NEURAL NETWORKS WITH JAVASCRIPT

SUCCINCTLY

BY JAMES MCCAFFREY

SUCCINCTLY EBOOK SERIES

Syncfusion

Neural Networks with JavaScript Succinctly

By
James McCaffrey

Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

Important licensing information. Please read.

Technical Reviewer: Chris Lee
Copy Editor: Courtney Wright
Acquisitions Coordinator: Tres Watkins, content development manager, Syncfusion, Inc.
Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

James McCaffrey works for Microsoft Research in Redmond, Wash. He holds a B.A. in psychology from the University of California at Irvine, a B.A. in applied mathematics from California State University at Fullerton, an M.S. in information systems from Hawaii Pacific University, and a doctorate in cognitive psychology and computational statistics from the University of Southern California. James enjoys exploring all forms of activity that involve human interaction and combinatorial mathematics, such as the analysis of betting behavior associated with professional sports, machine learning algorithms, and data mining.

Chapter 1 Getting Started

The best way to see what this e-book is all about is to take a look at the screenshot in Figure 1-1. The image shows an example of using a neural network written using the JavaScript language without any significant external libraries. The network predicts the size of the company (small, medium, or large) that a college student will work at after graduation, based on sex (male or female), GPA, major (such as finance), and parents' annual income.

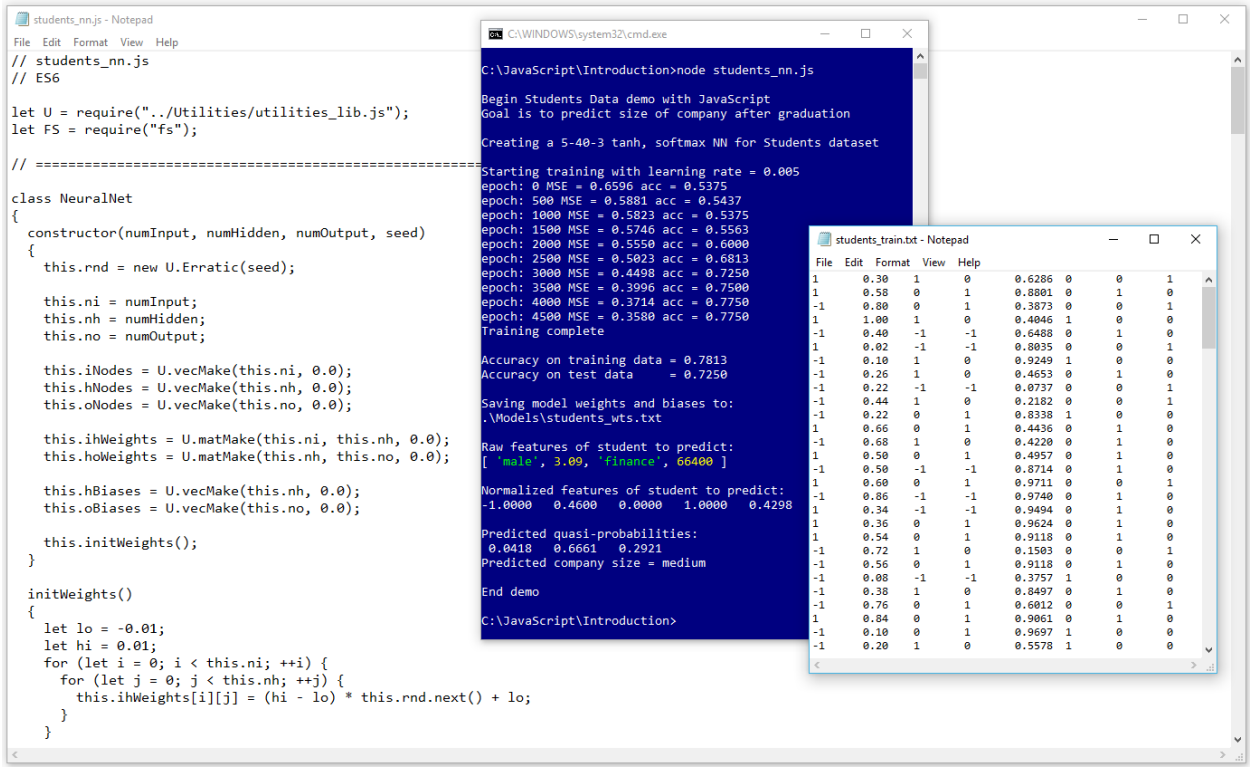


Figure 1-1: Neural Networks with JavaScript

The system shown in Figure 1-1 is running on a Windows machine, but the code and information in this e-book also applies to Linux and macOS machines. The system uses Notepad as the text editor, but you can use any editor you wish to edit the programs.

If you look at the shell carefully, you'll see that the program is hosted by the Node.js JavaScript environment. The code in this e-book was run on Node.js version 10.14.1, but any relatively recent version will work. The key programming language dependency is JavaScript ES6, also known as ECMAScript 6, ECMAScript 2015, and JavaScript 6. You can run the code examples in a web browser environment if the browser supports ES6, and if you refactor Node.js **`console.log()`** statements and use the HTML5 File API or the **`XMLHttpRequest`** for file reading and writing.

This e-book assumes you have intermediate or better programming skills, but does not assume you know anything about neural networks.

This e-book presents complete example programs for the three major types of neural network problems. A multiclass classification problem predicts a discrete value where there are three or more possible values, for example, predicting the size of company (small, medium, large). A regression problem predicts a single numeric value, for example, predicting a student's GPA. A binary classification problem predicts a discrete value where there are only two possible values, for example, predicting the sex of a student.

Everyone I know learns a new programming technology in the same way: you get an example program up and running and then experiment with the code. All the program code is presented in this e-book so you can copy and paste it, and run the demo programs. The data files used by the programs are in the appendix of this e-book. You can find the program code and data on GitHub. To prepare, all you have to do is install Node.js and use the code and data presented here. No other software is needed.

If you look at the screenshot in Figure 1-1 carefully, you'll see that there is a lot going on. Some of the fundamental concepts shown include neural network architecture, neural network input-output, tanh and softmax activation, back-propagation, error and accuracy, normalization and encoding, and model interpretation. This illustrates the major challenge for learning about neural networks: although most concepts are relatively simple, there are many concepts, and they interact with each other in nonobvious ways. But you can learn all important neural network concepts by running and examining the code in this e-book.

All the neural network examples in this e-book use a single hidden layer of nodes. Neural networks that have two or more hidden layers are called deep neural networks. In theory (the *Universal Approximation Theorem*, sometimes called the *Cybenko Theorem*), a neural network with a single hidden layer can compute anything that a neural network with multiple hidden layers can. But in practice, deep neural networks can solve some problems that simple neural networks cannot.

Other forms of deep neural networks use complex architectures. Examples include convolutional neural networks (often used to classify images) and recurrent neural networks (often used for natural language processing). Implementing a neural network from scratch gives you complete control over your system and enables you to have a complete understanding of your system.

The techniques used in this e-book can be extended to neural networks with two hidden layers. But for three or more hidden layers, or for deep neural networks with complex architectures, implementing from scratch is not practical in most situations. For these scenarios, you should use a deep neural network code library, such as TensorFlow, Keras, or PyTorch.

However, at my company, where we have hundreds of engineers working on deep neural systems, it's considered essential to have a solid foundation that can be best gained by understanding single, hidden-layer neural networks and implementing them from scratch. Much of the content presented in this e-book was developed for engineers who are new to the field of machine learning and neural networks.

Enough chit-chat already—let's start looking at some examples.

Chapter 2 Input and Output

One of several ways to think about a neural network (NN) is that it's a complex math function that accepts two or more numeric input values and emits one or more numeric output values. A solid understanding of the neural network input-output mechanism is essential in order to understand how a NN prediction system works.

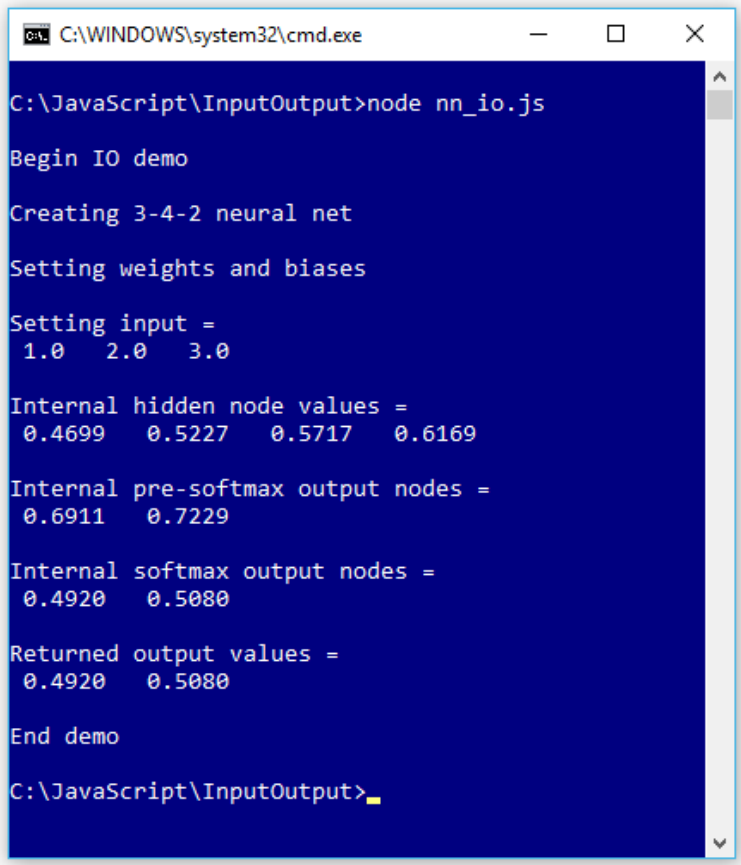


Figure 2-1: Neural Network Input-Output Demo

The screenshot in Figure 2-1 shows a demonstration of the NN input-output process. The demo program creates a 3-4-2 network, which means there are three input values, four so-called hidden nodes where most of the computations are performed, and two output values.

Behind the scenes, the demo program configures the NN by setting the values of 12 input-to-hidden weights, four hidden biases, eight hidden-to-output weights, and two output biases (a total of 26 weights and biases). Initializing the values of a network's weights and biases will be explained shortly.

After setting the weights and biases, the demo program sets up three input values: (1.0, 2.0, 3.0). These values are fed to the network, and the two final output values are (0.4920, 0.5080). Note that the output values sum to 1.0, which is not a coincidence.

The demo program displays the computed values of the four internal hidden nodes and the preliminary values of the output nodes (0.6911, 0.7229) before the application of an important function called softmax activation.

Understanding the input-output mechanism

Before looking at the demo code, it's important to understand the neural network input-output mechanism. The diagram in Figure 2-2 corresponds to the demo program.

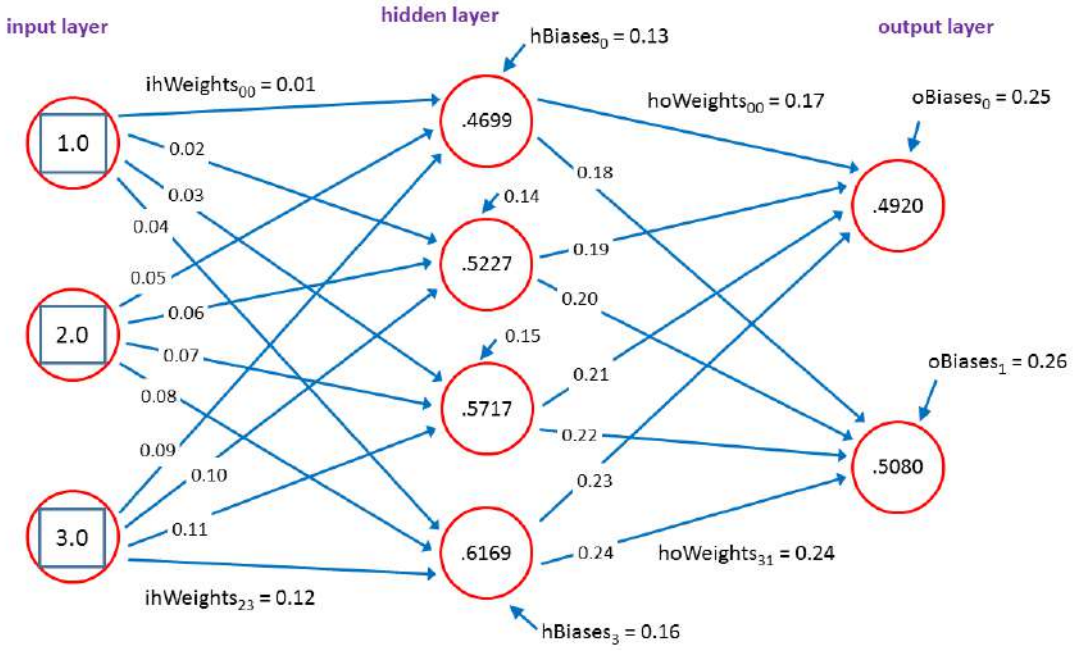


Figure 2-2: Neural Network Input-Output Calculations

The input node values are (1.0, 2.0, 3.0). Each blue line connecting input-to-hidden and hidden-to-output nodes represents a numeric constant called a weight. If nodes are zero-based indexed with node [0] at the top of the diagram, then the weight from input[0] to hidden[0] is 0.01, and the weight from hidden[3] to output[1] is 0.24. Weight values can be positive or negative.

Each hidden node and each output node (but not the input nodes) has an additional special weight called a bias. The bias value for hidden[3] is 0.16, and the bias for output[0] is 0.25.

Notice that if there are **ni** input nodes, **nh** hidden nodes, and **no** output nodes, then there are a total of (**ni** * **nh**) + (**nh** * **no**) + **nh** + **no** weights and biases. For the demo 3-4-2 neural net, there are (3 * 4) + (4 * 2) + 4 + 2 = 26 weights and bias values.

The first step in the input-output mechanism is to compute the values of the hidden nodes. Note that in most cases no processing occurs in the input nodes. For this reason, it's common to use different symbols for the input nodes (for example, squares inside circles) than for the hidden and output nodes (just circles). Additionally, the architecture of a neural network with a single hidden layer is usually called a two-layer neural network rather than a three-layer network.

To compute the value of a hidden node, you multiply each input value by its associated input-to-hidden weight, add the products up, then add the bias value, and then apply the hyperbolic tangent function (abbreviated tanh) to the sum. For hidden node [0] this is the following.

```
sum[0] = (1.0)(0.01) + (2.0)(0.05) + (3.0)(0.09) + 0.13
            = 0.5100

hidden[0] = tanh(0.5100)
               = 0.4699
```

The hyperbolic tangent function used in this way is called the hidden layer activation function. Two common alternatives are the logistic sigmoid function (often called just sigmoid) and rectified linear unit (ReLU). Using the **tanh** activation function forces all hidden node values to be between -1.0 and +1.0. Logistic sigmoid forces values between 0.0 and 1.0.

The output nodes are calculated similarly, but instead of using **tanh**, logistic sigmoid, or ReLU activation, a special function called softmax is used. The function is best explained by example. The preliminary sum of products plus bias values of **output[0]** and **output[1]** are the following.

```
sum[0] = (0.4699)(0.17) + (0.5227)(0.19) + (0.5717)(0.21) + (0.6169)(0.23) +
0.25
            = 0.6911

sum[1] = (0.4699)(0.18) + (0.5227)(0.20) + (0.5717)(0.22) + (0.6169)(0.24) +
0.26
            = 0.7229
```

A term called the divisor is computed by applying the **exp()** function to each **sum** term, and then summing those values.

```
divisor = exp(0.6911) +  exp(0.7229)
            = 1.9960 + 2.0604
            = 4.0563
```

The **exp(x)** function is x raised to Euler's number, approximately 2.71828. The result of softmax is the **exp()** of each term divided by the divisor term.

```
output[0] = 1.9960 / 4.0563
               = 0.4920

ouput[1] = 2.0604 / 4.0563
              = 0.5080
```

The purpose of softmax activation is to scale output values so that they sum to 1.0 and can be loosely interpreted as probabilities. Suppose the demo corresponded to a problem where the goal is to predict if a person is male or female based on three predictor variables, such as annual income, years of education, and height. If male is encoded as (1, 0) and female is encoded as (0, 1) then the prediction is female because the second output value (0.5080) is larger than the first (0.4920).

It's relatively uncommon to use (1, 0) and (0, 1) encoding for a binary classification problem, but I used this encoding in the explanation to match the demo neural network architecture.

The Input-Output demo program

The complete source code for the demo program is presented in Code Listing 2-1. The overall structure of the program is:

```
// nn_io.js
// several helper functions defined here.
class NeuralNet
{
 . . . // define NN
}

function main
{
  let nn = new NeuralNet(3, 4, 2);  // create NN.
  // set the weights and biases values.
  let X = [1.0, 2.0, 3.0];  // set input values.
  let oupt = nn.eval(X);
  console.log("Returned output values = ");
  vecShow(oupt, 4);
}

main();
```

The core neural network functionality is defined in an ES6 class named **`NeuralNet`**. Some of the class methods call external helper functions, such as **`matMake()`**, to construct a matrix, and **`hyperTan()`** to apply hyperbolic tangent hidden layer activation. All of the control logic is contained in a **`main()`** function.

Code Listing 2-1: Neural Network Input-Output Demo Program Code

```
// nn_io.js
// ES6
//
================================================================================
==

function vecMake(n, val)
{
  let result = [];
  for (let i = 0; i < n; ++i) {
    result[i] = val;
  }
  return result;
}
```

```
function matMake(rows, cols, val)
{
  let result = [];
  for (let i = 0; i < rows; ++i) {
    result[i] = [];
    for (let j = 0; j < cols; ++j) {
      result[i][j] = val;
    }
  }
  return result;
}

function vecShow(v, dec)
{
  for (let i = 0; i < v.length; ++i) {
    if (v[i] >= 0.0) {
      process.stdout.write(" ");
    }
    process.stdout.write(v[i].toFixed(dec));
    process.stdout.write("  ");
  }
  process.stdout.write("\n");
}

function matShow(m, dec)
{
  let rows = m.length;
  let cols = m[0].length;
  for (let i = 0; i < rows; ++i) {
    for (let j = 0; j < cols; ++j) {
      if (m[i][j] >= 0.0) {
        process.stdout.write(" ");
      }
      process.stdout.write(m[i][j].toFixed(dec));
      process.stdout.write("  ");
    }
    process.stdout.write("\n");
  }
}

function hyperTan(x)
{
  if (x < -20.0) {
    return -1.0;
  }
  else if (x > 20.0) {
    return 1.0;
  }
```

```
  else {
    return Math.tanh(x);
  }
}

function vecMax(vec)
{
  let mx = vec[0];
  for (let i = 0; i < vec.length; ++i) {
    if (vec[i] > mx) {
      mx = vec[i];
    }
  }
  return mx;
}

function softmax(vec)
{
  let mx = vecMax(vec);  // or Math.max(...vec)
  let result = [];
  let sum = 0.0;
  for (let i = 0; i < vec.length; ++i) {
    result[i] = Math.exp(vec[i] - mx);
    sum += result[i];
  }
  for (let i = 0; i < result.length; ++i) {
    result[i] = result[i] / sum;
  }
  return result;
}

//
==========================================================================
==

class NeuralNet
{
  constructor(numInput, numHidden, numOutput)
  {
    this.ni = numInput;
    this.nh = numHidden;
    this.no = numOutput;

    this.iNodes = vecMake(this.ni, 0.0);
    this.hNodes = vecMake(this.nh, 0.0);
    this.oNodes = vecMake(this.no, 0.0);

    this.ihWeights = matMake(this.ni, this.nh, 0.0);
    this.hoWeights = matMake(this.nh, this.no, 0.0);
```

```
  this.hBiases = vecMake(this.nh, 0.0);
  this.oBiases = vecMake(this.no, 0.0);
}

eval(X)
{
  let hSums = vecMake(this.nh, 0.0);
  let oSums = vecMake(this.no, 0.0);

  this.iNodes = X;

  for (let j = 0; j < this.nh; ++j) {
    for (let i = 0; i < this.ni; ++i) {
      hSums[j] += this.iNodes[i] * this.ihWeights[i][j];
    }
    hSums[j] += this.hBiases[j];
    this.hNodes[j] = hyperTan(hSums[j]);
  }
  console.log("\nInternal hidden node values = ");
  vecShow(this.hNodes, 4);

  for (let k = 0; k < this.no; ++k) {
    for (let j = 0; j < this.nh; ++j) {
      oSums[k] += this.hNodes[j] * this.hoWeights[j][k];
    }
    oSums[k] += this.oBiases[k];
  }

  console.log("\nInternal pre-softmax output nodes = ");
  vecShow(oSums, 4);

  this.oNodes = softmax(oSums);
  console.log("\nInternal softmax output nodes = ");
  vecShow(this.oNodes, 4);

  let result = [];
  for (let k = 0; k < this.no; ++k) {
    result[k] = this.oNodes[k];
  }
  return result;
} // eval()

setWeights(wts)
{
  // order: ihWts, hBiases, hoWts, oBiases
  let p = 0;

  for (let i = 0; i < this.ni; ++i) {
```

```
      for (let j = 0; j < this.nh; ++j) {
        this.ihWeights[i][j] = wts[p++];
      }
    }

    for (let j = 0; j < this.nh; ++j) {
      this.hBiases[j] = wts[p++];
    }

    for (let j = 0; j < this.nh; ++j) {
      for (let k = 0; k < this.no; ++k) {
        this.hoWeights[j][k] = wts[p++];
      }
    }

    for (let k = 0; k < this.no; ++k) {
      this.oBiases[k] = wts[p++];
    }
  } // setWeights()

} // NeuralNet

//
================================================================================
==

function main()
{
  process.stdout.write("\033[0m");  // reset
  process.stdout.write("\x1b[1m" + "\x1b[37m");  // bright white
  console.log("\nBegin IO demo ");

  console.log("\nCreating 3-4-2 neural net ");
  let nn = new NeuralNet(3, 4, 2);

  let wts = [
    0.01, 0.02, 0.03, 0.04, 0.05, 0.06,  // ihWeights
    0.07, 0.08, 0.09, 0.10, 0.11, 0.12,

    0.13, 0.14, 0.15, 0.16,  // hBiases

    0.17, 0.18, 0.19, 0.20,  // hoWeights
    0.21, 0.22, 0.23, 0.24,

    0.25, 0.26];  // oBiases

  console.log("\nSetting weights and biases ");
  nn.setWeights(wts);
```

```
  let X = [1.0, 2.0, 3.0];
  console.log("\nSetting input = ");
  vecShow(X, 1);

  let oupt = nn.eval(X);
  console.log("\nReturned output values = ");
  vecShow(oupt, 4);

  process.stdout.write("\033[0m");  // reset
  console.log("\nEnd demo");
}

main();
```

The **main()** function begins by using the **process.stdout.write()** function to send escape characters to set the shell font to bright-white for better readability, but this does not affect the functionality of the neural network.

Neural networks are relatively complex structures. This means there are many different ways to organize and define a neural network object.

Matrix and vector helper functions

A neural network implementation is based on vectors (numeric arrays) and matrices (numeric array-of-arrays). The demo program defines helper functions to instantiate and display vectors and matrices.

Creating a vector is implemented by the **vecMake()** function.

```
function vecMake(n, val)
{
  let result = [];
  for (let i = 0; i < n; ++i) {
    result[i] = val;
  }
  return result;
}
```

A statement like **let v = vecMake(3, 0.0)** creates a vector named **v** with three cells, each initialized to a 0.0 value. If you are new to JavaScript, note that there is only a single floating point-based numeric type (no integer type). Additionally, JavaScript arrays are more like lists in other programming languages.

When using the **++** increment operator in a standalone way, I prefer the prefix form (such as **++i**) rather than the more common postfix form (**i++**).

Creating a matrix is implemented by the **matMake()** helper function.

```
function matMake(rows, cols, val)
{
  let result = [];
  for (let i = 0; i < rows; ++i) {
     result[i] = [];
     for (let j = 0; j < cols; ++j) {
       result[i][j] = val;
     }
  }
  return result;
}
```

A statement such as **`let m = matMake(2, 3, 0.0)`** creates a matrix named **`m`** with two rows and three columns, where each of the six cells is initialized to a 0.0 value. The returned matrix is really an array-of-arrays rather than a true matrix, but it's convenient to think of it having rows and columns. Once created, the matrix can be used in an intuitive way, for example, the following.

```
let m = matMake(4, 3, 0.0);
m[0][2] = 5.5;
let x = m[2][2];
let nRows = m.length;  // 4 rows
let nCols = m[0].length;  // 3 columns
let row1 = m[1];  // entire row
```

The demo program defines two helper functions to display vectors and matrices. To display a vector to the shell, see the following.

```
function vecShow(v, dec)
{
  for (let i = 0; i < v.length; ++i) {
    if (v[i] >= 0.0) {
      process.stdout.write(" ");  // + or -
    }
    process.stdout.write(v[i].toFixed(dec));
    process.stdout.write("  ");
  }
  process.stdout.write("\n");
}
```

The function **`vecShow(v, dec)`** displays each cell in vector **`v`** using **`dec`** decimals. The function is quite crude, but gives you slightly nicer output than the **`console.log()`** function. You might want to add a third parameter to **`vecShow()`** to limit the number of values displayed on a line in situations with vectors with many cells. For example, the following code will print up to **`limit`** values on each line of the shell, and then print a new line.

```
function vecShow(v, dec, limit)
{
  for (let i = 0; i < v.length; ++i) {
    if (i > 0 && i % limit == 0) {
      process.stdout.write("\n");
    }
. . . (as before)
```

The function to display a matrix is as follows.

```
function matShow(m, dec)
{
  let rows = m.length;
  let cols = m[0].length;
  for (let i = 0; i < rows; ++i) {
    for (let j = 0; j < cols; ++j) {
      if (m[i][j] >= 0.0) {
        process.stdout.write(" ");
      }
      process.stdout.write(m[i][j].toFixed(dec));
      process.stdout.write("  ");
    }
    process.stdout.write("\n");
  }
}
```

One of the advantages of using your own lightweight code is that you can choose to add as much or as little error-checking code as you wish. The **`matShow()`** function has no error checking, which keeps the code small and easy to understand and modify. Adding error-checking code often increases the size of a code base significantly. For example, you could check parameter **`m`** to make sure it's not **`undefined`** or **`null`**, and you could check parameter **`dec`** to make sure it's positive and is integer-like.

```
function matShow(m, dec)
{
  if (typeof m == 'undefined') {
    console.log("OOPS");
  }
  if (m == null) {
    console.log("UGH");
  }
  if (dec < 0) {
    console.log("ARGH");
  }
  if (dec - dec.toFixed(0) != 0) {
    console.log("DANG");
  }
. . .
```

Even these basic parameter checks have nearly doubled the size of the function implementation. As a general rule of thumb, if you are the sole intended user of your code, you can get away with less error checking than if your code is intended for use by others.

Defining a neural network

The **NeuralNet** class constructor defines the key data structures for a single, hidden-layer neural network.

```
class NeuralNet
{
  constructor(numInput, numHidden, numOutput)
  {
    this.ni = numInput;
    this.nh = numHidden;
    this.no = numOutput;

    this.iNodes = vecMake(this.ni, 0.0);
    this.hNodes = vecMake(this.nh, 0.0);
    this.oNodes = vecMake(this.no, 0.0);

    this.ihWeights = matMake(this.ni, this.nh, 0.0);
    this.hoWeights = matMake(this.nh, this.no, 0.0);

    this.hBiases = vecMake(this.nh, 0.0);
    this.oBiases = vecMake(this.no, 0.0);
  }
. . .
```

A neural network defined with this code would be instantiated with a statement like the following.

```
let nn = new NeuralNet(3, 4, 2);
```

The **new** keyword transfers control to the associated **constructor()** method. Syntactically, notice that methods defined in an ES6 class are functions, but you do not use the **function** keyword.

Variables (more accurately "class members") **ni**, **nh**, and **no** hold the number of input, hidden, and output nodes. In JavaScript, all class variables have public scope—there is no private scope mechanism as found in languages like Java and C#.

The class vectors **iNodes**, **hNodes**, and **oNodes** are the input, hidden, and output nodes, respectively. The class matrix **ihWeights** holds the input-to-hidden weights where the first index represents the "from" input node, and the second index represents the "to" hidden node. For example, **ihWeights[0][3]** holds the weight value connecting input node [0] to hidden node [3].

Similarly, class matrix **hoWeights** holds the hidden-to-output weights where the first index represents the hidden node and the second index represents the output node. For example, **hoWeights[2][0]** holds the weight value connecting hidden node [2] to output node [0].

The class vector **hBiases** holds the biases values for the hidden nodes, with one value for each node, and the class vector **oBiases** holds the bias values for the output nodes.

Notice that the **NeuralNet** class definition assumes that helper functions **vecMake()** and **matMake()** are in the same source code file as the class definition. A more manageable design would place all the helper functions in a separate library file.

For example, suppose you placed all the helper functions in a separate file named Utilities_lib.js, like the following.

```
// file utilities_lib.js

function vecMake(n, val)
{
  let result = [];
  for (let i = 0; i < n; ++i) {
    result[i] = val;
  }
  return result;
}

function matMake(rows, cols, val)
{
. . .
// etc.

// -----------------------------------------------

module.exports = {
  vecMake,
  matMake,
. . .
// etc.
};
```

Then the helper functions could be accessed like in the following.

```
// file nn_io.js

let U = require("./utilities_lib.js");

class NeuralNet {
  constructor(numInput, numHidden, numOutput)
  {
    this.ni = numInput;
    this.nh = numHidden;
    this.no = numOutput;

    this.iNodes = U.vecMake(this.ni, 0.0);
    this.hNodes = U.vecMake(this.nh, 0.0);
    this.oNodes = U.vecMake(this.no, 0.0);
. . .
```

In summary, to create an external library file, write JavaScript code as usual, and add a **`module.exports`** statement at the end of the library file that lists the names of the defined functions. To access the external library file, use the **`require()`** function.

Setting values for weights and biases

The output values of a neural network depend on the input values and the values of the weights and biases. The demo program defines a class method **`setWeights()`** to assign values to the weights and biases.

```
setWeights(wts)
{
  // order: ihWts, hBiases, hoWts, oBiases
  let p = 0;

  for (let i = 0; i < this.ni; ++i) {
    for (let j = 0; j < this.nh; ++j) {
      this.ihWeights[i][j] = wts[p++];
    }
  }

  for (let j = 0; j < this.nh; ++j) {
    this.hBiases[j] = wts[p++];
  }

  for (let j = 0; j < this.nh; ++j) {
    for (let k = 0; k < this.no; ++k) {
      this.hoWeights[j][k] = wts[p++];
    }
  }
```

```
  for (let k = 0; k < this.no; ++k) {
    this.oBiases[k] = wts[p++];
  }
}
```

The method is called using these statements.

```
let wts = [
  0.01, 0.02, 0.03, 0.04, 0.05, 0.06,  // ihWeights
  0.07, 0.08, 0.09, 0.10, 0.11, 0.12,
  0.13, 0.14, 0.15, 0.16,  // hBiases
  0.17, 0.18, 0.19, 0.20,  // hoWeights
  0.21, 0.22, 0.23, 0.24,
  0.25, 0.26  // oBiases
];

nn.setWeights(wts);
```

The ability to set the values of weights and biases is useful for experimentation purposes. In a nondemo environment, the values of the weights and biases are first initialized using one of several algorithms, and then the values are updated during training. Training is the process of determining values of weights and biases so that computed output values closely match target output values in a set of training data that has known input values and known correct output values.

The strategy used by the **setWeights()** method is to accept a single vector that contains all the weights and biases, and then sequentially places each value into the input-hidden weights, followed by the hidden node biases, followed by the hidden-output weight, followed by the output node biases. For the input-hidden and hidden-output weights, values are stored in row-major order, that is, all of row [0], followed by all of row [1], etc.

The order in which the weights and biases values are placed into the network is arbitrary, but it's important to document the order used and be consistent. For example, after training a neural network, you may want to write the values of the trained network's weights and biases to file so that you can reconstruct the network at a later time without having to retrain the network.

The demo program does not define a corresponding **getWeights()** function. Such a function could be called like the following.

```
let wts = nn.getWeights();
```

Here's a possible implementation of a **getWeights()** method.

```
getWeights()
{
  // order: ihWts, hBiases, hoWts, oBiases
  let numWts = (this.ni * this.nh) + this.nh + (this.nh * this.no) + this.no;
  let result = vecMake(numWts, 0.0);
  let p = 0;
```

```
  for (let i = 0; i < this.ni; ++i) {
    for (let j = 0; j < this.nh; ++j) {
      result[p++] = this.ihWeights[i][j];
    }
  }

  for (let j = 0; j < this.nh; ++j) {
    result[p++] = this.hBiases[j];
  }

  for (let j = 0; j < this.nh; ++j) {
    for (let k = 0; k < this.no; ++k) {
      result[p++] = this.hoWeights[j][k];
    }
  }

  for (let k = 0; k < this.no; ++k) {
    result[p++] = this.oBiases[k];
  }

  return result;
}
```

The **getWeights()** function begins by creating a **result** vector with the appropriate number of cells. For a neural network with a single hidden layer with **ni** input nodes, **nh** hidden nodes, and **no** output nodes, there will be **ni** * **nh** input-hidden weights, **nh** * **no** hidden-output weights, **nh** hidden biases, and **no** output biases.

An alternative design choice for the **setWeights()** and **getWeights()** class methods is to pass four separate vectors as parameters, instead of serializing the input-hidden and hidden-output weights and the hidden and output biases. For example:

```
setWeights(ihWts, hBs, hoWts, oBs)
{
  // copy ihWts param values into this.ihWeights matrix.
  // copy hBs param values into this.hBiases vector.
  // copy hoWts param values into this.hoWeights matrix.
  // copy oBs param values into this.oBiases vector.
}
```

and

```
getWeights(ihWts, hBs, hoWts, oBs)
{
  // copy this.ihWeights matrix values into ihWts out-param.
  // copy this.hBiases vector values into hBs out-param.
  // copy this.hoWeights matrix values into hoWts out-param.
  // copy this.oBiases vector values into oBs out-param.
}
```

Using this design pattern gives you a bit more flexibility at the expense of a slightly less clean method interface.

Computing output values

The core class **`eval()`** method that computes output values (with **`console.log()`** statements removed) is defined as the following.

```
eval(X)
{
  let hSums = vecMake(this.nh, 0.0);
  let oSums = vecMake(this.no, 0.0);

  this.iNodes = X;

  for (let j = 0; j < this.nh; ++j) {
    for (let i = 0; i < this.ni; ++i) {
      hSums[j] += this.iNodes[i] * this.ihWeights[i][j];
    }
    hSums[j] += this.hBiases[j];
    this.hNodes[j] = hyperTan(hSums[j]);
  }

  for (let k = 0; k < this.no; ++k) {
    for (let j = 0; j < this.nh; ++j) {
      oSums[k] += this.hNodes[j] * this.hoWeights[j][k];
    }
    oSums[k] += this.oBiases[k];
  }

  this.oNodes = softmax(oSums);

  let result = [];
  for (let k = 0; k < this.no; ++k) {
    result[k] = this.oNodes[k];
  }

  return result;
}
```

The **eval()** method sets up local vectors **hSums** and **oSums** to hold the pre-activation sum of products for the hidden nodes and the output nodes, respectively. An alternative design is to compute the sums of products directly into class members **this.hNodes** and **this.oNodes**, but then you'd have to write code to reset their values all to 0.0 at the beginning of the **eval()** method because the sums of products are accumulated using the **+=** operator.

The input values in parameter **X** are assigned to class vector **this.iNodes** by reference. An alternative design, which can be useful if the input values need to be processed in some way, is to assign by value, for example, the following.

```
for (let i = 0; i < this.ni; ++i) {
  this.iNodes[i] = X[i];
}
```

A third design choice for dealing with the input values into a neural network is to eliminate the explicit network input nodes altogether and just use the **X** vector values directly, without copying those values into the network. Several deep neural network libraries use this implicit-input nodes approach.

The hidden node values are computed using the following statements.

```
. . .
  for (let j = 0; j < this.nh; ++j) {  // each hidden node.
    for (let i = 0; i < this.ni; ++i) {  // process each input node.
      hSums[j] += this.iNodes[i] * this.ihWeights[i][j];  // accumulate.
    }
    hSums[j] += this.hBiases[j];  // add the bias.
    this.hNodes[j] = hyperTan(hSums[j]);  // apply activation.
  }
. . .
```

As explained earlier, the product of each input node and its associated input-hidden weight is accumulated, then the hidden node bias value is added, and then an activation function (**tanh**) is applied to the sum.

A significantly different approach to the one used in the demo program is to use matrix operations. Accumulating the sum of products is essentially matrix multiplication. Therefore, if you had a helper function **matProduct(A, B)** that returned the result of matrix multiplication, then computing the values of the hidden nodes would resemble the following.

```
. . .
hSums = matProduct(this.ihWeights, X);
this.hNodes = vecHyperTan(hSums);
. . .
```

The matrix operations approach is used by all deep neural network code libraries so that they can take advantage of GPU processing. However, the matrix operations approach introduces quite a bit of additional complexity. For example, the input values **X** must be stored as an **n×1** matrix rather than as a more natural vector with **n** cells.

After computing the values of the hidden nodes, the output node values are computed using the following statements.

```
. . .
  for (let k = 0; k < this.no; ++k) {
    for (let j = 0; j < this.nh; ++j) {
      oSums[k] += this.hNodes[j] * this.hoWeights[j][k];
    }
    oSums[k] += this.oBiases[k];
  }
  this.oNodes = softmax(oSums);
. . .
```

The output nodes sums of products are accumulated using the values in the hidden nodes and the associated hidden-output weights, plus the output node bias values. Then, instead of applying an activation function to each accumulated sum, softmax activation is applied to the entire accumulated vector. The result of softmax activation applied to a vector is a vector with the same size as the input vector, where all values have been scaled so that they sum to 1.0, which allows those values to be loosely interpreted as probabilities.

The core **`eval()`** method concludes by copying the output node values into a vector, and returning that vector.

```
. . .
  let result = [];
  for (let k = 0; k < this.no; ++k) {
    result[k] = this.oNodes[k];
  }
  return result;
}  // eval()
```

In effect, the **`eval()`** method computes output values and returns the results in two ways: first, by storing into the internal **`this.oNodes`** vector, and second, by storing into an explicit return value. This approach is done for programming convenience, and allows **`eval()`** to be called like the following.

```
let oupt = nn.eval(X);  // results stored into this.oNodes and also returned.
// do something with the oupt vector.
```

The explicit return value could have been omitted. If so, the call to **`eval()`** would resemble the following.

```
nn.eval(X);  // results stored into this.oNodes.
let oupt = vecMake(numOutput);
for (let k = 0; k < numOutput; ++k) {
  oupt[k] = nn.oNodes[k];
}
// do something with oupt.
```

The demo program hard codes the hidden layer activation function (**tanh**) and the output layer activation function (**softmax**) used in the **eval()** method. A more flexible design would pass this information in as two parameters to the class constructor. For example, the constructor would look like the following.

```
class NeuralNet
{
  constructor(numInput, numHidden, numOutput, hiddenAct, outAct)
  {
. . .
```

And then creating a neural network would look like the following.

```
let nn = new NeuralNet(3, 4, 2, "tanh", "softmax");
```

The point is that when creating a neural network from scratch, you have many design options. These design options usually involve a tradeoff between implementation simplicity and calling flexibility.

