Goncalo Mendes

ID #1035947

CPEG 586 – DEEP LEARNING

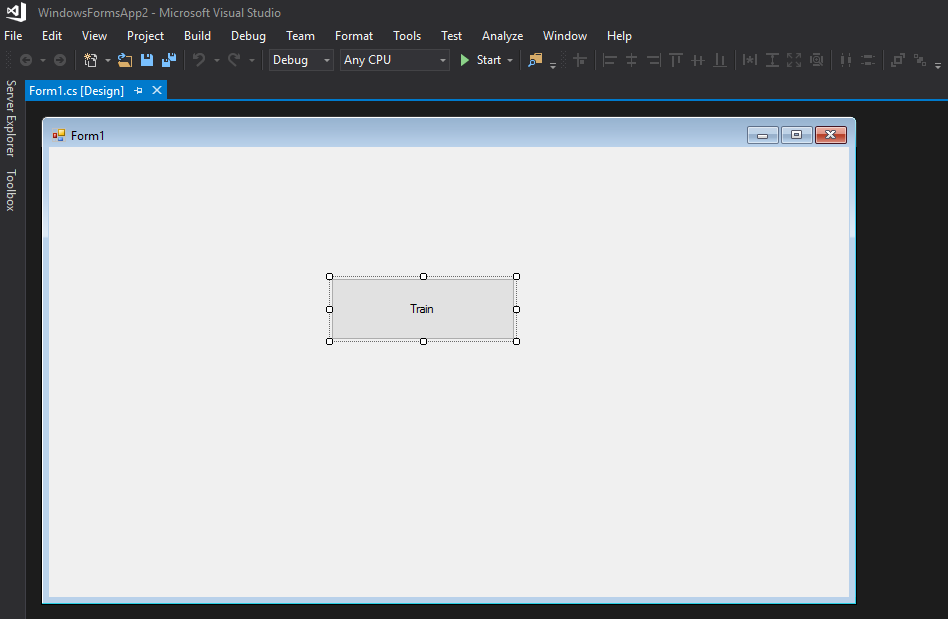
Assignment 1

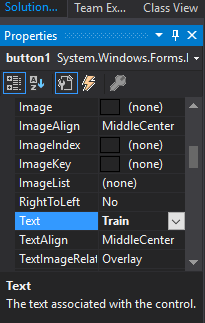
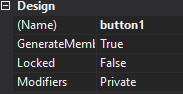
Problem #1

The example in the section Handouts was tried to train a single neuron.

A Windows Forms is created with C# code and a button which I labeled as “train” is placed. To do this I changed the attribute “text” of the button from the toolbox. Noticed if the attribute “name” was changed, it would create a conflict with the example code since the function uses this attribute name to execute. The name must match the function.

After using the code and reading through its logic, I obtained the results of training.



Source Code for example of single neuron training. Code imported from handout only for the purposes of interpreting it and experimenting the forms buttons and running logistics.

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace SingleNeuronBackProp

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

private void btnTrain\_Click(object sender, EventArgs e)

{

double w = 0.1;

double b = 0.4;

double x = 0;

for (int j = 0; j < 1000; j++)

{

for (int i = 1; i < 10; i++)

{

x = i;

double newW = newWeight(x, w, b);

double newb = newBias(x, w, b);

w = newW;

b = newb;

}

}

MessageBox.Show("w = " + w.ToString() + " b =" + b.ToString());

}

double newWeight(double x, double w, double b)

{

// compute output

double a = w \* x + b;

double y = 0.3 \* x + 2;

double gradw = -1 \* (y - a) \* x;

w = w - 0.01 \* gradw;

return w;

}

double newBias(double x, double w, double b)

{

// compute output

double a = w \* x + b;

double y = 0.3 \* x + 2;

double gradb = -1 \* (y - a) \* 1;

