ECE457B Assignment 4

David YeounJun Park
Student ID: 20434264

April 4, 2016

Problem 1

1)

$f(x) = \tanh(ax) = \dfrac{e^{ax} - e^{-ax}}{e^{ax} + e^{-ax}}$

For MLP,

$E(k) = \frac{1}{2} \sum_{i=1} [t_i(k) - O_i(k)]^2$, t = target value, O = output

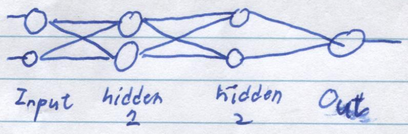
Using gradient descent,

$\Delta W_{ij}^{(L)}(n) = -\eta \frac{dE(n)}{dw} y_i(n)$, $y_i(n)$ = output of previous neuron, η = learning rate

$\frac{dE(n)}{dw} = e_j(n) \cdot \frac{df}{dx}$, $e_j(n)$ = error in output node j in the nth data point $= t_i(k) - O_i(k)$

$\frac{df}{dx} = a\,\mathrm{sech}^2(ax)$

$\therefore \Delta W_{ij}(n) = -\eta\,(t_i(k) - O_i(k)) \cdot a\,\mathrm{sech}^2(ax)$



Step 1. Initialize weights and thresholds to small random values

Step 2. Choose an input-output pattern from the training data set $(x(k), t(k))$

Step 3. Propagate the k^{th} signal forward through the network and compute the output values for all i neurons at every layer (l) using $o_i^l(k) = \tanh(\sum_{p=0}^{n_{l-1}} W_{ip}^l o_p^{(l-1)})$.

Step 4. Compute the total error value $E = E(k) + E$ and the error signal $\delta_i^{(L)}$ using the formula, $\delta_i^{(L)} = [t_i - O_i^{(L)}][\mathrm{sech}^2(tot)_i^{(L)}]$

Step 5. Update the weights according to $\Delta W_{ij}^{(l)} = \eta(t_i(n) - O_i(n))\,\mathrm{sech}^2(n)$.

Step 6. Repeat for epoch or convergence.

Step 7. Check for cumulative value.

Problem 2

$$f_a(x) = exp(-x^2)$$

$$f_b(x) = arctan(x)$$

Section A

The network uses sigmoid function. It has 1 input, 1 hidden layer with 3 nodes, and 1 output layer. The learning rate is 0.1, with 0 momentum.

The number of hidden nodes are fixed to 3 nodes. First, 5 points were generated with the even distribution for the range of each function for training. Then, 5 more points were generated with the even distribution for the range of each function for validation. Then the network was trained until convergence. The total errors were

$$Error_{tot}(f_a(x)) = 0.822$$

$$Error_{tot}(f_b(x)) = 0.0669$$

Then, 10 points were generated with the even distribution for the range of each function for the training. The validation list remained same. Then the network was trained until convergence.

$$Error_{tot}(f_a(x)) = 0.0233$$

$$Error_{tot}(f_b(x)) = 0.00169$$

Then, 15 points were generated with the even distribution for the range of each function for the training. The validation list remained same. Then the network was trained until convergence.

$$Error_{tot}(f_a(x)) = 0.198$$

$$Error_{tot}(f_b(x)) = 0.00113$$

Section B

There are 5 training points, evenly distributed within given limits for the function. This time, however, there are 100 validation points in the validation set, all evenly distributed within given limits for the function. The networks will be trained until convergence. The learning rate is 0.1 with 0 momentum

To find the best performance for function A, different number of nodes were tried.

$$Error_{tot}(f_a(x)) = 5.89, \text{number of node} = 1$$
$$Error_{tot}(f_a(x)) = 4.89, \text{number of nodes} = 2$$
$$Error_{tot}(f_a(x)) = 2.83, \text{number of nodes} = 3$$
$$Error_{tot}(f_a(x)) = 6.36, \text{number of nodes} = 4$$

Now, to find the best performance for function B, different number of nodes were tried.

$$Error_{tot}(f_a(x)) = 2.30, \text{number of node} = 1$$
$$Error_{tot}(f_a(x)) = 1.84, \text{number of nodes} = 2$$
$$Error_{tot}(f_a(x)) = 1.49, \text{number of nodes} = 3$$
$$Error_{tot}(f_a(x)) = 3.26, \text{number of nodes} = 4$$

As can be seen, the relationship between the number of nodes and the total error is quadratic. In both cases, it best performed when the number of nodes are 3.

Section C

As can be seen, both the relationships between the number of nodes and the number of training data are in quadratic relationship with the total error. The number of training data helps, but too much data can over train the network, causing it to memorize the training set instead to infer from the training data set. For function A, the "good" amount of training data points is 10 points. The effect of the number of node is similar to the effects of the number of points. Too much node can be hindrance. The "good" amount of nodes are 3 nodes, for both function A and B.

Problem 3

The network only has 1 hidden layer.

Section A

Training Data:

$$5 \text{ points from -1 to 1 for } x_1, 10 \text{ points from -1 to 1 for } x_2$$

Validation Data:

$$10 \text{ points from -1 to 1 for } x_1, 5 \text{ points from -1 to 1 for } x_2$$

Section B

The network was trained until the error value could not be decreased below 0.2. Mean Square Value was used to calculate the error. The error was 0.187 when tested against the validation set.