Softmax activation

The three most common forms of neural network systems are NN regression, NN multiclass classification, and NN binary classification. In NN regression, the goal is to predict a single numeric value, for example, predicting the annual income of a person based on age, sex, years and of education.

In NN multiclass classification, the goal is to predict a discrete value where there are three or more possible values. For example, you might want to predict the political leaning of a person (conservative, moderate, or liberal) based on things such as age and annual income.

In NN binary classification, the goal is to predict a discrete value where there are exactly two possible values. For example, you might want to predict the sex of a person (male or female) based on things such as political leaning, age, and height. As it turns out, the programming techniques used for neural binary classification are quite a bit different from the programming techniques used for multiclass classification.

Softmax activation is used for neural multiclass classification. The idea is best explained by example. Suppose you want to predict the political leanings of a person (conservative, moderate, or liberal) based on their age, height, annual income, and years of education. You could implement a 4-10-3 neural network, meaning there are four input nodes (one for each predictor value), 10 hidden processing nodes, and three output nodes. The number of hidden nodes is a free parameter, also called a hyperparameter, which means you must determine the value using trial and error, experience, and intuition.

Suppose the input values into the trained neural network are X = (3.2, 7.0, 6.5, 2.0), which could correspond to a 32-year old person who is 70 inches tall, makes $65,000 per year, and has two years of education past high school. And suppose the three raw output values before softmax activation are (3.0, 5.0, 2.0).

The result of applying softmax to (3.0, 5.0, 2.0) is (0.1142, 0.8438, 0.0420). The resulting values sum to 1.0 and loosely represent the probabilities that the person is conservative, moderate, or liberal. Because the second value (0.8434) is largest, the prediction is that the person is a political moderate.

The math equation for softmax is shown in Figure 2-3. There are alternatives for output layer activation for neural classifiers, such as Taylor softmax and sparsemax, but softmax is by far the most common.

$$softmax(x_i) = \frac{e^{x_i}}{\sum_k e^{x_k}}$$

Figure 2-3: The Softmax Function

In words, the softmax of one of the values in a vector is the **exp()** of the value divided by the sum of the **exp()** function applied to each of the values in the vector. For (3.0, 5.0, 2.0), the **exp()** applied to each value and their sum is the following.

```
exp(3.0) =  20.0855
exp(5.0) =  148.4132
exp(2.0) =  7.3891

sum = 175.8878
```

Then the computed softmax values.

```
softmax(3.0) = 20.0855 / 175.8878 = 0.1142
softmax(5.0) = 148.4132 / 175.8878 = 0.8438
softmax(2.0) = 7.3891 / 175.8878 = 0.0420
```

A naive implementation of the softmax of a vector could be the following.

```
function softmax(vec)
{
  let result = [];
  let sum = 0.0;
  for (let i = 0; i < vec.length; ++i) {
    result[i] = Math.exp(vec[i]);  // find exp() of each value.
    sum += result[i];  // sum the exp()s
  }
  for (let i = 0; i < result.length; ++i) {
    result[i] = result[i] / sum;
  }
  return result;
}
```

Unfortunately, a naive implementation could easily cause arithmetic problems because **Math.exp(x)** can be astronomically large for even moderate values of **x**. For example, **Math.exp(200.0) = 7.23 × 1086**, which is much larger than the JavaScript Number type can handle.

One technique for greatly reducing the likelihood of arithmetic overflow is to use the max trick. The trick relies on the algebra facts that **exp(x + y) = exp(x) * exp(y)** and **exp(x - y) = exp(x) / exp(y)**.

The trick is to find the maximum value in the input vector, subtract the max value from each value in the input vector, and then compute softmax as usual on the differences.

For example, for an input of X = (3.0, 5.0, 2.0) the largest value is 5.0. Subtracting the max from each input value gives (-2.0, 0.0, -3.0). Then the following.

```
exp(-2.0) = 0.1353
exp(0.0)  = 1.0000
exp(-3.0) = 0.0498

sum = 1.1851
```

Then the computed softmax values on the differences.

```
softmax(3.0) = exp(-2.0) / sum = 0.1353 / 1.1851 = 0.1142
softmax(5.0) = exp(0.0) / sum  = 1.0000 / 1.1851 = 0.8438
softmax(2.0) = exp(-3.0) / sum = 0.0498 / 1.1851 = 0.0420
```

This is the same result as the direct calculation. Notice that by subtracting the max value, one modified value will be 0.0, and all other values will be negative. This prevents trying to compute Math.exp(x) for any value larger than x = 0.0.

The demo program implements the softmax function using the max trick.

```
function softmax(vec)
{
  let mx = vecMax(vec);  // or Math.max(...vec)
  let result = [];
  let sum = 0.0;
  for (let i = 0; i < vec.length; ++i) {
    result[i] = Math.exp(vec[i] - mx);  // use max trick.
    sum += result[i];
  }
  for (let i = 0; i < result.length; ++i) {
    result[i] = result[i] / sum;
  }
  return result;
}
```

```
function vecMax(vec)
{
  let mx = vec[0];  // assume first cell holds largest.
  for (let i = 0; i < vec.length; ++i) {  // check each cell.
    if (vec[i] > mx) {  // found a larger value.
      mx = vec[i];
    }
  }
  return mx;
}
```

Instead of defining a **vecMax()** function to return the largest value in a vector, you can use the quirky ES6 **...** spread operator (three consecutive period characters) with the built-in **Math.max()** function.

Hidden layer activation functions

The demo program uses the hyperbolic tangent function for activation on the hidden layer nodes. The function is implemented.

```
function hyperTan(x)
{
  if (x < -20.0) {
    return -1.0;
  }
  else if (x > 20.0) {
    return 1.0;
  }
  else {
    return Math.tanh(x);
  }
}
```

Notice that the program-defined **hyperTan(x)** function is just a wrapper around the built-in **Math.tanh(x)** function. To understand why the wrapper approach is used, take a look at the graph of the tanh function in Figure 2-4.

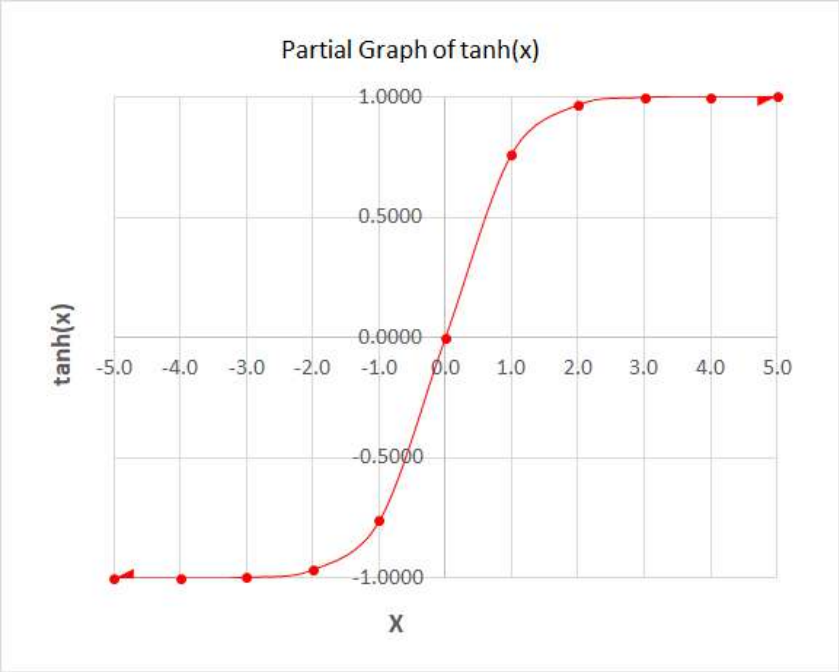


Figure 2-4: Graph of tanh(x) on [-5.0, +5.0]

The **tanh(x)** function accepts any value **x**, from negative infinity to positive infinity, and returns a value between -1.0 and +1.0. Notice that for **x** values less than -5.0 and greater than +5.0, the **tanh(x)** result is nearly at its extreme value.

Because pre-activation hidden node values of less than about -5.0 or greater than +5.0 result in -1.0 or +1.0, respectively, an effective neural network often has pre-activation hidden node values close to 0.0. This means that you should avoid very large or very small input values. This is accomplished by normalizing input data, as described in Chapter 4.

The limits of -20.0 and +20.0 used in the demo program implementation of the **hyperTan(x)** function are somewhat arbitrary. Other common limit values for a program-defined tanh wrapper function are (-10.0, +10.0) and (-40.0, 40.0). In practice, different tanh limit values have little effect in the neural network in which they're used.

Another common hidden layer activation function is the logistic sigmoid function. The name of the function is often shortened to log-sig or sigmoid in a neural network context (there are many different kinds of sigmoid functions in addition to logistic sigmoid). The math definition of the sigmoid function is the following.

$$y = 1 / (1 + e^{-x}) = 1 / (1 + \exp(-x))$$

The graph of the sigmoid function is shown in Figure 2-5. Notice that the sigmoid function has a shape similar to the tanh function. The sigmoid function accepts any value from negative infinity to positive infinity and returns a value between 0.0 and 1.0.

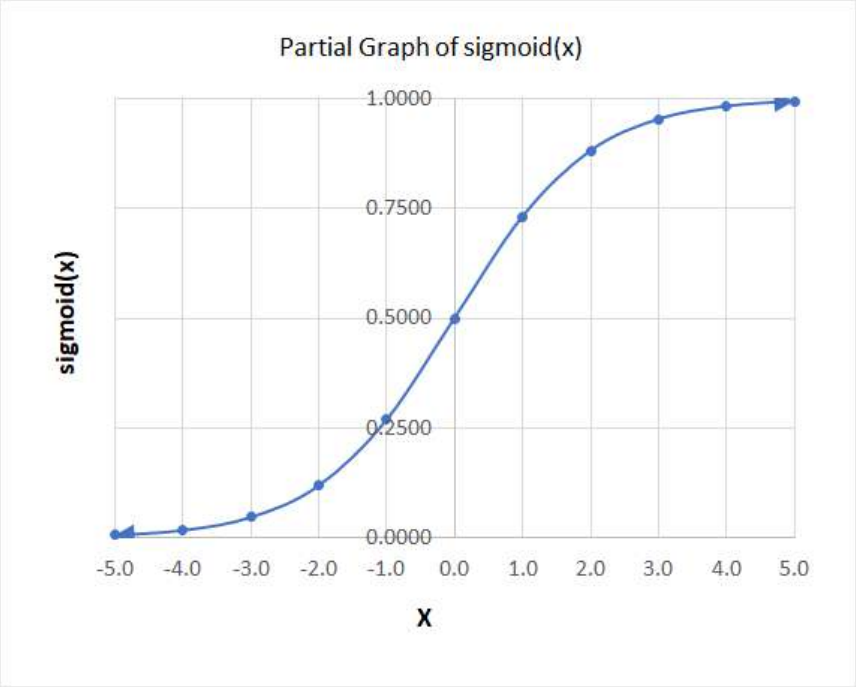


Figure 2-5: Graph of sigmoid(x) on [-5.0, +5.0]

One possible implementation of the logistic sigmoid function follows.

```
function logSig(x)
{
  if (x < -20.0) {
    return 0.0;
  }
  else if (x > 20.0) {
    return 1.0;
  }
  else {
    return 1.0 / (1.0 + Math.exp(-x));
  }
}
```

During neural network training, most training algorithms use the calculus derivative of the hidden layer activation function. One of the reasons the tanh function is often used for hidden layer activation is that the function has a very convenient derivative. If y = tanh(x), the derivative is y' = (1 - y) * (1 + y).

The derivatives of most math functions are expressed in terms of the input value x. For example, if y = x2 + sin(x) + 3x, then y' = 2x + cos(x) + 3. By an algebra coincidence, the derivative of y = tanh(x) can be expressed in terms of the calculated value y instead of x. This characteristic is useful, as explained in Chapter 3. The derivative of the sigmoid function is y' = y * (1 - y), which is also computationally convenient.

In the early days of neural networks, the sigmoid function was used most often for hidden layer activation. However, experience has shown that for many, but not all, problem scenarios, the tanh function tends to give a slightly better predictive model.

A generic deep neural network has two or more hidden layers. For deep neural networks, the rectified linear unit (ReLU) function is often used for hidden layer activation. ReLU activation is a topic that is outside the scope of this book.

Questions

1. How many weights and biases does a 4-10-3 neural network have?

2. Suppose you have a 2-2-2 neural network with X = (1.0, 2.0), input-hidden weights = (0.2, 0.4, 0.6, 0.8), hidden biases = (0.10, 0.12), hidden-output weights = (0.14, 0.16, 0.18, 0.20), output biases = (0.22, 0.24), hidden layer activation = tanh, output layer activation = none. What are the output node values?

3. Suppose you have a 5-8-3 neural network that uses softmax activation on the output nodes. If the pre-activation output nodes values are (1.5, 0.0, 2.5), what are the final output values?

4. Suppose a neural network uses sigmoid activation on the hidden layer. If a hidden node has pre-activation value of 2.0, what is the approximate value of the node after activation?

Chapter 3 Training

Training a neural network is the process of finding values for the network's weights and biases so that the network can make predictions. Training a neural network is complex and difficult, both conceptually and in practice.

The screenshot in Figure 3-1 shows a demo of neural network training. The goal of the program is to create a prediction model (a trained neural network is often called a model) for the well-known Iris data set.

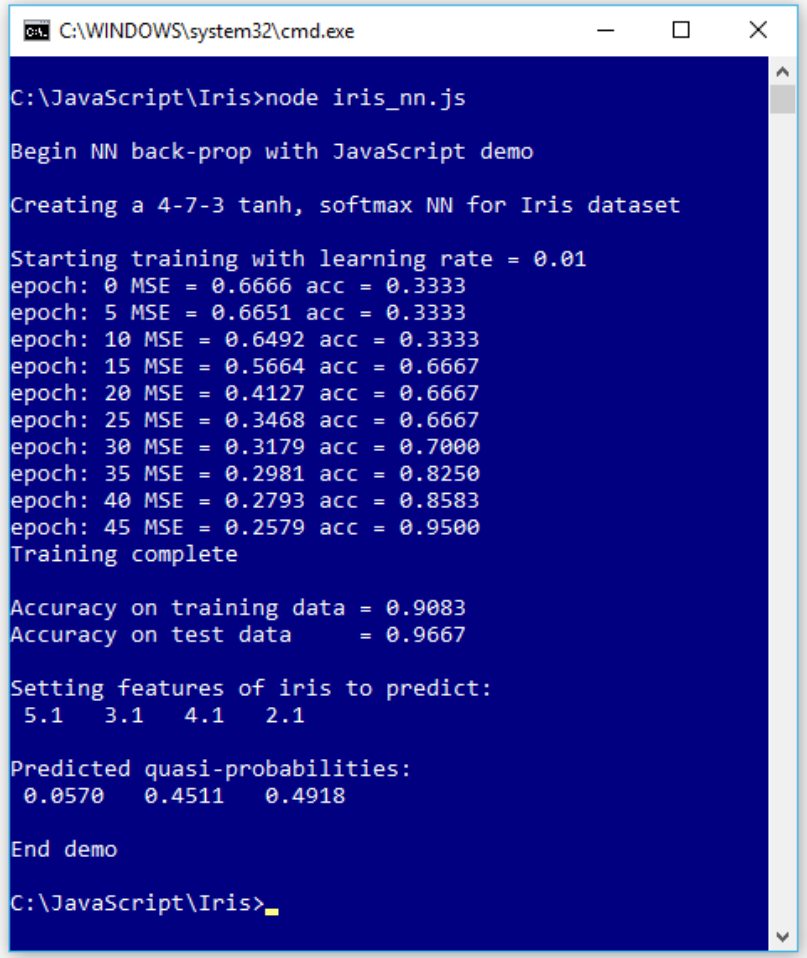


Figure 3-1: Neural Network Training on the Iris Data Set

The Iris data set has 150 data items. Each item represents an iris flower. There are four numeric predictor values (often called features in NN machine learning terminology) and three possible species to predict: setosa, versicolor, and virginica. The demo program creates a 4-7-3 neural network. There are four input nodes, one for each predictor value, and three output nodes, because there are three classes (discrete values to predict).

The neural network uses tanh activation on the hidden nodes and softmax output on the output nodes. Behind the scenes, the 150-item iris dataset had been split into two files: a 120-item (80% of the items) training data set to be used for determining the model weights and biases values, and a 30-item (the remaining 20%) test data set to be used to evaluate the quality of the trained network.

The demo program uses the back-propagation technique (also called stochastic gradient descent, SGD, even though the terms have somewhat different meanings) to train the network. Training is an iterative process. The demo program trains the network using 50 epochs. One epoch consists of processing every item in the training data set once.

During training, after each set of five epochs, the demo program displays the mean squared error associated with the current set of weights and biases, and the current prediction accuracy. As you can see in Figure 3-1, the error slowly decreases and the accuracy slowly increases, suggesting that training is working.

After training is completed, the demo computes and displays the prediction accuracy of the model on the 120-item training data (0.9083 = 109 out of 120 correct) and the 30-item test data (0.9667 = 29 out of 30 correct). The accuracy of the model on the test data set is a very rough estimate of the expected accuracy of the model on new, previously unseen data.

The demo program concludes by using the trained model to make a prediction. The demo sets up an iris flower with predictor values of (5.1, 3.1, 4.1, 2.1) and sends those values to the model. The model's output values are (0.0570, 0.4511, 0.4918). Because the third output value is the largest, the model predicts that the unknown flower is the third species (virginica).

Understanding the Iris data set

Fischer's Iris data set is one of the most well-known benchmark data sets in statistics and machine learning. The data set has 150 items. The raw data looks like this.

```
5.1, 3.5, 1.4, 0.2, setosa
7.0, 3.2, 4.7, 1.4, versicolor
6.3, 3.3, 6.0, 2.5, virginica
. . .
```

The raw data was prepared by one-hot encoding the class labels, but the feature values were not normalized as is usually done.

```
5.1, 3.5, 1.4, 0.2, 1, 0, 0
7.0, 3.2, 4.7, 1.4, 0, 1, 0
6.3, 3.3, 6.0, 2.5, 0, 0, 1
. . .
```

One-hot encoding is also known as 1-of-N encoding. The first class label gets a 1 in the first position, the second item gets a 1 in the second position, and so on. For example, if you are trying to predict the color of something, and there are four possible values (red, blue, green, yellow) then red = (1, 0, 0, 0), blue = (0, 1, 0, 0), green = (0, 0, 1, 0), and yellow = (0, 0, 0, 1).

The four predictor values are: sepal length, sepal width, petal length, and petal width. A sepal is a leaf-like structure.

After encoding, the full data set was split into a 120-item set for training and a 30-item test set to be used after training for model evaluation. The complete iris training and test data sets are presented in the appendix to this e-book.

Because the data has four dimensions, it's not possible to easily visualize it in a two-dimensional graph. But you can get a rough idea of the data from the partial graph in Figure 3-2.

As the graph shows, the Iris data set is quite simple. The **`setosa`** class can be easily distinguished from **`versicolor`** and **`virginica`**. Furthermore, the classes **`versicolor`** and **`virginica`** are nearly linearly separable. In spite of its simplicity, the Iris data set serves well as a classification example.

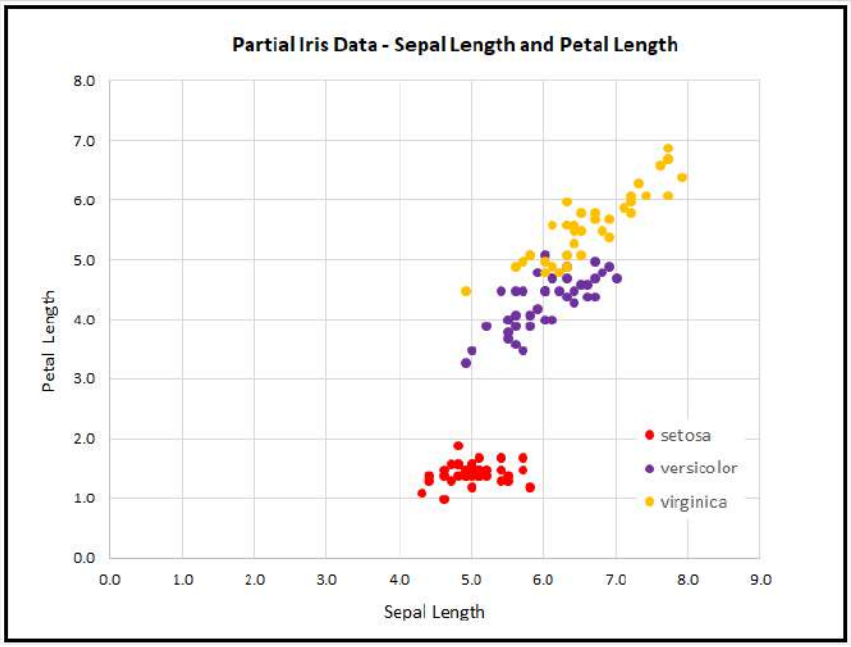


Figure 3-2: The Iris data set

By the way, there are actually at least two different versions of Fisher's Iris data set that are in common use. The original data was collected in 1935 and published by Fisher in 1936. However, at some point in time, a couple of the original values for setosa items were incorrectly transcribed, and years later made their way onto the internet. This isn't serious because the data set is now used just for a teaching example rather than for serious research.

The Iris data set demo program utilities

The complete source code for the demo program is presented in Code Listings 3-1 and 3-2. The demo uses a file named Utilities_lib.js, which contains helper functions. The demo program is organized so that there is a top-level directory named JavaScript with subdirectories named Utilities and Iris. The Iris directory contains the Iris_nn.js demo program and a subdirectory named Data, which holds the files Iris_train.txt and Iris_test.txt.

The overall structure of the program is as follows.

```
// iris_nn.js
// ES6
let U = require("../Utilities/utilities_lib.js");

class NeuralNet
{
  . . . // define NN.
}

function main
{
  // load training and test data into memory.
  let seed = 0;
  let nn = new NeuralNet(4, 7, 3, seed);  // create NN.
  // train the network.
  // evaluate the trained network.
  // use the trained network to make a prediction.
}

main();
```

It's possible to place all the utility functions in the same file as the core neural network functionality, but then the file becomes very large. When working with neural networks, you should not underestimate the importance of keeping your files organized.

Code Listing 3-1: File Utilities_lib.js Source Code

```
// utilities_lib.js
// ES6

let FS = require('fs');

function loadTxt(fn, delimit, usecols)
{
  let all = FS.readFileSync(fn, "utf8");  // giant string
  all = all.trim();  // strip final crlf in file.
  let lines = all.split("\n");
  let rows = lines.length;
  let cols = usecols.length;
```

```
  let result = matMake(rows, cols, 0.0);
  for (let i = 0; i < rows; ++i) {  // each line
    let tokens = lines[i].split(delimit);
    for (let j = 0; j < cols; ++j) {
      result[i][j] = parseFloat(tokens[usecols[j]]);
    }
  }
  return result;
}

function arange(n)
{
  let result = [];
  for (let i = 0; i < n; ++i) {
    result[i] = Math.trunc(i);
  }
  return result;
}

class Erratic
{
  constructor(seed)
  {
    this.seed = seed + 0.5;  // avoid 0
  }

  next()
  {
    let x = Math.sin(this.seed) * 1000;
    let result = x - Math.floor(x);  // [0.0,1.0)
    this.seed = result;  // for next call
    return result;
  }

  nextInt(lo, hi)
  {
    let x = this.next();
    return Math.trunc((hi - lo) * x + lo);
  }
}

function vecMake(n, val)
{
  let result = [];
  for (let i = 0; i < n; ++i) {
    result[i] = val;
  }
  return result;
}
```

```
function matMake(rows, cols, val)
{
  let result = [];
  for (let i = 0; i < rows; ++i) {
    result[i] = [];
    for (let j = 0; j < cols; ++j) {
      result[i][j] = val;
    }
  }
  return result;
}

function vecShow(v, dec, len)
{
  for (let i = 0; i < v.length; ++i) {
    if (i != 0 && i % len == 0) {
      process.stdout.write("\n");
    }
    if (v[i] >= 0.0) {
      process.stdout.write(" ");  // + or - space
    }
    process.stdout.write(v[i].toFixed(dec));
    process.stdout.write("  ");
  }
  process.stdout.write("\n");
}

function matShow(m, dec)
{
  let rows = m.length;
  let cols = m[0].length;
  for (let i = 0; i < rows; ++i) {
    for (let j = 0; j < cols; ++j) {
      if (m[i][j] >= 0.0) {
        process.stdout.write(" ");  // + or - space
      }
      process.stdout.write(m[i][j].toFixed(dec));
      process.stdout.write("  ");
    }
    process.stdout.write("\n");
  }
}

function argmax(v)
{
  let result = 0;
  let m = v[0];
  for (let i = 0; i < v.length; ++i) {
```

```
    if (v[i] > m) {
      m = v[i];
      result = i;
    }
  }
  return result;
}

function hyperTan(x)
{
  if (x < -20.0) {
    return -1.0;
  }
  else if (x > 20.0) {
    return 1.0;
  }
  else {
    return Math.tanh(x);
  }
}

function logSig(x)
{
  if (x < -20.0) {
    return 0.0;
  }
  else if (x > 20.0) {
    return 1.0;
  }
  else {
    return 1.0 / (1.0 + Math.exp(-x));
  }
}

function vecMax(vec)
{
  let mx = vec[0];
  for (let i = 0; i < vec.length; ++i) {
    if (vec[i] > mx) {
      mx = vec[i];
    }
  }
  return mx;
}

function softmax(vec)
{
  //let m = Math.max(...vec);  // or 'spread' operator.
  let m = vecMax(vec);
```

```
  let result = [];
  let sum = 0.0;
  for (let i = 0; i < vec.length; ++i) {
    result[i] = Math.exp(vec[i] - m);
    sum += result[i];
  }
  for (let i = 0; i < result.length; ++i) {
    result[i] = result[i] / sum;
  }
  return result;
}

module.exports = {
  vecMake,
  matMake,
  vecShow,
  matShow,
  argmax,
  loadTxt,
  arange,
  Erratic,
  hyperTan,
  logSig,
  vecMax,
  softmax
};
```

The Utilities_lib.js file contains 10 functions that work with vectors and matrices, one function named **loadTxt()** that reads data from a text file into memory, and one program-defined Erratic class to generate simple pseudo-random numbers.

The functions **vecMake()**, **matMake()**, **vecShow()**, and **matShow()** create and display JavaScript vectors and array-of-arrays style matrices.

The function **softmax(vec)** returns a vector of values scaled so that they sum to 1.0. For example, if **vec = (3.0, 5.0, 2.0)**, then **let sm = softmax(vec)** returns a vector **sm** holding (0.1142, 0.8438, 0.0420). The function **softmax()** calls function **vecMax()**, which returns the largest value in a vector. For example, if a vector contained values (4.0, 3.0, 7.0, 5.0) then **vecMax()** returns **7.0**.

The function **hyperTan(x)** returns the hyperbolic tangent of **x**. For example, **hyperTan(-8.5)** returns **-1.0**, **hyperTan(9.5)** returns **1.0**, and **hyperTan(1.1)** returns **0.8005**. The function **logSig(x)** returns the logistic sigmoid value of **x**.

The **argmax(vec)** function returns the index of its vector input parameter that has the largest value. For example, if a vector **v** contained values (4.0, 3.0, 7.0, 5.0), then **argmax(v)** returns **2**.

The function **arange(n) ("array-range")** creates and returns a vector with values **(0, 1, . . . n-1)**. For example, **let v = arange(5)** returns a vector **v** holding (0, 1, 2, 3, 4). The function is implemented.

```
function arange(n)
{
  let result = [];
  for (let i = 0; i < n; ++i) {
    result[i] = Math.trunc(i);
  }
  return result;
}
```

Because JavaScript does not have an integer data type, the values in the return vector are floating point values like (0.0, 1.0, 2.0, 3.0, 4.0), but it's conceptually convenient to think of the values as integers because they're used to represent array indices. Note that the call to the **Math.trunc()** function here has no significant effect, so it could have been omitted.

Reading data into memory

When training a neural network, the training and test data are usually stored as text files, so you need to read the data into memory. If your neural network code is client-side (in a browser), then you can use the **XMLHttpRequest** object to read from the server or the HTML5 **FileReader** object to read from the client machine. These techniques are outside the scope of this e-book.

If your neural network code is server-side and using Node.js, then you can use the Node.js File System library. The File System library is large (approximately 200 methods), and there are many ways to read a text file. The simplest approach when working with neural network data files is to use the **readFileSync()** function.

File utilities_lib.js implements a program-defined **loadTxt()** function. The implementation begins as follows.

```
let FS = require('fs');

function loadTxt(fn, delimit, usecols)
 {
  let all = FS.readFileSync(fn, "utf8");
  all = all.trim();  // strip away last newline char.
. . .
```

The File System library is globally available and is accessed through the **fs** alias. The function **loadTxt()** has three parameters: **fn**, which is the name of the text file to read (including the path); **delimit**, which is a string defining the character that separates the numeric values on each line; and **usecols**, which is an array of indices indicating which columns to read. For example, suppose a text file named Dummy.txt is saved with UTF-8 encoding and has four lines.

```
1.3, 1.1, 1.9, 1.7
2.6, 2.8, 2.2, 2.4
3.2, 3.3, 3.0, 3.8
```

Then the statement **`let m = loadTxt(".\\dummy.txt", ",", [0,2,3])`** would create a matrix **`m`** holding the following values.

```
1.30  1.90  1.70
2.60  2.20  2.40
3.20  3.00  3.80
```

The **`readFileSync()`** method reads the entire content of the text file into a variable as one giant string, including the newline characters at the end of each line. The File System library has a **`readFile()`** method that reads asynchronously, but the simpler **`readFileSync()`** works fine in most neural network scenarios.

The **`readFileSync()`** method assumes UTF-8 format by default, and also supports **`ascii`**, **`bae64`**, **`binary`**, **`hex`**, and **`latin1`** encoding. The **`trim()`** function removes the trailing newline character, so variable **`all`** holds the following.

```
"1.3, 1.1, 1.9, 1,7 \n 2.6, 2.8, 2.2, 2.4 \n 3.2, 3.3, 3.0, 3.8 \n"
```

The implementation continues.

```
. . .
  let lines = all.split("\n");
  let rows = lines.length;
  let cols = usecols.length;
  let result = matMake(rows, cols, 0.0);
```

The call to **`split("\n")`** separates the one large string into separate strings.

```
lines[0] = "1.3, 1.1, 1.9, 1,7"
lines[1] = "2.6, 2.8, 2.2, 2.4"
lines[2] = "3.2, 3.3, 3.0, 3.8"
```

The implementation concludes.

```
. . .
   for (let i = 0; i < rows; ++i) {  // each line
    let tokens = lines[i].split(delimit);
    for (let j = 0; j < cols; ++j) {
      result[i][j] = parseFloat(tokens[usecols[j]]);
    }
  }

  return result;
}
```

The call to **split(",")** peels off each numeric value, and then the call to **parseFloat()** converts the text representation of the current value into its numeric representation, which is then stored into the return-result matrix.

Reading data into memory for use by a neural network is simple in principle but is time-consuming and error-prone in practice. The demo code presented here can be used as a template for many neural network scenarios, but in situations where you need to write a different data reader, you should plan to spend several hours, or more, on development and testing.

Simple, pseudo-random numbers with an Erratic class

Neural networks use pseudo-random numbers when initializing weights and biases before training and to scramble the order in which training items are processed during training. One of the strange quirks of the JavaScript language is that the built-in **Math.random()** function does not have any way to set the random seed value, and therefore, there is no way to get reproducible results when using the function. Reproducible results are essential when working with neural networks.

Implementing a good pseudo-random number generator (where "good" means cryptographically secure and suitable for security-related usage) is extraordinarily difficult. Fortunately, neural networks don't require a sophisticated random number generator.

The Utilities_lib.js library contains a lightweight program-defined **Erratic** class. At a high level, in order to generate a pseudo-random value between 0.0 and 1.0, the idea is to compute the trigonometric sine function of the previous random value, and then peel off some of the trailing digits of the result.

The class is named Erratic rather than Random to emphasize the fact that the class is not a true pseudo-random number generator. The implementation looks like the following.

```
class Erratic
{
  constructor(seed) {
    this.seed = seed + 0.5;  // avoid 0
  }

  next() {
    // implementation here
  }

  nextInt(lo, hi) {
    // implementation here
  }
}
```

The class can be used as in the following.

```
let rnd = new Erratic(0);
let p = rnd.next();  // a pseudo-random value between [0.0, 1.0)
let ri = rnd.nextInt(0, 5);  // int-like value between [0, 5) = [0, 4]
```

The **next()** method returns a number in the range [0.0, 1.0), which means greater than or equal to 0.0 and strictly less than 1.0. The **nextInt(lo, hi)** method returns an integer-like number that is greater than or equal to **lo** and strictly less than **hi**. The **nextInt()** method is often used to select one cell of a vector. For example:

```
let v = vecMake(5, 0.0);  // a vector with 5 cells, each holding 0.0
let n = v.length;  // n is 5
let idx = rnd.nextInt(0, n);  // idx holds an index into v
```

The Erratic **constructor()** method accepts a seed value, which should be an integer-like number greater than or equal to 0. The internal **this.seed** value is set to the input seed plus 0.5 to avoid a seed of exactly 0, which would cause the **next()** method to generate a stream of only 0.0 values.

The **next()** method is implemented.

```
  next()
  {
    let x = Math.sin(this.seed) * 1000;
    let result = x - Math.floor(x);  // [0.0,1.0)
    this.seed = result;  // for next call
    return result;
  }
```

Initially, suppose a seed parameter of **0** was passed to the **constructor()** method. Then **this.seed** is 0.5 and **x** is computed as **sin(0.5) * 1000 = 0.479425538604203 * 1000 = 479.425538604203**. The **Math.floor()** function gives the integer part of its argument, and so **x - Math.floor(x) = 0.425538604203**. This value is returned by **next()**, and is also used as the new seed value the next time the **next()** method is called.

Notice that if the original seed value was exactly 0.0, then **sin(0.0) = 0.0** and **x - Math.floor(x) = 0.0**, and therefore, the generator would emit an unending sequence of 0.0 values.

It's very possible, but extremely unlikely, that after many thousands of calls to the **next()** method, a value of 0.0 could be generated. You could modify the **next()** method by adding a check.

```
  next()
  {
    let x = Math.sin(this.seed) * 1000;
    let result = x - Math.floor(x);  // [0.0,1.0)
    if (result == 0.0) {  // unlikely but possible
      throw("Zero error in call to Erratic.next()");
    }
    this.seed = result;  // for next call
    return result;
  }
```

There's usually a tradeoff between adding error-checking code and simplicity. The extent to which you'll want to add error checks will depend on your problem scenario.

It's convenient to have a method that returns an integer-like number, so the **Erratic** class defines a **nextInt()** method. The implementation is the following.

```
  nextInt(lo, hi)
  {
    let x = this.next();
    return Math.trunc((hi - lo) * x + lo);
  }
```

Suppose **lo** = 1 and **hi** = 7 (think the roll of a single dice). The internal call to **this.next()** will return a value in the range [0.0. 1.0). Multiplying that value by 7 - 1 = 6 will give a value in the range [0.0, 5.999999999]. Adding **1** will give a value in the range [1.0, 6.999999999]. Truncating will give a final value in the range [1, 6], or equivalently in the range [0, 7).

The Iris demo program code

The complete source code for the demo program is presented in Code Listing 3-2. The demo program begins by importing the functions in file Utilities_lib.js.

```
let U = require("../Utilities/utilities_lib.js");
```

The **require()** statement can handle either Windows-style file path strings with double backslash characters, or Linux-style strings with single forward slashes. Some programmers prefer to use the **const** keyword instead of **let** when importing a Node.js library using **require()**.

