BRSU

Neural Networks
Assignment 5

Bastian Lang

November 14, 2015

This report contains the summary, ex3.1 and ex3.2 without any programming/plotting.

1 Outline

- Introduction
  - Three distinct characteristics of MLP:
    * Model of each neuron includes nonlinear activation function that is differentiable everywhere
    * One or more layers of hidden units
    * High degree of connectivity
  - Organization of the chapter
- Some Preliminaries
  - Signals in an MLP:
    * Function Signals: Propagating forward through the network
    * Error Signals: Propagating backwards through the network
  - Computations for each neuron:
    * Function signal
    * Gradient vector of error surface
  - Notation
- Back-Propagation Algorithm
  - Error signal at output neuron: $e_j(n) = d_j(n) - y_j(n)$
  - Instantaneous value of total error energy: $\xi(n) = 0.5 \sum_{j \epsilon C} e_j^2(n)$
  - Induced Local Field $v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n)$
  - Function signal $y_i(n) = \phi_j(v_j(n))$
  - Derivation of delta rule
  - Local gradient for output neuron: $\delta_j(n) = e_j(n) \phi'_j(v_j(n))$
  - Local gradient for hidden neuron: $\delta_j(n) = \phi'_j(v_j(n)) \sum_k \delta_k(n) w_{kj}(n)$
  - Weight correction: $\Delta w_{ji}(n) = \eta * \delta_j(n) * y_i(n)$
  - Two passes of Computation
    * Forward pass: Compute net output
    * Backward pass: Adjust weights according to error
  - Activation Function
    * Logistic Function
    * Hyperbolic tangent function

  - Rate of Learning
    * Including momentum term for stability
  - Sequential and Batch Modes of Training
    * Epoch: Complete presentation of training set
    * Sequential Mode of back-propagation learning: Update weights directly
    * Batch Mode of back-propagation learning: Update weights only after an epoch
  - Summary of the Back-Propagation Algorithm
    * Cycle for sequential updating:
      · Initialization
      · Presentations of Training Examples
      · Forward Computation
      · Backward Computation
      · Iteration

- Summary of the back-propagation algorithm
- XOR problem
  - Special case of classifying points in the unit hypercube
  - Elemental perceptron cannot solve XOR
  - Example model using McCulloch-Pits neurons
- Heuristics for making the back-propagation algorithm perform better
  - Methods to improve back-prop
    * Sequential versus batch update
    * Maximizing information content
      · Use of example that results in largest training error
      · Use of example that is radically different than previous ones
    * Activation Function
      · Better antisymmetric than non-symmetric
    * Target values
      · Offset of target values to lie within function range
    * Normalizing the inputs
    * Initialization
    * Learning from hints
    * Learning rates

2 MLP FOR XOR

I implemented the back propagation algorithm for the fixed 2-2-1-mlp after my other solutions were not working. But it turned out that my solution still is not able to adjust the weights correctly. I tried different learning rates and I even used 0.99 and 0.01 instead of 0 and 1 as desired outputs.
Following is the python code.
I started by implementing an OO solution using python, but python seemed to be doing some unexpected things, so I switched to Java and implemented backprop for dynamically sized MLPs. But as with the fixed solution it does not seem to work. I tried random sequential learning as well. I add the Java code as well. Because of these problems I did not spend any time on exercise 3.