Section C

Now, to find the best performance for function B, different number of nodes were tried.

$$Error(f(x)) = 0.194, \text{number of node} = 1$$
$$Error(f(x)) = 0.187, \text{number of node} = 2$$
$$Error(f(x)) = 0.205, \text{number of node} = 3$$
$$Error(f(x)) = 0.185, \text{number of node} = 4$$
$$Error(f(x)) = 0.202, \text{number of node} = 5$$
$$Error(f(x)) = 0.196, \text{number of node} = 6$$

After number of node = 6, the network moved between 0.18 - 0.20 without converging to a single value. It can be concluded, however, that 4 nodes in the hidden network performed best.

Problem 4

The network is trained until it has less than 1% error. If the network could not achieve the targeted error, then "N/A" will be indicated in the table.

Section A

Error Percentage on Validation Set			
Number of Nodes in Hidden Layers	1 Hidden Layer	2 Hidden Layer	3 Hidden Layer
01	N/A	N/A	N/A
02	4.55%	0.00%	N/A
03	2.27%	4.55%	N/A
04	4.55%	0.00%	2.27%
05	4.55%	0.00%	0.00%
06	0.00%	2.27%	9.09%
07	0.00%	0.00%	4.55%
08	2.27%	4.55%	2.27%
09	2.27%	4.55%	6.82%
10	0.00%	0.00%	2.27%

The network best performed when there were 2 hidden layers with half of 0% error. From 1 node per layer to 10 nodes per layer, 2, 4, 5, 7, 10 nodes per layer showed 0% error value.

Section B

A) Class 1

B) Class 2

C) Class 3

Code Snippet

Listing 1: Solution to Question 4A

```
from pybrain.datasets            import
   ↪ ClassificationDataSet
from pybrain.utilities           import percentError
from pybrain.tools.shortcuts     import buildNetwork
from pybrain.supervised.trainers import BackpropTrainer
from numpy import genfromtxt, zeros

def main():
        # Get Data
        dataSets = genfromtxt('normalizedData.csv',
           ↪ delimiter=',')
        alldata = ClassificationDataSet(13, 1,
           ↪ nb_classes=3)
        for dataSet in dataSets:
                alldata.addSample(dataSet[1:14], int(
                   ↪ dataSet[0])-1)

        # Split the data
        tstdata_temp, trndata_temp = alldata.
           ↪ splitWithProportion(0.25)
        tstdata = ClassificationDataSet(13, 1,
           ↪ nb_classes=3)
        for n in range(0, tstdata_temp.getLength()):
                tstdata.addSample(tstdata_temp.
                   ↪ getSample(n)[0], tstdata_temp.
                   ↪ getSample(n)[1])
        trndata = ClassificationDataSet(13, 1,
           ↪ nb_classes=3)
        for n in range(0, trndata_temp.getLength()):
                trndata.addSample(trndata_temp.
                   ↪ getSample(n)[0], trndata_temp.
                   ↪ getSample(n)[1])
        trndata._convertToOneOfMany()
        tstdata._convertToOneOfMany()
        # Build Network
        fnn = buildNetwork(trndata.indim, 4, 4, 4,
           ↪ trndata.outdim)

        # Construct Trainer
        trainer = BackpropTrainer(fnn, trndata,
           ↪ learningrate=0.1)

        # Train
        while True:
                trainer.trainEpochs(1)
                trnresult = percentError(trainer.
                   ↪ testOnClassData(), trndata['class
                   ↪ '])
                print("Training␣Test␣Error:␣%5.2f%%" %
                   ↪ trnresult)
                if trnresult < 1:
                        break

        tstresult = percentError(trainer.
           ↪ testOnClassData(dataset=tstdata), tstdata
           ↪ ['class'])
        print("test␣error:␣%5.2f%%" % tstresult)

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

Listing 2: Feature Scaling Script

```
import os

fileName = "./dataFile.txt"
outputFileName = "./normalizedData.csv"

def writeToNewFile(newLines):
        with open(outputFileName, "w") as f:
                for newLine in newLines:
```

```
            for i in range(len(newLine)):
                f.write(str(newLine[i])
                  ↪ )
                if not i == (len(
                  ↪ newLine) - 1):
                    f.write(",")
            f.write("\n")

def normalize(newLines):
    for i in range(1,14):
        minimum = 10000000;
        maximum = 0;
        for j in range(len(newLines)):
            minimum = min(newLines[j][i],
              ↪ minimum)
            maximum = max(newLines[j][i],
              ↪ maximum)
        print(minimum, maximum)
        for j in range(len(newLines)):
            newLines[j][i] = round((
              ↪ newLines[j][i] - minimum)
              ↪ /(maximum - minimum), 8)
    return newLines


def readAndFormat():
    newLines = []
    with open(fileName) as f:
        content = f.read().split('\n')
        for line in content:
            formatedLine = []
            [formatedLine.append(float(
              ↪ intVal)) for intVal in
              ↪ line.split(",")]
            newLines.append(formatedLine)
            print(formatedLine)
    return newLines
```

```
def main():
        newLines = readAndFormat()
        newLines = normalize(newLines)
        [print(newLine) for newLine in newLines]
        writeToNewFile(newLines)

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