Code Listing 3-2: File Iris_nn.js Source Code

```
// iris_nn.js
// ES6

let U = require("../Utilities/utilities_lib.js");
```

```
//
================================================================================
==

class NeuralNet
{
  constructor(numInput, numHidden, numOutput, seed)
  {
    this.rnd = new U.Erratic(seed);

    this.ni = numInput;
    this.nh = numHidden;
    this.no = numOutput;

    this.iNodes = U.vecMake(this.ni, 0.0);
    this.hNodes = U.vecMake(this.nh, 0.0);
    this.oNodes = U.vecMake(this.no, 0.0);

    this.ihWeights = U.matMake(this.ni, this.nh, 0.0);
    this.hoWeights = U.matMake(this.nh, this.no, 0.0);

    this.hBiases = U.vecMake(this.nh, 0.0);
    this.oBiases = U.vecMake(this.no, 0.0);

    this.initWeights();
  }

  initWeights()
  {
    let lo = -0.01;
    let hi = 0.01;
    for (let i = 0; i < this.ni; ++i) {
      for (let j = 0; j < this.nh; ++j) {
        this.ihWeights[i][j] = (hi - lo) * this.rnd.next() + lo;
      }
    }
    for (let j = 0; j < this.nh; ++j) {
      for (let k = 0; k < this.no; ++k) {
        this.hoWeights[j][k] = (hi - lo) * this.rnd.next() + lo;
      }
    }
  }

  eval(X)
  {
    let hSums = U.vecMake(this.nh, 0.0);
    let oSums = U.vecMake(this.no, 0.0);

    this.iNodes = X;
```

```
  for (let j = 0; j < this.nh; ++j) {
    for (let i = 0; i < this.ni; ++i) {
      hSums[j] += this.iNodes[i] * this.ihWeights[i][j];
    }
    hSums[j] += this.hBiases[j];
    this.hNodes[j] = U.hyperTan(hSums[j]);
  }
  //console.log("\nHidden node values: ");
  //vecShow(this.hNodes, 4);

  for (let k = 0; k < this.no; ++k) {
    for (let j = 0; j < this.nh; ++j) {
      oSums[k] += this.hNodes[j] * this.hoWeights[j][k];
    }
    oSums[k] += this.oBiases[k];
  }

  this.oNodes = U.softmax(oSums);
  // console.log("\nPre-softmax output nodes: ");
  // vecShow(this.oNodes, 4);

  let result = [];
  for (let k = 0; k < this.no; ++k) {
    result[k] = this.oNodes[k];
  }
  return result;
} // eval()

setWeights(wts)
{
  // order: ihWts, hBiases, hoWts, oBiases
  let p = 0;

  for (let i = 0; i < this.ni; ++i) {
    for (let j = 0; j < this.nh; ++j) {
      this.ihWeights[i][j] = wts[p++];
    }
  }

  for (let j = 0; j < this.nh; ++j) {
    this.hBiases[j] = wts[p++];
  }

  for (let j = 0; j < this.nh; ++j) {
    for (let k = 0; k < this.no; ++k) {
      this.hoWeights[j][k] = wts[p++];
    }
  }
```

```
    for (let k = 0; k < this.no; ++k) {
      this.oBiases[k] = wts[p++];
    }
  } // setWeights()

  getWeights()
  {
    // order: ihWts, hBiases, hoWts, oBiases
    let numWts = (this.ni * this.nh) + this.nh + (this.nh * this.no) +
this.no;
    let result = vecMake(numWts, 0.0);
    let p = 0;
    for (let i = 0; i < this.ni; ++i) {
      for (let j = 0; j < this.nh; ++j) {
        result[p++] = this.ihWeights[i][j];
      }
    }

    for (let j = 0; j < this.nh; ++j) {
      result[p++] = this.hBiases[j];
    }

    for (let j = 0; j < this.nh; ++j) {
      for (let k = 0; k < this.no; ++k) {
        result[p++] = this.hoWeights[j][k];
      }
    }

    for (let k = 0; k < this.no; ++k) {
      result[p++] = this.oBiases[k];
    }
    return result;
  } // getWeights()

  shuffle(v)
  {
    // Fisher-Yates
    let n = v.length;
    for (let i = 0; i < n; ++i) {
      let r = this.rnd.nextInt(i, n);
      let tmp = v[r];
      v[r] = v[i];
      v[i] = tmp;
    }
  }

  train(trainX, trainY, lrnRate, maxEpochs)
  {
```

```
    let hoGrads = U.matMake(this.nh, this.no, 0.0);
    let obGrads = U.vecMake(this.no, 0.0);
    let ihGrads = U.matMake(this.ni, this.nh, 0.0);
    let hbGrads = U.vecMake(this.nh, 0.0);

    let oSignals = U.vecMake(this.no, 0.0);
    let hSignals = U.vecMake(this.nh, 0.0);

    let n = trainX.length;  // 120
    let indices = U.arange(n);  // [0,1,..,119]
    let freq = Math.trunc(maxEpochs / 10);  // when to print error.

    for (let epoch = 0; epoch < maxEpochs; ++epoch) {
      this.shuffle(indices);  //
      for (let ii = 0; ii < n; ++ii) {  // each item
        let idx = indices[ii];
        let X = trainX[idx];
        let Y = trainY[idx];
        this.eval(X);  // output stored in this.oNodes.

        // compute output node signals.
        for (let k = 0; k < this.no; ++k) {
          let derivative = (1 - this.oNodes[k]) * this.oNodes[k];  //
assumes softmax
          oSignals[k] = derivative * (this.oNodes[k] - Y[k]); // E=(t-o)^2
        }

        // compute hidden-to-output weight gradients using output signals.
        for (let j = 0; j < this.nh; ++j) {
          for (let k = 0; k < this.no; ++k) {
            hoGrads[j][k] = oSignals[k] * this.hNodes[j];
          }
        }

        // compute output node bias gradients using output signals.
        for (let k = 0; k < this.no; ++k) {
          obGrads[k] = oSignals[k] * 1.0;  // 1.0 dummy input can be
dropped.
        }

        // compute hidden node signals.
        for (let j = 0; j < this.nh; ++j) {
          let sum = 0.0;
          for (let k = 0; k < this.no; ++k) {
            sum += oSignals[k] * this.hoWeights[j][k];
          }
          let derivative = (1 - this.hNodes[j]) * (1 + this.hNodes[j]);  //
tanh
          hSignals[j] = derivative * sum;
```

```
        }

        // compute input-to-hidden weight gradients using hidden signals.
        for (let i = 0; i < this.ni; ++i) {
          for (let j = 0; j < this.nh; ++j) {
            ihGrads[i][j] = hSignals[j] * this.iNodes[i];
          }
        }

        // compute hidden node bias gradients using hidden signals.
        for (let j = 0; j < this.nh; ++j) {
          hbGrads[j] = hSignals[j] * 1.0;  // 1.0 dummy input can be
dropped.
        }

        // update input-to-hidden weights.
        for (let i = 0; i < this.ni; ++i) {
          for (let j = 0; j < this.nh; ++j) {
            let delta = -1.0 * lrnRate * ihGrads[i][j];
            this.ihWeights[i][j] += delta;
          }
        }

        // update hidden node biases.
        for (let j = 0; j < this.nh; ++j) {
          let delta = -1.0 * lrnRate * hbGrads[j];
          this.hBiases[j] += delta;
        }

        // update hidden-to-output weights.
        for (let j = 0; j < this.nh; ++j) {
          for (let k = 0; k < this.no; ++k) {
            let delta = -1.0 * lrnRate * hoGrads[j][k];
            this.hoWeights[j][k] += delta;
          }
        }

        // update output node biases.
        for (let k = 0; k < this.no; ++k) {
          let delta = -1.0 * lrnRate * obGrads[k];
          this.oBiases[k] += delta;
        }
      } // ii

      if (epoch % freq == 0) {
        let mse = this.meanSqErr(trainX, trainY).toFixed(4);
        let acc = this.accuracy(trainX, trainY).toFixed(4);

        let s1 = "epoch: " + epoch.toString();
```

```
        let s2 = " MSE = " + mse.toString();
        let s3 = " acc = " + acc.toString();
        console.log(s1 + s2 + s3);
      }

    } // epoch
  } // train()

  meanSqErr(dataX, dataY)
  {
    let sumSE = 0.0;
    for (let i = 0; i < dataX.length; ++i) {  // each data item
      let X = dataX[i];
      let Y = dataY[i];  // target output like (0, 1, 0)
      let oupt = this.eval(X);  // computed like (0.23, 0.66, 0.11)
      for (let k = 0; k < this.no; ++k) {
        let err = Y[k] - oupt[k]  // target - computed
        sumSE += err * err;
      }
    }
    return sumSE / dataX.length;
  } // meanSqErr()

  accuracy(dataX, dataY)
  {
    let nc = 0; let nw = 0;
    for (let i = 0; i < dataX.length; ++i) {  // each data item
      let X = dataX[i];
      let Y = dataY[i];  // target output like (0, 1, 0)
      let oupt = this.eval(X);  // computed like (0.23, 0.66, 0.11)
      let computedIdx = U.argmax(oupt);
      let targetIdx = U.argmax(Y);
      if (computedIdx == targetIdx) {
        ++nc;
      }
      else {
        ++nw;
      }
    }
    return nc / (nc + nw);
  } // accuracy()

} // NeuralNet

//
================================================================================
==
```

```
function main()
{
  process.stdout.write("\033[0m");  // reset
  process.stdout.write("\x1b[1m" + "\x1b[37m");  // bright white
  console.log("\nBegin NN back-prop with JavaScript demo ");

  // 1. load data
  // data looks like: 5.1, 3.5, 1.4, 0.2, 1, 0, 0
  let trainX = U.loadTxt(".\\Data\\iris_train.txt", ",", [0, 1, 2, 3]);
  let trainY = U.loadTxt(".\\Data\\iris_train.txt", ",", [4, 5, 6]);
  let testX = U.loadTxt(".\\Data\\iris_test.txt", ",", [0, 1, 2, 3]);
  let testY = U.loadTxt(".\\Data\\iris_test.txt", ",", [4, 5, 6]);

  // 2. create network
  console.log("\nCreating a 4-7-3 tanh, softmax NN for Iris dataset");
  let seed = 0;
  let nn = new NeuralNet(4, 7, 3, seed);

  // 3. train network
  let lrnRate = 0.01;
  let maxEpochs = 50;
  console.log("\nStarting training with learning rate = 0.01 ");
  nn.train(trainX, trainY, lrnRate, maxEpochs);
  console.log("Training complete");

  // 4. evaluate model
  let trainAcc = nn.accuracy(trainX, trainY);
  let testAcc = nn.accuracy(testX, testY);
  console.log("\nAccuracy on training data = " +
    trainAcc.toFixed(4).toString());
  console.log("Accuracy on test data     = " +
    testAcc.toFixed(4).toString());

  // 5. use trained model
  let unknown = [5.1, 3.1, 4.1, 2.1];  // set., set., ver., vir.
  let predicted = nn.eval(unknown);
  console.log("\nSetting features of iris to predict: ");
  U.vecShow(unknown, 1, 12);
  console.log("\nPredicted quasi-probabilities: ");
  U.vecShow(predicted, 4, 12);

  process.stdout.write("\033[0m");  // reset
  console.log("\nEnd demo");
} // main()

main();
```

The **NeuralNet** class **constructor()** accepts the number of input, hidden, and output nodes, and a seed value for an internal pseudo-random number generator.

```
class NeuralNet
{
  constructor(numInput, numHidden, numOutput, seed)
  {
    this.rnd = new U.Erratic(seed);
    this.ni = numInput;
    this.nh = numHidden;
    this.no = numOutput;
. . .
```

Notice that Erratic is defined in the Utilities_lib.js library file that's identified by **U**. Similarly, the calls to functions **vecMake()** and **matMake()** use the **U** identifier.

```
. . .
    this.iNodes = U.vecMake(this.ni, 0.0);
    this.hNodes = U.vecMake(this.nh, 0.0);
    this.oNodes = U.vecMake(this.no, 0.0);
    this.ihWeights = U.matMake(this.ni, this.nh, 0.0);
    this.hoWeights = U.matMake(this.nh, this.no, 0.0);
    this.hBiases = U.vecMake(this.nh, 0.0);
    this.oBiases = U.vecMake(this.no, 0.0);

    this.initWeights();

  } // constructor()
```

The last statement in the **constructor()** calls an **initWeights()** class method. This statement assigns small, random values to each of the network's weights and biases. Initialization of weights and biases is surprisingly subtle and important and will be explained shortly.

All of the program control logic is defined in a **main()** function. Program execution begins with the following.

```
function main()
{
  process.stdout.write("\033[0m");  // reset
  process.stdout.write("\x1b[1m" + "\x1b[37m");  // bright white
  console.log("\nBegin NN back-prop with JavaScript demo ");

  // 1. load data
  // data looks like: 5.1, 3.5, 1.4, 0.2, 1, 0, 0

  let trainX = U.loadTxt(".\\Data\\iris_train.txt", ",", [0,1,2,3]);
  let trainY = U.loadTxt(".\\Data\\iris_train.txt", ",", [4,5,6]);
  let testX = U.loadTxt(".\\Data\\iris_test.txt", ",", [0,1,2,3]);
  let testY = U.loadTxt(".\\Data\\iris_test.txt", ",", [4,5,6]);
. . .
```

When working with neural networks, it's common to load data into four matrices, as shown. It's good practice to add a comment in your code to explain the format of the data. The **loadTxt()** function does not support embedding comments in data files, but you can consider adding code to **loadTxt()** to allow comment lines.

The neural network is created like so.

```
  // 2. create network
  console.log("\nCreating a 4-7-3 tanh, softmax NN for Iris dataset");
  let seed = 0;
  let nn = new NeuralNet(4, 7, 3, seed);
. . .
```

Next, the network is trained with these statements.

```
  // 3. train network
  let lrnRate = 0.01;
  let maxEpochs = 50;
  console.log("\nStarting training with learning rate = 0.01 ");
  nn.train(trainX, trainY, lrnRate, maxEpochs);
  console.log("Training complete");
. . .
```

The **train()** method requires the known input values stored in the **trainX** matrix, and the known correct output values stored in the **trainY** matrix. The **lrnRate** (learning rate) variable controls how much each weight and bias value is changed on each training iteration. The **maxEpochs** variable sets a hard limit on the number of epochs, where an epoch consists of visiting and processing each training item one time.

Neural network training is extremely sensitive to the values used for the learning rate and the maximum number of training epochs. For example, a learning rate of 0.02 might work well, but a learning rate of 0.03 might cause the network to not learn at all.

The learning rate and maximum number of training epochs are called hyperparameters, which means you must specify their values and good values must be determined by trial and error. It's sometimes said that neural network expertise is part art and part science. The art component often refers to the idea that determining good hyperparameter values relies on experience.

During training, the **train()** method displays the values of the mean squared error and the classification accuracy, using the current values of the weights and biases. This is important because in nondemo problem scenarios, training failure is the rule rather than the exception. You want to identify situations where training is not working as quickly as possible.

After training, the trained model is evaluated.

```
  // 4. evaluate model
  let trainAcc = nn.accuracy(trainX, trainY);
  let testAcc = nn.accuracy(testX, testY);
  console.log("\nAccuracy on training data = " +
    trainAcc.toFixed(4).toString());
  console.log("Accuracy on test data     = " +
    testAcc.toFixed(4).toString());
. . .
```

The **accuracy()** method returns the classification accuracy of the model in decimal format, for example 0.9500, rather than in percentage format, like 95.00%. The demo program design has **accuracy()** as a class method, so it is called **let acc = nn.accuracy(trainX, trainY)**. An alternative design is to implement **accuracy()** as a standalone function, in which case it would be called **let acc = accuracy(nn, trainX, trainY)**.

Because the Iris data set problem is so simple, training is very quick. But realistic problems can require hours of training. The demo program does not save the trained model to file, but if training takes a long time, you'll usually want to save the values of the weights and biases to file.

The demo program concludes by making a prediction on a new, previously unseen iris flower.

```
  // 5. use trained model
  let unknown = [5.1, 3.1, 4.1, 2.1];  // set., set., ver., vir.
  let predicted = nn.eval(unknown);
  console.log("\nSetting features of iris to predict: ");
  U.vecShow(unknown, 1, 12);
  console.log("\nPredicted quasi-probabilities: ");
  U.vecShow(predicted, 4, 12);
. . .
```

The input values of (5.1, 3.1, 4.1, 2.1) are designed to be a challenge for the model. The first two predictor values (sepal length and width, 5.1, 3.1) are typical of those for the setosa species. The third predictor value (petal length, 4.1) is typical of versicolor, and the fourth predictor value (petal width, 2.1) is typical of virginica.

The **eval()** method returns a set of values (0.0570, 0.4511, 0.4918), and it's up to you to interpret the results. Because the third value is the largest, the model predicts the species of the unknown flower is the third species (virginica). If you wish, you can programmatically display the results.

```
  let species = ["setosa", "versicolor", "virginica"];
  let predIdx = U.argmax(predicted);
  let predSpecies = species[predIdx];
  console.log("Predicted species = " + predSpecies);
```

Even though the output values sum to 1.0 and can be loosely interpreted as probabilities, the values really aren't probabilities in the mathematical sense. However, the relative values do give you insights into how confident you can be of the prediction. In this case, because the pseudo-probabilities for **versicolor** and **virginica** are so close, you shouldn't have strong confidence in the prediction, compared to a result of something like (0.1000, 0.2000, 0.7000).

Weight and bias initialization

A neural network is very sensitive to the initial values of the weights and biases. The demo program uses the most basic form of initialization, which is to initialize all weights to small random values between -0.01 and +0.01, and to initialize all biases to 0.0 values. The class method **`initWeights()`** is defined.

```
initWeights()
{
  let lo = -0.01;
  let hi = 0.01;
  for (let i = 0; i < this.ni; ++i) {
    for (let j = 0; j < this.nh; ++j) {
      this.ihWeights[i][j] = (hi - lo) * this.rnd.next() + lo;
    }
  }

  for (let j = 0; j < this.nh; ++j) {
    for (let k = 0; k < this.no; ++k) {
      this.hoWeights[j][k] = (hi - lo) * this.rnd.next() + lo;
    }
  }
}
```

Because the neural network architecture is 4-7-3, there are 4 * 7 = 28 input-to-hidden weights and 7 * 3 = 21 hidden-to-output weights. Each of these 49 weights is initialized to a random value between **`lo`** = -0.01 and **`hi`** = +0.01. All of the 7 + 3 = 10 biases in the network are left initialized to their original 0.0 values assigned in the **`constructor()`** method.

The limit values of (-0.01, +0.01) used for the weights are hyperparameters. Fixed weight initialization limit values usually work well with simple, single-hidden-layer networks, but often don't work well with deep neural networks. Researchers have devised several new advanced neural network weight initialization algorithms, including Glorot uniform, Glorot normal, He uniform, He normal, and LeCun normal.

Even though these advanced initialization algorithms were designed for use with deep neural networks, they often work very well with simple neural networks. The Glorot uniform algorithm can be implemented as in the following.

```
  initGlorotUniform()
  {
    let fanIn = this.ni;
    let fanOut = this.nh;

    let variance = 6.0 / (fanIn + fanOut);
    let lo = -variance; let hi = variance;

    for (let i = 0; i < this.ni; ++i) {
      for (let j = 0; j < this.nh; ++j) {
        this.ihWeights[i][j] = (hi - lo) * this.rnd.next() + lo;
      }
    }

    fanIn = this.nh;
    fanOut = this.no;

    variance = 6.0 / (fanIn + fanOut);
    lo = -variance; hi = variance;

    for (let j = 0; j < this.nh; ++j) {
      for (let k = 0; k < this.no; ++k) {
        this.hoWeights[j][k] = (hi - lo) * this.rnd.next() + lo;
      }
    }
  } // initGlorotUniform()
```

If you examine the code, you'll see that instead of initializing weights to small random values between arbitrary limits, the Glorot uniform algorithm determines the limit values based on the architecture of the network. In addition to usually working well, the Glorot uniform algorithm for initialization eliminates two hyperparameters that you have to deal with.

Error and accuracy

Classification accuracy and model prediction error are closely related. The key high-level takeaway idea is that you use error during training, and you use accuracy after training.

Suppose you have only three iris data items for training, and at some point during training, the current input and output values are the following.

```
input               target (correct)   computed
-------------------------------------------------------
5.0  3.5  1.5  0.5   1  0  0           0.70  0.20  0.10
6.5  2.9  4.2  1.2   0  1  0           0.40  0.35  0.25
5.9  3.2  2.3  1.8   0  0  1           0.30  0.15  0.55
```

Accuracy is just the percentage of correct predictions. The input values and the known correct output values (often called the target values) are contained in the training data. The computed outputs are determined by the input values and the current values of the network weights and biases. For this data, the first and third items are correctly predicted, but the second item is incorrectly predicted. Therefore, the classification accuracy is 2 / 3 = 0.6667.

From a computational point of view, note that to determine if a set of computed output values gives a correct prediction, you can compare argmax(computed) and argmax(target) and check if the return indices are the same.

Accuracy is a relatively crude metric. You can imagine situations where the computed outputs are nearly perfect, for example, (0.9990, 0.0004, 0.0006) for the first item. And you can imagine situations where the computed outputs are just barely correct, for example, (0.3335, 0.3332, 0.3333) for the first item. But accuracy only counts correct-incorrect predictions.

In pseudo-code, the **NeuralNet** accuracy method is the following.

```
loop through each training data item
  computed = result of feeding item X vales to network
  target = item Y values
  if computed gives correct prediction
    increment correct counter
  else
    increment wrong counter
end-loop
return number correct / (number correct + number wrong)
```

The demo implementation.

```
  accuracy(dataX, dataY)
  {
    let nc = 0; let nw = 0;

    for (let i = 0; i < dataX.length; ++i) {  // each data item
      let X = dataX[i];
      let Y = dataY[i];  // target output like (0, 1, 0)

      let oupt = this.eval(X);  // computed output like (0.23, 0.66, 0.11)
      let computedIdx = U.argmax(oupt);
      let targetIdx = U.argmax(Y);

      if (computedIdx == targetIdx) {
        ++nc;
      }
      else {
        ++nw;
      }
    }

    return nc / (nc + nw);
  } // accuracy()
```

Compared to using a heavyweight neural network code library, such as TensorFlow or PyTorch, writing a lightweight implementation from scratch allows you to easily customize your code. For example, you might want to only count a prediction as correct if the computed output values are close to within some threshold value. Or you might want to insert **`console.log()`** messages to see exactly which input items are correctly predicted, and which are incorrectly predicted.

Mean squared error (MSE) is the average of the sum of the squared differences between the computed outputs and the correct target outputs. For example, the squared error terms for the three previous hypothetical data items are the following.

```
error(item 0) = (1 - 0.70)² + (0 - 0.20)² + (0 - 0.10)²  = 0.09 + 0.04 + 0.01
= 0.14
error(item 1) = (0 - 0.40)² + (1 - 0.35)² + (0 - 0.25)²  = 0.16 + 0.42 + 0.06
= 0.64
error(item 2) = (0 - 0.30)² + (0 - 0.15)² + (1 - 0.55)²  = 0.09 + 0.02 + 0.20
= 0.31
```

Therefore, `MSE = (0.14 + 0.64 + 0.31) / 3 = 1.09 / 3 = 0.36`.

Notice that, mathematically, it doesn't matter if you compute `(target - output)²` or `(output - target)²`. For example, `(1 - 0.30)² = 0.70² = 0.49` and `(0.30 - 1)² = -0.70² = 0.49`. However, the order does matter when using the back-propagation technique, which relies on error. By far the more common of the two approaches is to use `(target - output)²`, as the demo code does.

Additionally, instead of computing the error for each term and then summing those errors, you can just sum the squared differences between each component.

```
MSE = [ (1 - 0.70)² + (0 - 0.20)² + . . . + (1 - 0.55)² ] / 3 = (0.09 + 0.04 +
. . . +0.20) / 3 = 0.36
```

In pseudo-code, the **`NeuralNet`** error method is this.

```
sumError = 0
loop through each training data item
  computed = result of feeding item X vales to network
  target = item Y values
  for each component in target, computed
    sumError += squared difference between component
  end-for
end-loop
return sumError / number training items
```

The demo implementation is this.

```
  meanSqErr(dataX, dataY)
  {
    let sumSE = 0.0;
    for (let i = 0; i < dataX.length; ++i) {  // each data item
      let X = dataX[i];
      let Y = dataY[i];  // target output like (0, 1, 0)
```

```
      let oupt = this.eval(X);  // computed like (0.23, 0.66, 0.11)

      for (let k = 0; k < this.no; ++k) {  // each component of target and
computed
        let err = Y[k] - oupt[k]  // target - computed
        sumSE += err * err;
      }
    }

    return sumSE / dataX.length;
  }
```

Notice the parameter names for meanSqErr() are **dataX** and **dataY** rather than **trainX** and **trainY** or **testX** and **testY** to suggest that the function can be applied to either training data or test data. The statements **let X = dataX[i]** and **let Y = dataY[i]** retrieve a single row from the **dataX** and **dataY** matrices. The statements are just for clarity and the code could have used **dataX[i]** and **dataY[i]** directly.

Cross-entropy error

Cross-entropy error is a very important alternative to squared error. Cross-entropy error is also called log loss. The mathematical equation for cross-entropy error in the context of a neural network classifier is shown in Figure 3-3.

$$E = -\sum \ln(o_j) * t_j$$

Figure 3-3: Cross-Entropy Error

In words, the cross-entropy error (CEE) for a single data item is the negative sum of the log of each computed output component times the target component. The idea is best explained by example.

Suppose you have only three iris data items for training a 4-7-3 neural network, and at some point during training, the current input and output values are (as in the previous section) this.

```
input               target (correct)   computed
-----------------------------------------------------
5.0  3.5  1.5  0.5   1  0  0           0.70  0.20  0.10
6.5  2.9  4.2  1.2   0  1  0           0.40  0.35  0.25
5.9  3.2  2.3  1.8   0  0  1           0.30  0.15  0.55
```

The three cross-entropy error terms are the following.

```
error(item 0) = - [ ln(0.70) * 1  +  ln(0.20) * 0  + ln(0.10) * 0 ] = -
ln(0.70) = 0.36
error(item 0) = - [ ln(0.40) * 0  +  ln(0.35) * 1  + ln(0.25) * 0 ] = -
ln(0.35) = 1.05
error(item 0) = - [ ln(0.30) * 0  +  ln(0.15) * 0  + ln(0.55) * 1 ] = -
ln(0.55) = 0.60
```

Therefore, the mean cross-entropy error = `(0.36 + 1.05 + 0.60) / 3 = 2.01 / 3 = 0.67`.

Cross-entropy error is quite subtle and isn't as easy to understand as squared error. Notice that when cross entropy is used for neural network classification, because there is only one 1 component in the target vector, only one term contributes to the result.

In practical terms, the key idea is that cross-entropy error is a value where smaller values mean less error between computed outputs and correct target outputs. Unlike squared error, which will always be a value between 0.0 and 1.0, cross-entropy error will always be greater than or equal to 0.0, but doesn't have an upper limit.

One possible implementation of mean cross-entropy error is this.

```
  meanCrossEntErr(dataX, dataY)
  {
    let sumCEE = 0.0;  // sum of the cross-entropy errors.
    for (let i = 0; i < dataX.length; ++i) {  // each data item.
      let X = dataX[i];
      let Y = dataY[i];  // target output like (0, 1, 0).

      let oupt = this.eval(X);  // computed like (0.23, 0.66, 0.11).

      let idx = U.argmax(Y);  // find location of the 1 in target.
      sumCEE += Math.log(oupt[idx]);
    }

    return -1 * sumCEE / dataX.length;
  }
```

Because the log(0) is negative infinity, this implementation could fail, for example, if target = (1, 0, 0) and computed = (0.0000, 0.7500, 0.2500). In practice, getting a computed output that has a 0.0000 component value associated with the target 1 component is extremely unlikely. But you could add an error check like this.

```
  . . .
  let idx = U.argmax(Y);  // find location of the 1 in target.
  if (oupt[idx] == 0.0) { // a really wrong output component.
    sumCEE += 100.0;  // an arbitrary large value, or could throw an
Exception.
  }
  else {
    sumCEE += Math.log(oupt[idx]);
  }
```

As usual, adding error-checking code adds complexity. When writing your own lightweight neural network code, you can decide when and what type of error checking is needed.

Back-propagation training

Neural network training is the process of finding values for the network's weights and biases so that computed output values closely match the known, correct target values in the training data.

The back-propagation algorithm for neural network training is arguably one of the most important algorithms in the history of computer science. Back-propagation (often spelled as one word, backpropagation, due to a typo in a research paper decades ago that has been repeated ever since) is based on the general mathematical idea of stochastic gradient descent (SGD), and so the terms back-propagation and SGD are often used interchangeably.

Back-propagation is extraordinarily complex, but fortunately, a complete understanding of the theory is not necessary to implement the algorithm in practice. At a very high level, back-propagation training is as follows.

```
loop maxEpochs times
  for each training item
    get training item inputs X and correct target values Y
    compute output values using X  ("forward pass")

    (this is the "backward pass")
    compute the gradient of each hidden-output weight
    compute the gradient of each output node bias
    compute the gradient of each input-hidden weight
    compute the gradient of each hidden node bias

    use the gradients to adjust each weight and bias
  end-for
end-loop
```

Each neural network weight and bias has an associated value called a gradient. Technically, each value is a partial derivative of the weight with respect to the error function, but *gradient* is almost always used because it's much easier to say and write.

A gradient value has information about how the associated weight or bias value must change in order for computed output values to get closer to the target output values. This idea is illustrated in Figure 3-4. The graph represents one weight in a neural network. The possible values of the weight are shown on the horizontal axis at the bottom of the graph.

Each possible value of the weight will give a different error value. Therefore, the goal is to find the value of the weight that gives the smallest error. In the graph, you can see that the value of the weight that minimizes error is w = 5.0. Unfortunately, the error function is never completely known, so you can only take snapshots by sampling different values for w.

The gradient is the calculus derivative. A calculus derivative is the slope of the tangent line at some point. For example, if w = -5.0, you can see the error is about 7.0. The gradient at this point is -0.80 where the sign (negative in this case) indicates that the gradient runs from upper left to lower right. The magnitude of the gradient (0.80 in this case) indicates the steepness of the gradient.

If you examine the graph again, you can see than when w = -5.0 the gradient is negative, which means that to reduce error you must increase the value of w (towards 0).

The idea is also shown when w = 7.0. The gradient at that point is +2.15, and because the gradient is positive, to reduce the error you would decrease the value of w.

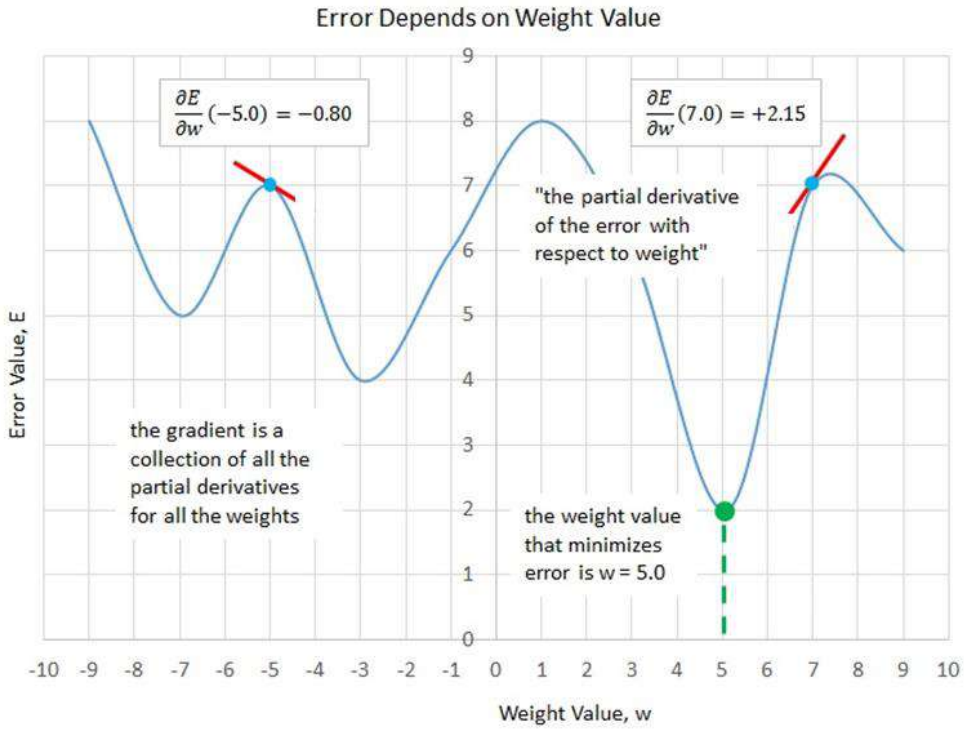


Figure 3-4: The Gradient of a Neural Network Weight

To summarize, if you can somehow compute the values of the gradients of each weight and bias, you can adjust the values of each weight and bias to reduce the error between computed output values and known correct output values. If the gradient is negative, increase the weight. If the gradient is positive, decrease the weight. Okay, but how can you compute gradients?

It turns out that calculating the gradients for the output node biases and the hidden-to-output weight is relatively simple, but calculating the gradients for the hidden node biases and the input-to-hidden weights is quite tricky.

Expressed as a math equation, the technique for updating the hidden-to-output weights and the output node biases is shown in Figure 3-5.

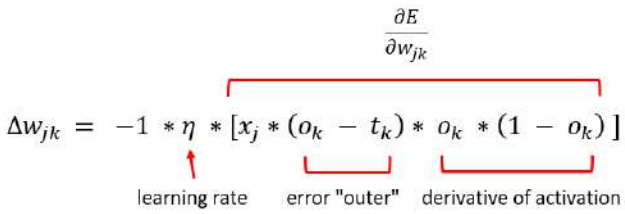


Figure 3-5: The Hidden-to-Output Weights Update Equation

The delta-w represents an increment of the weight from hidden node j to output node k that will decrease error. The -1 factor is an algebra consequence so that the delta value is added to the hidden-output weight rather than subtracted.

Greek letter eta (resembles a lower-case script "n") is the learning rate, which is a small value, typically something like 0.01 or 0.05. The x-j term is the value in the hidden node associated with the weight connecting hidden node j to output node k.

The (o - t) term ("output" minus "target") is part of a calculus derivative of the error function, in this case mean squared error. Notice that the back-propagation algorithm does not explicitly compute the error. Instead, the calculus derivative of the error function is used. If you use an error function other than mean squared error, the weight update equation changes.

The o * (1 - o) term is the calculus derivative of the output activation function, in this case the softmax function. If you use a different output activation function, the weight update equation changes. A technical exception to the fact that a change in activation function requires a change to the update calculation is that a neural network that uses logistic sigmoid output node activation has the same weight update equation as the equation assuming softmax. This happens because, mathematically, the softmax function and the logistic sigmoid function are very closely related.

Implementing back-propagation

Most of my software developer colleagues believe that the best way to understand back-propagation is to look at the code. The demo **`NeuralNet`** class implementation begins with the following.

```
 train(trainX, trainY, lrnRate, maxEpochs)
 {
   let hoGrads = U.matMake(this.nh, this.no, 0.0);
   let obGrads = U.vecMake(this.no, 0.0);
   let ihGrads = U.matMake(this.ni, this.nh, 0.0);
   let hbGrads = U.vecMake(this.nh, 0.0);

   let oSignals = U.vecMake(this.no, 0.0);
   let hSignals = U.vecMake(this.nh, 0.0);
. . .
```

Because there is a gradient value for each weight and bias in the network, it makes sense to store those gradient values in matrices and vectors that have the same shape as the weights and the biases. For instance, **hoGrads[3][0]** holds the gradient for **hoWeights[3][0]**, that is, the weight connecting hidden node [3] to output node [0].

The **oSignals** and **hSignals** vectors are scratch variables for computing the gradients, as you'll see shortly. The implementation continues as in the following.

```
  let n = trainX.length;  // 120
  let indices = U.arange(n);  // [0,1,..,119]
  let freq = Math.trunc(maxEpochs / 10);  // when to print error.
. . .
```

Variable **n** is the number of training items, 120 in this demo. The vector **indices** uses the **arange()** helper function to create a vector containing values (0, 1, . . n-1), which represent the indices of each training item. The idea here is that it's important in most cases to visit the training items in a different order every pass through the data set.

Variable **freq** (frequency) is a convenience to specify when to compute and print error and accuracy during training. In most nondemo situations, you don't want to compute and print these values for every training item because computing error and accuracy is computationally expensive, requiring iteration through the entire data set.

The 10 in the denominator means error and accuracy will be computed and displayed 10 times during training. For example, if **maxEpochs** is 350, then there will be information displayed 10 times (once every 35 epochs).

The main training loop begins with the following.

```
  for (let epoch = 0; epoch < maxEpochs; ++epoch) {
    this.shuffle(indices);
    for (let ii = 0; ii < n; ++ii) {  // each item
      let idx = indices[ii];

      let X = trainX[idx];
      let Y = trainY[idx];

      this.eval(X);  // output stored in this.oNodes.
. . .
```

Each epoch represents one complete pass through the training data. The demo code scrambles the order of the training items with the statement **this.shuffle(indices)**. The class method **shuffle()** uses the Fisher-Yates algorithm to randomly rearrange the order of the index values in vector **indices**. The method is defined as follows.

```
  shuffle(v)
  {
    let n = v.length;

    for (let i = 0; i < n; ++i) {
      let r = this.rnd.nextInt(i, n);  // pick a random index.

      let tmp = v[r];  // exchange values.
      v[r] = v[i];
      v[i] = tmp;
    }
  }
```

The **shuffle()** method walks through its vector argument **v** and exchanges the values at randomly selected indices. The algorithm is surprisingly subtle. For example, incorrectly coding the statement **this.rnd.nextInt(i, n)** as **this.rnd.nextInt(0, n)** will generate strongly biased results. Also, notice that the loop condition **i < n** can be changed to **i < n-1** because the last iteration exchanges the value in the last cell of vector **v** with itself.

The hidden-to-output weight gradients are computed by the following statements.

```
  // compute output node signals.
  for (let k = 0; k < this.no; ++k) {
    let derivative = (1 - this.oNodes[k]) * this.oNodes[k];  // assumes
softmax.
    oSignals[k] = derivative * (this.oNodes[k] - Y[k]);  // E=(t-o)^2 do
E'=(o-t)
  }

  // compute hidden-to-output weight gradients using output signals.
  for (let j = 0; j < this.nh; ++j) {
    for (let k = 0; k < this.no; ++k) {
      hoGrads[j][k] = oSignals[k] * this.hNodes[j];
    }
  }
. . .
```

The derivative term of the output signal depends on the output layer activation function. For softmax activation, the derivative is **o * (1 - o)**, where **o** is the output node value. The computation for the **oSignals** vector values assumes that squared error is defined as (target - output)^2, which gives (output - target) because of the minus sign in the error term.

After the scratch output signals have been computed, the gradients are computed as the associated hidden node value times the output signal. None of these ideas is simple or obvious. From a developer's point of view, the idea to remember is that hidden-to-output weight gradients depend on the choice of error function and the choice of output layer activation function.

The gradients for the output node biases are computed like the following.

```
  // compute output node bias gradients using output signals.
  for (let k = 0; k < this.no; ++k) {
    obGrads[k] = oSignals[k] * 1.0;  // 1.0 dummy input can be dropped.
  }
. . .
```

The multiplication by the dummy 1.0 can be dropped because it doesn't affect the result. The idea here is that because a weight connects two nodes, you can think of a bias value as connecting one node with a constant 1.0 value. The use of the dummy 1.0 input value shows the symmetry between the computation of hidden-to-output weight gradients and the computation of the output node biases.

Next, the hidden node signals are computed using the output node signals and the calculus derivative of the hidden layer activation function, the hyperbolic tangent in this case.

```
  // compute hidden node signals.
  for (let j = 0; j < this.nh; ++j) {
    let sum = 0.0;
    for (let k = 0; k < this.no; ++k) {
      sum += oSignals[k] * this.hoWeights[j][k];
    }

    let derivative = (1 - this.hNodes[j]) * (1 + this.hNodes[j]);  // tanh
    hSignals[j] = derivative * sum;
  }
. . .
```

The ideas underlying this code are some of the deepest in all of machine learning. However, from a practical point of view, the only statement you'll ever need to change is the computation of the derivative term (if you decide to use a different hidden layer activation function). For example, if you use logistic sigmoid activation, because the calculus derivative of y = logsig(x) is y * (1 - y), the code changes to the following.

```
   let derivative = this.hNodes[j] * (1 - this.hNodes[j]);  // logistic
sigmoid
```

Next, the gradients for the input-to-hidden weights and the gradients for the hidden node biases are computed.

```
  // compute input-to-hidden weight gradients using hidden signals.
  for (let i = 0; i < this.ni; ++i) {
    for (let j = 0; j < this.nh; ++j) {
      ihGrads[i][j] = hSignals[j] * this.iNodes[i];
    }
  }

  // compute hidden node bias gradients using hidden signals.
  for (let j = 0; j < this.nh; ++j) {
    hbGrads[j] = hSignals[j] * 1.0;  // 1.0 dummy input can be dropped.
  }
. . .
```

To summarize, the goal is to find a small delta value to add to each weight and bias so that computed outputs get closer to target output values (or equivalently, the error is reduced). To compute the delta values, you need to compute a gradient value for each weight and bias. The gradient values are composed of a signal and an input value (where the input value is a dummy 1.0 for biases).

After the gradients have been computed, the next step is to compute the delta values and then use them to modify the weights and biases. When computing the gradients, you must work backward from output to hidden to input (which is why back-propagation is named as it is). But when updating the weights and biases, you can work in any order. The demo code works from input to hidden to output to emphasize that the order of weights and biases updates can be different from the order of gradient computations.

The **`train()`** method continues by updating the input-to-hidden weights.

```
  // update input-to-hidden weights
  for (let i = 0; i < this.ni; ++i) {
    for (let j = 0; j < this.nh; ++j) {
      let delta = -1.0 * lrnRate * ihGrads[i][j];
      this.ihWeights[i][j] += delta;
    }
  }
. . .
```

If you compare this code with the weight update equation in Figure 3-5, you will see a one-to-one correspondence between the code and the math. Recall that the purpose of the -1 term is just so that the weight value is updated by adding using the **`+=`** operator. Instead, you could write the following equivalent code.

```
  let delta = lrnRate * ihGrads[i][j];
  this.ihWeights[i][j] -= delta;
```

In other words, instead of multiplying by -1 to get a negative value and then adding, you can just subtract. Even though this alternative approach is a tiny bit more efficient (saving a multiplication by -1 operation), it's rarely used. Next, the hidden node biases are updated using their associated gradients.

```
  // update hidden node biases.
  for (let j = 0; j < this.nh; ++j) {
    let delta = -1.0 * lrnRate * hbGrads[j];
    this.hBiases[j] += delta;
  }
. . .
```

The pattern to update biases is exactly the same as the pattern used to update weights. Next, the hidden-to-output weights and the output node biases are updated.

```
    // update hidden-to-output weights.
    for (let j = 0; j < this.nh; ++j) {
      for (let k = 0; k < this.no; ++k) {
        let delta = -1.0 * lrnRate * hoGrads[j][k];
        this.hoWeights[j][k] += delta;
      }
    }

    // update output node biases.
    for (let k = 0; k < this.no; ++k) {
      let delta = -1.0 * lrnRate * obGrads[k];
      this.oBiases[k] += delta;
    }
  } // each item
. . .
```

At this point, every data item in the training data set has been read and processed once, and each item generates an update for each weight and bias. In the demo, there are 120 training items, and the neural network architecture is 4-7-3, so there are (4 * 7) + 7 + (7 * 3) + 3 = 59 weights and biases. Therefore, one pass through the training data set (called an epoch) produces 120 * 59 = 7,080 update operations in one epoch. The demo program sets the number of epochs to 50, so the entire program generates 7080 * 50 = 354,000 update operations.