2.1 PYTHON

```
# -*- coding: utf-8 -*-
"""
Created on Sat Nov 14 19:03:39 2015

@author: bastian
"""
import numpy as np

f = lambda x : 1 / ( 1 + np.exp(-x))
f_d = lambda x : f(x) * (1 - f(x))

class MLP:

    def __init__(self, initial_weights):
        # input to hidden
        self.w31 = initial_weights
        self.w41 = initial_weights
        self.w32 = initial_weights
        self.w42 = initial_weights
        # hidden to output
        self.w53 = initial_weights
        self.w54 = initial_weights
        # bias
        self.w30 = initial_weights
        self.w40 = initial_weights
        self.w50 = initial_weights

    def propagate(self, net_input):
        # input
```

```
        self.output_1 = net_input[0]
        self.output_2 = net_input[1]

        # hidden
        self.input_3 = self.w30+self.w31*self.output_1 + self.w32*self.output_2
        self.output_3 = f(self.input_3)
        self.input_4 = self.w40 + self.w41*self.output_1 + self.w42*self.output_2
        self.output_4 = f(self.input_4)

        #output
        self.input_5 = self.w50+self.w53*self.output_3 + self.w54*self.output_4
        self.output_5 = f(self.input_5)

        return self.output_5

    def backpropagate(self, net_input, desired_output, learning_rate):
        net_output = self.propagate(net_input)
        error = desired_output - net_output

        self.delta_5 = f_d(self.input_5) * error
        self.delta_4 = f_d(self.input_4) * self.w54 * self.delta_5
        self.delta_3 = f_d(self.input_3) * self.w53 * self.delta_5

        self.w50 = self.w50 + learning_rate*self.delta_5
        self.w53 = self.w53 + learning_rate*self.delta_5*self.output_3
        self.w54 = self.w54 + learning_rate*self.delta_5*self.output_4

        self.w40 = self.w40 + learning_rate * self.delta_4
        self.w42 = self.w42 + learning_rate * self.delta_4*self.output_2
        self.w41 = self.w41 + learning_rate * self.delta_4*self.output_1

        self.w30 = self.w30 + learning_rate * self.delta_3
        self.w32 = self.w32 + learning_rate * self.delta_3*self.output_2
        self.w31 = self.w31 + learning_rate * self.delta_3*self.output_1


mlp = MLP(0.5)
print mlp.propagate((1,1))
mlp.backpropagate((1,1),0.01, 0.1)

learning_rate = 0.1
for i in range(1000):
    mlp.backpropagate((1,1), 0.01, learning_rate)
    mlp.backpropagate((0,1), 0.99, learning_rate)
```

```
    mlp.backpropagate((1,0), 0.99, learning_rate)
    mlp.backpropagate((0,0), 0.01, learning_rate)
print mlp.propagate((0,0))
print mlp.propagate((1,0))
print mlp.propagate((0,1))
print mlp.propagate((1,1))
```

