1) + '\n' + "Class␣
    2:␣" + str(MCR_2))


# Evaluating test error:
x1_1000_data = open('x1_1000.txt')
x2_1000_data = open('x2_1000.txt')

x1_1000 = openData(x1_1000_data)
x2_1000 = openData(x2_1000_data)


classX1eval = []
classX2eval = []
for exes in x1_1000:
        y,z,a = ffnn(exes.T,W1_trained,
            W2_trained)
        classX1eval.append(y[0,0])

for exes in x2_1000:
        y,z,a = ffnn(exes.T,W1_trained,
            W2_trained)
        classX2eval.append(y[0,0])

misclassification_1 = 0
for value in classX1eval:
        if value < 0.5:
                misclassification_1 += 1

misclassification_2 = 0
for value in classX2eval:
        if value > 0.5:
                misclassification_2 += 1

print("Training␣MCR:")
print("Class␣1:␣" + str(misclassification_1) + '
    \n' + "Class␣2:␣" + str(misclassification_2))

outputfile.write(str(Niter) + '␣' + str(eta) + '
    ␣' + str(MCR_1) + '␣' + str(MCR_2) + '␣' +
    str(misclassification_1) + '␣' + str(
    misclassification_2) + '\n')
```

B.5 findBestArchitecture.py

```
#!/usr/local/bin/python3

from sys import argv

script, inputfile = argv

data = []
with open(inputfile) as Optimization_datafile:
        file_lines = Optimization_datafile.readlines()[7:]
        for lines in file_lines:
                data.append(lines.strip('\n').split(' '))



        i = 0
        optimal = 0
        sum_of_errors = 1000
        for thing in data:
                if ((float(thing[4]) + float(thing[5])) <
                    sum_of_errors):
                        sum_of_errors = float(thing[4]) + float(
                            thing[5])
                        optimal = i
                i += 1

        print("The optimal settings are:\n" + "Niter: " + str(
            data[optimal][0]) + "\n" + "eta: " + str(data[optimal
            ][1]))
        print("The sum of errors is: " + str(sum_of_errors))
```