The **train()** method concludes by checking to see if it's time to show a progress message.

```
. . .
      if (epoch % freq == 0) {
        let mse = this.meanSqErr(trainX, trainY).toFixed(4);
        let acc = this.accuracy(trainX, trainY).toFixed(4);

        let s1 = "epoch: " + epoch.toString();
        let s2 = " MSE = " + mse.toString();
        let s3 = " acc = " + acc.toString();

        console.log(s1 + s2 + s3);
      }
    } // epoch
  } // train()
```

Checking for progress messages is placed at the end of the main processing loop. You could check at the top of the main loop.

```
. . .
  for (let epoch = 0; epoch < maxEpochs; ++epoch) {  // each epoch.
    this.shuffle(indices);                           // scramble order of
items.
    for (let ii = 0; ii < n; ++ii) {                 // each training item.
    if (epoch % freq == 0) {                         // it's time to show
progress.
      let mse = this.meanSqErr(trainX, trainY).toFixed(4);
. . .
```

Using this approach would display the mean squared error and classification accuracy immediately (at epoch = 0), which is useful to get baseline values. The disadvantage is that during all other epochs, you see error and accuracy of the model before weights and biases are updated, not after the updates.

Back-propagation using cross-entropy error

In spite of decades of research on neural networks, many fundamental questions remain unanswered. One such question is, "For neural multiclass classification, should you use mean squared error or cross-entropy error?" In my opinion, there are no definitive research results that answer this question. For some problems, mean squared error seems to work better, but for other problems, mean cross-entropy error appears to work better.

For a multiclass classification problem, to use mean cross-entropy error instead of mean squared error, you only need to make one small (but truly surprising) change to the demo code. When using mean squared error, the statements that compute the output node signals are the following.

```
  // compute output node signals (assuming mean squared error).
  for (let k = 0; k < this.no; ++k) {
    let derivative = (1 - this.oNodes[k]) * this.oNodes[k];  // assumes
softmax.
    oSignals[k] = derivative * (this.oNodes[k] - Y[k]);  // E=(t-o)^2 do
E'=(o-t)
  }
```

Because of some algebra cancellations, when using mean cross-entropy error, the computation of the output node signals becomes the following.

```
  // compute output node signals (assuming mean cross-entropy error).
  for (let k = 0; k < this.no; ++k) {
    // assumes softmax.
    oSignals[k] = this.oNodes[k] - Y[k];  // E=-Sum(ln(o)*t)
  }
```

The derivative term essentially goes away, leaving just the (output - target) term. This is one of the most surprising results in all of neural machine learning.

The demo program displays error and accuracy progress messages during training. If you use mean cross-entropy error instead of mean squared error, it makes sense to display mean cross-entropy error rather than mean squared error. But because mean squared error is slightly easier to interpret than cross-entropy error, you might want to display both error metrics.

```
  if (epoch % freq == 0) {  // time to display progress.
    let mse = this.meanSqErr(trainX, trainY).toFixed(4);
    let mcee = this.meanCrossEntErr(trainX, trainY).toFixed(4);
    let acc = this.accuracy(trainX, trainY).toFixed(4);

    let s1 = "epoch: " + epoch.toString();
    let s2 = " MSE = " + mse.toString();
    let s3 = " MCEE = " + mcee.toString();
    let s4 = " acc = " + acc.toString();

    console.log(s1 + s2 + s3 + s4);
  }
```

Until the year 2012 or so, the use of mean squared error was more common than the use of mean cross-entropy error. However, cross-entropy error is now far more common. The graph in Figure 3-6 shows an example of squared error and cross-entropy error applied to the same problem.

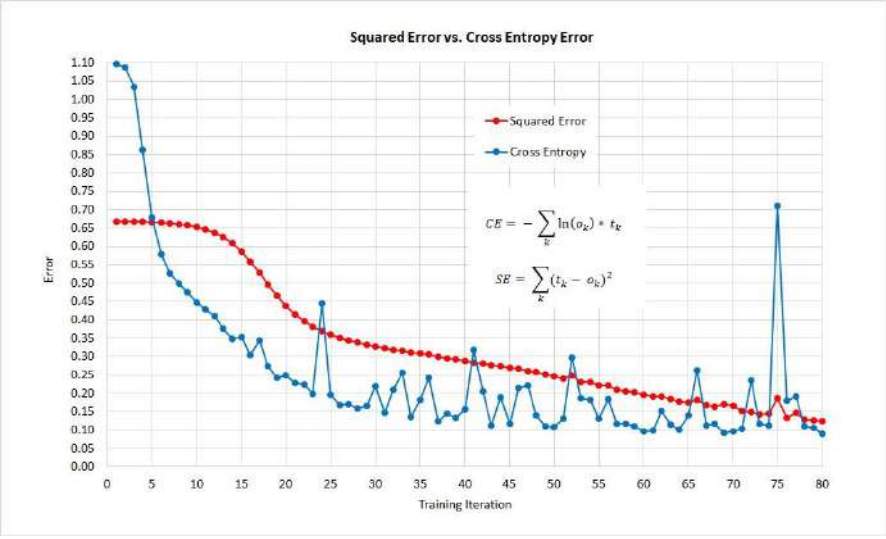


Figure 3-6: Comparison of Squared Error and Cross-Entropy Error on Same Problem

The graph shows that, for this one problem, cross-entropy error gives a good neural network model more quickly than squared error, but that cross entropy is more volatile than squared error.

In practice, if you use enough training iterations, for most multiclass classification problems mean squared error and mean cross-entropy error produce similar neural network models in terms of prediction accuracy. However, because cross-entropy error is used much more often than squared error, you should use cross-entropy error if you are collaborating with others. And note that for regression problems, where the goal is to predict a single numeric value, cross-entropy error is not an option.

Although the terms back-propagation and stochastic gradient descent are often used interchangeably, they're different. Technically, the back-propagation algorithm computes calculus gradients, and stochastic gradient descent uses the gradients to reduce error.

Stochastic gradient descent often works quite well for neural networks with a single hidden layer, but often fails for deep neural networks. Between 2014 and 2017, remarkable progress was made in devising powerful new learning algorithms for deep neural networks. Most of these algorithms use a different learning rate for each weight and bias, and adaptively change the learning rates during training.

Advanced algorithms supported by most deep neural network code libraries such as TensorFlow and PyTorch include Adagrad (adaptive gradient), Adadelta (improved variation of Adagrad), RMSProp (adaptive gradient plus resilient back-prop), Adam ("adaptive moment estimation," similar to RMSProp with momentum), and AdaMax (a variation of Adam).

All of these learning algorithms are based on calculus gradients. There are exotic learning algorithms that don't rely on gradients. For example, evolutionary optimization is similar to genetic algorithms that loosely mimic the behavior of biological chromosomes. And particle swarm optimization loosely mimics the behavior of swarming organisms. These bio-inspired techniques have great promise, but are currently not practical because they require truly enormous computational power.

Questions

1. Suppose that at some point during training, a neural network's current input and output values are the following.

```
input                   target        computed
-------------------------------------------------------
4.8  3.9  1.1  0.8      1  0  0       0.80  0.10  0.10
6.7  2.6  4.6  1.3      0  1  0       0.15  0.60  0.25
5.6  3.1  2.2  1.7      0  0  1       0.20  0.50  0.30
```

A. What is the classification accuracy of the network?

B. What is the mean squared error of the network?

C. What is the mean cross-entropy error of the network?

Chapter 4 Overfitting

One of the big challenges when working with neural networks is a phenomenon called overfitting. A trained model is overfitted if it predicts well on the training data but predicts poorly on test data and new, previously unseen data. There are many techniques to reduce the likelihood of model overfitting, including dropout, train-validate-test, and regularization.

The screenshot in Figure 4-1 shows a demo of neural network overfitting. The goal of the program is to predict the political leaning of a person (conservative, moderate, or liberal) based on sex (male or female), age, geographical region (eastern, western, or central), and annual income.

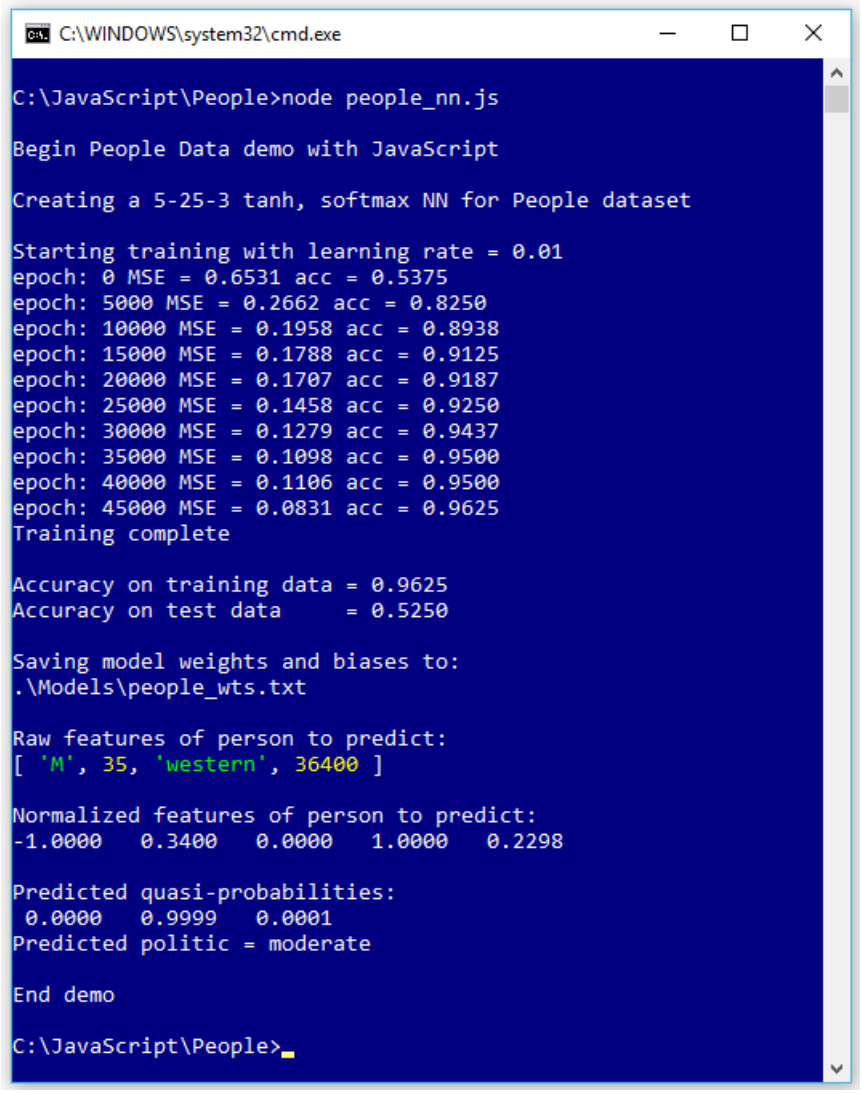


Figure 4-1: An Example of Neural Network Model Overfitting

The data is completely artificial and was generated programmatically. The raw data has 240 items that look like the following.

```
M 32 eastern 59200.00 | moderate
F 33 central 38400.00 | moderate
M 35 central 30800.00 | liberal
M 26 western 53800.00 | conservative
. . .
```

The first column is the person's sex, represented by M or F. The second column is the person's age. The third column is the geo-region (eastern, western, or central) where the person lives. The fifth column is an arbitrary pipe character for readability. The sixth column is the political leaning to predict (conservative, moderate, or liberal).

The 240-item raw data was split into a 160-item data set for training and a 40-item data set for testing. After splitting, the two data sets were normalized and encoded for use by a neural network (the remaining 40 items are used later).

The demo created a 5-25-3 neural network. The raw data has four predictor variables, but when normalized and encoded, there are five predictor values. There are three output nodes because there are three classes to predict. The number of hidden nodes (25) is a hyperparameter and must be determined by trial and error.

The demo program trains the network for 50,000 epochs using back-propagation with the online training technique. After training, the model's classification accuracy is an impressive 96.25% correct (154 out of 160), but a rather unimpressive 52.50% accuracy (21 out of 40) on the test data. The difference in prediction accuracy on the training and test datasets suggest that the model has been overfitted.

After the model evaluation metrics have been displayed, the demo saves the model's weights and biases to a text file so that they can be retrieved later. For all but the quickest training, you'll want to save your model weights and biases so you don't have to retrain a good model.

The demo concludes by making a prediction for a new, previously unseen person with raw predictor values of sex = male, age = 35, region = "western", and income = $36,400.00. These values are normalized and encoded to sex = -1, age = 0.3400, region = (0, 1), and income = 0.2298. The prediction probabilities are (0.0000, 0.9999, 0.0001), and because the second value is largest, the prediction is that the person is a political moderate. The extreme values of the predicted probabilities also suggest the model may be overfitted.

The graph in Figure 4-2 illustrates the basic concepts of overfitting. In the figure, there are two predictor variables, X1 and X2. There are two possible values to predict, indicated by the red (class = 0) and blue (class = 1) dots. You can imagine this corresponds to the problem of predicting if a person is male (0) or female (1), based on normalized age (X1) and normalized income (X2). For example, the left-most data point at (X1 = 1, X2 = 4) is colored blue (female).

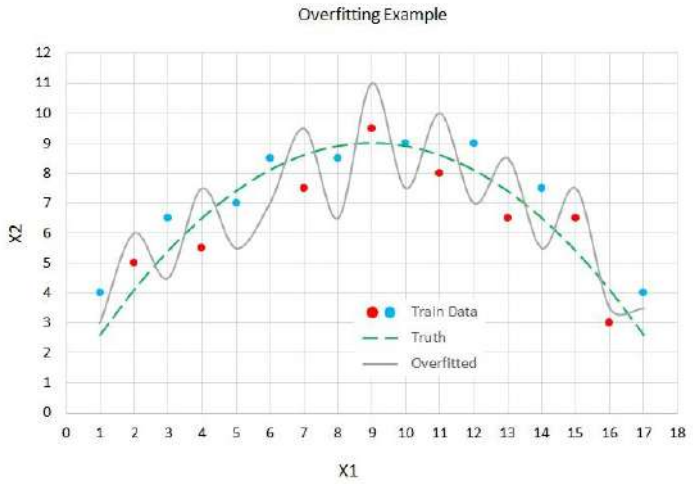


Figure 4-2: Model Overfitting

The dashed green line represents the true (but unknown) decision boundary between the two classes. Notice that data items that are above the green line are mostly blue (seven of ten), and that data items below the green line are mostly red (five of seven). The five misclassifications in the training data are due to the randomness inherent in all real-life data.

A good neural network model would find the true decision boundary represented by the green line. However, if you train a neural network model long enough, it will essentially get too good and generate a model indicated by the spiky, solid gray line.

The gray line makes perfect predictions on the test data. All the blue dots are above the gray line and all the red dots are below the gray line.

But when the overfitted model is presented with new, previously unseen data, there's a good chance the model will make an incorrect prediction. For example, a new data item at (X1 = 4, X2 = 7) is above the truth boundary, and so should be classified as blue. But because the data item is below the gray line overfitted boundary, it will be incorrectly classified as red.

Data normalization

Preparing data for use by a neural network is often tedious and time-consuming. In most scenarios there are three distinct data preparation tasks: cleaning, normalizing numeric data, and encoding non-numeric data.

Cleaning data usually means dealing with missing values and extreme or anomalous values. For data items that have one or more missing predictor values, in all but rare cases the best approach (in my opinion) is to delete those data items. Alternatives such as supplying an average value of some sort are sometimes used in classical statistics scenarios, but such techniques rarely work well with neural networks.

In almost all neural network scenarios, you should normalize numeric predictor values so that the values are scaled to roughly the same magnitude. Note that the Iris data set in Chapter 3 is an exception because the data is so simple and is already scaled to values between 0.1 and 7.9 (this is a major reason why the Iris data set is used so often for neural network examples).

The three most common forms of data normalization are min-max normalization, z-score normalization, and constant factor normalization. All three techniques are best explained by example.

Suppose you have a tiny set of training data with just three items.

```
25   68000.00   male     moderate
41   73000.00   male     conservative
36   54000.00   female   liberal
```

The goal is to predict political leaning from age, annual income, and sex. Without normalization, the large annual income values would overwhelm the small age values.

To apply min-max normalization to a numeric predictor, you find the minimum and maximum value in its column, then compute (x - min) / (max - min).

For the age variable, min = 25 and max = 41, and so the normalized age values are the following.

```
(25 - min) / (max - min) = (25 - 25) / (41 - 25) = 0.0000
(41 - min) / (max - min) = (41 - 25) / (41 - 25) = 1.0000
(36 - min) / (max - min) = (36 - 25) / (41 - 25) = 0.6875
```

The annual income values could be normalized in the same way.

```
(68000 - min) / (max - min) = (68000 - 54000) / (73000 - 54000) = 0.7368
(73000 - min) / (max - min) = (73000 - 54000) / (73000 - 54000) = 1.0000
(54000 - min) / (max - min) = (54000 - 54000) / (73000 - 54000) = 0.0000
```

After min-max normalizing a numeric predictor variable, all values will be scaled between 0.0 and 1.0.

To apply z-score normalization, you find the mean and standard deviation for each numeric column, then compute (x - mean) / sd.

For the age variable, mean = 34.00 and sd = 6.68, and so the normalized values are the following.

```
(25 - mean) / sd = (25 - 34.00) / 6.68 = -1.3473
(41 - mean) / sd = (41 - 34.00) / 6.68 = 1.0479
(36 - mean) / sd = (36 - 34.00) / 6.68 = 0.2994
```

After z-score normalizing a predictor variable, the values will usually be between -5.0 and +5.0. Normalized values less than -5.0 or greater than 5.0 are extreme in the sense that they're much different from the average value.

To apply constant factor normalization, you divide or multiply all values by a constant. For example, for the age values, you could divide each value by 100.

```
25 / 100 = 0.25
41 / 100 = 0.41
36 / 100 = 0.36
```

In general, there is no one best normalization technique. In practice, the type of normalization technique used usually doesn't have a huge effect on the prediction accuracy of the trained network. It is possible to use different normalization techniques on different predictors in your training data. For example, you could normalize age values using min-max normalization and normalize annual income values using z-score normalization.

After normalizing your training data, you must use the same normalizing parameters on the test data. For example, suppose you have a set of training data where one of the predictors is a person's height in inches. If you use min-max normalization on the training data where min = 58.0 and max = 77.0, then when you normalize the test data, you do not calculate a new min and max value. You use the min = 58.0 and max = 77.0 on the test data, regardless of the actual min and max values in the test data.

In short, the steps are:

1. Split all data into a training and a test set.
2. Normalize the training data.
3. Normalize the test data using the normalization parameters from the training data.

Data encoding

A neural network works directly only with numeric values, so predictor variables that are non-numeric (sometimes called categorical) must be converted to numbers. For categorical predictor variables that can take one of three or more values, the two most common encoding techniques are 1-of-(N-1) encoding and one-hot encoding (sometimes called 1-of-N encoding). For categorical predictor variables that can take one of just two possible values (binary predictors), the three most common encoding techniques are one-hot encoding, -1 +1 encoding, and 0-1 encoding.

Before explaining these techniques, let me say that my recommendations are to use 1-of-(N-1) encoding for predictors that can be from three to nine possible values, use one-hot encoding for predictors that can be 10 or more possible values, and use -1 +1 encoding for binary predictors that can be one of two possible values.

Suppose you have a categorical predictor variable, color, that can be one of four possible values: red, blue, green, yellow. If you use one-hot / 1-of-N encoding, then red = (1, 0, 0, 0), blue = (0, 1, 0, 0), green = (0, 0, 1, 0), and yellow = (0, 0, 0, 1).

If you use 1-of-(N-1) encoding, then red = (1, 0, 0), blue = (0, 1, 0), green = (0, 0, 1), and yellow = (-1, -1, -1). The 1-of-(N-1) encoding technique is sometimes called effect coding or contrast coding, but there is quite a bit of inconsistency with the terminology.

Suppose you have a binary predictor variable, sex, that can be either male or female. If you use one-hot encoding, then male = (1, 0) and female = (0, 1). If you use -1 +1 encoding, then male = -1 and female = +1. If you use 0-1 encoding, then male = 0 and female = 1.

If you have a classification problem, then you must encode the values of the variable to predict (the dependent variable), too. For dependent variables that can be one of three or more values (such as political leaning of conservative, moderate, or liberal), then you should always use one-hot / 1-of-N encoding: conservative = (1, 0, 0), moderate = (0, 1, 0), liberal = (0, 0, 1).

For binary classification problems (such as predicting a person's sex of male or female), by far the most common approach is to use 0-1 encoding: male = 0, female = 1. However, you can also use one-hot / 1-of-N encoding: male = (1, 0), female = (0, 1). Encoding a binary dependent variable for a neural network is surprisingly subtle, and is explained in Chapter 6, Binary Classification.

Understanding the People Dataset

The People Dataset consists of 240 tab-delimited items. The data was split into a training set of 160 items and a test set of 40 items. The training set raw data looks like the following.

```
M 32 eastern 59200.00 | moderate
F 33 central 38400.00 | moderate
M 35 central 30800.00 | liberal
M 26 western 53800.00 | conservative
. . .
```

The training set was normalized using min-max normalization on both the age (min = 18, max = 68) and the annual income (min = 20,500.00, max = 89,700.00) predictor variables.

The sex predictor variable was encoded as M = -1 and F = +1. The region predictor variable was 1-of-(N-1) encoded as eastern = (1, 0), western = (0, 1), central = (-1, -1). The dependent variable to predict, political leaning, was one-hot encoded as conservative = (1, 0, 0), moderate = (0, 1, 0), liberal = (0, 0, 1). The normalized and encoded training data looks like the following.

```
-1  0.28   1  0  0.5592   0 1 0
 1  0.30  -1 -1  0.2587   0 1 0
-1  0.34  -1 -1  0.1488   0 0 1
-1  0.16   0  1  0.4812   1 0 0
. . .
```

The 40-item test data was min-max normalized using the training data age parameters (min = 18, max = 68) and annual income parameters (min = 20,500, max = 89,700), and was encoded. The test data looks like the following.

```
 1  1.00  -1 -1  0.4581   1 0 0
 1  0.68   1  0  0.9682   1 0 0
-1  0.70   1  0  0.1705   0 1 0
. . .
```

Even this tiny data set took quite some time to prepare. In practice, it's rarely possible to experiment with different normalization and encoding schemes, so you must decide beforehand which techniques to use.

The normalized and encoded data used by the demo programs are in the appendix to this e-book, and on GitHub.

The People Dataset demo program

The complete demo program shown running in Figure 4-1 is presented in Code Listing 4-1. The demo assumes there is a top-level directory named JavaScript that contains subdirectories named Utilities and People. The Utilities directory contains the Utilities_lib.js library file. The People directory contains the demo program People_nn.js file and subdirectories named Data and Models. The Data directory contains files People_train.txt and People_test.txt. The Models directory is used to store trained model weights and biases values.

Code Listing 4-1: The People Data Demo Program

```
// people_nn.js
// ES6

let U = require("../Utilities/utilities_lib.js");
let FS = require("fs");

//
================================================================================
==

class NeuralNet
{
  constructor(numInput, numHidden, numOutput, seed)
  {
    this.rnd = new U.Erratic(seed);

    this.ni = numInput;
    this.nh = numHidden;
    this.no = numOutput;

    this.iNodes = U.vecMake(this.ni, 0.0);
    this.hNodes = U.vecMake(this.nh, 0.0);
    this.oNodes = U.vecMake(this.no, 0.0);

    this.ihWeights = U.matMake(this.ni, this.nh, 0.0);
    this.hoWeights = U.matMake(this.nh, this.no, 0.0);

    this.hBiases = U.vecMake(this.nh, 0.0);
    this.oBiases = U.vecMake(this.no, 0.0);
```

```
  this.initWeights();
}

initWeights()
{
  let lo = -0.01;
  let hi = 0.01;
  for (let i = 0; i < this.ni; ++i) {
    for (let j = 0; j < this.nh; ++j) {
      this.ihWeights[i][j] = (hi - lo) * this.rnd.next() + lo;
    }
  }

  for (let j = 0; j < this.nh; ++j) {
    for (let k = 0; k < this.no; ++k) {
      this.hoWeights[j][k] = (hi - lo) * this.rnd.next() + lo;
    }
  }
}

eval(X)
{
  let hSums = U.vecMake(this.nh, 0.0);
  let oSums = U.vecMake(this.no, 0.0);

  this.iNodes = X;

  for (let j = 0; j < this.nh; ++j) {
    for (let i = 0; i < this.ni; ++i) {
      hSums[j] += this.iNodes[i] * this.ihWeights[i][j];
    }
    hSums[j] += this.hBiases[j];
    this.hNodes[j] = U.hyperTan(hSums[j]);
  }

  for (let k = 0; k < this.no; ++k) {
    for (let j = 0; j < this.nh; ++j) {
      oSums[k] += this.hNodes[j] * this.hoWeights[j][k];
    }
    oSums[k] += this.oBiases[k];
  }

  this.oNodes = U.softmax(oSums);

  let result = [];
  for (let k = 0; k < this.no; ++k) {
    result[k] = this.oNodes[k];
  }
  return result;
```

```
} // eval()

setWeights(wts)
{
  // order: ihWts, hBiases, hoWts, oBiases
  let p = 0;

  for (let i = 0; i < this.ni; ++i) {
    for (let j = 0; j < this.nh; ++j) {
      this.ihWeights[i][j] = wts[p++];
    }
  }

  for (let j = 0; j < this.nh; ++j) {
    this.hBiases[j] = wts[p++];
  }

  for (let j = 0; j < this.nh; ++j) {
    for (let k = 0; k < this.no; ++k) {
      this.hoWeights[j][k] = wts[p++];
    }
  }

  for (let k = 0; k < this.no; ++k) {
    this.oBiases[k] = wts[p++];
  }
} // setWeights()

getWeights()
{
  // order: ihWts, hBiases, hoWts, oBiases
  let numWts = (this.ni * this.nh) + this.nh +
    (this.nh * this.no) + this.no;
  let result = U.vecMake(numWts, 0.0);
  let p = 0;
  for (let i = 0; i < this.ni; ++i) {
    for (let j = 0; j < this.nh; ++j) {
      result[p++] = this.ihWeights[i][j];
    }
  }

  for (let j = 0; j < this.nh; ++j) {
    result[p++] = this.hBiases[j];
  }

  for (let j = 0; j < this.nh; ++j) {
    for (let k = 0; k < this.no; ++k) {
      result[p++] = this.hoWeights[j][k];
```

```
      }
    }

    for (let k = 0; k < this.no; ++k) {
      result[p++] = this.oBiases[k];
    }
    return result;
  } // getWeights()

  shuffle(v)
  {
    // Fisher-Yates
    let n = v.length;
    for (let i = 0; i < n; ++i) {
      let r = this.rnd.nextInt(i, n);
      let tmp = v[r];
      v[r] = v[i];
      v[i] = tmp;
    }
  }

  train(trainX, trainY, lrnRate, maxEpochs)
  {
    let hoGrads = U.matMake(this.nh, this.no, 0.0);
    let obGrads = U.vecMake(this.no, 0.0);
    let ihGrads = U.matMake(this.ni, this.nh, 0.0);
    let hbGrads = U.vecMake(this.nh, 0.0);

    let oSignals = U.vecMake(this.no, 0.0);
    let hSignals = U.vecMake(this.nh, 0.0);

    let n = trainX.length;  // 160
    let indices = U.arange(n);  // [0,1,..,159]
    let freq = Math.trunc(maxEpochs / 10);

    for (let epoch = 0; epoch < maxEpochs; ++epoch) {
      this.shuffle(indices);  //
      for (let ii = 0; ii < n; ++ii) {  // each item
        let idx = indices[ii];
        let X = trainX[idx];
        let Y = trainY[idx];
        this.eval(X);  // output stored in this.oNodes.

        // compute output node signals.
        for (let k = 0; k < this.no; ++k) {
          let derivative = (1 - this.oNodes[k]) * this.oNodes[k];  //
softmax
          oSignals[k] = derivative * (this.oNodes[k] - Y[k]);  // E=(t-o)^2
        }
```

```
        // compute hidden-to-output weight gradients using output signals.
        for (let j = 0; j < this.nh; ++j) {
          for (let k = 0; k < this.no; ++k) {
            hoGrads[j][k] = oSignals[k] * this.hNodes[j];
          }
        }

        // compute output node bias gradients using output signals.
        for (let k = 0; k < this.no; ++k) {
          obGrads[k] = oSignals[k] * 1.0;  // 1.0 dummy input can be
dropped.
        }

        // compute hidden node signals.
        for (let j = 0; j < this.nh; ++j) {
          let sum = 0.0;
          for (let k = 0; k < this.no; ++k) {
            sum += oSignals[k] * this.hoWeights[j][k];
          }
          let derivative = (1 - this.hNodes[j]) * (1 + this.hNodes[j]);  //
tanh
          hSignals[j] = derivative * sum;
        }

        // compute input-to-hidden weight gradients using hidden signals.
        for (let i = 0; i < this.ni; ++i) {
          for (let j = 0; j < this.nh; ++j) {
            ihGrads[i][j] = hSignals[j] * this.iNodes[i];
          }
        }

        // compute hidden node bias gradients using hidden signals.
        for (let j = 0; j < this.nh; ++j) {
          hbGrads[j] = hSignals[j] * 1.0;  // 1.0 dummy input can be
dropped.
        }

        // update input-to-hidden weights.
        for (let i = 0; i < this.ni; ++i) {
          for (let j = 0; j < this.nh; ++j) {
            let delta = -1.0 * lrnRate * ihGrads[i][j];
            this.ihWeights[i][j] += delta;
          }
        }

        // update hidden node biases.
        for (let j = 0; j < this.nh; ++j) {
          let delta = -1.0 * lrnRate * hbGrads[j];
```

```
        this.hBiases[j] += delta;
      }

      // update hidden-to-output weights.
      for (let j = 0; j < this.nh; ++j) {
        for (let k = 0; k < this.no; ++k) {
          let delta = -1.0 * lrnRate * hoGrads[j][k];
          this.hoWeights[j][k] += delta;
        }
      }

      // update output node biases.
      for (let k = 0; k < this.no; ++k) {
        let delta = -1.0 * lrnRate * obGrads[k];
        this.oBiases[k] += delta;
      }
    } // ii

    if (epoch % freq == 0) {
      let mse = this.meanSqErr(trainX, trainY).toFixed(4);
      let acc = this.accuracy(trainX, trainY).toFixed(4);

      let s1 = "epoch: " + epoch.toString();
      let s2 = " MSE = " + mse.toString();
      let s3 = " acc = " + acc.toString();

      console.log(s1 + s2 + s3);
    }

  } // epoch
} // train()

meanCrossEntErr(dataX, dataY)
{
  let sumCEE = 0.0;  // sum of the cross-entropy errors.
  for (let i = 0; i < dataX.length; ++i) {  // each data item
    let X = dataX[i];
    let Y = dataY[i];  // target output like (0, 1, 0)
    let oupt = this.eval(X);  // computed like (0.23, 0.66, 0.11)
    let idx = U.argmax(Y);  // find location of the 1 in target.
    sumCEE += Math.log(oupt[idx]);
  }

  return -1 * sumCEE / dataX.length;
}

meanSqErr(dataX, dataY)
{
  let sumSE = 0.0;
```

```
  for (let i = 0; i < dataX.length; ++i) {  // each data item
    let X = dataX[i];
    let Y = dataY[i];  // target output like (0, 1, 0)
    let oupt = this.eval(X);  // computed like (0.23, 0.66, 0.11)
    for (let k = 0; k < this.no; ++k) {
      let err = Y[k] - oupt[k]  // target - computed
      sumSE += err * err;
    }
  }
  return sumSE / dataX.length;
}

accuracy(dataX, dataY)
{
  let nc = 0; let nw = 0;
  for (let i = 0; i < dataX.length; ++i) {  // each data item
    let X = dataX[i];
    let Y = dataY[i];  // target output like (0, 1, 0)
    let oupt = this.eval(X);  // computed like (0.23, 0.66, 0.11)
    let computedIdx = U.argmax(oupt);
    let targetIdx = U.argmax(Y);
    if (computedIdx == targetIdx) {
      ++nc;
    }
    else {
      ++nw;
    }
  }
  return nc / (nc + nw);
}

saveWeights(fn)
{
  let wts = this.getWeights();
  let n = wts.length;
  let s = "";
  for (let i = 0; i < n - 1; ++i) {
    s += wts[i].toString() + ",";
  }
  s += wts[n - 1];

  FS.writeFileSync(fn, s);
}

loadWeights(fn)
{
  let n = (this.ni * this.nh) + this.nh + (this.nh * this.no) + this.no;
  let wts = U.vecMake(n, 0.0);
  let all = FS.readFileSync(fn, "utf8");
```

```
    let strVals = all.split(",");
    let nn = strVals.length;
    if (n != nn) {
      throw ("Size error in NeuralNet.loadWeights()");
    }
    for (let i = 0; i < n; ++i) {
      wts[i] = parseFloat(strVals[i]);
    }
    this.setWeights(wts);
  }

} // NeuralNet


//
================================================================================
==

function main() {
  process.stdout.write("\033[0m");  // reset
  process.stdout.write("\x1b[1m" + "\x1b[37m");  // bright white
  console.log("\nBegin People Data demo with JavaScript ");

  // 1. load data
  // raw data looks like:   M   32   western  38400.00    |  moderate
  // norm data looks like: -1  0.28    0 1    0.2587          0 1 0
  let trainX = U.loadTxt(".\\Data\\people_train.txt", "\t", [0, 1, 2, 3,
4]);
  let trainY = U.loadTxt(".\\Data\\people_train.txt", "\t", [5, 6, 7]);
  let testX = U.loadTxt(".\\Data\\people_test.txt", "\t", [0, 1, 2, 3, 4]);
  let testY = U.loadTxt(".\\Data\\people_test.txt", "\t", [5, 6, 7]);

  // 2. create network
  console.log("\nCreating a 5-25-3 tanh, softmax NN for People dataset");
  let seed = 0;
  let nn = new NeuralNet(5, 25, 3, seed);

  // 3. train network
  let lrnRate = 0.01;
  let maxEpochs = 50000;
  console.log("\nStarting training with learning rate = 0.01 ");
  nn.train(trainX, trainY, lrnRate, maxEpochs);
  console.log("Training complete");

  // 4. evaluate model
  let trainAcc = nn.accuracy(trainX, trainY);
  let testAcc = nn.accuracy(testX, testY);
  console.log("\nAccuracy on training data = " +
    trainAcc.toFixed(4).toString());
```

```
  console.log("Accuracy on test data     = " +
    testAcc.toFixed(4).toString());

  // 5. save model
  let fn = ".\\Models\\people_wts.txt";
  console.log("\nSaving model weights and biases to: ");
  console.log(fn);
  nn.saveWeights(fn);

  // 6. use trained model
  let unknownRaw = ["M", 35, "western", 36400.00];
  let unknownNorm = [-1, 0.34, 0, 1, 0.2298];
  let predicted = nn.eval(unknownNorm);
  console.log("\nRaw features of person to predict: ");
  console.log(unknownRaw);
  console.log("\nNormalized features of person to predict: ");
  U.vecShow(unknownNorm, 4, 12);
  console.log("\nPredicted quasi-probabilities: ");
  U.vecShow(predicted, 4, 12);

  let politics = ["conservative", "moderate", "liberal"];
  let predIdx = U.argmax(predicted);
  let predPolitic = politics[predIdx];
  console.log("Predicted politic = " + predPolitic);

  process.stdout.write("\033[0m");  // reset
  console.log("\nEnd demo");
} // main()

main();
```

All of the control logic is contained in a top-level **`main()`** function. Program execution begins by loading the training and test data into memory.

```
  // 1. load data
  // raw data looks like:   M   32   western  38400.00    |  moderate
  // norm data looks like: -1  0.28    0 1    0.2587          0 1 0
  let trainX = U.loadTxt(".\\Data\\people_train.txt", "\t", [0,1,2,3,4]);
  let trainY = U.loadTxt(".\\Data\\people_train.txt", "\t", [5,6,7]);
  let testX = U.loadTxt(".\\Data\\people_test.txt", "\t", [0,1,2,3,4]);
  let testY = U.loadTxt(".\\Data\\people_test.txt", "\t", [5,6,7]);
. . .
```

The **`loadTxt()`** function is contained in the Utilities_lib.js file, which is located in the Utilities directory. Notice that the data files are tab-delimited. Next, a neural network is created.

```
  // 2. create network
  console.log("\nCreating a 5-25-3 tanh, softmax NN for People dataset");
  let seed = 0;
  let nn = new NeuralNet(5, 25, 3, seed);
. . .
```

The raw data has four predictor variables (sex, age, region, and income), but there are five predictor values after normalization because region has been 1-of-(N-1) encoded. The number of hidden nodes, 25, is a hyperparameter that must be determined by trial and error. In general, more hidden nodes can produce a more accurate model at the expense of an increased risk of model overfitting. The seed value is for random initialization of weights and random order processing of training items. Next, the network is trained.

```
  // 3. train network
  let lrnRate = 0.01;
  let maxEpochs = 50000;
  console.log("\nStarting training with learning rate = 0.01 ");
  nn.train(trainX, trainY, lrnRate, maxEpochs);
  console.log("Training complete");
. . .
```

The large number of training iterations (50,000) could lead to overfitting. Next, the trained mode is evaluated.

```
  // 4. evaluate model
  let trainAcc = nn.accuracy(trainX, trainY);
  let testAcc = nn.accuracy(testX, testY);

  console.log("\nAccuracy on training data = " +
    trainAcc.toFixed(4).toString());

  console.log("Accuracy on test data     = " +
    testAcc.toFixed(4).toString());
. . .
```

The end goal of a neural network classifier is prediction accuracy, so it makes sense to compute accuracy after training. In many cases you'll also want to compute and display the trained model's error (either mean squared error or mean cross-entropy error, or both), for example, with code similar to the following.

```
let trainError = nn.MeanSqErr(trainX, trainY);
console.log(trainError.toFixed(4).toString());
```

Error is a useful metric for comparing different trained models because error is a more granular metric than accuracy. Next, the trained model's weights and biases are saved to a text file.

```
  // 5. save model
  let fn = ".\\Models\\people_wts.txt";
  console.log("\nSaving model weights and biases to: ");
  console.log(fn);
  nn.saveWeights(fn);
. . .
```

The demo program concludes by making a prediction for a new, previously unseen person.

```
  // 6. use trained model
  let unknownRaw = ["M", 35, "western", 36400.00];
  let unknownNorm = [-1, 0.34, 0, 1, 0.2298];

  let predicted = nn.eval(unknownNorm);

  console.log("\nRaw features of person to predict: ");
  console.log(unknownRaw);

  console.log("\nNormalized features of person to predict: ");
  U.vecShow(unknownNorm, 4, 12);

  console.log("\nPredicted quasi-probabilities: ");
  U.vecShow(predicted, 4, 12);
. . .
```

The demo program normalizes and encodes the new person item offline. An alternative that is useful when making many predictions is to write a problem-specific function to normalize and encode. For example, using such a function might look like the following.

```
let unknownRaw = ["F", 27, "eastern", 42500.00];
let unkNorm = normAndEncode(unknownRaw);
let predicted = nn.eval(unkNorm);
```

Note that a program-defined function to normalize and encode raw data needs to know the normalization parameters, such as min and max if min-max normalization is used.