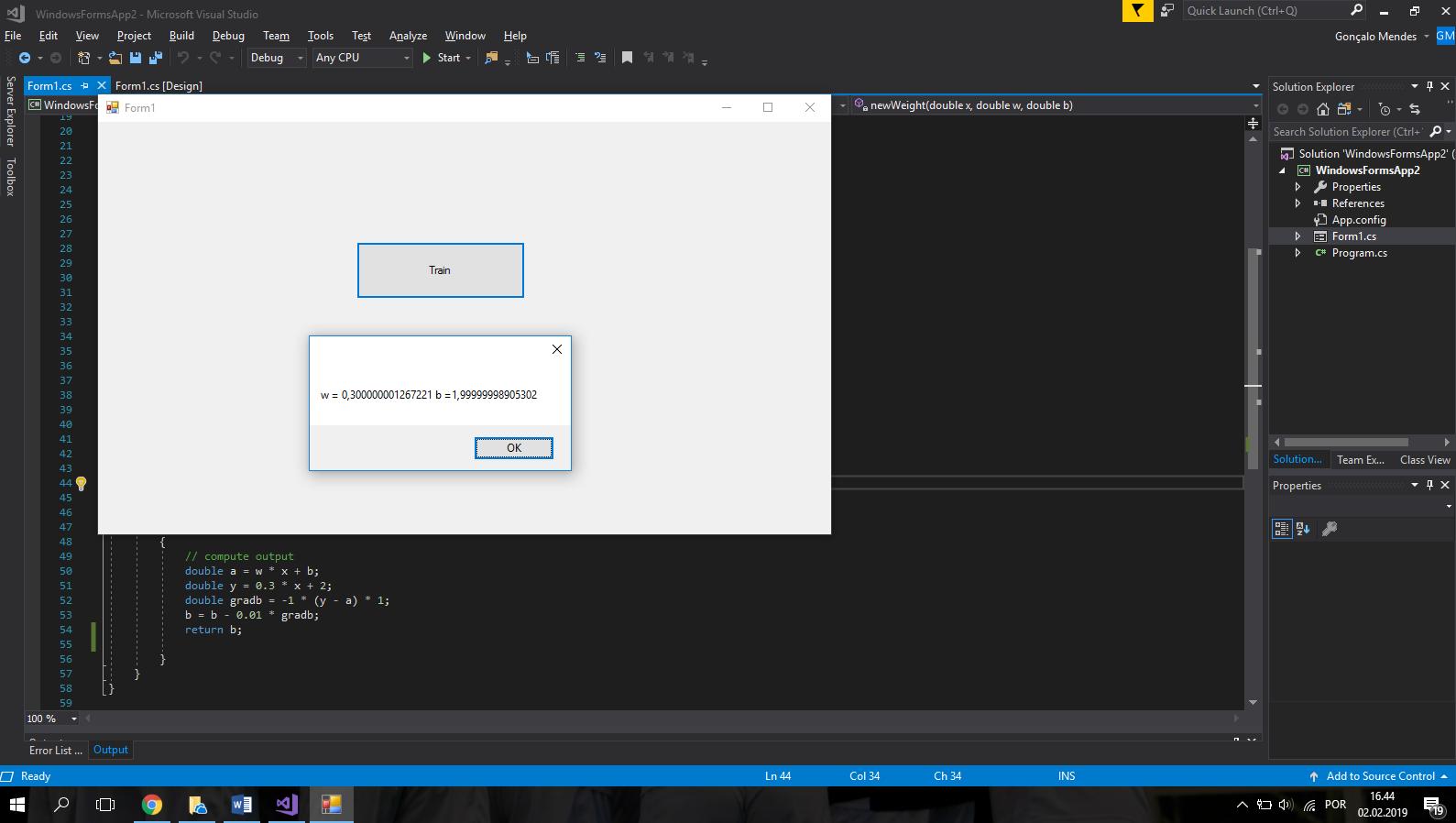
b = b - 0.01 \* gradb;

return b;