2.2 JAVA

```
package model;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class MLP {

    private final List<Neuron> inputLayer = new LinkedList<Neuron>();
    private final List<Neuron> hiddenLayer = new LinkedList<Neuron>();
    private final List<Neuron> outputLayer = new LinkedList<Neuron>();
    private final Neuron bias = new BiasNeuron();

    public MLP(int numberInputNeurons, int numberHiddenNeurons, int numberOutputNeurons,

        createInputLayer(numberInputNeurons);
        createFullyConnectedLayer(inputLayer, hiddenLayer, numberHiddenNeurons, initialW
        createFullyConnectedLayer(hiddenLayer, outputLayer, numberOutputNeurons, initialW

    }

    private void createInputLayer(int numberInput) {
        for (int i = 0; i < numberInput; i++) {
            Neuron inputNeuron = new InputNeuron();
            inputLayer.add(inputNeuron);
        }
    }

    private void createFullyConnectedLayer(List<Neuron> originLayer, List<Neuron> targetL
            double initialWeight) {
        for (int i = 0; i < layerSize; i++) {
            Neuron targetNeuron = new StandardNeuron();
            for (Neuron originNeuron : originLayer) {
                Connection connection = new Connection(originNeuron, targetNeuron, initia
                originNeuron.addOutgoingConnection(connection);
```

```
            targetNeuron.addIncomingConnection(connection);
        }
        Connection connection = new Connection(bias, targetNeuron, initialWeight);
        targetNeuron.addIncomingConnection(connection);
        targetLayer.add(targetNeuron);
    }
}

public List<Double> propagate(List<Double> input) {
    for (int i = 0; i < inputLayer.size(); i++) {
        Neuron inputNeuron = inputLayer.get(i);
        inputNeuron.setLastInducedLocalField(input.get(i));
        inputNeuron.activate();
    }

    propagateThroughLayer(hiddenLayer);
    propagateThroughLayer(outputLayer);

    ArrayList<Double> result = new ArrayList<Double>();
    for (Neuron neuron : outputLayer) {
        result.add(neuron.getLastOutput());
    }
    return result;
}

private void propagateThroughLayer(List<Neuron> layer) {
    for (Neuron neuron : layer) {
        neuron.computeInducedLocalField();
        neuron.activate();
    }
}

public void backpropagate(List<Double> input, List<Double> desiredOutput, double lea
    List<Double> netOutput = propagate(input);
    // Compute delta for output neurons and change incoming weights
    for (int i = 0; i < outputLayer.size(); i++) {
        double error = desiredOutput.get(i) - netOutput.get(i);
        Neuron neuron = outputLayer.get(i);
        neuron.setLastDelta(error * neuron.derivative());
        changeIncomingWeightsForNeuron(learningRate, neuron);
    }

    // Compute delta for hidden neurons and change their incoming weights
    for (Neuron neuron : hiddenLayer) {
```

```
            double summedDelta = 0;
            for (Connection outgoingConnection : neuron.getOutgoingConnections()) {
                summedDelta += outgoingConnection.getWeight() * outgoingConnection.getTa
            }
            neuron.setLastDelta(neuron.derivative() * summedDelta);
            changeIncomingWeightsForNeuron(learningRate, neuron);
        }
    }

    private void changeIncomingWeightsForNeuron(double learningRate, Neuron neuron) {
        for (Connection incomingConnection : neuron.getIncomingConnections()) {
            double weightChange = neuron.getLastDelta() * learningRate
                    * incomingConnection.getOriginNeuron().getLastOutput();
            incomingConnection.setWeight(incomingConnection.getWeight() + weightChange);
        }
    }

    @Override
    public String toString() {
        StringBuilder result = new StringBuilder();
        result.append("MLP:\nInput Layer:\n");
        for (Neuron neuron : inputLayer) {
            result.append(neuron.toString() + "\n");
        }
        result.append("\nHidden Layer:\n");
        for (Neuron neuron : hiddenLayer) {
            result.append(neuron.toString() + "\n");
        }
        result.append("\nOutput Layer:\n");
        for (Neuron neuron : outputLayer) {
            result.append(neuron.toString() + "\n");
        }
        return result.toString();
    }
}
package model;

import java.util.LinkedList;
import java.util.List;

public abstract class Neuron {

    private double lastInducedLocalField;
    private double lastDelta;
```

```java
private double lastOutput;
private final List<Connection> outgoingConnections;
private final List<Connection> incomingConnections;

public Neuron() {
    outgoingConnections = new LinkedList<Connection>();
    incomingConnections = new LinkedList<Connection>();
}

public void addOutgoingConnection(Connection connection) {
    outgoingConnections.add(connection);
}

public void addIncomingConnection(Connection connection) {
    incomingConnections.add(connection);
}

public List<Connection> getIncomingConnections() {
    return incomingConnections;
}

public List<Connection> getOutgoingConnections() {
    return outgoingConnections;
}

public double getLastDelta() {
    return lastDelta;
}

public double getLastOutput() {