The demo program wraps up by displaying the predicted political leaning in user-friendly form.

```
  let politics = ["conservative", "moderate", "liberal"];
  let predIdx = U.argmax(predicted);
  let predPolitic = politics[predIdx];
  console.log("Predicted politic = " + predPolitic);
```

The pattern used in the demo is: load data, create neural network, train network, evaluate network, save weights and biases, and use model. This pattern, or one very close to it, is a very common pattern when working with neural networks.

Saving and loading model weights

The **NeuralNet** class method **saveWeights()** is defined as the following.

```
  saveWeights(fn)
  {
    let wts = this.getWeights();
    let n = wts.length;
    let s = "";

    for (let i = 0; i < n-1; ++i) { // all except last value.
      s += wts[i].toString() + ",";
    }
    s += wts[n-1];  // last value, no trailing comma.

    FS.writeFileSync(fn, s);
  }
```

The statement **let wts = this.getWeights()** returns a vector named **wts** with the weights and biases serialized as the input-to-hidden weights, followed by the hidden node biases, followed by the hidden-to-output weights, followed by the output node biases. Because the demo network has a 5-25-3 architecture, there are a total of (5 * 25) + 25 + (25 * 3) + 3 = 228 weights and biases. The weights and biases are serialized to a single, comma-separated string (without a trailing comma), which is written to a text file.

There are many alternative designs you can use. For example, you might want to write an architecture such as 5,25,3 and then follow with the weights and biases, or you might want to write the weights using JSON or XML format, and so on.

Almost all widely used deep neural network libraries, such as TensorFlow, Keras, and PyTorch, save models in a proprietary binary format. However, many of these libraries have added support for the ONNX (open neural network exchange) standard.

The counterpart to **saveWeights()** is a method **loadWeights()** that loads saved weights from file into a neural network. A **loadWeights()** method can be defined as the following.

```
  loadWeights(fn)
  {
    let n = (this.ni * this.nh) + this.nh + (this.nh * this.no) + this.no;
    let wts = U.vecMake(n, 0.0);

    let all = FS.readFileSync(fn, "utf8");
    let strVals = all.split(",");
    let nn = strVals.length;

    if (n != nn) {
      throw("Size error in NeuralNet.loadWeights()");
    }

    for (let i = 0; i < n; ++i) {
      wts[i] = parseFloat(strVals[i]);
    }
    this.setWeights(wts);
  }
```

The method sets up a numeric vector to hold the serialized weights and biases, then reads all those values from file into a single large string. The string is split on the comma characters, and then the individual values are converted to numeric form and placed into the vector. The vector is then passed to the class **`setWeights()`** method, which places the values into the neural network.

Method **`loadWeights()`** can be called along the lines of the following.

```
// assume a 5-25-3 neural network has been trained and saved earlier.
let nn = new NeuralNet(5, 25, 3, 0);
nn.loadWeights(".\\Models\\people_wts.txt");
// use nn to make predictions.
```

An extension to the save-load weights approach is to implement a **`saveModel()`** function that saves the network weights and biases values and also the architecture (number of input, hidden, and output nodes; the hidden layer activation function; the output layer activation function; the error function; and so on). Then you could implement a **`loadModel()`** function that both creates a neural network and loads the trained weights and biases values.

Regularization and weight decay

Regularization is one of many techniques that are intended to reduce model overfitting. There are several forms of neural network regularization. The most common form is called L2 regularization. Briefly, the graph of a model where overfitting has occurred has a spiky appearance, such as the graph in Figure 4-2. Such graphs resemble the graph of a polynomial function that has large-magnitude coefficients. These coefficients correspond to neural network weights. Therefore, the idea behind L2 regularization is to keep the magnitudes of the weight values small, which will reduce the likelihood of overfitting. Regularization can be used for classification or regression problems.

The screenshot in Figure 4-3 shows an example of training using L2 regularization. The goal is to predict a person's political leaning (conservative moderate, or liberal) from sex, age, geo-region, and annual income.

The use of L2 regularization requires an additional hyperparameter lambda, which is set to 0.0005 in the demo. Without regularization, the final training and test classification accuracies (shown in Figure 4-1) were 96.25% and 52.50% but with regularization, the final accuracies are 76.88% and 77.50%.

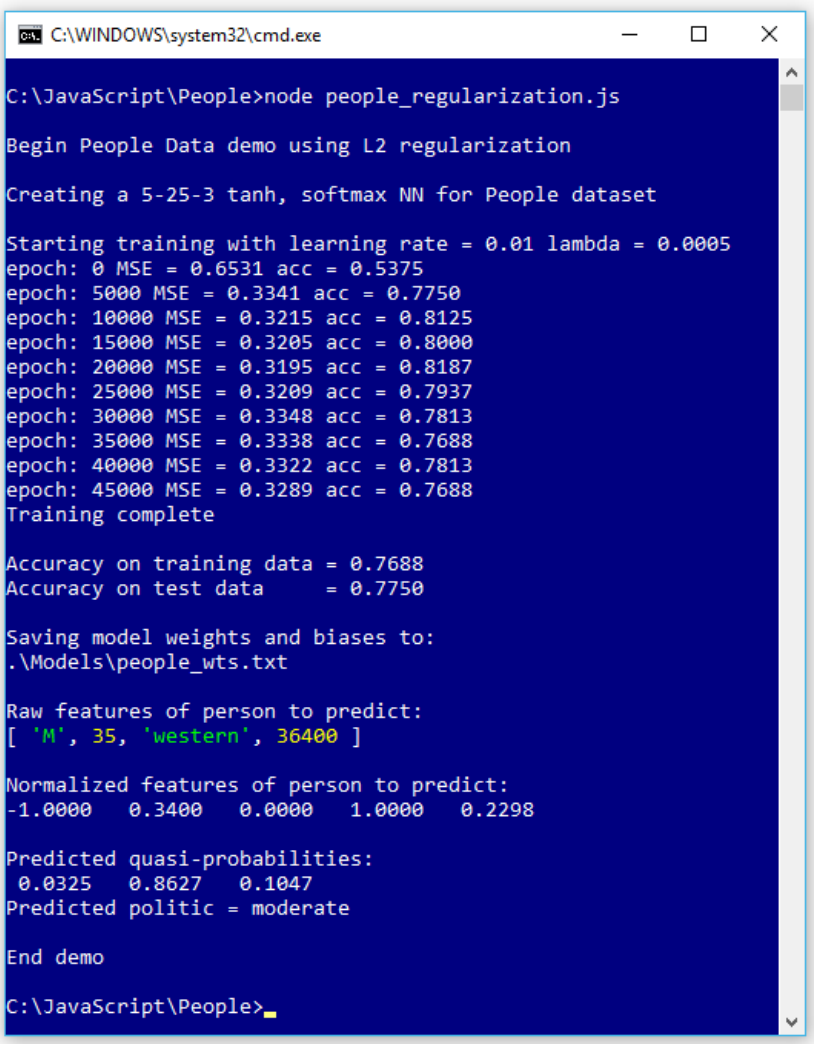


Figure 4-3: L2 Regularization Can Reduce Overfitting and Improve Accuracy

When regularization works properly, model accuracy on the training data is often lower than accuracy on the training data without regularization, but model accuracy on the test data, which is the key metric, is higher.

The underlying math principle is that L2 regularization works by adding a term that penalizes weight values to the error function. The ideas are illustrated by the equations in Figure 4-4.

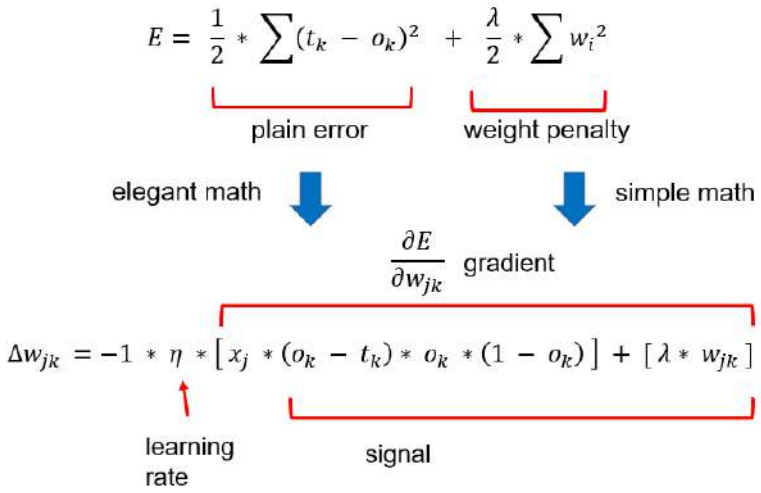


Figure 4-4: Adding a Weight Penalty to Error Changes Weight Deltas

You add a fraction (often called the “L2 regularization constant” and given the lowercase lambda Greek letter) of the sum of the squared weight values to the error. The top-most equation is squared error augmented with the L2 weight penalty: one-half the sum of the squared differences between the target values and the computed output values, plus one-half a constant lambda times the sum of the squared weight values.

During ordinary training, the back-propagation algorithm iteratively adds a weight-delta (which can be positive or negative) to each weight. The weight-delta is a fraction (called the learning rate, usually given lowercase eta Greek symbol, which resembles a script 'n') of the weight gradient. The weight gradient is the calculus derivative of the error function.

To use L2 regularization, you have to add the derivative of the weight penalty. Luckily, this is quite easy. You can see this as the last term in the gradient part of the weight-delta equation.

Let me emphasize that I've abused the formal math notation somewhat, and this explanation is not mathematically one hundred percent accurate. But a thorough, rigorous explanation of the relationship between back-propagation and L2 regularization with all the details would take many pages of messy math.

Although the L2 regularization math concepts are complex, modifying basic back-propagation training to use L2 regularization is surprisingly simple. The program People_regularization.js makes only a few changes to the People_nn.js program.

The definition of class method **`train()`** requires a new parameter for the L2 regularization weight term.

```
  train(trainX, trainY, lrnRate, maxEpochs, lamda)
  {
. . .
```

I use the misspelled "lamda" rather than "lambda" because lambda is a reserved word in many programming languages. In the **train()** implementation, the computations for the hidden-to-output gradients and the input-to-hidden gradients have to be changed slightly to add a weight penalty.

```
  // compute hidden-to-output weight gradients using output signals.
  for (let j = 0; j < this.nh; ++j) {
    for (let k = 0; k < this.no; ++k) {
      hoGrads[j][k] = oSignals[k] * this.hNodes[j];
      hoGrads[j][k] += lamda * this.hoWeights[j][k];  // L2 penalty
    }
  }

. . .

  // compute input-to-hidden weight gradients using hidden signals.
  for (let i = 0; i < this.ni; ++i) {
    for (let j = 0; j < this.nh; ++j) {
      ihGrads[i][j] = hSignals[j] * this.iNodes[i];
      ihGrads[i][j] += lamda * this.ihWeights[i][j];  // L2 penalty
    }
  }
```

In short, to add L2 regularization to standard back-propagation, you only need to add two lines of code.

To call the **train()** method, you just need to add the lambda L2 weighting term.

```
  // 3. train network
  let lrnRate = 0.01;
  let maxEpochs = 50000;
  let lamda = 0.0005;
  console.log("\nStarting training with learning rate = 0.01 lambda =
0.0005");
  nn.train(trainX, trainY, lrnRate, maxEpochs, lamda);
  console.log("Training complete");
```

As it turns out, L2 regularization is extremely sensitive to the value used for the lambda weighting term and finding a good value for lambda can be difficult. But when regularization works, it can be highly effective in reducing model overfitting.

Researchers worked out the details of L2 regularization for neural networks in the 1990s. But at the same time, engineers discovered the same basic idea independently, and called the technique “weight decay.” Formal L2 regularization decreases the value of a network's gradients, and then the gradients are used to decrease the magnitude of the network's weights and biases.

Weight decay skips the modification of the gradients and directly reduces the magnitude of the value of weights and biases on each iteration. The weight decay mechanism is much simpler than formal L2 regularization. The computation of gradients is done exactly as it is without regularization. Then during weight and biases updates, after each update, the magnitude is decreased. For example, the following.

```
// update input-to-hidden weights and apply weight decay.
for (let i = 0; i < this.ni; ++i) {
  for (let j = 0; j < this.nh; ++j) {
    let delta = -1.0 * lrnRate * ihGrads[i][j];
    this.ihWeights[i][j] += delta;
    this.ihWeights[i][j] *= lamda;  // decrease weight (different from L2
lambda).
  }
}
```

Somewhat unfortunately, both L2 regularization and weight decay regularization usually use a parameter named **`lambda`** as the scaling constant. However, by looking at the code, you can see that the lambda values are used very differently. The point is, if you're using a neural network library, the meaning of a regularization lambda can vary (although larger values of lambda ultimately reduce weights more than smaller values).

A technique that is closely related to L2 regularization is called max norm constraint. The idea is to put a fixed limit on the magnitude of all weights. For example, immediately after updating each weight and bias in the **`train()`** method, you check to see if the new weight value is greater than a fixed threshold value (perhaps 10.0) or less than the corresponding threshold value (-10.0), and if so, you adjust the weight to the threshold value.

Dropout

Dropout is an advanced technique used to reduce the likelihood of neural network overfitting. The technique sounds strange: on each training iteration, a portion (typically something like 0.50 or 0.20) of the network's hidden nodes are randomly dropped, meaning the training algorithm acts as if the nodes that have been selected to be dropped don't exist.

Dropout can be used for classification problems or regression problems. It can be used in conjunction with other techniques such as regularization, or it can be used by itself.

The screenshot in Figure 4-5 shows an example of training using dropout. The goal is to predict a person's political leaning (conservative, moderate, or liberal) from their sex, age, geo-region (eastern, western, or central), and annual income.

Because dropout effectively trains with just some of the hidden nodes, in general you want to use more hidden nodes than you'd use without dropout. The dropout demo uses twice as many hidden nodes (50 versus 25) as the nondropout demo. Additionally, in many situations, you may have to adjust the values of the learning rate and maximum-epochs to get good results.

Without dropout, the final training and test classification accuracies (shown in Figure 4-1) were 96.25% and 52.50%, but with dropout, the final accuracies are 65.63% and 75.00%. So, the model's accuracy on the training data is quite a bit worse, but the accuracy on the test data, which is the key metric, is significantly better.

The theory behind dropout is shown in Figure 4-6. The neural network has three input nodes, four hidden nodes, and two output nodes, and so does not correspond to the demo program. When using back-propagation, on each training iteration, each node-to-node weight (indicated by the blue arrows) and each node bias (indicated by the green arrows) are updated slightly.

With dropout, on each training iteration, a proportion of the hidden node is randomly selected to be dropped. Then the drop nodes essentially don't exist, so they don't take part in the computation of output node values, or in the computation of the weight and bias updates. In the diagram, if hidden nodes are 0-indexed, then hidden nodes [1] and [2] are selected as drop nodes on this training iteration.

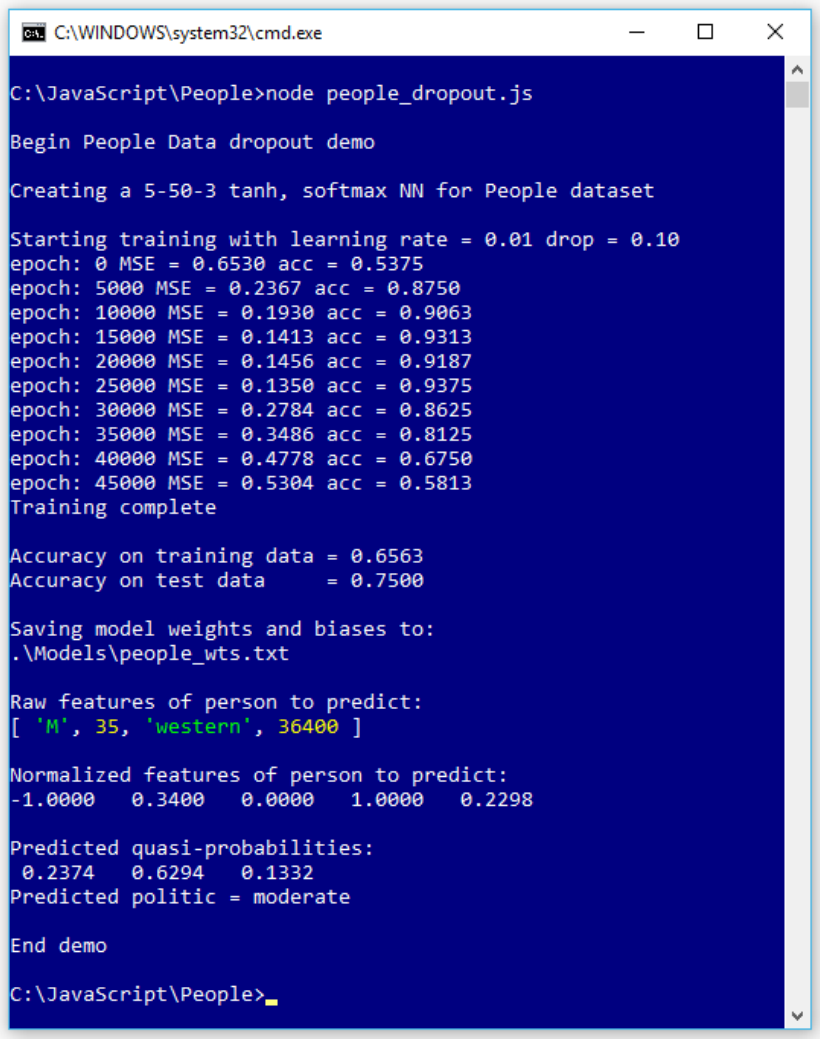


Figure 4-5: Neural Network Dropout Demo Run

Drop nodes aren't physically removed; instead, they are virtually removed by having any code that references them ignore the nodes. In this case, in addition to hidden nodes [1] and [2], the six input-to-hidden weights that terminate in the drop nodes are ignored. And the four hidden-to-output node weights that originate from the drop nodes are ignored.

It's not immediately obvious why dropout reduces overfitting. In fact, the effectiveness of dropout is not fully understood. One notion is that dropout essentially selects random subsets of neural networks, and then averages them together, giving a more robust final result. Another notion is that dropout prevents hidden nodes from relying on other hidden nodes (called *coadaptation*).

The dropout technique became well known in 2015 following the publication of a research paper. The paper presented a complex mechanism for implementing dropout, which involved many logic branches and code modification to both the training method and the forward-pass evaluation method.

The complex implementation of dropout was used for several years, until engineers noticed that the key ideas of dropout could be implemented in a much simpler way. This simpler technique for dropout implementation is called inverted dropout.

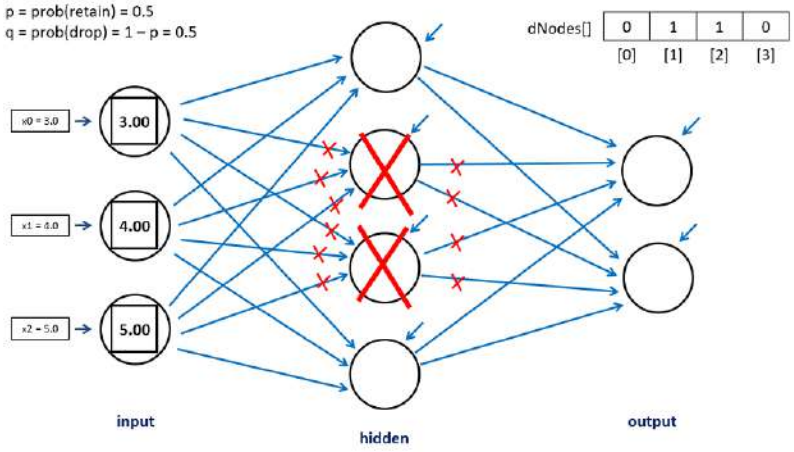


Figure 4-6: The Dropout Mechanism in Theory, but Not in Practice

There are several minor implementation variations that can be used for inverted dropout. The demo modifies the basic **`train()`** method by adding a parameter **`dropProb`** (drop probability) that controls how many hidden nodes to drop.

```
  train(trainX, trainY, lrnRate, maxEpochs, dropProb)
  {
    let keepProb = 1.0 - dropProb
    let keepNodes = U.vecMake(this.nh, 0.0);

    let hoGrads = U.matMake(this.nh, this.no, 0.0);
. . .
```

The numeric vector **keepNodes** tracks if a hidden node is marked for dropping (value = 0) or not (value = 1). An alternative is to use a vector **dropNodes** with reversed values, but **keepNodes** is more convenient programmatically. Another alternative is to use a JavaScript **boolean** type array instead of a numeric vector.

In the **train()** method, inside the for-loop that iterates through each training item, a new set of nodes to drop (or equivalently, keep) is determined for each training item.

```
  for (let ii = 0; ii < n; ++ii) {  // each item
    // select hidden nodes to drop.
    for (let j = 0; j < this.nh; ++j) {
      let p = this.rnd.next();

      if (p < dropProb) {
        keepNodes[j] = 0;  // this is a node to drop.
      }
      else {
        keepNodes[j] = 1;  // keep the node.
      }
    }
```

Immediately after calling the forward-pass **eval()** method, dropout is applied.

```
  let X = trainX[idx];
  let Y = trainY[idx];

  this.eval(X);  // output stored in this.oNodes.
  // apply inverted dropout.
  for (let j = 0; j < this.nh; ++j) {
    if (keepNodes[j] == 0) {
      this.hNodes[j] = 0.0;
    }
    else {
      this.hNodes[j] /= keepProb;
    }
  }
```

Hidden nodes that have been selected to be dropped have their values overwritten with 0.0, and nodes that aren't dropped have their values scaled larger by dividing by **keepProb**, the complement of the drop probability. Because **keepProb** is always less than 1.0, dividing by **keepProb** always increases a node value. For example, suppose a node's value is 0.90 and **dropProb** is 0.20; then **keepProb** is 0.80 and 0.90 / 0.80 = 1.125. Scaling the values of nondrop nodes larger compensates for the drop nodes not contributing to the final output node values, because the drop node values are 0.0.

A call to the **train()** method with a 10% probability of dropping each hidden node looks like the following.

```
dropProb = 0.10;
nn.train(trainX, trainY, lrnRate, maxEpochs, dropProb);  // use dropout
```

Notice that by setting the value of **dropProb** to 0.0, you get normal back-propagation training without dropout. Also note that the exact number of hidden nodes that will be dropped on each training iteration will likely vary. For example, with 50 hidden nodes, on average 0.10 * 50 = 5 nodes will be dropped. But the exact number of nodes dropped can be between 0 and 50.

The dropout technique was created mostly to deal with overfitting in deep neural networks with many hidden layers and hundreds or even thousands of nodes in each layer. But when used carefully, dropout can improve the prediction accuracy of neural networks with a single hidden layer.

It is possible to apply dropout to input nodes. This technique is most often effective when training data has many predictor variables. A closely related technique is to add random noise to input values. This technique is called *jittering*.

Train-validate-test

The train-validate-test (TVT) technique can be used to reduce the likelihood of model overfitting. Model overfitting often occurs if a neural network is trained too long. Briefly, instead of splitting your data into a training set and a test set, you split your data into three sets: a training set (typically about 70% of the data), a validation set (often about 15% of the data), and a test set (about 15% of the data).

You use the training data as usual, but every few epochs, you compute and display the error and accuracy of the current model on the validation data. By observing how the model deals with the validation data, you can sometimes see when model overfitting is starting to occur, and then stop training before reaching the max epochs limit.

After training, you compute the error and accuracy of the model on the test data as usual, to get a rough estimate of how the model will perform on new, previously unseen data. The main principle of train-validate-test is shown in the graph in Figure 4-7.

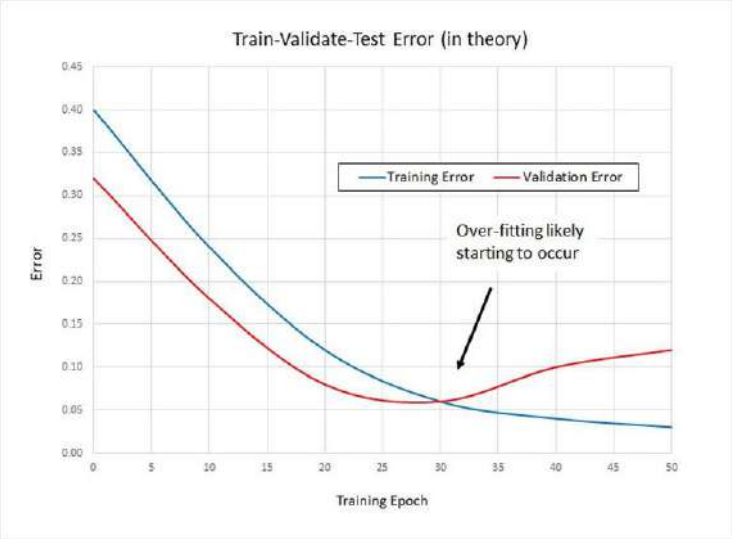


Figure 4-7: Train-Validate-Test in Theory

The blue line represents error on the training data. That error will tend to decrease steadily the longer the network is trained. The red line represents error on the validation data. Initially error will decrease, but at some point, overfitting will start to occur and error on the validation data will start to increase. This can indicate when to stop training.

Unfortunately, there are two reasons why train-validate-test often does not work well. First, you must use valuable labeled data for validation rather than for training. Second, the actual graph of training and validation error is rarely as smooth as shown in Figure 4-7. Instead, the error values for training and validation error often jump around wildly, making it very difficult to determine when overfitting is starting to occur.

The screenshot in Figure 4-8 shows an example of train-validate-test. The goal is to predict a person's political leaning (conservative, moderate, or liberal) from their sex, age, geo-region (eastern, western, or central), and annual income.

The demo program has a total of 240 labeled data items and uses three files: People_train.txt has 160 items, People_validate.txt has 40 items, and People_test.txt has 40 items. The data is synthetic and was generated programmatically.

During training, the demo program computes and displays error and classification prediction accuracy on both the training and validation data. In several runs of the program using 50,000 training epochs, the error and accuracy metrics that were being displayed suggested that overfitting was starting to occur at somewhere between 1,000 and 2,000 epochs. After a bit of experimentation, a value of 1,600 epochs was used to generate the results shown in Figure 4-8.

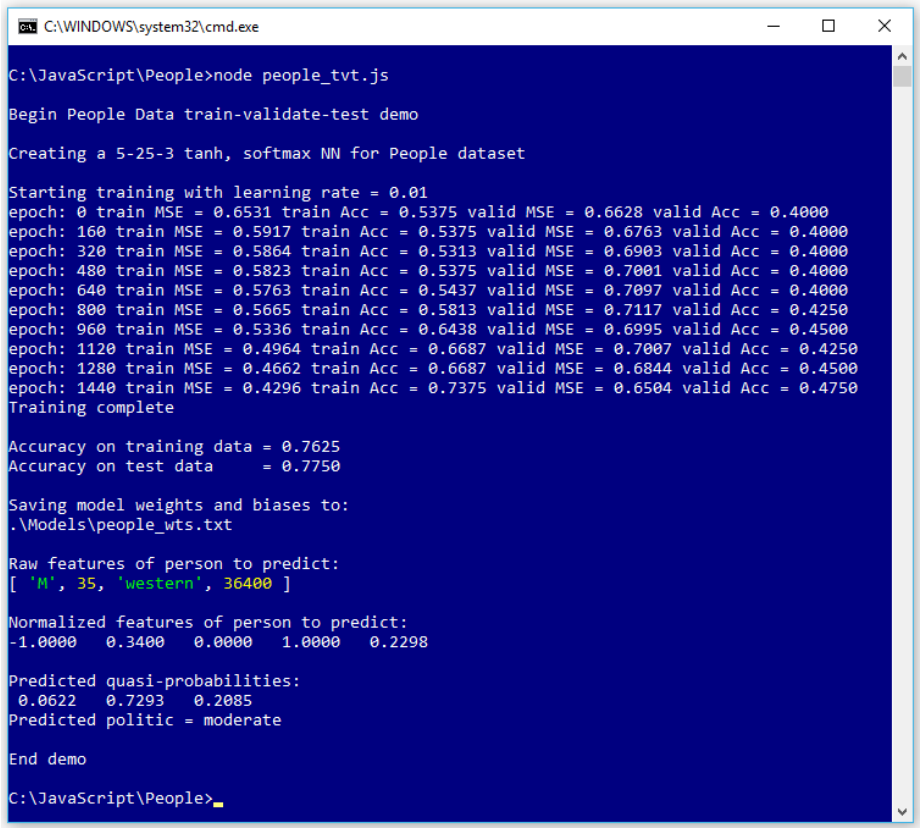


Figure 4-8: Train-Validate-Test Demo

Without train-validate-test, the final training and test classification accuracies (shown in Figure 4-1) were 96.25% and 52.50%, but by using train-validate-test to limit the number of training epochs, the final accuracies are 76.25% and 77.50%. So the model's accuracy on the training data is somewhat worse, but the accuracy on the test data, which is the key metric, is significantly better.

Implementing train-validate-test is straightforward. In the **`main()`** function you load the three files into six matrices.

```
let trainX = U.loadTxt(".\\Data\\people_train.txt", "\t", [0,1,2,3,4]);
let trainY = U.loadTxt(".\\Data\\people_train.txt", "\t", [5,6,7]);

let validX = U.loadTxt(".\\Data\\people_validate.txt", "\t", [0,1,2,3,4]);
let validY = U.loadTxt(".\\Data\\people_validate.txt", "\t", [5,6,7]);

let testX = U.loadTxt(".\\Data\\people_test.txt", "\t", [0,1,2,3,4]);
let testY = U.loadTxt(".\\Data\\people_test.txt", "\t", [5,6,7]);
```

In the **`train()`** method, you compute and display error and accuracy on the training and validation data every few epochs.

```
if (epoch % freq == 0) {
  let trainMSE = this.meanSqErr(trainX, trainY).toFixed(4);
  let trainAcc = this.accuracy(trainX, trainY).toFixed(4);
  let validMSE = this.meanSqErr(validX, validY).toFixed(4);
  let validAcc = this.accuracy(validX, validY).toFixed(4);

  let s1 = "epoch: " + epoch.toString();
  let s2 = " train MSE = " + trainMSE.toString();
  let s3 = " train Acc = " + trainAcc.toString();
  let s4 = " valid MSE = " + validMSE.toString();
  let s5 = " valid Acc = " + validAcc.toString();

  console.log(s1 + s2 + s3+ s4 + s5);
}
```

Determining when overfitting is starting to occur is part art and part science, as the saying goes. There have been attempts to design algorithms and code that programmatically determine when overfitting is starting to occur, but these efforts have not succeeded very often.

Using train-validate-test is sometimes called *early stopping*. However, early stopping is a general term that can be used to describe any technique to halt training before reaching the max-epochs value.

Questions

1. Suppose you want to predict political party using this tiny set of training data:

```
Age Sex     IQ   Party
-----------------------------------------
46  male    120  independent
24  female  100  democrat
38  male    116  republican
```

A. Normalize the Age values using min-max normalization.

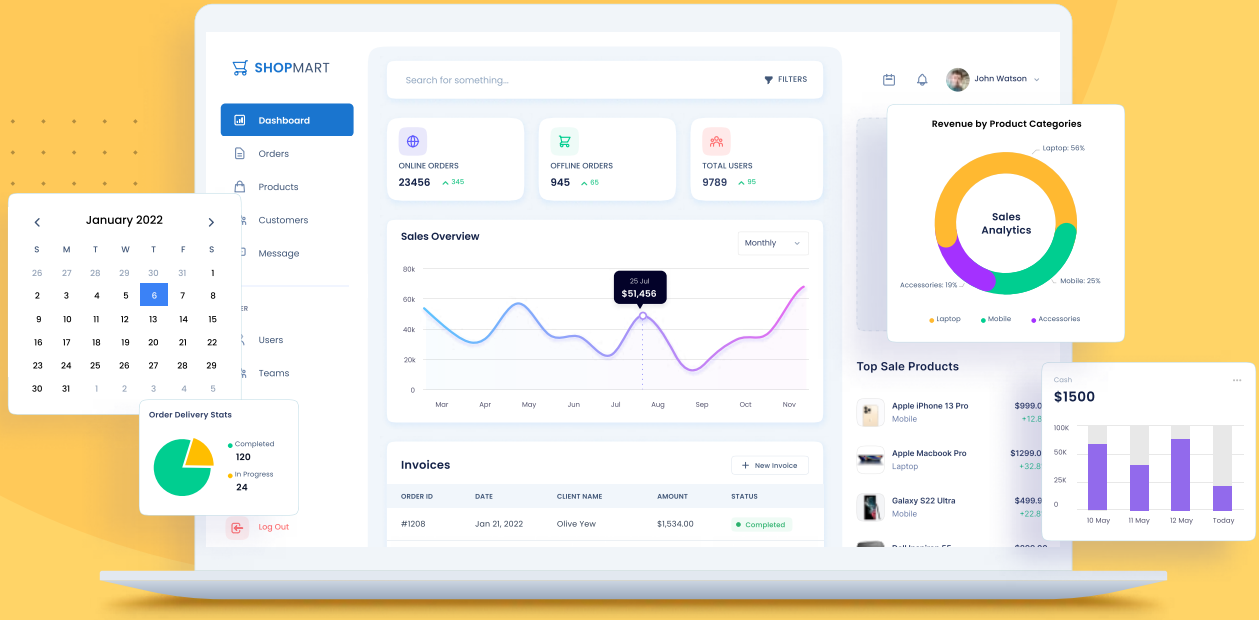
B. Encode the Sex values using -1 +1 encoding.

C. Normalize the IQ values using z-score normalization (note: mean = 112.0, sd = 8.64).

D. Encode the Party values using one-hot / 1-of-N encoding.

The Pure JavaScript UI controls library

4.6 out of 5 stars



GET YOUR FREE JAVASCRIPT UI COMPONENTS

syncfusion.com/communitylicense

- 65+ JavaScript UI controls for building mobile and web apps
- Responsive and touch friendly on all devices.
- Stunning built-in themes available with UI customization.
- One of the best JavaScript control libraries in the market.
- Expect new features and widgets frequently.
- A wide range of product demos, documentation, and video tutorials.

Trusted by the world's leading companies

Syncfusion®

Chapter 5 Regression

A regression problem is one where the goal is to predict a single numeric value. For example, you might want to predict the college GPA of an incoming freshman based on their high school GPA, parent's annual income, and so on.

Designing and training a neural network for a regression problem is quite similar to designing and creating a neural network for a classification problem. The key differences are that a neural network for regression has a single output node and uses identity activation rather than softmax activation, and regression accuracy requires a different approach than classification accuracy.

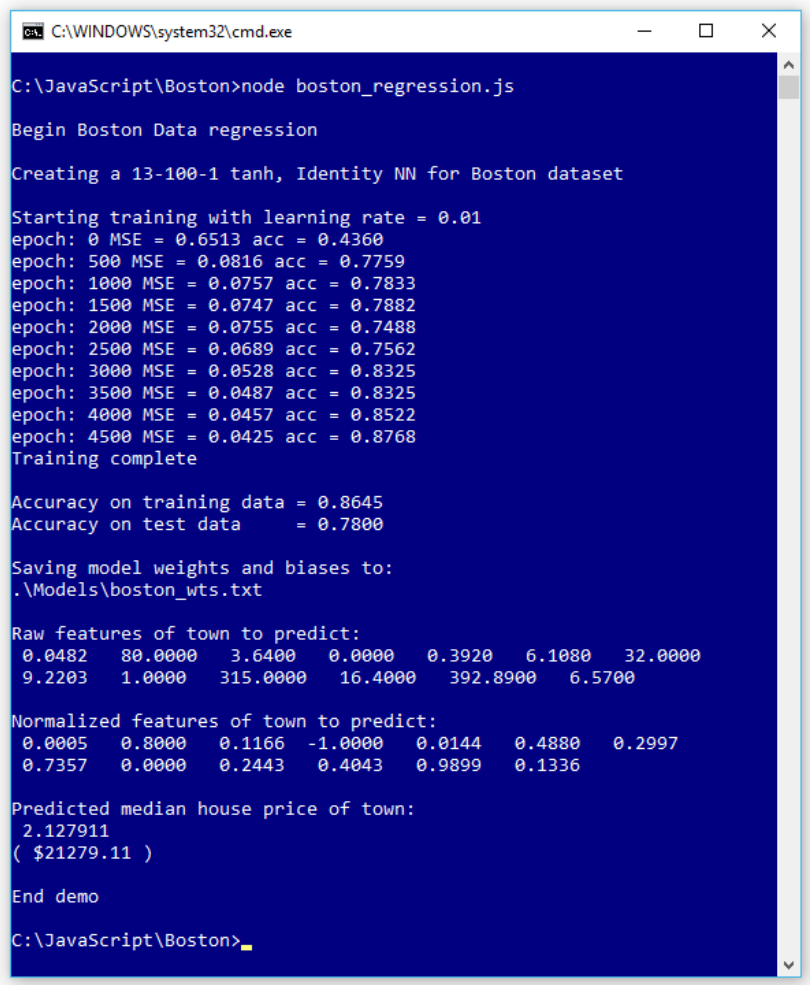


Figure 5-1: Neural Network Regression on the Boston Housing Dataset

The screenshot in Figure 5-1 shows a demo of neural network regression. The goal is to predict the median house price of a town near Boston. The data comes from a 1978 research paper, so the median house prices are very small (between $5,000 and $50,000).

The Boston Dataset is a well-known collection containing 506 data items. Each item represents one of 506 towns near Boston. There are 13 predictor variables, including things such as the crime rate in the town and whether the town is next to a river (0 = no, 1 = yes).

The demo program sets up a 13-100-1 neural network. The network has just one output node because the goal is to predict a single numeric value. The network has 100 hidden nodes. The number of hidden nodes is a hyperparameter that must be determined by trial and error.

Before training, the 506-item data set was randomly split into a 406-item set for training and a 100-item set for testing. The training data set was normalized and encoded. The demo program uses the online back-propagation training technique for 5,000 epochs.

After training, the model accuracy on the training data was 86.45% (351 out of 406) and was 78.00% (78 out of 100) on the test data. Here, a correct median house price prediction is one that is within 15% of the true price. For example, if a town's median house price is $10,000, then any prediction from $8,500 to $11,500 would be considered correct.

The demo concludes by making a prediction for a new, previously unseen town. A set of 13 raw values was set up and then normalized using the normalization parameters from the training data. The predicted median house price of the new town is $21,279.11.

Understanding the Boston Dataset

The Boston Dataset has a total of 14 variables:

- Crime rate in the town.
- Percentage of land area zoned for large lots.
- Percentage of area zoned for industrial use.
- Whether the town is adjacent to a river (0 = no, 1 = yes).
- Air pollution.
- Average number of rooms per house.
- Percentage of old homes.
- Distance to Boston metric.
- Accessibility to highway metric.
- Property tax rate.
- Pupil-teacher ratio.
- Percentage of black residents metric.
- Percentage of low social status residents.
- Median value of home in town.