}

}

}

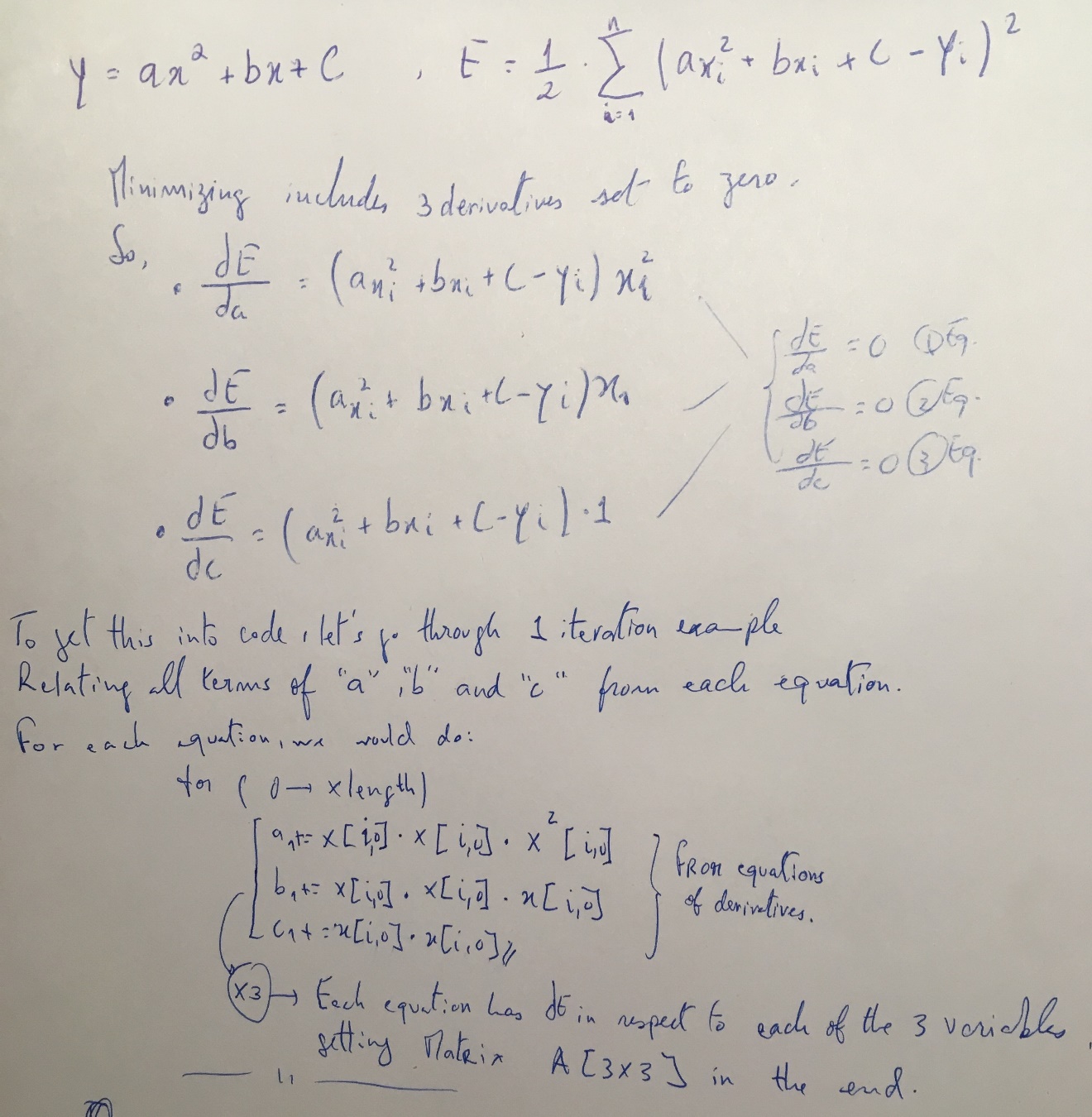


Problem #2

Use the Linear Least Squares Optimization to estimate the model parameters of a line to fit the data given in the table, for a second order polynomial.

The function is now of the type y = ax2+bx+c which means we have three unknowns. When doing the Optimization algorithm, we will have 3 derivatives in respect to each of the variables a,b and c.

In the example given for this optimization, the matrix A with the resulting sums of the error function of each data point is hard coded. Instead of doing the same and computing those sums by hand, I calculated the derivative by hand and then created logic that loops through the data points, plugs them into the derived expression and adds them. Those are the elements of the matrix A, that is then inverted to solve the system of equations.