    return lastOutput;
}

public double getLastInducedLocalField() {
    return lastInducedLocalField;
}

public abstract double activate();

public double computeInducedLocalField() {
    double result = 0;
    for (Connection connection : incomingConnections) {
        result += connection.getWeight() * connection.getOriginNeuron().getLastOutpu
    }
```

```
        setLastInducedLocalField(result);
        return result;
    }

    @Override
    public String toString() {
        StringBuilder result = new StringBuilder();
        result.append(String.format("Number of incomming connections: %d. ", getIncoming
        result.append("weights:[");
        for (Connection connection : incomingConnections) {
            result.append(connection.getWeight() + " ");
        }
        result.append("]");
        result.append(String.format("Number of outgoing connections: %d. ", getOutgoingC
        return result.toString();
    }

    public void setLastOutput(double lastOutput) {
        this.lastOutput = lastOutput;
    }

    public void setLastDelta(double lastDelta) {
        this.lastDelta = lastDelta;
    }

    public void setLastInducedLocalField(double lastInducedLocalField) {
        this.lastInducedLocalField = lastInducedLocalField;
    }

    public abstract double derivative();
}
package model;

public class InputNeuron extends Neuron {

    @Override
    public String toString() {
        StringBuilder result = new StringBuilder();
        result.append("Input Neuron: ");
        result.append(super.toString());
        return result.toString();
    }

    @Override
```

```java
    public double activate() {
        setLastOutput(getLastInducedLocalField());
        return getLastOutput();
    }

    @Override
    public double derivative() {
        System.out.println("Derivative of input neuron should never be needed.");
        return 0;
    }
}
package model;

public class StandardNeuron extends Neuron {

    @Override
    public double activate() {
        double activation = computeActivation();
        setLastOutput(activation);
        return activation;
    }

    private double computeActivation() {
        double activation = 1.0 / (1 + Math.exp(getLastInducedLocalField() * (-1)));
        return activation;
    }

    @Override
    public String toString() {
        StringBuilder result = new StringBuilder();
        result.append("Standard Neuron: ");
        result.append(super.toString());
        return result.toString();
    }

    @Override
    public double derivative() {
        return computeActivation() * (1 - computeActivation());
    }

}
package model;

public class BiasNeuron extends Neuron {
```

```java
    public BiasNeuron() {
        setLastInducedLocalField(1);
        setLastOutput(1);
    }

    @Override
    public double activate() {
        setLastOutput(1);
        return 1;
    }

    @Override
    public String toString() {
        StringBuilder result = new StringBuilder();
        result.append("Bias Neuron: ");
        result.append(super.toString());
        return result.toString();
    }

    @Override
    public double derivative() {
        System.out.println("Derivative for bias should never be needed");
        return 0;
    }

}
package model;

public class Connection {

    private Neuron originNeuron;
    private Neuron targetNeuron;
    private double weight;

    public Connection(Neuron originNeuron, Neuron targetNeuron, double initialWeight) {
        this.originNeuron = originNeuron;
        this.targetNeuron = targetNeuron;
        this.weight = initialWeight;
    }

    public Neuron getOriginNeuron() {
        return originNeuron;
    }
```

```java
    public double getWeight() {
        return weight;
    }

    public Neuron getTargetNeuron() {
        return targetNeuron;
    }

    public void setOriginNeuron(Neuron originNeuron) {
        this.originNeuron = originNeuron;
    }

    public void setTargetNeuron(Neuron targetNeuron) {
        this.targetNeuron = targetNeuron;
    }

    public void setWeight(double weight) {
        this.weight = weight;
    }

}
import java.util.Arrays;
import java.util.List;

import model.MLP;

/**
 *
 * @author bastian
 *
 */
public class Main {
    public static void main(String[] args) {
        MLP mlp = new MLP(2, 2, 1, 0.5);
        System.out.println(mlp);
        List<Double> input = Arrays.asList(0.0, 0.0);
        System.out.println(mlp.propagate(input));

        for (int i = 0; i < 10000; i++) {
            mlp.backpropagate(Arrays.asList(0.0, 0.0), Arrays.asList(0.01), 0.1);
            mlp.backpropagate(Arrays.asList(1.0, 0.0), Arrays.asList(0.99), 0.1);
            mlp.backpropagate(Arrays.asList(0.0, 1.0), Arrays.asList(0.99), 0.1);
            mlp.backpropagate(Arrays.asList(1.0, 1.0), Arrays.asList(0.01), 0.1);
```

```
        }

        System.out.println(mlp);
        System.out.println(mlp.propagate(Arrays.asList(0.0, 0.0)));
        System.out.println(mlp.propagate(Arrays.asList(1.0, 0.0)));
        System.out.println(mlp.propagate(Arrays.asList(0.0, 1.0)));
        System.out.println(mlp.propagate(Arrays.asList(1.0, 1.0)));
    }
}
```