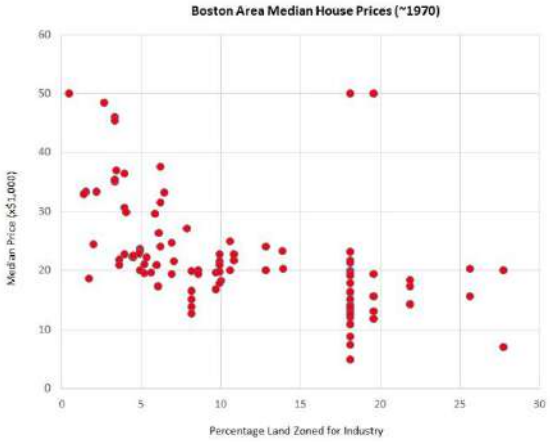


Figure 5-2: Partial Boston House Data

Because there are 14 variables, it's not possible to visualize the data set, but you can get a rough idea of the data from the graph in Figure 5-2. The graph shows median house price in a town as a function of the percent area in the town zoned for industrial use.

The industry variable, by itself, cannot make a good prediction of the median house price. For example, if you knew a town had 18% of land area zoned for industrial use, the median house price could be anything between $5,000 and $25,000 (house prices were very low in the 1970s).

The raw data looks like the following.

```
0.19802  80  10.59  0  0.489  6.182  42.4  3.9454  4  277  18.6  393.63  9.47  25
```

There is no missing data. All of the 13 predictor variables, except for the binary adjacent-to-river variable, were min-max normalized. The adjacent-to-river variable was -1 +1 encoded (-1 = no, +1 = yes).

The variable to predict, median house price, was already normalized by dividing by 1,000. For example, in the sample raw data shown previously, the median house price is $25,000. I re-normalized median house price by dividing by 10 because smaller values are usually easier for neural regression to predict.

After normalization and encoding, the data looks like the following.

```
0.0021 0.0534 0.3713 -1 0.2139 0.5022 0.4067 0.2560 0.1304 0.1717 0.6382 0.9917 0.2135  2.50
```

Because there is only one categorical predictor (adjacent-to-river), and it was -1 +1 encoded, the normalized data has 13 predictor variables just like the raw data. After the training data was normalized and encoded, and the 100-item test data was normalized using the same min-max parameters from the training data, the adjacent-to-river values were -1 +1 encoded, and the median house prices were divided by 10.

The Regression Dataset demo program

The complete demo program shown in Figure 5-1 is presented in Code Listing 5-1. The demo assumes there is a top-level directory named JavaScript that contains subdirectories named Utilities and Boston. The Utilities directory contains the Utilities_lib.js library file. The Boston directory contains the demo program Boston_regression.js and subdirectories named Data and Models. The Data directory contains files Boston_train.txt and Boston_test.txt. The Models directory is used to store trained model weights and biases values.

Code Listing 5-1: The Boston Dataset Regression Demo Program Source Code

```
// boston_regression.js
// ES6

let U = require("../Utilities/utilities_lib.js");
let FS = require("fs");

//
================================================================================
==

class NeuralNet
{
  constructor(numInput, numHidden, numOutput, seed) {
    this.rnd = new U.Erratic(seed);

    this.ni = numInput;
    this.nh = numHidden;
    this.no = numOutput;

    this.iNodes = U.vecMake(this.ni, 0.0);
    this.hNodes = U.vecMake(this.nh, 0.0);
    this.oNodes = U.vecMake(this.no, 0.0);

    this.ihWeights = U.matMake(this.ni, this.nh, 0.0);
    this.hoWeights = U.matMake(this.nh, this.no, 0.0);

    this.hBiases = U.vecMake(this.nh, 0.0);
    this.oBiases = U.vecMake(this.no, 0.0);

    this.initWeights();
  }

  initWeights()
  {
    let lo = -0.01;
    let hi = 0.01;
    for (let i = 0; i < this.ni; ++i) {
      for (let j = 0; j < this.nh; ++j) {
```

```
      this.ihWeights[i][j] = (hi - lo) * this.rnd.next() + lo;
    }
  }

  for (let j = 0; j < this.nh; ++j) {
    for (let k = 0; k < this.no; ++k) {
      this.hoWeights[j][k] = (hi - lo) * this.rnd.next() + lo;
    }
  }
}
eval(X)
{
  // regression: no output activation.
  let hSums = U.vecMake(this.nh, 0.0);
  let oSums = U.vecMake(this.no, 0.0);

  this.iNodes = X;

  for (let j = 0; j < this.nh; ++j) {
    for (let i = 0; i < this.ni; ++i) {
      hSums[j] += this.iNodes[i] * this.ihWeights[i][j];
    }
    hSums[j] += this.hBiases[j];
    this.hNodes[j] = U.hyperTan(hSums[j]);
  }

  for (let k = 0; k < this.no; ++k) {
    for (let j = 0; j < this.nh; ++j) {
      oSums[k] += this.hNodes[j] * this.hoWeights[j][k];
    }
    oSums[k] += this.oBiases[k];
  }

  // this.oNodes = U.softmax(oSums);
  for (let k = 0; k < this.no; ++k) {  // aka "Identity"
    this.oNodes[k] = oSums[k];
  }

  let result = [];
  for (let k = 0; k < this.no; ++k) {
    result[k] = this.oNodes[k];
  }
  return result;
} // eval()

setWeights(wts)
{
  // order: ihWts, hBiases, hoWts, oBiases
  let p = 0;
```

```
    for (let i = 0; i < this.ni; ++i) {
      for (let j = 0; j < this.nh; ++j) {
        this.ihWeights[i][j] = wts[p++];
      }
    }

    for (let j = 0; j < this.nh; ++j) {
      this.hBiases[j] = wts[p++];
    }

    for (let j = 0; j < this.nh; ++j) {
      for (let k = 0; k < this.no; ++k) {
        this.hoWeights[j][k] = wts[p++];
      }
    }

    for (let k = 0; k < this.no; ++k) {
      this.oBiases[k] = wts[p++];
    }
  } // setWeights()

  getWeights()
  {
    // order: ihWts, hBiases, hoWts, oBiases
    let numWts = (this.ni * this.nh) + this.nh +
      (this.nh * this.no) + this.no;
    let result = U.vecMake(numWts, 0.0);
    let p = 0;
    for (let i = 0; i < this.ni; ++i) {
      for (let j = 0; j < this.nh; ++j) {
        result[p++] = this.ihWeights[i][j];
      }
    }

    for (let j = 0; j < this.nh; ++j) {
      result[p++] = this.hBiases[j];
    }

    for (let j = 0; j < this.nh; ++j) {
      for (let k = 0; k < this.no; ++k) {
        result[p++] = this.hoWeights[j][k];
      }
    }

    for (let k = 0; k < this.no; ++k) {
      result[p++] = this.oBiases[k];
    }
    return result;
```

```
  } // getWeights()

  shuffle(v)
  {
    // Fisher-Yates
    let n = v.length;
    for (let i = 0; i < n; ++i) {
      let r = this.rnd.nextInt(i, n);
      let tmp = v[r];
      v[r] = v[i];
      v[i] = tmp;
    }
  }

  train(trainX, trainY, lrnRate, maxEpochs)
  {
    // regression: no output activation => f(x)=x => f'(x)=1
    let hoGrads = U.matMake(this.nh, this.no, 0.0);
    let obGrads = U.vecMake(this.no, 0.0);
    let ihGrads = U.matMake(this.ni, this.nh, 0.0);
    let hbGrads = U.vecMake(this.nh, 0.0);

    let oSignals = U.vecMake(this.no, 0.0);
    let hSignals = U.vecMake(this.nh, 0.0);

    let n = trainX.length;  // 406
    let indices = U.arange(n);  // [0,1,..,405]
    let freq = Math.trunc(maxEpochs / 10);

    for (let epoch = 0; epoch < maxEpochs; ++epoch) {
      this.shuffle(indices);  //
      for (let ii = 0; ii < n; ++ii) {  // each item
        let idx = indices[ii];
        let X = trainX[idx];
        let Y = trainY[idx];
        this.eval(X);  // output stored in this.oNodes.

        // compute output node signals.
        for (let k = 0; k < this.no; ++k) {
          // let derivative = (1 - this.oNodes[k]) * this.oNodes[k];  //
softmax
          let derivative = 1;  // identity activation
          oSignals[k] = derivative * (this.oNodes[k] - Y[k]);  // E=(t-o)^2
        }

        // compute hidden-to-output weight gradients using output signals.
        for (let j = 0; j < this.nh; ++j) {
          for (let k = 0; k < this.no; ++k) {
            hoGrads[j][k] = oSignals[k] * this.hNodes[j];
```

```
      }
    }

    // compute output node bias gradients using output signals.
    for (let k = 0; k < this.no; ++k) {
      obGrads[k] = oSignals[k] * 1.0;  // 1.0 dummy input can be
dropped.
    }

    // compute hidden node signals.
    for (let j = 0; j < this.nh; ++j) {
      let sum = 0.0;
      for (let k = 0; k < this.no; ++k) {
        sum += oSignals[k] * this.hoWeights[j][k];
      }
      let derivative = (1 - this.hNodes[j]) * (1 + this.hNodes[j]);  //
tanh
      hSignals[j] = derivative * sum;
    }

    // compute input-to-hidden weight gradients using hidden signals.
    for (let i = 0; i < this.ni; ++i) {
      for (let j = 0; j < this.nh; ++j) {
        ihGrads[i][j] = hSignals[j] * this.iNodes[i];
      }
    }

    // compute hidden node bias gradients using hidden signals.
    for (let j = 0; j < this.nh; ++j) {
      hbGrads[j] = hSignals[j] * 1.0;  // 1.0 dummy input can be
dropped.
    }

    // update input-to-hidden weights.
    for (let i = 0; i < this.ni; ++i) {
      for (let j = 0; j < this.nh; ++j) {
        let delta = -1.0 * lrnRate * ihGrads[i][j];
        this.ihWeights[i][j] += delta;
      }
    }

    // update hidden node biases.
    for (let j = 0; j < this.nh; ++j) {
      let delta = -1.0 * lrnRate * hbGrads[j];
      this.hBiases[j] += delta;
    }

    // update hidden-to-output weights.
    for (let j = 0; j < this.nh; ++j) {
```

```
        for (let k = 0; k < this.no; ++k) {
          let delta = -1.0 * lrnRate * hoGrads[j][k];
          this.hoWeights[j][k] += delta;
        }
      }

      // update output node biases.
      for (let k = 0; k < this.no; ++k) {
        let delta = -1.0 * lrnRate * obGrads[k];
        this.oBiases[k] += delta;
      }
    } // ii

    if (epoch % freq == 0) {
      let mse = this.meanSqErr(trainX, trainY).toFixed(4);
      let acc = this.accuracy(trainX, trainY, 0.15).toFixed(4);

      let s1 = "epoch: " + epoch.toString();
      let s2 = " MSE = " + mse.toString();
      let s3 = " acc = " + acc.toString();

      console.log(s1 + s2 + s3);
    }

  } // epoch
} // train()

// cross-entropy error not applicable to regression problems.
meanSqErr(dataX, dataY)
{
  let sumSE = 0.0;
  for (let i = 0; i < dataX.length; ++i) {  // each data item
    let X = dataX[i];
    let y = dataY[i];  // target output like [2.3] as matrix
    let oupt = this.eval(X);  // computed like [2.07]

    for (let k = 0; k < this.no; ++k) {
      let err = y[k] - oupt[k];
    }
    let err = y[0] - oupt[0];
    sumSE += err * err;
  }
  return sumSE / dataX.length;
}

accuracy(dataX, dataY, pctClose)
{
  // correct if predicted is within pctClose of target.
  let nc = 0; let nw = 0;
```

```
    for (let i = 0; i < dataX.length; ++i) {  // each data item
      let X = dataX[i];
      let y = dataY[i];  // target output
      let oupt = this.eval(X);  // computed output

      if (Math.abs(oupt[0] - y[0]) < Math.abs(pctClose * y[0])) {
        ++nc;
      }
      else {
        ++nw;
      }
    }
    return nc / (nc + nw);
  }

  saveWeights(fn)
  {
    let wts = this.getWeights();
    let n = wts.length;
    let s = "";
    for (let i = 0; i < n - 1; ++i) {
      s += wts[i].toString() + ",";
    }
    s += wts[n - 1];

    FS.writeFileSync(fn, s);
  }

  loadWeights(fn)
  {
    let n = (this.ni * this.nh) + this.nh + (this.nh * this.no) + this.no;
    let wts = U.vecMake(n, 0.0);
    let all = FS.readFileSync(fn, "utf8");
    let strVals = all.split(",");
    let nn = strVals.length;
    if (n != nn) {
      throw ("Size error in NeuralNet.loadWeights()");
    }
    for (let i = 0; i < n; ++i) {
      wts[i] = parseFloat(strVals[i]);
    }
    this.setWeights(wts);
  }
} // NeuralNet

//
================================================================================
==
```

```
function main()
{
  process.stdout.write("\033[0m");  // reset
  process.stdout.write("\x1b[1m" + "\x1b[37m");  // bright white
  console.log("\nBegin Boston Data regression ");

  // 1. load data
  let trainX = U.loadTxt(".\\Data\\boston_train.txt", "\t",
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]);
  let trainY = U.loadTxt(".\\Data\\boston_train.txt", "\t", [13]);
  let testX = U.loadTxt(".\\Data\\boston_test.txt", "\t",
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]);
  let testY = U.loadTxt(".\\Data\\boston_test.txt", "\t", [13]);

  // 2. create network
  console.log("\nCreating a 13-100-1 tanh, Identity NN for Boston
dataset");
  let seed = 0;
  let nn = new NeuralNet(13, 100, 1, seed);

  // 3. train network
  let lrnRate = 0.01;
  let maxEpochs = 5000;
  console.log("\nStarting training with learning rate = 0.01 ");
  nn.train(trainX, trainY, lrnRate, maxEpochs);
  console.log("Training complete");

  // 4. evaluate model
  let trainAcc = nn.accuracy(trainX, trainY, 0.15);
  let testAcc = nn.accuracy(testX, testY, 0.15);
  console.log("\nAccuracy on training data = " +
    trainAcc.toFixed(4).toString());
  console.log("Accuracy on test data     = " +
    testAcc.toFixed(4).toString());

  // 5. save model
  let fn = ".\\Models\\boston_wts.txt";
  console.log("\nSaving model weights and biases to: ");
  console.log(fn);
  nn.saveWeights(fn);

  // 6. use trained model
  let unknownRaw = [0.04819, 80, 3.64, 0, 0.392, 6.108, 32, 9.2203, 1, 315,
    16.4, 392.89, 6.57];
  let unknownNorm = [0.000471, 0.800000, 0.116569, -1, 0.014403, 0.488025,
    0.299691, 0.735726, 0.000000, 0.244275, 0.404255, 0.989889, 0.133554];
  let predicted = nn.eval(unknownNorm);

  console.log("\nRaw features of town to predict: ");
```

```
  U.vecShow(unknownRaw, 4, 7);

  console.log("\nNormalized features of town to predict: ");
  U.vecShow(unknownNorm, 4, 7);

  console.log("\nPredicted median house price of town: ");
  U.vecShow(predicted, 6, 1);  // predicted is a vector.

  let predPrice = predicted[0] * 10000;
  console.log("( $" + predPrice.toFixed(2).toString() + " )");

  process.stdout.write("\033[0m");  // reset
  console.log("\nEnd demo");

} // main()

main();
```

All of the control logic is contained in a top-level **main()** function. Program execution begins by loading the training and test data into memory.

```
  // 1. load data
  let trainX = U.loadTxt(".\\Data\\boston_train.txt", "\t",
    [0,1,2,3,4,5,6,7,8,9,10,11,12]);
  let trainY = U.loadTxt(".\\Data\\boston_train.txt", "\t", [13]);
  let testX = U.loadTxt(".\\Data\\boston_test.txt", "\t",
    [0,1,2,3,4,5,6,7,8,9,10,11,12]);
  let testY = U.loadTxt(".\\Data\\boston_test.txt", "\t", [13]);
. . .
```

The **loadTxt()** function is contained in the File utilities_lib.js file, which is located in the Utilities directory. Notice that the data files are tab-delimited. Next, a neural network is created.

```
  // 2. create network
  console.log("\nCreating a 13-100-1 tanh, Identity NN for Boston dataset");
  let seed = 0;
  let nn = new NeuralNet(13, 100, 1, seed);
. . .
```

The normalized and encode data has 13 predictor variables. There is just one output node because the goal is to predict a single numeric value. The number of hidden is a hyperparameter that must be determined by trial and error. In general, more hidden nodes can produce a more accurate model at the expense of an increased risk of model overfitting. Next, the network is trained.

```
  // 3. train network
  let lrnRate = 0.01;
  let maxEpochs = 5000;
  console.log("\nStarting training with learning rate = 0.01 ");
  nn.train(trainX, trainY, lrnRate, maxEpochs);
  console.log("Training complete");
. . .
```

The learning rate and maximum number of training epochs are hyperparameters. Next, the trained mode is evaluated.

```
  // 4. evaluate model
  let trainAcc = nn.accuracy(trainX, trainY, 0.15);
  let testAcc = nn.accuracy(testX, testY, 0.15);
  console.log("\nAccuracy on training data = " +
    trainAcc.toFixed(4).toString());
  console.log("Accuracy on test data     = " +
    testAcc.toFixed(4).toString());
. . .
```

The **0.15** argument passed to the **accuracy()** method is how close to the predicted median house value the computed value must be in order to be considered correct. The closeness parameter will vary from problem to problem.

Next, the trained model's weights and biases are saved to a text file.

```
  // 5. save model
  let fn = ".\\Models\\boston_wts.txt";
  console.log("\nSaving model weights and biases to: ");
  console.log(fn);
  nn.saveWeights(fn);
. . .
```

The demo program concludes by making a prediction for a new, previously unseen hypothetical town near Boston.

```
  // 6. use trained model
  let unknownRaw = [0.04819, 80, 3.64, 0, 0.392, 6.108, 32, 9.2203, 1, 315,
    16.4, 392.89, 6.57];
  let unknownNorm = [0.000471, 0.800000, 0.116569, -1, 0.014403, 0.488025,
    0.299691, 0.735726, 0.000000, 0.244275, 0.404255, 0.989889, 0.133554];

  let predicted = nn.eval(unknownNorm);

  console.log("\nRaw features of town to predict: ");
  U.vecShow(unknownRaw, 4, 7);

  console.log("\nNormalized features of town to predict: ");
  U.vecShow(unknownNorm, 4, 7);
```

```
  console.log("\nPredicted median house price of town: ");
  U.vecShow(predicted, 6, 1);

  let predPrice = predicted[0] * 10000;
  console.log("( $" + predPrice.toFixed(2).toString() + " )" );
```

The demo program normalizes and encodes the new town item offline. An alternative that is useful when making many predictions is to write a problem-specific function to normalize and encode. For example, using such a function might look like the following.

```
let unknownRaw = [0.04819, 80, 3.64, 0, 0.392, (etc.)];
let unkNorm = normAndEncode(unknownRaw);
let predicted = nn.eval(unkNorm);
```

Note that a program-defined function to normalize and encode raw data needs to know the normalization parameters, such as **min** and **max**, if min-max normalization is used.

Because the predicted output value is a house price divided by 10,000, the demo program wraps up by displaying the predicted price in a slightly more friendly format.

Identity activation for regression

A neural network classifier has multiple output nodes and uses softmax activation on the output nodes so that their values sum to 1.0 and can be loosely interpreted as probabilities. In back-propagation training, the derivative of the function used for output-layer activation is used to compute the gradients, which in turn are used to update the network's weights and biases.

A neural network for regression uses just a single output node and uses the identity function for output layer activation. The identity function is just f(x) = x, or in other words, the identity function doesn't do anything. So it's also somewhat correct to say that a neural network for regression has no output layer activation function.

If y = softmax(x), then the calculus derivative is y' = y * (1 - y). If y = x (the identity function), then the calculus derivative is the constant value 1.

In method **train()**, the output node signals are computed using the following statements.

```
  // compute output node signals.
  for (let k = 0; k < this.no; ++k) {
    // let derivative = (1 - this.oNodes[k]) * this.oNodes[k];  // softmax
    let derivative = 1;  // identity activation
    oSignals[k] = derivative * (this.oNodes[k] - Y[k]);  // E=(t-o)^2
  }
```

I left the statement used for a neural network classifier in as a comment so you can see the difference between softmax activation and identity activation.

In the **eval()** method, the code for regression is the following.

```
. . .
  // compute output node before activation.
  for (let k = 0; k < this.no; ++k) {
    for (let j = 0; j < this.nh; ++j) {
      oSums[k] += this.hNodes[j] * this.hoWeights[j][k];
    }
    oSums[k] += this.oBiases[k];
  }

  // this.oNodes = U.softmax(oSums);  // for classifier
  for (let k = 0; k < this.no; ++k) {  // aka "Identity"
    this.oNodes[k] = oSums[k];  // copy as-is
  }
```

No activation function is applied when performing regression. There are some very rare situations in which you might want to apply an activation function on the output node for a regression problem. For example, if you applied $y = x^2$ as an output activation, then the calculus derivative is $y' = 2x$, and you'd apply that derivative in the training method. Additionally, note that, except for rare situations, neural networks for regression cannot use cross-entropy error because the output is not a probability distribution.

Momentum

Momentum is a technique intended to speed up training. Momentum can be applied to a neural network classification problem or a regression problem. The screenshot in Figure 5-3 shows an example of training with momentum applied to the Boston Dataset problem.

If you compare the training progress accuracy without momentum (from Figure 5-1) and with momentum, this is the result.

```
epoch   no momentum   with momentum
----------------------------------
   0     0.4360        0.5099
 500     0.7759        0.7931
1000     0.7833        0.8079
1500     0.7882        0.8128
. . .
```

You'll notice that the prediction accuracy for training with momentum improves slightly faster than training without momentum.

Recall that during training, each weight in the network is iteratively updated a little bit so that the computed output values get closer and closer to the known correct output values contained in the training data. The idea of momentum is that if an update to a weight is good in the sense that the network improves, then on the next update a bonus update is added.

On each training iteration, the weight delta that is added will get larger and larger until at some point the delta becomes too big, and the bonus is reset to zero. The demo program implements momentum by adding a momentum rate parameter to the **`train()`** method.

```
  train(trainX, trainY, lrnRate, maxEpochs, momentRate)
  {
    let hoGrads = U.matMake(this.nh, this.no, 0.0);
. . .
```

The **train()** method maintains the value of the weight deltas from the previous iteration.

```
  // for momentum
  let ihPrevWtsDeltas = U.matMake(this.ni, this.nh, 0.0);
  let hPrevBiasDeltas = U.vecMake(this.nh, 0.0);
  let hoPrevWtsDeltas = U.matMake(this.nh, this.no, 0.0);
  let oPrevBiasDeltas = U.vecMake(this.no, 0.0);
```

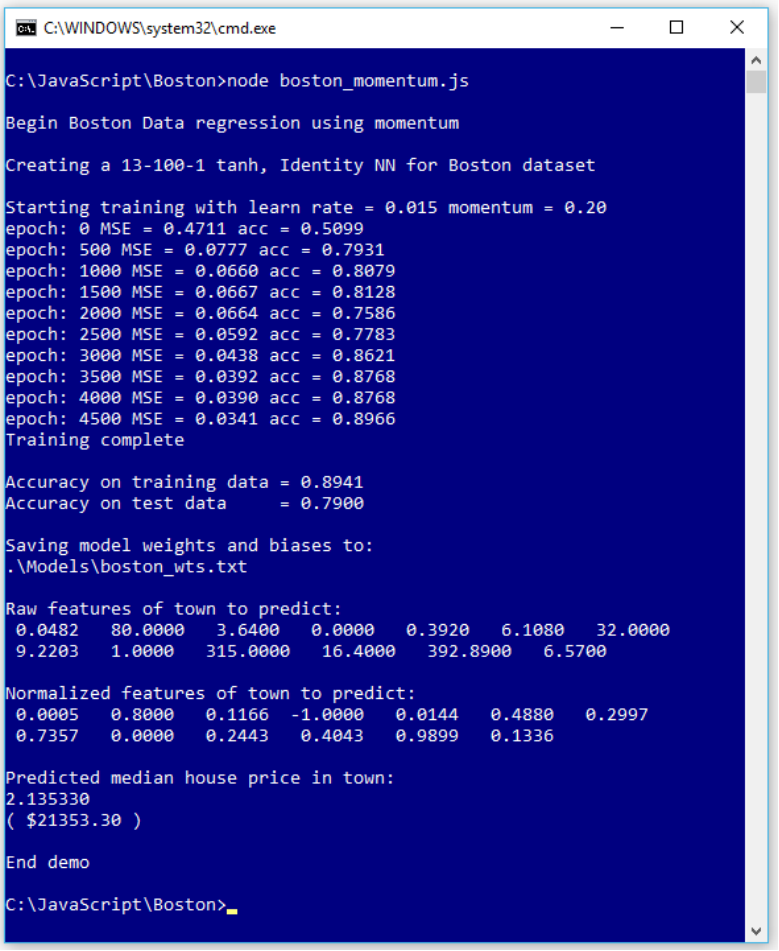


Figure 5-3: Momentum Training Demo Run

Then, on each update, a small portion of the previous delta is added as a bonus. For example, the input-to-hidden weights are updated like the following.

```
  // update input-to-hidden weights.
  for (let i = 0; i < this.ni; ++i) {
    for (let j = 0; j < this.nh; ++j) {
      let delta = -1.0 * lrnRate * ihGrads[i][j];
      this.ihWeights[i][j] += delta;
      this.ihWeights[i][j] += momentRate * ihPrevWtsDeltas[i][j];  // add a
bonus.
      ihPrevWtsDeltas[i][j] = delta;  // save the delta for next iteration.
    }
  }
```

The call to the the **`train()`** method looks like the following.

```
  // 3. train network
  let lrnRate = 0.015;
  let maxEpochs = 5000;
  let momentRate = 0.20;  // momentum typically 0.90.
  console.log("\nStarting training with learn rate = 0.015 momentum = 0.20");
  nn.train(trainX, trainY, lrnRate, maxEpochs, momentRate);
  console.log("Training complete");
```

When momentum works, it can speed up training and improve accuracy. A disadvantage is that it introduces a new hyperparameter to deal with: the momentum rate. The demo uses a momentum rate of 0.20, but a value like 0.90 is more typical. Many deep neural libraries have an advanced form of momentum called *Nesterov momentum*.

Batch and mini-batch training

Batch training is a technique that used to be the most common approach for neural network training, but is now used less often. The screenshot in Figure 5-4 shows an example of batch training applied to the Boston Dataset problem.

The most common technique for neural network training is called *online training*. Online training updates weights after processing each training item. Batch training processes all training items, and then updates weights. Batch training can be used for classification or regression problems.

In pseudo-code, the two training approaches are the following.

```
// online training
for-each training item
  compute the gradients
  use gradients from item to update weights
end-loop
```

```
// batch training
for-each training item
  compute the gradients
  accumulate the gradients
end-loop
use average of accumulated gradients from all items to update weights
```

In the early days of neural networks, batch training was very common because it is more principled. The online training approach uses a single training item to estimate the true gradient. The batch approach uses all training items to estimate the true gradient.

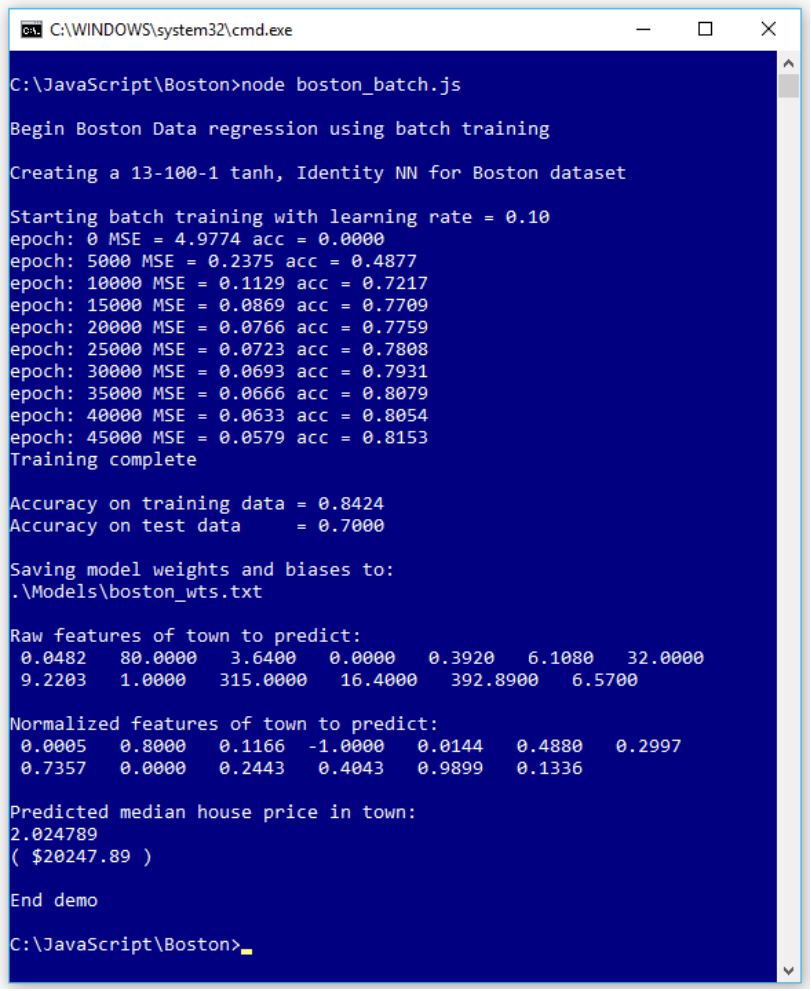


Figure 5-4: Batch Training on the Boston Dataset

But in practice, online training usually, but not always, proved to work better than batch training for simple, single-hidden-layer neural networks. However, a variation of batch training, called *mini-batch training*, is now the most common technique used for deep neural networks.

The demo program implements batch training using the same method signature as online training.

```
  batchTrain(trainX, trainY, lrnRate, maxEpochs)
  {
    let oSignals = U.vecMake(this.no, 0.0);
    let hSignals = U.vecMake(this.nh, 0.0);

    let n = trainX.length;  // 406
    let indices = U.arange(n);  // [0,1,..,405]
    let freq = Math.trunc(maxEpochs / 10);
. . .
```

In online training, you declare a method-scope set of matrices and vectors to hold the gradients for the weights and biases. But in batch training, you instantiate matrices and vectors for accumulated gradients inside the loop that iterates through each training item.

```
  for (let epoch = 0; epoch < maxEpochs; ++epoch) {
    // this.shuffle(indices);  // no shuffle needed for batch training.

    let ihWtsAccGrads = U.matMake(this.ni, this.nh, 0.0);  // accumulated
grads
    let hBiasAccGrads = U.vecMake(this.nh, 0.0);
    let hoWtsAccGrads = U.matMake(this.nh, this.no, 0.0);
    let oBiasAccGrads = U.vecMake(this.no, 0.0);
. . .
```

You could declare the storage for accumulated gradients outside the training items loop, but then you'd have to zero-out all values immediately inside the loop. Because batch training processes all training items before updating weights, it's not necessary to visit the training items in random order, because doing so leads to the identical accumulated gradients.

Batch training begins with the forward pass in the same way as online training.

```
  for (let ii = 0; ii < n; ++ii) {  // each item
    let idx = indices[ii];
    let X = trainX[idx];
    let Y = trainY[idx];

    this.eval(X);  // output stored in this.oNodes.
. . .
```

After the **eval()** method is called, although gradients are computed in the usual way, they are then accumulated rather than used immediately. For example, the hidden-to-output gradients are accumulated like in the following.

```
  // compute output node signals.
  for (let k = 0; k < this.no; ++k) {
    let derivative = 1;  // identity activation.
    oSignals[k] = derivative * (this.oNodes[k] - Y[k]);  // E=(t-o)^2
  }
```

```
  // compute and accumulate hidden-to-output weight gradients.
  for (let j = 0; j < this.nh; ++j) {
    for (let k = 0; k < this.no; ++k) {
      let grad = oSignals[k] * this.hNodes[j];
      hoWtsAccGrads[j][k] += grad;  // accumulate -- don't use yet.
    }
  }
```

After the hidden-to-output weight gradients, the output node bias gradients, the input-to-hidden weight gradients, and the hidden node bias gradients have been computed and accumulated, the loop iterating through all training items terminates, and then the accumulated gradients are used to update the weights and biases. For example, the input-to-hidden weights are updated like in the following.

```
. . .
  } // ii end-each item

  // update input-to-hidden weights.
  for (let i = 0; i < this.ni; ++i) {
    for (let j = 0; j < this.nh; ++j) {
      let delta = -1.0 * lrnRate * (ihWtsAccGrads[i][j] / n);  // average
grad
      this.ihWeights[i][j] += delta;
    }
  }
```

In the code, the variable **n** is an alias for the number of training items and is used to compute the average gradient over all training items. Some neural network library implementations use the raw accumulated gradients (not averaged), but using the average is better because it doesn't introduce a dependency on the size of the training data.

In addition to online and batch training, there is a third technique called *mini-batch training*. The mini-batch training technique is a variation of (full) batch training. A mini-batch is just a subset of the training data, typically something like 10 or 16 items. In pseudo-code, mini-batch training looks the following.

```
for-each epoch
  for-each mini-batch
    fetch next mini-batch
    use batch training technique on the mini-batch subset
  end each mini-batch
end each-epoch
```

Although mini-batch training is conceptually simple, the implementation details are a bit messy. The source code repository has an implementation of mini-batch training on the Boston Dataset problem that you can examine and run.

Questions

1. True or false?

   A. Neural nets for regression problems use logistic sigmoid output layer activation.

   B. Neural nets for regression most often use mean squared error rather than cross-entropy error.

   C. Momentum is a technique for reducing the likelihood of model overfitting.

   D. For simple neural networks, online training usually works better than batch training.

Chapter 6 Binary Classification

A binary classification problem is one where the goal is to predict a discrete value that can be one of just two possible values. For example, you might want to predict the sex of a person (male or female) based on predictor variables such as age, occupation, annual income, and so on. Somewhat surprisingly, the techniques used for neural binary classification are quite a bit different than those used for neural multiclass classification, where the variable to predict can be one of three or more possible values.

The screenshot in Figure 6-1 shows a demo of neural network binary classification. The goal of the program is to predict if a patient has heart disease (0 = no, 1 = yes) based on 13 predictor variables such as age, sex, blood pressure, and so on.

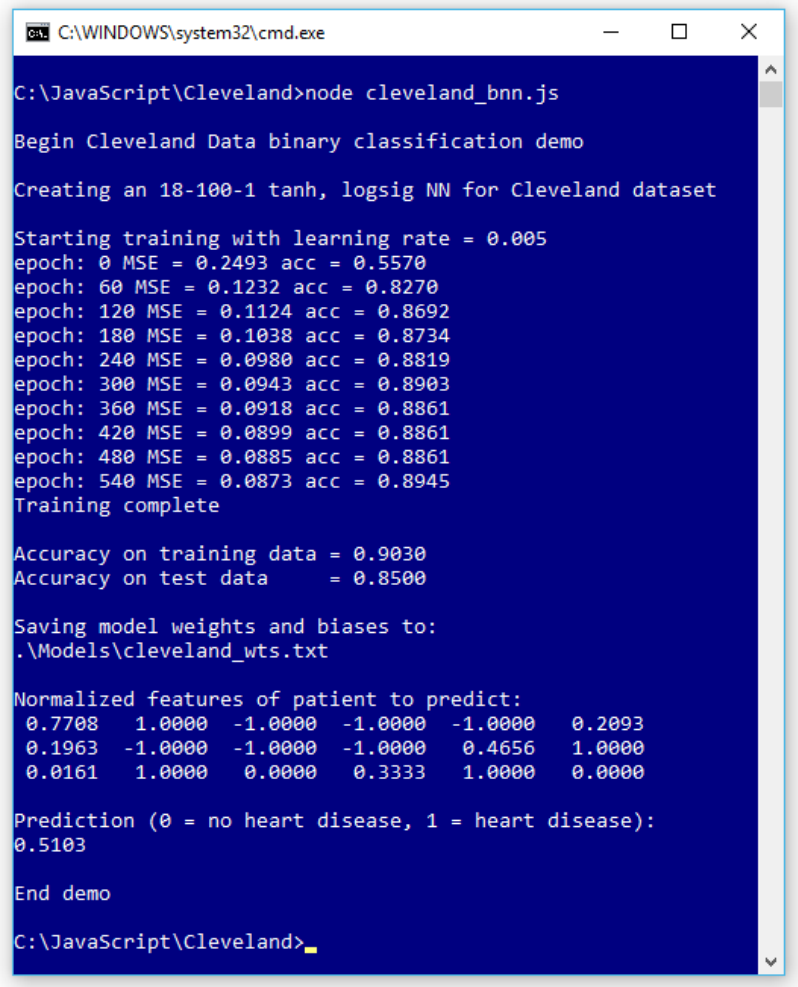


Figure 6-1: Neural Binary Classification

The demo creates an 18-100-1 neural network. The raw training data has 13 predictor variables, but after normalization and encoding, there are 18 values.

The most common technique for neural binary classification is to encode the two possible values to predict as 0 or 1 and use a single output node and coerce the value to be in the range [0.0, 1.0]. If the output value is less than 0.5, then the prediction is class 0; otherwise (if the output value is greater than 0.5), the prediction is class 1.

The number of hidden nodes, 100 in the demo, is a hyperparameter that must be determined by trial and error. The demo uses a learning rate of 0.005 and 600 training epochs (both values are hyperparameters).

The demo program uses the well-known Cleveland Heart Disease data set. The raw data has 303 items, but six items have a missing value, leaving 297 items. The raw data was randomly split into a 237-item set (approximately 80% of the data) for training, and a 60-item set for testing.

The training data has six numeric predictor variables, three binary predictor variables, and four categorical predictor variables. The dependent variable was encoded as 0 = no heart disease and 1 = heart disease. The numeric predictors were min-max normalized. The binary predictors were -1 +1 encoded. The categorical predictors were 1-of-(N-1) encoded. The test data was normalized and encoded using the normalization parameters and encoding scheme from the training data.

After training, the model scored 90.30% accuracy on the training data (214 out of 237 correct) and 85.00% accuracy on the test data (51 out of 60 correct).

The demo program concluded by making a prediction for a new, previously unseen patient. The output of the model is 0.5130, and because that value is greater than 0.5, the prediction is class 1, which is a prediction that the patient has heart disease.

Understanding the Cleveland data set

The demo program uses the Cleveland Heart Disease data set, a well-known classification benchmark data set for statistics and machine learning. The raw data looks like the following.

```
56.0,  1,  2,  120.0,  236.0,  0,  0,  178.0,  0,  0.8,  1,   3,   3,   0
62.0,  0,  4,  140.0,  268.0,  0,  2,  160.0,  0,  3.6,  3,   1,   6,   3
63.0,  1,  4,  130.0,  254.0,  0,  1,  147.0,  0,  1.4,  2,   2,   ?,   2
53.0,  1,  1,  140.0,  203.0,  1,  2,  155.0,  1,  3.1,  3,   0,   7,   1
[0]   [1] [2]  [3]     [4]    [5] [6]  [7]    [8]  [9]  [10] [11] [12] HD
```

The first 13 values on each line are the predictor values. The last value is 0 to 4, where 0 indicates no heart disease, and 1 to 4 indicate heart disease of some kind. Predictor [0] is patient age (numeric). Predictor [1] is a sex (0 = female, 1 = male). Predictor [2] is categorical chest pain type encoded as 1 to 4.