Source Code for Least Squares Regression

import sys

import numpy as np

import matplotlib.pyplot as plt

def main():

#d = 2\*5.2 + 4 \* 6.7 + 6 \* 9.1 + 8 \* 10.9

#print(d)

#e = 2\*5.2 + 2\*6.7 +2\*9.1 + 2\*10.9

#print(e)

x = np.ndarray((6,1))

y = np.ndarray((6,1))

x[0,0] = 1

x[1,0] = 2

x[2,0] = 3

x[3,0] = 4

x[4,0] = 5

x[5,0] = 6

y[0,0] = 3.2

y[1,0] = 6.4

y[2,0] = 10.5

y[3,0] = 17.7

y[4,0] = 28.1

y[5,0] = 38.5

A = np.ndarray((3,3))

a1=a2=a3=b1=b2=b3=c1=c2=c3=d1=d2=d3=0

i=j=0

for i in range(len(x)):

a1+=x[i,0]\*x[i,0]\*x[i,0]\*x[i,0]

a2+=x[i,0]\*x[i,0]\*x[i,0]

a3+=x[i,0]\*x[i,0]

b1+=x[i,0]\*x[i,0]\*x[i,0]

b2+=x[i,0]\*x[i,0]

b3+=x[i,0]

c1+= x[i,0]\*x[i,0]

c2+= x[i,0]

c3+= 1

d1+=y[i,0]\*x[i,0]\*x[i,0]

d2+=y[i,0]\*x[i,0]

d3+=y[i,0]

A[0,0] = a1

A[0,1] = b1

A[0,2] = c1

A[1,0] = a2

A[1,1] = b2

A[1,2] = c2

A[2,0] = a3

A[2,1] = b3

A[2,2] = c3

z = np.ndarray((3,1))

z[0,0] = d1

z[1,0] = d2

z[2,0] = d3

#A[0,0] = 60

#A[0,1] = 20

#A[1,0] = 20

#A[1,1] = 8

ainv = np.linalg.inv(A)

#z[0,0] = 179

#z[1,0] = 63.8

res = np.dot(ainv,z) # a = res[0,0] and b=[1,0]

print(res)

# do a scatter plot of the data

area = 10

colors =['black']

plt.scatter(x, y, s=area, c=colors, alpha=0.5, linewidths=8)

plt.title('Linear Least Squares Regression')

plt.xlabel('x')

plt.ylabel('y')

#plot the fitted line

yfitted = x\*x\*res[0,0] + res[1,0]\*x + res[2,0]

line,=plt.plot(x, yfitted, '--', linewidth=2) #line plot

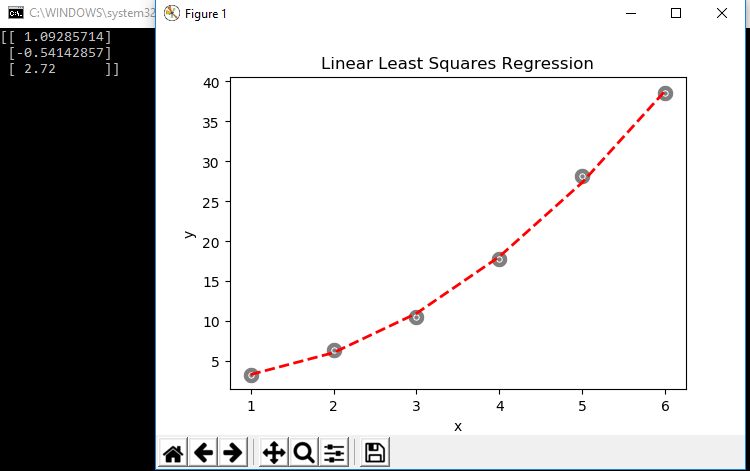
line.set\_color('red')

plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

sys.exit(int(main() or 0))