Predictor [3] is blood pressure (numeric). Predictor [4] is cholesterol (numeric). Predictor [5] is a related to blood sugar (0 = low, 1 = high). Predictor [6] is categorical electrocardiographic result encoded as (0, 1, 2). Predictor [7] is maximum heart rate (numeric). Predictor [8] is a Boolean for angina (0 = no, 1 = yes). Predictor [9] is ST ("S-wave, T-wave") graph depression.

Predictor [10] is a categorical ST metric encoded as (1, 2, 3). Predictor [11] is a categorical count of colored fluoroscopy vessels encoded as (0, 1, 2, 3). Predictor [12] is a categorical value related to thalassemia encoded as (3, 6, 7).

Because the Cleveland Heart Disease data set has 13 dimensions, it's not possible to easily visualize it in a two-dimensional graph. But you can get a rough idea of the data from the partial graph in Figure 6-2. The graph shows just patient age and blood pressure for the first 160 items of the full data set.

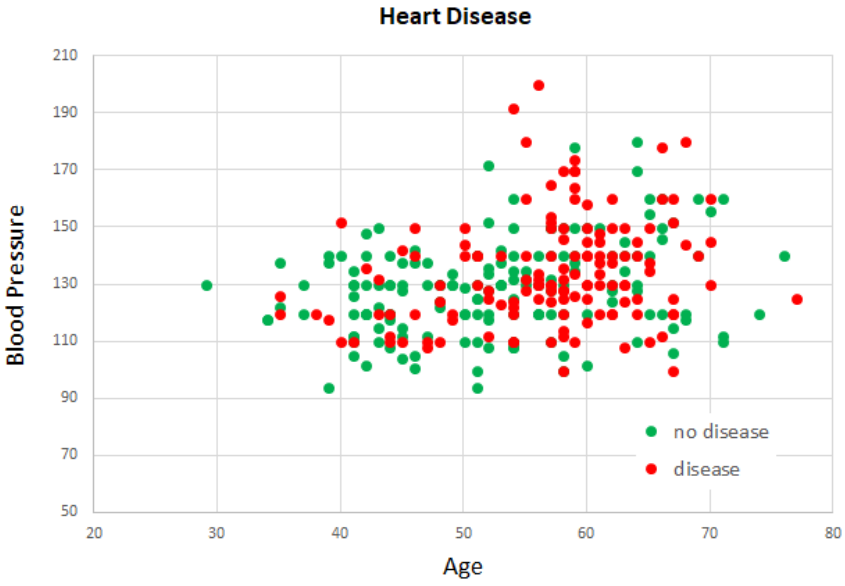


Figure 6-2: Partial Graph of the Cleveland Data Set

As you can see from the graph, it's not possible to get a good prediction model using a simple linear technique like logistic regression or a base support vector machine model.

The Cleveland data set demo program

The complete code for the demo program shown running in Figure 6-1 is presented in Code Listing 6-1. The demo assumes there is a top-level directory named JavaScript that contains subdirectories named Utilities and Cleveland. The Utilities directory contains the Utilities_lib.js library file. The Cleveland directory contains the demo program Cleveland_bnn.js file and subdirectories named Data and Models. The Data directory contains the files Cleveland_train.txt and Cleveland_test.txt. The Models directory is used to store trained model weights and biases values.

Code Listing 6-1: The Cleveland Data Demo Program

```
// cleveland_bnn.js
// ES6

let U = require("../Utilities/utilities_lib.js");
```

```
let FS = require("fs");

//
================================================================================
==

class NeuralNet
{
  constructor(numInput, numHidden, numOutput, seed) {
    this.rnd = new U.Erratic(seed);

    this.ni = numInput;
    this.nh = numHidden;
    this.no = numOutput;

    this.iNodes = U.vecMake(this.ni, 0.0);
    this.hNodes = U.vecMake(this.nh, 0.0);
    this.oNodes = U.vecMake(this.no, 0.0);

    this.ihWeights = U.matMake(this.ni, this.nh, 0.0);
    this.hoWeights = U.matMake(this.nh, this.no, 0.0);

    this.hBiases = U.vecMake(this.nh, 0.0);
    this.oBiases = U.vecMake(this.no, 0.0);

    this.initWeights();
  }

  initWeights()
  {
    let lo = -0.01;
    let hi = 0.01;
    for (let i = 0; i < this.ni; ++i) {
      for (let j = 0; j < this.nh; ++j) {
        this.ihWeights[i][j] = (hi - lo) * this.rnd.next() + lo;
      }
    }

    for (let j = 0; j < this.nh; ++j) {
      for (let k = 0; k < this.no; ++k) {
        this.hoWeights[j][k] = (hi - lo) * this.rnd.next() + lo;
      }
    }

  }

  eval(X)
  {
    let hSums = U.vecMake(this.nh, 0.0);
```

```
    let oSums = U.vecMake(this.no, 0.0);

    this.iNodes = X;

    for (let j = 0; j < this.nh; ++j) {
      for (let i = 0; i < this.ni; ++i) {
        hSums[j] += this.iNodes[i] * this.ihWeights[i][j];
      }
      hSums[j] += this.hBiases[j];
      this.hNodes[j] = U.hyperTan(hSums[j]);
    }

    for (let k = 0; k < this.no; ++k) {
      for (let j = 0; j < this.nh; ++j) {
        oSums[k] += this.hNodes[j] * this.hoWeights[j][k];
      }
      oSums[k] += this.oBiases[k];
    }

    for (let k = 0; k < this.no; ++k) {
      this.oNodes[k] = U.logSig(oSums[k]);  // logistic sigmoid activation
    }

    let result = [];
    for (let k = 0; k < this.no; ++k) {
      result[k] = this.oNodes[k];
    }
    return result;
  } // eval()

  setWeights(wts)
  {
    // order: ihWts, hBiases, hoWts, oBiases
    let p = 0;

    for (let i = 0; i < this.ni; ++i) {
      for (let j = 0; j < this.nh; ++j) {
        this.ihWeights[i][j] = wts[p++];
      }
    }

    for (let j = 0; j < this.nh; ++j) {
      this.hBiases[j] = wts[p++];
    }

    for (let j = 0; j < this.nh; ++j) {
      for (let k = 0; k < this.no; ++k) {
        this.hoWeights[j][k] = wts[p++];
      }
```

```
    }

    for (let k = 0; k < this.no; ++k) {
      this.oBiases[k] = wts[p++];
    }
  } // setWeights()

  getWeights()
  {
    // order: ihWts, hBiases, hoWts, oBiases
    let numWts = (this.ni * this.nh) + this.nh +
      (this.nh * this.no) + this.no;
    let result = U.vecMake(numWts, 0.0);
    let p = 0;
    for (let i = 0; i < this.ni; ++i) {
      for (let j = 0; j < this.nh; ++j) {
        result[p++] = this.ihWeights[i][j];
      }
    }

    for (let j = 0; j < this.nh; ++j) {
      result[p++] = this.hBiases[j];
    }

    for (let j = 0; j < this.nh; ++j) {
      for (let k = 0; k < this.no; ++k) {
        result[p++] = this.hoWeights[j][k];
      }
    }

    for (let k = 0; k < this.no; ++k) {
      result[p++] = this.oBiases[k];
    }
    return result;

  } // getWeights()

  shuffle(v)
  {
    // Fisher-Yates
    let n = v.length;
    for (let i = 0; i < n; ++i) {
      let r = this.rnd.nextInt(i, n);
      let tmp = v[r];
      v[r] = v[i];
      v[i] = tmp;
    }
  }
```

```
  train(trainX, trainY, lrnRate, maxEpochs)
  {
    let hoGrads = U.matMake(this.nh, this.no, 0.0);
    let obGrads = U.vecMake(this.no, 0.0);
    let ihGrads = U.matMake(this.ni, this.nh, 0.0);
    let hbGrads = U.vecMake(this.nh, 0.0);

    let oSignals = U.vecMake(this.no, 0.0);
    let hSignals = U.vecMake(this.nh, 0.0);

    let n = trainX.length;  // 237
    let indices = U.arange(n);  // [0,1,..,236]
    let freq = Math.trunc(maxEpochs / 10);

    for (let epoch = 0; epoch < maxEpochs; ++epoch) {
      this.shuffle(indices);  //
      for (let ii = 0; ii < n; ++ii) {  // each item
        let idx = indices[ii];
        let X = trainX[idx];
        let Y = trainY[idx];
        this.eval(X);  // output stored in this.oNodes.

        // compute output node signals.
        for (let k = 0; k < this.no; ++k) {
          let derivative = (1 - this.oNodes[k]) * this.oNodes[k];  //
logsig
          oSignals[k] = derivative * (this.oNodes[k] - Y[k]);  // E=(t-o)^2
        }

        // compute hidden-to-output weight gradients using output signals.
        for (let j = 0; j < this.nh; ++j) {
          for (let k = 0; k < this.no; ++k) {
            hoGrads[j][k] = oSignals[k] * this.hNodes[j];
          }
        }

        // compute output node bias gradients using output signals.
        for (let k = 0; k < this.no; ++k) {
          obGrads[k] = oSignals[k] * 1.0;  // 1.0 dummy input can be
dropped.
        }

        // compute hidden node signals.
        for (let j = 0; j < this.nh; ++j) {
          let sum = 0.0;
          for (let k = 0; k < this.no; ++k) {
            sum += oSignals[k] * this.hoWeights[j][k];
          }
```

```
          let derivative = (1 - this.hNodes[j]) * (1 + this.hNodes[j]);  //
tanh
          hSignals[j] = derivative * sum;
        }

        // compute input-to-hidden weight gradients using hidden signals.
        for (let i = 0; i < this.ni; ++i) {
          for (let j = 0; j < this.nh; ++j) {
            ihGrads[i][j] = hSignals[j] * this.iNodes[i];
          }
        }

        // compute hidden node bias gradients using hidden signals.
        for (let j = 0; j < this.nh; ++j) {
          hbGrads[j] = hSignals[j] * 1.0;  // 1.0 dummy input can be
dropped.
        }

        // update input-to-hidden weights.
        for (let i = 0; i < this.ni; ++i) {
          for (let j = 0; j < this.nh; ++j) {
            let delta = -1.0 * lrnRate * ihGrads[i][j];
            this.ihWeights[i][j] += delta;
          }
        }

        // update hidden node biases.
        for (let j = 0; j < this.nh; ++j) {
          let delta = -1.0 * lrnRate * hbGrads[j];
          this.hBiases[j] += delta;
        }

        // update hidden-to-output weights.
        for (let j = 0; j < this.nh; ++j) {
          for (let k = 0; k < this.no; ++k) {
            let delta = -1.0 * lrnRate * hoGrads[j][k];
            this.hoWeights[j][k] += delta;
          }
        }

        // update output node biases.
        for (let k = 0; k < this.no; ++k) {
          let delta = -1.0 * lrnRate * obGrads[k];
          this.oBiases[k] += delta;
        }
      } // ii

      if (epoch % freq == 0) {
        let mse = this.meanSqErr(trainX, trainY).toFixed(4);
```

```
      let acc = this.accuracy(trainX, trainY).toFixed(4);

      let s1 = "epoch: " + epoch.toString();
      let s2 = " MSE = " + mse.toString();
      let s3 = " acc = " + acc.toString();

      console.log(s1 + s2 + s3);
    }

  } // epoch

} // train()

meanCrossEntErr(dataX, dataY)  // doesn't work for binary classification.
{
  let sumCEE = 0.0;  // sum of the cross-entropy errors.
  for (let i = 0; i < dataX.length; ++i) {  // each data item
    let X = dataX[i];
    let Y = dataY[i];  // target output like (0, 1, 0).
    let oupt = this.eval(X);  // computed like (0.23, 0.66, 0.11).
    let idx = U.argmax(Y);  // find location of the 1 in target.
    sumCEE += Math.log(oupt[idx]);
  }
  return -1 * sumCEE / dataX.length;
}

meanBinCrossEntErr(dataX, dataY)  // for binary problems
{
  let sum = 0.0;
  for (let i = 0; i < dataX.length; ++i) {  // each data item
    let oupt = this.eval(dataX[i]);
    if (dataY[i] == 1) {  // target is 1
      sum += Math.log(oupt);
    }
    else {  // target is 0
      sum += Math.log(1.0 - oupt);
    }
  }
  return -1 * sum / dataX.length;
}

meanSqErr(dataX, dataY)
{
  let sumSE = 0.0;
  for (let i = 0; i < dataX.length; ++i) {  // each data item
    let X = dataX[i];
    let Y = dataY[i];  // target output 0 or 1
    let oupt = this.eval(X);  // computed like 0.2345
    for (let k = 0; k < this.no; ++k) {
```

```
      let err = Y[k] - oupt[k]  // target - computed
      sumSE += err * err;
    }
  }
  return sumSE / dataX.length;
}

accuracy(dataX, dataY)
{
  let nc = 0; let nw = 0;
  for (let i = 0; i < dataX.length; ++i) {  // each data item
    let X = dataX[i];
    let Y = dataY[i];  // target output 0 or 1
    let oupt = this.eval(X);  // computed like 0.2345

    if (Y == 0 && oupt < 0.5 || Y == 1 && oupt >= 0.5) {
      ++nc;
    }
    else {
      ++nw;
    }
  }
  return nc / (nc + nw);
}

saveWeights(fn)
{
  let wts = this.getWeights();
  let n = wts.length;
  let s = "";
  for (let i = 0; i < n - 1; ++i) {
    s += wts[i].toString() + ",";
  }
  s += wts[n - 1];

  FS.writeFileSync(fn, s);
}

loadWeights(fn)
{
  let n = (this.ni * this.nh) + this.nh + (this.nh * this.no) + this.no;
  let wts = U.vecMake(n, 0.0);
  let all = FS.readFileSync(fn, "utf8");
  let strVals = all.split(",");
  let nn = strVals.length;
  if (n != nn) {
    throw ("Size error in NeuralNet.loadWeights()");
  }
  for (let i = 0; i < n; ++i) {
```

```
      wts[i] = parseFloat(strVals[i]);
    }
    this.setWeights(wts);
  }

} // NeuralNet

//
================================================================================
==

function main()
{
  process.stdout.write("\033[0m");  // reset
  process.stdout.write("\x1b[1m" + "\x1b[37m");  // bright white
  console.log("\nBegin Cleveland Data binary classification demo  ");

  // 1. load data
  // raw data has 13 predictor values such as age, sex, BP, ECG.
  // normalized and encoded data has 18 values.
  // value to predict: 0 = no heart disease, 1 = heart disease.
  let trainX = U.loadTxt(".\\Data\\cleveland_train.txt", "\t", [0, 1, 2, 3,
4,
    5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]);
  let trainY = U.loadTxt(".\\Data\\cleveland_train.txt", "\t", [18]);

  let testX = U.loadTxt(".\\Data\\cleveland_test.txt", "\t", [0, 1, 2, 3,
4,
    5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]);
  let testY = U.loadTxt(".\\Data\\cleveland_test.txt", "\t", [18]);

  // 2. create network
  console.log("\nCreating an 18-100-1 tanh, logsig NN for Cleveland
dataset");
  let seed = 0;
  let nn = new NeuralNet(18, 100, 1, seed);

  // 3. train network
  let lrnRate = 0.005;
  let maxEpochs = 600;
  console.log("\nStarting training with learning rate = 0.005 ");
  nn.train(trainX, trainY, lrnRate, maxEpochs);
  console.log("Training complete");

  // 4. evaluate model
  let trainAcc = nn.accuracy(trainX, trainY);
  let testAcc = nn.accuracy(testX, testY);
  console.log("\nAccuracy on training data = " +
    trainAcc.toFixed(4).toString());
```

```
  console.log("Accuracy on test data     = " +
    testAcc.toFixed(4).toString());

  // 5. save model
  let fn = ".\\Models\\cleveland_wts.txt";
  console.log("\nSaving model weights and biases to: ");
  console.log(fn);
  nn.saveWeights(fn);

  // 6. use trained model
  let unknownNorm = [0.7708, 1, -1, -1, -1, 0.2093, 0.1963, -1, -1, -1,
    0.4656, 1, 0.0161, 1, 0, 0.3333, 1, 0];
  let prediction = nn.eval(unknownNorm);

  console.log("\nNormalized features of patient to predict: ");
  U.vecShow(unknownNorm, 4, 6);
  console.log("\nPrediction (0 = no heart disease, 1 = heart disease): ");
  console.log(prediction[0].toFixed(4).toString());

  process.stdout.write("\033[0m");  // reset
  console.log("\nEnd demo");
} // main()

main();
```

All of the control logic is contained in a top-level **`main()`** function. Program execution begins by loading the training and test data into memory.

```
  // 1. load data
  // raw data has 13 predictor values such as age, sex, BP, ECG.
  // normalized and encoded data has 18 values.
  // value to predict: 0 = no heart disease, 1 = heart disease.

  let trainX = U.loadTxt(".\\Data\\cleveland_train.txt", "\t", [0,1,2,3,4,
    5,6,7,8,9,10,11,12,13,14,15,16,17]);
  let trainY = U.loadTxt(".\\Data\\cleveland_train.txt", "\t", [18]);

  let testX = U.loadTxt(".\\Data\\cleveland_test.txt", "\t", [0,1,2,3,4,
    5,6,7,8,9,10,11,12,13,14,15,16,17]);
  let testY = U.loadTxt(".\\Data\\cleveland_test.txt", "\t", [18]);
. . .
```

The **`loadTxt()`** function is contained in file Utilities_lib.js, which is located in the Utilities directory. Notice that the data files are tab-delimited. Next, a neural network is created.

```
  // 2. create network
  console.log("\nCreating an 18-100-1 tanh, logsig NN for Cleveland
dataset");
  let seed = 0;
  let nn = new NeuralNet(18, 100, 1, seed);
. . .
```

A neural network classifier has a single output node and uses logistic sigmoid activation on that node to force the value to be between 0.0 and 1.0. Technically, the output value represents the probability of class 1. Next, the network is trained.

```
  // 3. train network
  let lrnRate = 0.005;
  let maxEpochs = 600;
  console.log("\nStarting training with learning rate = 0.005 ");
  nn.train(trainX, trainY, lrnRate, maxEpochs);
  console.log("Training complete");
. . .
```

The demo uses standard online training. Options include using dropout, using regularization, using momentum, and using the batch or mini-batch techniques.

Next, the trained mode is evaluated.

```
  // 4. evaluate model
  let trainAcc = nn.accuracy(trainX, trainY);
  let testAcc = nn.accuracy(testX, testY);

  console.log("\nAccuracy on training data = " +
    trainAcc.toFixed(4).toString());

  console.log("Accuracy on test data     = " +
    testAcc.toFixed(4).toString());
. . .
```

With binary classification, you have to be careful interpreting accuracy when your data is highly skewed towards one of the two classes. For example, if you have a set of training data with 1,000 items and 950 of those items are class 0, then you could get 95% accuracy by predicting class 0 regardless of input values.

Next, the trained model's weights and biases are saved to a text file.

```
  // 5. save model
  let fn = ".\\Models\\cleveland_wts.txt";
  console.log("\nSaving model weights and biases to: ");
  console.log(fn);
  nn.saveWeights(fn);
. . .
```

The demo program concludes by making a prediction for a new, previously unseen patient.

```
  // 6. use trained model
  let unknownNorm = [0.7708, 1, -1, -1, -1, 0.2093, 0.1963, -1, -1, -1,
    0.4656, 1, 0.0161, 1, 0, 0.3333, 1, 0];
  let prediction = nn.eval(unknownNorm);

  console.log("\nNormalized features of patient to predict: ");
  U.vecShow(unknownNorm, 4, 6);  // 4 decimals, 6 values per line

  console.log("\nPrediction (0 = no heart disease, 1 = heart disease): ");
  console.log(prediction[0].toFixed(4).toString());
. . .
```

The demo program normalizes and encodes the new person item offline. An alternative that is useful when making many predictions is to write a problem-specific function to normalize and encode. For example, using such a function might look like the following.

```
let unknownRaw = [36, 0, etc.];  // 36 years old, female, etc.
let unkNorm = normAndEncode(unknownRaw);
let predicted = nn.eval(unkNorm);
```

Note that a program-defined function to normalize and encode raw data needs to know the normalization parameters, such as **min** and **max** if min-max normalization is used, and the encoding scheme for non-numeric data. Writing a function to normalize and encode raw data is not difficult conceptually, but it is often very time-consuming.

The demo program leaves it up to the user to interpret the output value. You could add code to display the output value in user-friendly form.

```
  if (prediction[0] < 0.5) {
    console.log("patient is predicted to have low chance of heart disease");
  }
  else {
    console.log("patient is predicted to have high chance of heart disease");
  }
```

Notice that the output value is stored in a 1×1 matrix rather than in a scalar variable.

Logistic sigmoid activation

A neural network classifier has multiple output nodes and uses softmax activation on the output nodes so that their values sum to 1.0 and can be loosely interpreted as probabilities. In back-propagation training, the derivative of the function used for output layer activation is used to compute the gradients, which in turn are used to update the network's weights and biases.

A neural network for binary classification uses just a single output node and uses the logistic sigmoid function for output layer activation. The logistic sigmoid function is defined as y = 1.0 / (1.0 + exp(-x)). Although it's not at all obvious, the calculus derivative is y' = y * (1 - y). Somewhat surprisingly, this is the same as the derivative of the softmax activation function.

In the method **train()** for binary classification, the output node signals are computed using these statements.

```
  // compute output node signals.
  for (let k = 0; k < this.no; ++k) {
    let derivative = (1 - this.oNodes[k]) * this.oNodes[k];  // log-sig
    oSignals[k] = derivative * (this.oNodes[k] - Y[k]);  // E=(t-o)^2
  }
```

The **eval()** method for binary classification is the following.

```
. . .
  // compute output node before activation.
  for (let k = 0; k < this.no; ++k) {
    for (let j = 0; j < this.nh; ++j) {
      oSums[k] += this.hNodes[j] * this.hoWeights[j][k];
    }
    oSums[k] += this.oBiases[k];
  }

  for (let k = 0; k < this.no; ++k) {
    this.oNodes[k] = U.logSig(oSums[k]);  // logistic sigmoid activation
  }
```

To summarize, to perform binary classification, you encode the variable to predict as 0 or 1. Your neural network has a single output node, uses logistic sigmoid activation for the output layer, and y * (1 - y) as the derivative term for the output node signals.

Binary cross entropy

When creating a neural network for binary classification, you can use mean squared error as in the demo program, or you can use binary cross-entropy error. Regular cross-entropy error compares a set of predicted probabilities with a set of correct probabilities. With only one output node, there are no sets of probabilities to compare, so a special form of cross entropy is used.

Suppose t is a target probability value and y is a predicted output probability (a value between 0.0 and 1.0). For just two values, the cross-entropy error equation is CEE = -[log(y) * t + log(1-y) * (1-t)]. But in neural binary classification, the target is always a 1 or a 0.

If t = 1, the log(1-y) * (1-t) term drops out, leaving just -log(y). If t = 0, the log(y) * t term drops out, leaving just log(1-y). To compute binary cross-entropy error for a neural network, if the target is 1, use -log(y), and if the target is 0, use -log(1-y).

Suppose you have just three training items with the following target and computed output values.

```
target   computed output
------------------------
  1         0.80
  0         0.30
  1         0.90
```

The three binary cross entropy terms are -log(0.80) = 0.223, -log(0.70) = 0.357, and -log(0.9) = 0.105, and therefore, the mean binary cross-entropy error is (0.223 + 0.357 + 0.105) / 3 = 0.228.

A possible implementation is the following.

```
  meanBinCrossEntErr(dataX, dataY)  // for binary classification problems.
  {
    let sum = 0.0;

    for (let i = 0; i < dataX.length; ++i) {  // each data item
      let oupt = this.eval(dataX[i]);

      if (dataY[i] == 1) {  // target is 1
        sum += Math.log(oupt);
      }
      else {  // target is 0
        sum += Math.log(1.0 - oupt);
      }
    }

    return -1 * sum / dataX.length;
  }
```

Recall that back-propagation training does not explicitly compute error; instead, back-propagation uses the calculus derivative of the error/loss function. For mean squared error, the relevant code is the following.

```
  // compute output node signals.
  for (let k = 0; k < this.no; ++k) {
    let derivative = (1 - this.oNodes[k]) * this.oNodes[k];  // logsig
    oSignals[k] = derivative * (this.oNodes[k] - Y[k]);  // E=(t-o)^2
  }

  // now compute hidden-to-output weight gradients using output signals.
```

But if you assume binary cross-entropy error instead of mean squared error, many of the terms cancel each other out and, in one of the most surprising results in all of machine learning (well, surprising to me, anyway), the computation of the output node signals reduces to the following.

```
  // compute output node signals.
  for (let k = 0; k < this.no; ++k) {
    oSignals[k] = this.oNodes[k] - Y[k];  // E = binary cross entropy
  }
```

In words, everything cancels out, leaving just the computed output value and the target value. Quite remarkable.

When training, it's good practice to print the current error and accuracy every few epochs so that you can see when training is not working. For binary classification problems, regardless of whether you assume mean squared error or binary cross-entropy error, you can compute and display either error term because both will indicate when training is not working. Mean squared error is slightly easier to interpret because the value will always be between 0.0 and 1.0.

The two-node technique for binary classification

By far the most common approach to neural network binary classification is to use a single output node scaled with logistic sigmoid activation and target values of 0 or 1. An alternative is to encode target values as (1, 0) or (0, 1) and use two output nodes scaled with softmax activation. The problem becomes a standard multiclass classification problem. Research results comparing the one-node and two-node techniques are inconclusive, in my opinion.

Questions

1. Suppose you have just three training items with the following target and computed output values.

```
target   computed output
-----------------------
  1         0.70
  0         0.20
  1         0.60
```

A. Compute the mean squared error.

B. Compute the mean binary cross-entropy error.

Appendix—Data Files

Data Listing 3-1: Iris_train.txt

```
5.1, 3.5, 1.4, 0.2, 1, 0, 0
4.9, 3.0, 1.4, 0.2, 1, 0, 0
4.7, 3.2, 1.3, 0.2, 1, 0, 0
4.6, 3.1, 1.5, 0.2, 1, 0, 0
5.0, 3.6, 1.4, 0.2, 1, 0, 0
5.4, 3.9, 1.7, 0.4, 1, 0, 0
4.6, 3.4, 1.4, 0.3, 1, 0, 0
5.0, 3.4, 1.5, 0.2, 1, 0, 0
4.4, 2.9, 1.4, 0.2, 1, 0, 0
4.9, 3.1, 1.5, 0.1, 1, 0, 0
5.4, 3.7, 1.5, 0.2, 1, 0, 0
4.8, 3.4, 1.6, 0.2, 1, 0, 0
4.8, 3.0, 1.4, 0.1, 1, 0, 0
4.3, 3.0, 1.1, 0.1, 1, 0, 0
5.8, 4.0, 1.2, 0.2, 1, 0, 0
5.7, 4.4, 1.5, 0.4, 1, 0, 0
5.4, 3.9, 1.3, 0.4, 1, 0, 0
5.1, 3.5, 1.4, 0.3, 1, 0, 0
5.7, 3.8, 1.7, 0.3, 1, 0, 0
5.1, 3.8, 1.5, 0.3, 1, 0, 0
5.4, 3.4, 1.7, 0.2, 1, 0, 0
5.1, 3.7, 1.5, 0.4, 1, 0, 0
4.6, 3.6, 1.0, 0.2, 1, 0, 0
5.1, 3.3, 1.7, 0.5, 1, 0, 0
4.8, 3.4, 1.9, 0.2, 1, 0, 0
5.0, 3.0, 1.6, 0.2, 1, 0, 0
5.0, 3.4, 1.6, 0.4, 1, 0, 0
5.2, 3.5, 1.5, 0.2, 1, 0, 0
5.2, 3.4, 1.4, 0.2, 1, 0, 0
4.7, 3.2, 1.6, 0.2, 1, 0, 0
4.8, 3.1, 1.6, 0.2, 1, 0, 0
5.4, 3.4, 1.5, 0.4, 1, 0, 0
5.2, 4.1, 1.5, 0.1, 1, 0, 0
5.5, 4.2, 1.4, 0.2, 1, 0, 0
4.9, 3.1, 1.5, 0.2, 1, 0, 0
5.0, 3.2, 1.2, 0.2, 1, 0, 0
5.5, 3.5, 1.3, 0.2, 1, 0, 0
4.9, 3.6, 1.4, 0.1, 1, 0, 0
4.4, 3.0, 1.3, 0.2, 1, 0, 0
5.1, 3.4, 1.5, 0.2, 1, 0, 0
7.0, 3.2, 4.7, 1.4, 0, 1, 0
6.4, 3.2, 4.5, 1.5, 0, 1, 0
6.9, 3.1, 4.9, 1.5, 0, 1, 0
5.5, 2.3, 4.0, 1.3, 0, 1, 0
6.5, 2.8, 4.6, 1.5, 0, 1, 0
5.7, 2.8, 4.5, 1.3, 0, 1, 0
6.3, 3.3, 4.7, 1.6, 0, 1, 0
4.9, 2.4, 3.3, 1.0, 0, 1, 0
6.6, 2.9, 4.6, 1.3, 0, 1, 0
5.2, 2.7, 3.9, 1.4, 0, 1, 0
5.0, 2.0, 3.5, 1.0, 0, 1, 0
5.9, 3.0, 4.2, 1.5, 0, 1, 0
6.0, 2.2, 4.0, 1.0, 0, 1, 0
6.1, 2.9, 4.7, 1.4, 0, 1, 0
```

```
5.6, 2.9, 3.6, 1.3, 0, 1, 0
6.7, 3.1, 4.4, 1.4, 0, 1, 0
5.6, 3.0, 4.5, 1.5, 0, 1, 0
5.8, 2.7, 4.1, 1.0, 0, 1, 0
6.2, 2.2, 4.5, 1.5, 0, 1, 0
5.6, 2.5, 3.9, 1.1, 0, 1, 0
5.9, 3.2, 4.8, 1.8, 0, 1, 0
6.1, 2.8, 4.0, 1.3, 0, 1, 0
6.3, 2.5, 4.9, 1.5, 0, 1, 0
6.1, 2.8, 4.7, 1.2, 0, 1, 0
6.4, 2.9, 4.3, 1.3, 0, 1, 0
6.6, 3.0, 4.4, 1.4, 0, 1, 0
6.8, 2.8, 4.8, 1.4, 0, 1, 0
6.7, 3.0, 5.0, 1.7, 0, 1, 0
6.0, 2.9, 4.5, 1.5, 0, 1, 0
5.7, 2.6, 3.5, 1.0, 0, 1, 0
5.5, 2.4, 3.8, 1.1, 0, 1, 0
5.5, 2.4, 3.7, 1.0, 0, 1, 0
5.8, 2.7, 3.9, 1.2, 0, 1, 0
6.0, 2.7, 5.1, 1.6, 0, 1, 0
5.4, 3.0, 4.5, 1.5, 0, 1, 0
6.0, 3.4, 4.5, 1.6, 0, 1, 0
6.7, 3.1, 4.7, 1.5, 0, 1, 0
6.3, 2.3, 4.4, 1.3, 0, 1, 0
5.6, 3.0, 4.1, 1.3, 0, 1, 0
5.5, 2.5, 4.0, 1.3, 0, 1, 0
6.3, 3.3, 6.0, 2.5, 0, 0, 1
5.8, 2.7, 5.1, 1.9, 0, 0, 1
7.1, 3.0, 5.9, 2.1, 0, 0, 1
6.3, 2.9, 5.6, 1.8, 0, 0, 1
6.5, 3.0, 5.8, 2.2, 0, 0, 1
7.6, 3.0, 6.6, 2.1, 0, 0, 1
4.9, 2.5, 4.5, 1.7, 0, 0, 1
7.3, 2.9, 6.3, 1.8, 0, 0, 1
6.7, 2.5, 5.8, 1.8, 0, 0, 1
7.2, 3.6, 6.1, 2.5, 0, 0, 1
6.5, 3.2, 5.1, 2.0, 0, 0, 1
6.4, 2.7, 5.3, 1.9, 0, 0, 1
6.8, 3.0, 5.5, 2.1, 0, 0, 1
5.7, 2.5, 5.0, 2.0, 0, 0, 1
5.8, 2.8, 5.1, 2.4, 0, 0, 1
6.4, 3.2, 5.3, 2.3, 0, 0, 1
6.5, 3.0, 5.5, 1.8, 0, 0, 1
7.7, 3.8, 6.7, 2.2, 0, 0, 1
7.7, 2.6, 6.9, 2.3, 0, 0, 1
6.0, 2.2, 5.0, 1.5, 0, 0, 1
6.9, 3.2, 5.7, 2.3, 0, 0, 1
5.6, 2.8, 4.9, 2.0, 0, 0, 1
7.7, 2.8, 6.7, 2.0, 0, 0, 1
6.3, 2.7, 4.9, 1.8, 0, 0, 1
6.7, 3.3, 5.7, 2.1, 0, 0, 1
7.2, 3.2, 6.0, 1.8, 0, 0, 1
6.2, 2.8, 4.8, 1.8, 0, 0, 1
6.1, 3.0, 4.9, 1.8, 0, 0, 1
6.4, 2.8, 5.6, 2.1, 0, 0, 1
7.2, 3.0, 5.8, 1.6, 0, 0, 1
7.4, 2.8, 6.1, 1.9, 0, 0, 1
7.9, 3.8, 6.4, 2.0, 0, 0, 1
6.4, 2.8, 5.6, 2.2, 0, 0, 1
6.3, 2.8, 5.1, 1.5, 0, 0, 1
6.1, 2.6, 5.6, 1.4, 0, 0, 1
7.7, 3.0, 6.1, 2.3, 0, 0, 1
```

```
6.3, 3.4, 5.6, 2.4, 0, 0, 1
6.4, 3.1, 5.5, 1.8, 0, 0, 1
6.0, 3.0, 4.8, 1.8, 0, 0, 1
6.9, 3.1, 5.4, 2.1, 0, 0, 1
```

Data Listing 3-2: Iris_test.txt

```
5.0, 3.5, 1.3, 0.3, 1, 0, 0
4.5, 2.3, 1.3, 0.3, 1, 0, 0
4.4, 3.2, 1.3, 0.2, 1, 0, 0
5.0, 3.5, 1.6, 0.6, 1, 0, 0
5.1, 3.8, 1.9, 0.4, 1, 0, 0
4.8, 3.0, 1.4, 0.3, 1, 0, 0
5.1, 3.8, 1.6, 0.2, 1, 0, 0
4.6, 3.2, 1.4, 0.2, 1, 0, 0
5.3, 3.7, 1.5, 0.2, 1, 0, 0
5.0, 3.3, 1.4, 0.2, 1, 0, 0
5.5, 2.6, 4.4, 1.2, 0, 1, 0
6.1, 3.0, 4.6, 1.4, 0, 1, 0
5.8, 2.6, 4.0, 1.2, 0, 1, 0
5.0, 2.3, 3.3, 1.0, 0, 1, 0
5.6, 2.7, 4.2, 1.3, 0, 1, 0
5.7, 3.0, 4.2, 1.2, 0, 1, 0
5.7, 2.9, 4.2, 1.3, 0, 1, 0
6.2, 2.9, 4.3, 1.3, 0, 1, 0
5.1, 2.5, 3.0, 1.1, 0, 1, 0
5.7, 2.8, 4.1, 1.3, 0, 1, 0
6.7, 3.1, 5.6, 2.4, 0, 0, 1
6.9, 3.1, 5.1, 2.3, 0, 0, 1
5.8, 2.7, 5.1, 1.9, 0, 0, 1
6.8, 3.2, 5.9, 2.3, 0, 0, 1
6.7, 3.3, 5.7, 2.5, 0, 0, 1
6.7, 3.0, 5.2, 2.3, 0, 0, 1
6.3, 2.5, 5.0, 1.9, 0, 0, 1
6.5, 3.0, 5.2, 2.0, 0, 0, 1
6.2, 3.4, 5.4, 2.3, 0, 0, 1
5.9, 3.0, 5.1, 1.8, 0, 0, 1
```