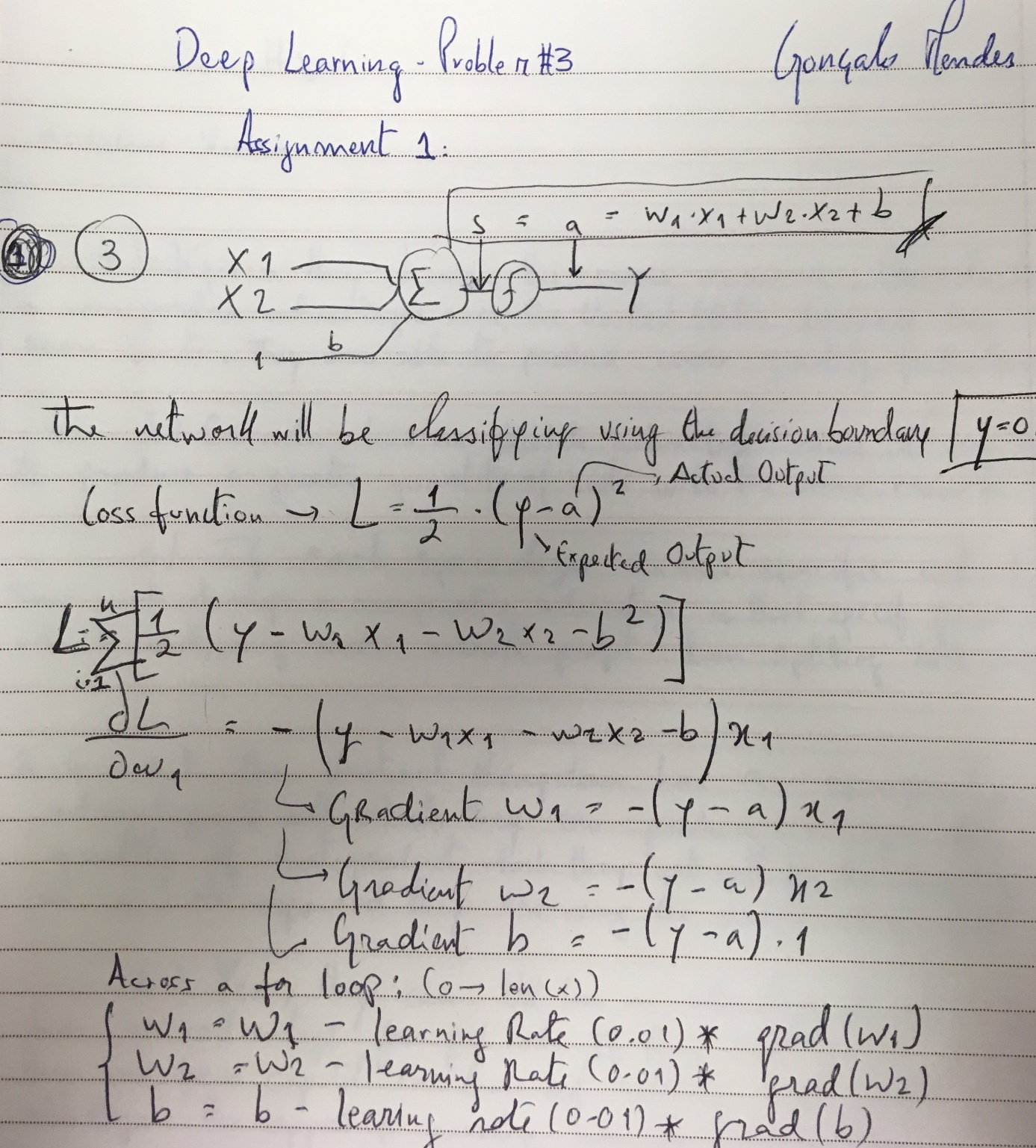
This way we were able to compute the derivatives at each of the data points and sum the components of each variable (a,b,c) in each equation with the for loop. The output looks like the following:



PROBLEM #3

In this problem, I picked up the example handout of the Backpropagation Algorithm and altered it to fit the problem in hand, by declaring two different weights instead of just one and coding the logic for classification. ( Class of points above the line – Output 1 an Class of points below the line – Output 0).

Computations :



After expanding the weights adjustment equation, computing the gradients in respect to all three variables w1,w2,b, we use the training data points to loop through and adjust the weights through 1000 epochs of backpropagation.

I also named the Y correspondent to the line equation as “y\_graph“ to avoid confusion with the expected result, named as “output”. And what the algorithm is learning is whether or not each point is above the line or not, which we determine by asking if the y-component of the data point is above the y-component of the y\_graph line for the same X.

Source Code for Backpropagation, in classification problem:

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace WindowsFormsApp2

{

public partial class Form1 : Form

{

double w1, w2;

double b;

public Form1()

{

InitializeComponent();

}

private void button1\_Click(object sender, EventArgs e)

{

double [] trainX = { 1, 1, 2, 2, 3, 3 };

double [] trainY = { 2.2, 2.4, 2.5, 2.7, 2.8, 3.0 };

/\* double x1[0] = 1.5;

double x2[0] = 2.4;

double x1[0] = 1.5;

double x2[0] = 2.5;

double x1[0] = 2.5;

double x2[0] = 2.7;

double x1[0] = 2.5;

double x2[0] = 2.8; \*/

w1 = 0.3;

w2 = 0.1;

b = -0.1;

int bin;

for (int i = 0; i < 1000; i++)

{

for (int j = 0; j < 6; j++)

{

double x1 = trainX[j];

double x2 = trainY[j];

double newW1 = newWeight(x1, x2, w1, w2, b, 0);

double newW2 = newWeight(x1, x2, w1, w2, b, 1);

double newb = newBias(x1, x2, w1, w2, b);

w1 = newW1;

w2 = newW2;

b = newb;

}

}

MessageBox.Show("w1 = " + w1.ToString() + "w1 = " + w2.ToString() + " b =" + b.ToString());

}

double newWeight(double x1, double x2, double w1, double w2, double b, int bin)

{

// compute output

double a = w1 \* x1 + w2 \* x2 + b;

double y\_graph = 0.3 \* x1 + 2;

double output;

double w;

if (y\_graph < x2) output = 1;

else output = 0; // 0 is including case of point being on the line.

if (bin == 0)

{

double gradw = -1 \* (output - a) \* x1;

w1 = w1 - 0.01 \* gradw;

w = w1;

}

else

{

double gradw = -1 \* (output - a) \* x2;

w2 = w2 - 0.01 \* gradw;

w = w2;

}

return w;

}

double newBias(double x1, double x2, double w1, double w2, double b)

{

// compute output

double a = w1 \* x1 + w2 \* x2 + b;

double y\_graph = 0.3 \* x1 + 2;

double output;

if (y\_graph < x2) output = 1;

else output = 0;

double gradb = -1 \* (output - a) \* 1;

b = b - 0.01 \* gradb;

return b;

}

private void button2\_Click(object sender, EventArgs e)

{

double[] test\_x = { 1.5, 1.5, 2.5, 2.5, 3.5, 3.5, 4.5, 4.5, 5.5, 5.5, 6.5, 6.5 };

double[] test\_y = { 2.3, 2.5, 2.7, 2.8, 3.1, 3.3, -1, 6, 20, 21, -10, -20 };

double output;

for (int i = 0; i < test\_x.Length; i++)

{

double result = test\_x[i] \* w1 + test\_y[i] \* w2 + b;

if (result > 0.5) output = 1; // Above the line

else output = 0; // Covers both on the line and below

MessageBox.Show("x = " + test\_x[i].ToString() + " Y = " + test\_y[i].ToString() + " Output:" + output + "\n");

}

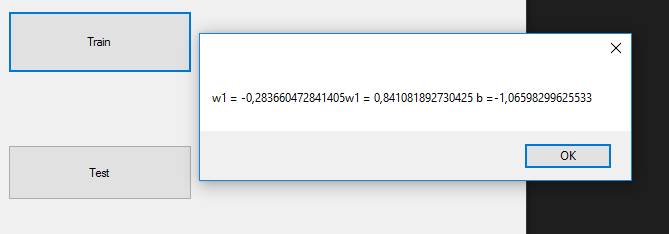
}

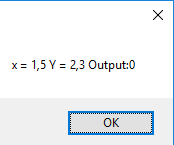
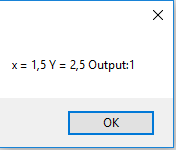
}

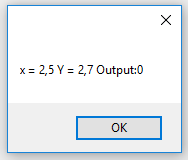
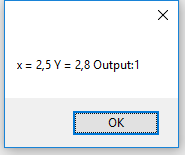
}

After training the neuron, we use the test data given and through computing the expected output we can classify if a point is either above or below the line. If the test data point multiplied by the trained weights is above 0.5 then the point is above. If less than 0.5, then it is below the line. Also, the more data we give it, the sharper it will get. I passed on a couple more points to test it and it seemed to be able to deal with different types of inputs.

We obtain the weights after going through the epochs. Then if tested, we get the correct output, a sequence of 0101 for the first 4 test data points.



CONCLUSION

In this assignment, we managed to review the concept of neuron in a practical approach and had to implement some of the mathematics that goes on in the background.

Finally, we learned how to implement our backpropagation algorithm for a classification problem.