Data Listing 4-1: People_train.txt

```
-1      0.28    1       0       0.5592 0       1       0
1       0.30    -1      -1      0.2587 0       1       0
-1      0.34    -1      -1      0.1488 0       0       1
-1      0.16    0       1       0.4812 1       0       0
1       0.60    0       1       0.7601 0       1       0
1       0.42    0       1       0.1676 0       1       0
-1      0.34    -1      -1      0.2298 0       1       0
1       0.22    -1      -1      0.1315 0       0       1
1       0.24    -1      -1      0.4740 1       0       0
-1      0.46    0       1       0.0000 0       1       0
-1      0.28    1       0       0.6734 0       1       0
1       0.36    1       0       0.3945 0       1       0
-1      0.38    -1      -1      0.0159 0       1       0
1       0.86    0       1       0.0116 1       0       0
```

```
1     0.96  1     0     0.1561 0     1     0
1     0.30  1     0     0.6286 0     0     1
1     0.58  0     1     0.8801 0     1     0
-1    0.80  0     1     0.3873 0     0     1
1     1.00  1     0     0.4046 1     0     0
-1    0.40  -1    -1    0.6488 0     1     0
1     0.02  -1    -1    0.8035 0     0     1
-1    0.10  1     0     0.9249 1     0     0
-1    0.26  1     0     0.4653 0     1     0
-1    0.22  -1    -1    0.0737 0     0     1
-1    0.44  1     0     0.2182 0     0     1
-1    0.22  0     1     0.8338 1     0     0
1     0.66  0     1     0.4436 0     1     0
-1    0.68  1     0     0.4220 0     1     0
1     0.50  0     1     0.4957 0     1     0
-1    0.50  -1    -1    0.8714 0     1     0
1     0.60  0     1     0.9711 0     0     1
-1    0.86  -1    -1    0.9740 0     1     0
1     0.34  -1    -1    0.9494 0     1     0
1     0.36  0     1     0.9624 0     1     0
1     0.54  0     1     0.9118 0     1     0
-1    0.72  1     0     0.1503 0     0     1
-1    0.56  0     1     0.9118 0     1     0
-1    0.08  -1    -1    0.3757 1     0     0
-1    0.38  1     0     0.8497 0     1     0
-1    0.76  0     1     0.6012 0     0     1
1     0.84  0     1     0.9061 0     1     0
-1    0.10  0     1     0.9697 1     0     0
-1    0.20  1     0     0.5578 1     0     0
1     0.68  0     1     0.9769 1     0     0
1     0.52  -1    -1    0.8902 0     1     0
1     0.48  1     0     0.8671 0     1     0
1     0.92  0     1     0.4942 1     0     0
1     0.66  0     1     0.1329 0     1     0
1     0.82  0     1     0.7760 1     0     0
1     0.44  0     1     0.5665 0     1     0
1     0.70  -1    -1    0.9610 0     0     1
1     0.22  1     0     0.1662 0     0     1
-1    0.48  0     1     0.2746 0     0     1
-1    0.04  1     0     0.3728 1     0     0
-1    0.10  -1    -1    0.6575 1     0     0
-1    0.50  1     0     0.7312 0     1     0
1     0.30  0     1     0.1517 0     1     0
-1    0.82  -1    -1    0.8223 0     0     1
1     0.66  1     0     0.6980 0     1     0
1     0.06  -1    -1    0.7803 0     0     1
-1    0.90  -1    -1    0.6893 0     0     1
-1    0.10  0     1     0.5665 1     0     0
-1    0.26  -1    -1    0.4249 0     0     1
1     0.82  1     0     0.9379 1     0     0
-1    0.76  1     0     0.8237 0     0     1
-1    0.12  0     1     0.1431 0     1     0
1     0.62  0     1     0.7962 0     1     0
1     0.62  1     0     0.5650 0     1     0
1     0.28  1     0     0.7832 0     1     0
-1    0.44  0     1     0.4509 0     1     0
1     0.62  1     0     0.0101 0     1     0
-1    0.16  1     0     0.1040 1     0     0
-1    0.28  0     1     0.9624 0     1     0
-1    0.00  1     0     0.6127 1     0     0
-1    0.34  0     1     0.1980 0     1     0
-1    0.78  1     0     0.2514 0     0     1
```

```
-1      0.34   1      0      0.5390 0      1      0
1       0.32   1      0      0.4408 0      1      0
-1      0.16   0      1      0.5231 1      0      0
-1      0.46   -1     -1     0.9624 1      0      0
-1      0.38   1      0      0.5043 0      1      0
1       0.16   0      1      0.1358 0      1      0
-1      0.28   0      1      0.8902 0      0      1
-1      0.88   1      0      0.3367 0      1      0
1       0.58   -1     -1     0.1864 0      1      0
1       0.36   -1     -1     0.3844 0      1      0
1       0.92   -1     -1     0.2413 1      0      0
1       0.34   0      1      0.3613 0      1      0
1       0.50   -1     -1     0.8078 0      1      0
-1      0.62   0      1      0.9581 0      1      0
-1      0.32   0      1      0.3078 0      1      0
1       0.88   0      1      0.4379 1      0      0
1       0.60   1      0      0.8584 0      1      0
1       0.64   -1     -1     1.0000 0      0      1
1       0.50   -1     -1     0.0694 0      1      0
1       0.36   -1     -1     0.0130 0      1      0
1       0.62   -1     -1     0.5087 0      1      0
1       1.00   -1     -1     0.9321 1      0      0
1       0.32   1      0      0.4075 0      1      0
-1      0.74   0      1      0.2919 1      0      0
1       0.80   0      1      0.4017 1      0      0
1       0.32   -1     -1     0.7442 0      1      0
1       0.34   1      0      0.6431 0      0      1
1       0.08   0      1      0.9162 0      1      0
1       0.28   1      0      0.1315 0      1      0
-1      0.74   0      1      0.9971 0      0      1
-1      0.02   -1     -1     0.5217 1      0      0
1       0.78   1      0      0.9798 1      0      0
1       0.26   -1     -1     0.9075 0      1      0
-1      0.40   0      1      0.0477 0      1      0
-1      0.12   0      1      0.1879 0      1      0
1       0.42   0      1      0.8150 0      0      1
-1      0.02   1      0      0.5564 1      0      0
1       0.72   1      0      0.0029 1      0      0
1       0.34   -1     -1     0.9812 0      0      1
1       0.24   -1     -1     0.6185 0      0      1
-1      0.86   1      0      0.7905 0      1      0
1       0.56   -1     -1     0.6806 1      0      0
1       0.08   1      0      0.7702 1      0      0
1       0.36   -1     -1     0.1777 0      1      0
-1      1.00   0      1      0.3165 0      0      1
1       0.50   1      0      0.2847 0      1      0
1       0.66   1      0      0.6633 0      1      0
-1      0.44   0      1      0.4003 0      1      0
1       0.42   0      1      0.7066 0      1      0
-1      0.54   -1     -1     0.9075 0      1      0
-1      0.26   0      1      0.8324 0      1      0
1       0.08   0      1      0.6243 0      0      1
-1      0.28   -1     -1     0.2254 0      1      0
1       0.16   -1     -1     0.1922 0      0      1
1       0.42   -1     -1     0.3974 0      1      0
-1      0.30   1      0      0.1893 0      1      0
-1      0.04   -1     -1     0.6142 1      0      0
1       0.24   1      0      0.6199 0      0      1
-1      0.72   0      1      0.7948 0      0      1
-1      0.40   0      1      0.1850 0      1      0
-1      0.02   0      1      0.7081 1      0      0
-1      0.30   -1     -1     0.9335 0      1      0
```

```
1       0.44    -1      -1      0.0997 0        1       0
-1      0.42    1       0       0.9306 0        1       0
-1      0.76    -1      -1      0.3988 0        0       1
-1      0.72    0       1       0.7832 0        0       1
1       0.84    -1      -1      0.9292 0        0       1
1       0.50    -1      -1      0.7775 0        1       0
1       0.40    -1      -1      0.8786 0        1       0
1       0.54    -1      -1      0.6936 0        1       0
1       0.56    -1      -1      0.9263 0        1       0
-1      0.88    1       0       0.6301 1        0       0
1       0.34    -1      -1      0.1647 0        1       0
1       0.40    0       1       0.6069 0        1       0
1       0.66    0       1       0.2919 0        1       0
1       0.72    1       0       0.0434 1        0       0
1       0.94    -1      -1      0.2789 1        0       0
1       0.10    -1      -1      0.7630 0        0       1
-1      0.90    1       0       0.7543 0        0       1
-1      0.70    0       1       0.8353 0        1       0
1       0.38    -1      -1      0.8844 0        1       0
-1      0.56    0       1       0.9552 1        0       0
-1      0.16    -1      -1      0.9581 1        0       0
1       0.32    1       0       0.3295 0        1       0
```

Data Listing 4-2: People_test.txt

```
1       1.00    -1      -1      0.4581 1        0       0
1       0.68    1       0       0.9682 1        0       0
-1      0.70    1       0       0.1705 0        1       0
1       0.52    0       1       0.9408 0        1       0
-1      0.36    -1      -1      0.4986 0        1       0
1       0.24    -1      -1      0.9147 0        0       1
1       0.44    1       0       0.3858 0        1       0
1       1.02    0       1       0.8887 0        1       0
-1      0.94    0       1       0.3309 0        0       1
-1      0.56    0       1       0.9971 0        1       0
-1      0.34    1       0       0.6647 0        1       0
1       0.96    -1      -1      0.4552 1        0       0
-1      0.16    0       1       0.6199 0        1       0
-1      0.00    -1      -1      0.9335 1        0       0
1       0.10    1       0       0.2153 0        0       1
-1      0.42    0       1       0.4191 0        1       0
-1      0.78    1       0       0.5014 0        0       1
-1      0.64    0       1       0.6705 1        0       0
1       0.36    -1      -1      0.6460 0        1       0
1       1.02    0       1       0.3425 1        0       0
1       0.24    0       1       0.9032 0        1       0
1       0.44    0       1       0.4061 0        1       0
1       0.50    -1      -1      0.8699 0        1       0
-1      0.50    1       0       0.2197 0        1       0
1       0.72    0       1       0.1272 1        0       0
-1      0.08    0       1       0.5303 1        0       0
1       0.10    -1      -1      0.5043 0        1       0
-1      0.54    0       1       0.9566 0        1       0
-1      0.48    0       1       0.6040 0        1       0
-1      1.02    0       1       0.0867 0        0       1
-1      0.26    1       0       0.8916 0        1       0
-1      1.02    1       0       0.3078 0        0       1
-1      0.38    0       1       0.8266 0        1       0
```

```
1       0.50    0       1       0.1965 0        1       0
1       0.26    1       0       0.7124 0        1       0
1       0.60    1       0       0.7890 0        0       1
1       0.70    1       0       0.1893 0        0       1
1       0.58    0       1       0.1431 0        1       0
1       0.94    -1      -1      0.9668 1        0       0
1       0.92    1       0       0.5390 1        0       0
```

Data Listing 4-3: People_validate.txt

```
-1      0.14    0       1       0.6243 0        1       0
1       0.82    1       0       0.1590 1        0       0
1       0.96    0       1       0.2948 1        0       0
-1      0.04    0       1       0.5968 1        0       0
1       0.86    -1      -1      0.3540 1        0       0
1       0.52    -1      -1      0.6676 0        1       0
1       0.30    -1      -1      0.2341 0        1       0
-1      0.22    0       1       0.7572 0        0       1
-1      0.26    1       0       0.4205 0        1       0
-1      0.18    0       1       0.5939 0        0       1
-1      0.24    0       1       0.4711 1        0       0
1       0.16    -1      -1      0.5159 0        0       1
1       0.70    1       0       0.1980 1        0       0
1       0.48    0       1       0.8931 0        1       0
-1      0.22    1       0       0.5997 0        0       1
-1      0.84    1       0       0.5968 0        0       1
1       0.48    -1      -1      0.9913 0        1       0
1       0.70    1       0       0.9740 0        1       0
-1      0.30    -1      -1      0.9017 0        1       0
-1      0.36    1       0       0.1214 0        1       0
-1      0.74    -1      -1      0.8772 1        0       0
1       0.30    1       0       0.9538 0        1       0
1       0.24    -1      -1      0.1676 0        0       1
1       0.72    0       1       0.0824 1        0       0
1       0.72    -1      -1      0.2341 1        0       0
-1      0.10    -1      -1      0.8772 1        0       0
1       0.46    0       1       0.3931 0        1       0
-1      0.64    1       0       0.5390 0        1       0
-1      0.14    1       0       0.7702 1        0       0
1       0.78    0       1       0.9393 0        0       1
1       0.60    0       1       0.9379 0        1       0
-1      0.24    -1      -1      0.6315 1        0       0
-1      0.06    -1      -1      0.7659 0        0       1
1       0.60    0       1       0.7832 0        1       0
1       0.32    0       1       0.6127 0        1       0
-1      0.32    -1      -1      0.1980 0        1       0
-1      0.72    -1      -1      0.9046 1        0       0
-1      0.70    1       0       0.9393 0        0       1
1       0.68    1       0       0.7514 0        0       1
1       0.20    -1      -1      0.0607 0        0       1
```

Data Listing 5-1: Boston_train.txt (tabs replaced with commas to save space)

```
0.002155,0.000000,0.371334,-1,0.213992,0.502203,0.406797,0.256054,0.130435,0.171756,0.638298,0.991755,0.213576,2.50
0.574683,0.000000,0.646628,-1,0.436214,0.420770,1.000000,0.025771,1.000000,0.914122,0.808511,0.005749,0.231236,1.50
```

0.006561,0.000000,0.785557,-1,0.491770,0.538609,0.978373,0.108922,0.130435,0.477099,0.914894,0.971910,0.259106,2.30
0.000815,0.800000,0.164589,-1,0.053498,0.687296,0.255407,0.362566,0.130435,0.110687,0.702128,1.000000,0.050497,3.73
0.001369,0.125000,0.205645,-1,0.049383,0.445296,0.309990,0.488174,0.130435,0.301527,0.670213,1.000000,0.194812,2.09
0.159940,0.000000,0.646628,-1,0.633745,0.533052,1.000000,0.040420,1.000000,0.914122,0.808511,1.000000,0.512969,0.72
0.134256,0.000000,0.646628,-1,0.563786,0.392221,1.000000,0.014149,1.000000,0.914122,0.808511,0.836578,0.286976,2.79
0.003876,0.000000,0.253666,-1,0.222222,0.475187,0.484037,0.326592,0.173913,0.190840,0.744681,1.000000,0.164735,2.04
0.000052,0.550000,0.065616,-1,0.008230,0.554129,0.298661,0.561767,0.000000,0.215649,0.287234,0.994503,0.179360,2.20
0.132696,0.000000,0.646628,-1,0.685185,0.625216,0.757981,0.060417,1.000000,0.914122,0.808511,0.121363,0.579746,0.84
0.009376,0.000000,0.281525,-1,0.314815,0.390496,0.852729,0.302358,0.130435,0.229008,0.893617,0.764285,0.407837,1.39
0.000271,0.000000,0.173387,-1,0.267490,0.447212,0.583934,0.407879,0.173913,0.070611,0.808511,0.994730,0.243653,1.85
0.000706,0.450000,0.109238,-1,0.106996,0.608929,0.287333,0.486519,0.173913,0.402672,0.276596,0.981870,0.081678,3.05
0.113027,0.000000,0.646628,-1,0.409465,0.626940,0.941298,0.087170,1.000000,0.914122,0.808511,0.204272,0.495585,1.41
0.000455,0.000000,0.108138,-1,0.213992,0.547231,0.650875,0.178478,0.043478,0.158397,0.553191,0.988098,0.195364,2.26
0.000328,0.800000,0.116569,-1,0.014403,0.443572,0.166838,0.735726,0.000000,0.244275,0.404255,0.995663,0.207506,2.09
0.137585,0.000000,0.646628,-1,0.409465,0.436099,0.584964,0.078931,1.000000,0.914122,0.808511,0.061350,0.385210,1.02
0.091553,0.000000,0.646628,-1,0.648148,0.350450,0.988671,0.054424,1.000000,0.914122,0.808511,1.000000,0.527594,1.15
0.000574,0.000000,0.147727,-1,0.131687,0.588044,0.547889,0.300821,0.086957,0.114504,0.627660,0.988401,0.132450,2.66
0.002073,0.220000,0.197947,-1,0.094650,0.604905,0.150360,0.608981,0.260870,0.272901,0.691489,0.992032,0.133278,2.62
0.001213,0.000000,0.492302,1,0.339506,0.538801,0.921730,0.203121,0.173913,0.169847,0.404255,0.992032,0.241998,2.30
0.002294,0.200000,0.105205,-1,0.119136,0.622916,0.301751,0.270176,0.173913,0.055344,0.244681,1.000000,0.086093,3.51
0.000000,0.180000,0.067815,-1,0.314815,0.577505,0.641607,0.269203,0.000000,0.208015,0.287234,1.000000,0.089680,2.40
0.092851,0.000000,0.646628,1,0.582305,0.443380,0.892894,0.000000,1.000000,0.914122,0.808511,0.876393,0.197296,5.00
0.043054,0.000000,0.646628,-1,0.792181,0.515424,0.908342,0.106021,1.000000,0.914122,0.808511,0.883378,0.343819,1.99
0.000255,0.400000,0.028959,-1,0.090535,0.647250,0.325438,0.696787,0.000000,0.282443,0.755319,0.982223,0.114790,2.66
0.001361,0.000000,0.236437,-1,0.129630,0.614869,0.000000,0.417509,0.086957,0.087786,0.563830,0.971027,0.085817,2.66
0.001090,0.250000,0.171188,-1,0.139918,0.453344,0.456231,0.527640,0.304348,0.185115,0.755319,1.000000,0.206678,1.96
0.000565,0.000000,0.108138,-1,0.213992,0.660280,0.858908,0.208431,0.043478,0.158397,0.553191,1.000000,0.104029,2.36
0.001127,0.800000,0.053152,-1,0.057613,0.455068,0.170958,0.859888,0.130435,0.280534,1.000000,0.947400,0.105960,2.06
0.168788,0.000000,0.646628,-1,0.471193,0.333972,0.972194,0.088307,1.000000,0.914122,0.808511,0.880428,0.639625,1.20
0.004283,0.000000,0.923387,-1,0.403292,0.393179,0.954686,0.057071,0.043478,0.001908,0.691489,0.905164,0.704470,1.57
0.003120,0.000000,0.253666,-1,0.222222,0.411381,0.735324,0.326592,0.173913,0.190840,0.744681,0.985451,0.276214,1.85
0.001173,0.000000,0.492302,1,0.339506,0.457942,0.936148,0.160018,0.173913,0.169847,0.404255,1.000000,0.446744,2.15
0.000139,0.850000,0.135264,-1,0.090535,0.566200,0.255407,0.673435,0.130435,0.312977,0.563830,0.988729,0.127759,2.31
0.002638,0.000000,0.253666,-1,0.222222,0.483234,0.420185,0.389773,0.173913,0.190840,0.744681,1.000000,0.305188,2.22
0.024085,0.000000,0.700880,-1,1.000000,0.411573,0.984552,0.044885,0.173913,0.412214,0.223404,0.659716,0.387969,1.94
0.070605,0.000000,0.646628,-1,0.730453,0.532669,0.962925,0.085697,1.000000,0.914122,0.808511,0.801074,0.443157,1.49
0.045320,0.000000,0.646628,-1,0.302469,0.511209,0.904222,0.179114,1.000000,0.914122,0.808511,0.996041,0.307395,1.96
0.146890,0.000000,0.646628,-1,0.401235,0.412340,0.554068,0.154053,1.000000,0.914122,0.808511,1.000000,0.359547,2.01
0.000685,0.000000,0.101906,-1,0.154321,0.531136,0.147271,0.371468,0.130435,0.463740,0.457447,0.945307,0.154801,2.26
0.008743,0.000000,0.281525,-1,0.314815,0.465415,0.811535,0.284471,0.130435,0.229008,0.893617,0.974406,0.357064,1.75
0.000316,0.825000,0.057551,-1,0.061728,0.498371,0.365602,0.467441,0.043478,0.307252,0.223404,0.992108,0.157285,2.41
0.002870,0.000000,0.346041,-1,0.327160,0.471738,0.901133,0.154989,0.130435,0.223282,0.617021,0.998487,0.275662,1.94
0.000633,0.000000,0.420455,-1,0.386831,0.580954,0.681771,0.122671,0.000000,0.164122,0.893617,0.987619,0.219095,2.24
0.000440,0.000000,0.101906,-1,0.154321,0.494922,0.301751,0.431394,0.130435,0.463740,0.457447,0.928564,0.203091,1.98
0.001403,0.000000,0.296921,-1,0.277778,0.491665,0.847580,0.090280,0.173913,0.375954,0.882979,0.976776,0.341060,2.04
0.000512,0.000000,0.147727,-1,0.131687,0.470205,0.434604,0.299866,0.086957,0.114504,0.627660,0.997705,0.307119,2.25
0.024151,0.000000,0.700880,-1,1.000000,0.396053,1.000000,0.035192,0.173913,0.412214,0.223404,0.426017,0.411700,1.56
0.763342,0.000000,0.646628,-1,0.633745,0.406591,1.000000,0.026898,1.000000,0.914122,0.808511,0.969918,0.586369,0.50
0.005273,0.000000,0.346041,-1,0.327160,0.488983,0.575695,0.261192,0.130435,0.223282,0.617021,0.998311,0.303532,2.10
0.161036,0.000000,0.646628,-1,0.648148,0.252730,1.000000,0.041821,1.000000,0.914122,0.808511,0.939533,0.797185,1.02
0.000374,0.000000,0.173387,-1,0.267490,0.474420,0.325438,0.441552,0.173913,0.070611,0.808511,1.000000,0.173289,2.11
0.001382,0.000000,0.493402,-1,0.106996,0.597241,0.290422,0.439287,0.130435,0.194656,0.361702,1.000000,0.125276,2.86
0.004079,0.000000,0.346041,-1,0.327160,0.575972,0.869207,0.224854,0.130435,0.223282,0.617021,0.996949,0.208333,2.38
0.178353,0.000000,0.646628,-1,0.588477,0.571757,0.990731,0.035428,1.000000,0.914122,0.808511,1.000000,0.533940,1.09
0.000549,0.330000,0.063050,-1,0.179012,0.704158,0.393409,0.263020,0.260870,0.066794,0.617021,0.991881,0.143488,3.61
0.002945,0.000000,0.338343,-1,0.411523,0.427860,0.697219,0.160327,0.217391,0.389313,0.702128,1.000000,0.341336,1.83
0.047825,0.000000,0.646628,-1,0.792181,0.488791,0.807415,0.125444,1.000000,0.914122,0.808511,0.984467,0.301876,2.26
0.001351,0.250000,0.171188,-1,0.139918,0.613336,0.417096,0.623021,0.304348,0.185115,0.755319,0.996672,0.214404,2.50
0.000970,0.340000,0.206378,-1,0.098765,0.562177,0.159629,0.396666,0.260870,0.270992,0.372340,0.966488,0.191501,2.64
0.015946,0.000000,0.700880,-1,1.000000,0.565051,1.000000,0.057862,0.173913,0.412214,0.223404,0.917822,0.156181,2.33
0.001665,0.250000,0.171188,-1,0.139918,0.495114,0.270855,0.607917,0.304348,0.185115,0.755319,0.984316,0.141556,2.33
0.006049,0.000000,0.785557,-1,0.491770,0.496264,0.978373,0.049023,0.130435,0.477099,0.914894,1.000000,0.461645,1.78
0.000076,0.900000,0.027859,-1,0.037037,0.706649,0.195675,0.688103,0.173913,0.074427,0.563830,0.997554,0.084989,3.54
0.001625,0.000000,1.000000,-1,0.460905,0.362713,0.924820,0.062863,0.130435,1.000000,0.797872,0.995436,0.450607,1.52
0.000878,0.000000,0.131598,-1,0.257202,0.440314,0.677652,0.142977,0.173913,0.208015,0.425532,0.990746,0.218267,2.26
0.012591,0.000000,0.700880,1,1.000000,0.278023,0.876416,0.043703,0.173913,0.412214,0.223404,0.864794,0.286700,1.53
0.009205,0.200000,0.128666,-1,0.539095,0.721594,0.943357,0.086315,0.173913,0.146947,0.042553,0.991225,0.262693,3.10
0.003763,0.000000,0.253666,-1,0.222222,0.546848,0.383110,0.326592,0.173913,0.190840,0.744681,1.000000,0.121137,2.50
0.000609,0.000000,0.073314,-1,0.211934,0.496647,0.678682,0.195519,0.086957,0.011450,0.553191,0.975314,0.315121,2.96
0.001896,0.000000,0.201613,-1,0.234568,0.460816,0.281153,0.247133,0.173913,0.175573,0.702128,0.991250,0.231788,2.47
0.003473,0.000000,0.210411,-1,0.244856,0.901514,0.776519,0.160482,0.304348,0.229008,0.510638,0.970120,0.066501,4.48
0.000129,0.950000,0.037023,-1,0.037037,0.684806,0.113285,0.593240,0.086957,0.410305,0.468085,0.968228,0.075055,3.29
0.003636,0.000000,0.785557,-1,0.491770,0.433225,0.952626,0.121880,0.130435,0.477099,0.914894,0.979298,0.366998,1.84
0.004153,0.000000,0.371334,1,0.213992,0.353133,0.882595,0.230556,0.130435,0.171756,0.638298,0.995814,0.613962,1.93
0.016747,0.000000,0.700880,-1,1.000000,0.353133,1.000000,0.042012,0.173913,0.412214,0.223404,0.860558,0.318709,1.96
0.002782,0.000000,0.236437,-1,0.129630,0.352175,0.951596,0.431067,0.086957,0.087786,0.563830,1.000000,0.802428,1.44
0.006429,0.200000,0.128666,-1,0.390947,0.907454,0.660144,0.117488,0.173913,0.146947,0.042553,0.968834,0.157561,5.00
0.000098,0.900000,0.027493,1,0.032922,0.835792,0.225541,0.432431,0.000000,0.020992,0.106383,0.996520,0.039459,5.00
0.018057,0.000000,0.281525,-1,0.314815,0.485725,0.968074,0.239176,0.130435,0.229008,0.893617,0.625321,0.513521,1.35
0.001877,0.000000,0.338343,-1,0.411523,0.411190,0.526262,0.113859,0.217391,0.389313,0.702128,1.000000,0.283664,2.18
0.000412,0.525000,0.178152,-1,0.041152,0.575589,0.205973,0.562668,0.217391,0.202290,0.425532,0.936507,0.214680,2.48
0.040040,0.000000,0.646628,-1,0.401235,0.551063,0.742533,0.160673,1.000000,0.914122,0.808511,0.991099,0.348510,2.32
0.005965,0.000000,0.210411,-1,0.244856,0.463690,0.671473,0.231147,0.304348,0.229008,0.510638,0.953225,0.273731,2.43
0.015022,0.000000,0.700880,-1,0.452675,0.479977,1.000000,0.057080,0.173913,0.412214,0.223404,0.891548,0.129691,2.43
0.018234,0.000000,0.785557,-1,0.491770,0.279364,1.000000,0.028172,0.130435,0.477099,0.914894,1.000000,0.901766,1.44
0.000858,0.000000,0.493402,-1,0.106996,0.491665,0.159629,0.397667,0.130435,0.194656,0.361702,1.000000,0.189018,2.39
0.013500,0.000000,0.700880,-1,0.452675,0.443380,0.944387,0.117879,0.173913,0.412214,0.223404,0.736220,0.350442,1.74
0.026664,0.000000,0.646628,-1,0.407407,0.442614,0.401648,0.235921,1.000000,0.914122,0.808511,0.934011,0.320364,2.06
0.054682,0.000000,0.646628,-1,0.471193,0.560069,0.934089,0.106912,1.000000,0.914122,0.808511,0.998260,0.467715,1.67
0.000622,0.000000,0.144062,-1,0.117284,0.447787,0.508754,0.625995,0.086957,0.314885,0.659574,0.918579,0.301876,1.72
0.002761,0.000000,0.371334,-1,0.213992,0.425752,0.718847,0.293292,0.130435,0.171756,0.638298,0.981164,0.450607,2.25
0.000563,0.400000,0.218109,1,0.127572,0.612569,0.308960,0.268076,0.130435,0.127863,0.531915,1.000000,0.049669,3.24
0.006112,0.200000,0.128666,-1,0.539095,0.698410,0.913491,0.072793,0.173913,0.146947,0.042553,0.977281,0.175773,3.65

```
0.000160,0.850000,0.010264,-1,0.051440,0.540717,0.337796,0.732752,0.043478,0.240458,0.500000,1.000000,0.111479,2.47
0.090474,0.000000,0.646628,-1,0.409465,0.357540,0.952626,0.118233,1.000000,0.914122,0.808511,0.888244,0.452815,1.38
0.002842,0.000000,0.785557,-1,0.491770,0.408507,0.958805,0.059899,0.130435,0.477099,0.914894,0.987922,0.426600,1.62
0.000618,0.200000,0.105205,1,0.119136,0.782525,0.481977,0.371223,0.173913,0.055344,0.244681,0.949997,0.035320,4.60
0.005133,0.000000,0.210411,-1,0.244856,0.737881,0.762101,0.231147,0.304348,0.229008,0.510638,0.947652,0.097130,3.17
0.000031,0.900000,0.092009,-1,0.030864,0.675800,0.184346,0.561767,0.000000,0.187023,0.287234,0.994503,0.168874,3.22
0.075435,0.000000,0.646628,-1,0.674897,0.610845,0.923790,0.108576,1.000000,0.914122,0.808511,0.000000,0.433499,1.34
0.006938,0.000000,0.210411,1,0.251029,0.635754,0.770340,0.194828,0.304348,0.229008,0.510638,0.983585,0.226269,2.75
0.092636,0.000000,0.646628,-1,0.674897,0.734240,0.992791,0.120316,1.000000,0.914122,0.808511,0.946972,0.414183,1.78
0.084520,0.000000,0.646628,-1,0.674897,0.547231,0.982492,0.095973,1.000000,0.914122,0.808511,0.766277,0.485099,1.30
0.061086,0.000000,0.646628,-1,0.674897,0.592834,0.981462,0.111450,1.000000,0.914122,0.808511,0.895078,0.441501,1.52
0.130062,0.000000,0.646628,-1,0.648148,0.282621,0.969104,0.058235,1.000000,0.914122,0.808511,1.000000,0.660872,0.97
0.016377,0.000000,0.700880,-1,0.452675,0.752635,0.905252,0.076503,0.173913,0.412214,0.223404,0.943341,0.000000,5.00
0.002370,0.000000,0.371334,1,0.213992,0.430351,0.524202,0.229428,0.130435,0.171756,0.638298,0.984972,0.394592,2.24
0.018549,0.000000,0.700880,-1,1.000000,0.490707,0.972194,0.044413,0.173913,0.412214,0.223404,0.939230,0.341336,2.15
0.005852,0.000000,0.210411,-1,0.244856,0.989462,0.824923,0.160482,0.304348,0.229008,0.510638,0.962429,0.080022,5.00
0.013782,0.000000,0.281525,-1,0.314815,0.494539,0.914521,0.258918,0.130435,0.229008,0.893617,1.000000,0.468819,1.52
0.188890,0.000000,0.646628,-1,0.648148,0.328799,0.980433,0.026962,1.000000,0.914122,0.808511,1.000000,0.802428,0.72
0.047388,0.000000,0.646628,1,0.792181,0.429584,0.886715,0.070483,1.000000,0.914122,0.808511,0.889404,0.356236,1.68
0.000438,0.000000,0.420455,-1,0.386831,0.490324,0.760041,0.105293,0.000000,0.164122,0.893617,1.000000,0.202815,2.06
0.001493,0.000000,0.131598,-1,0.257202,0.385323,0.881565,0.133356,0.173913,0.208015,0.425532,1.000000,0.357616,2.31
0.003199,0.000000,0.785557,-1,0.491770,0.500671,0.934089,0.043858,0.130435,0.477099,0.914894,0.977760,0.618929,1.40
0.001859,0.250000,0.171188,-1,0.139918,0.460816,0.932029,0.517319,0.304348,0.185115,0.755319,0.952544,0.350717,1.60
0.100895,0.000000,0.646628,1,0.792181,0.507952,0.973223,0.090262,1.000000,0.914122,0.808511,0.951662,0.437914,1.78
0.000425,0.700000,0.065249,-1,0.030864,0.634221,0.458290,0.609099,0.173913,0.326336,0.234043,0.984770,0.119757,2.48
0.106860,0.000000,0.646628,-1,0.674897,0.606821,0.939238,0.124262,1.000000,0.914122,0.808511,0.016037,0.468543,1.49
0.000545,0.330000,0.063050,-1,0.179012,0.585361,0.568486,0.203730,0.260870,0.066794,0.617021,0.991074,0.198675,2.84
0.050979,0.000000,0.646628,-1,0.792181,0.543591,0.876416,0.126272,1.000000,0.914122,0.808511,0.943668,0.167219,2.50
0.078861,0.000000,0.646628,-1,0.685185,0.468481,0.951596,0.067746,1.000000,0.914122,0.808511,0.806042,0.385486,1.42
0.000175,0.600000,0.090543,-1,0.032922,0.620617,0.072091,0.462858,0.000000,0.148855,0.319149,0.991099,0.091060,3.11
0.000301,0.000000,0.173387,-1,0.267490,0.478636,0.354274,0.334876,0.173913,0.070611,0.808511,0.998084,0.187086,2.06
0.000236,0.000000,0.242302,-1,0.172840,0.694386,0.599382,0.348962,0.043478,0.104962,0.553191,0.989737,0.063466,3.47
0.083147,0.000000,0.646628,-1,0.436214,0.393945,0.978373,0.029563,1.000000,0.914122,0.808511,0.792577,0.680740,1.72
0.157855,0.000000,0.646628,-1,0.436214,0.593217,1.000000,0.036183,1.000000,0.914122,0.808511,0.087574,0.537804,1.72
0.000673,0.000000,0.101906,-1,0.154321,0.442039,0.235839,0.371468,0.130435,0.463740,0.457447,0.963538,0.227373,1.93
0.178195,0.000000,0.646628,-1,0.604938,0.447404,0.952626,0.070929,1.000000,0.914122,0.808511,0.018559,0.625276,0.83
0.001751,0.200000,0.238270,-1,0.162551,0.513317,0.138002,0.300030,0.086957,0.068702,0.638298,1.000000,0.134106,2.52
0.171762,0.000000,0.646628,-1,0.588477,0.591684,0.930999,0.019578,1.000000,0.914122,0.808511,0.914570,0.593543,1.39
0.001397,0.000000,0.350073,-1,0.333333,0.442805,0.722966,0.122571,0.217391,0.467557,0.553191,0.853069,0.376380,2.04
0.032794,0.000000,0.700880,-1,0.452675,0.486683,0.927909,0.104920,0.173913,0.412214,0.223404,0.604771,0.222958,2.50
0.006198,0.000000,0.785557,-1,0.491770,0.531519,0.981462,0.089216,0.130435,0.477099,0.914894,0.994377,0.420254,1.81
0.274109,0.000000,0.646628,-1,0.648148,0.209044,1.000000,0.030700,1.000000,0.914122,0.808511,1.000000,0.732616,1.05
0.001117,0.000000,1.000000,-1,0.460905,0.464074,0.987642,0.067155,0.130435,1.000000,0.797872,0.982879,0.450883,1.36
0.003889,0.000000,0.785557,-1,0.491770,0.554321,0.983522,0.065491,0.130435,0.477099,0.914894,0.992889,0.354857,1.71
0.000380,0.800000,0.038856,-1,0.039095,0.713930,0.321318,0.561922,0.043478,0.270992,0.000000,1.000000,0.064845,3.33
0.001452,0.125000,0.205645,-1,0.049383,0.389538,0.349125,0.488174,0.130435,0.301527,0.670213,1.000000,0.313466,1.74
0.003184,0.000000,0.338343,-1,0.411523,0.350450,0.720906,0.151770,0.217391,0.389313,0.702128,1.000000,0.535596,1.97
0.003751,0.000000,0.785557,-1,0.491770,0.555087,0.988671,0.089925,0.130435,0.477099,0.914894,0.995310,0.299945,1.92
0.003643,0.000000,0.210411,-1,0.251029,0.483809,0.603502,0.229365,0.304348,0.229008,0.510638,0.949191,0.252483,2.40
0.004963,0.000000,0.210411,1,0.251029,0.606438,0.654995,0.229365,0.304348,0.229008,0.510638,0.907459,0.174393,2.90
0.002307,0.000000,0.296921,-1,0.277778,0.493581,0.870237,0.144141,0.173913,0.375954,0.882979,0.993873,0.323124,1.93
0.087062,0.000000,0.646628,-1,0.674897,0.525005,0.832132,0.150361,1.000000,0.914122,0.808511,0.685587,0.400110,1.49
0.124549,0.000000,0.646628,-1,0.685185,0.546082,1.000000,0.066319,1.000000,0.914122,0.808511,0.802940,0.366722,1.67
0.064088,0.000000,0.646628,-1,0.302469,0.611037,0.741504,0.200247,1.000000,0.914122,0.808511,0.990342,0.165839,2.37
0.125369,0.000000,0.646628,-1,0.730453,0.587852,0.944387,0.090489,1.000000,0.914122,0.808511,0.276186,0.594371,1.34
0.003655,0.000000,0.210411,-1,0.251029,0.897873,0.695160,0.229365,0.304348,0.229008,0.510638,0.954738,0.061258,4.83
0.009834,0.000000,0.785557,-1,0.491770,0.397777,0.945417,0.077322,0.130435,0.477099,0.914894,1.000000,0.458333,1.43
0.200746,0.000000,0.646628,-1,0.588477,0.510059,1.000000,0.023325,1.000000,0.914122,0.808511,0.992032,0.553256,1.02
0.000335,0.800000,0.164589,-1,0.053498,0.588044,0.211123,0.362566,0.130435,0.110687,0.702128,1.000000,0.081954,2.79
0.001308,0.000000,0.236437,-1,0.129630,0.480552,0.382080,0.417509,0.086957,0.087786,0.563830,0.981063,0.215784,2.12
0.000559,0.000000,0.073314,-1,0.211934,0.818164,0.522142,0.188198,0.086957,0.011450,0.553191,0.989233,0.075055,5.00
0.041271,0.000000,0.646628,-1,0.792181,0.345085,0.960865,0.088570,1.000000,0.914122,0.808511,0.959378,0.233444,2.08
0.000084,0.800000,0.000000,-1,0.076132,0.826595,0.299691,0.410916,0.130435,0.129771,0.191489,0.993267,0.034216,5.00
0.029551,0.000000,0.346041,-1,0.327160,0.270550,0.359423,0.126381,0.130435,0.223282,0.617021,0.882874,0.301049,1.61
0.000098,0.800000,0.056818,-1,0.102881,0.589002,0.276004,0.656039,0.130435,0.177481,0.468085,0.984972,0.117550,2.45
0.170517,0.000000,0.646628,-1,0.730453,0.496455,1.000000,0.071347,1.000000,0.914122,0.808511,0.022694,0.682119,0.87
0.000364,0.525000,0.178152,-1,0.041152,0.507377,0.292482,0.562668,0.217391,0.202290,0.425532,1.000000,0.149283,2.32
0.203196,0.000000,0.646628,-1,0.604938,0.550489,1.000000,0.064118,1.000000,0.914122,0.808511,0.067906,0.753863,0.72
0.000340,0.250000,0.161290,-1,0.084362,0.525196,0.301751,0.388391,0.130435,0.179389,0.680851,1.000000,0.137693,2.48
0.001142,0.000000,0.296921,-1,0.277778,0.504694,0.530381,0.149879,0.173913,0.375954,0.882979,0.991401,0.310982,2.17
0.000743,0.600000,0.045088,-1,0.053498,0.445104,0.160659,0.871218,0.130435,0.427481,0.606383,0.988476,0.167219,1.86
0.000240,0.750000,0.091276,-1,0.088477,0.581337,0.194645,0.388428,0.086957,0.124046,0.606383,0.996798,0.071468,3.08
0.002685,0.000000,0.346041,-1,0.327160,0.425560,0.708548,0.263902,0.130435,0.223282,0.617021,1.000000,0.392108,1.98
0.000236,0.000000,0.242302,-1,0.172840,0.547998,0.782698,0.348962,0.043478,0.104962,0.553191,1.000000,0.204470,2.16
0.000718,0.000000,0.131598,-1,0.257202,0.471163,0.456231,0.220544,0.173913,0.208015,0.425532,0.990746,0.231236,2.32
0.002125,0.000000,0.379399,-1,0.057613,0.514275,0.033986,0.378079,0.130435,0.225191,0.702128,0.950250,0.160320,2.34
0.000323,0.950000,0.081378,-1,0.063992,0.822380,0.312049,0.362684,0.130435,0.070611,0.223404,0.989611,0.057395,4.85
0.000872,0.000000,0.453446,-1,0.106996,0.443188,0.347065,0.306723,0.173913,0.402672,0.648936,0.997882,0.203366,2.03
0.038979,0.000000,0.646628,1,0.685185,1.000000,0.823893,0.070483,1.000000,0.914122,0.808511,0.893212,0.098234,2.19
1.000000,0.000000,0.646628,-1,0.588477,0.652807,0.916581,0.026089,1.000000,0.914122,0.808511,1.000000,0.427152,1.04
0.162212,0.000000,0.646628,-1,0.436214,0.630581,1.000000,0.030545,1.000000,0.914122,0.808511,0.451460,0.498068,2.75
0.030658,0.000000,0.700880,-1,1.000000,0.390113,0.947477,0.036019,0.173913,0.412214,0.223404,0.886404,0.544150,1.54
0.000500,0.000000,0.173387,-1,0.267490,0.527879,0.362513,0.484573,0.173913,0.070611,0.808511,0.981870,0.108996,2.22
0.000772,0.330000,0.063050,-1,0.179012,0.739414,0.710608,0.179105,0.260870,0.066794,0.617021,1.000000,0.130795,3.34
0.000826,0.400000,0.218109,-1,0.127572,0.559686,0.300721,0.273777,0.130435,0.127863,0.531915,1.000000,0.150662,2.91
0.041439,0.000000,0.646628,-1,0.674897,0.539375,0.880536,0.130719,1.000000,0.914122,0.808511,0.986207,0.356512,1.77
0.078182,0.000000,0.646628,-1,0.648148,0.412340,0.969104,0.072466,1.000000,0.914122,0.808511,0.993772,0.424393,1.51
0.001813,0.000000,0.253666,-1,0.222222,0.548956,0.508754,0.310160,0.173913,0.190840,0.744681,1.000000,0.150938,2.38
0.072359,0.000000,0.646628,-1,0.409465,0.548764,0.740474,0.097373,1.000000,0.914122,0.808511,0.246180,0.284216,1.61
0.143824,0.000000,0.646628,-1,0.730453,0.439356,0.964985,0.069656,1.000000,0.914122,0.808511,0.605679,0.608720,1.08
0.002779,0.000000,0.346041,-1,0.327160,0.410807,0.770340,0.256018,0.130435,0.223282,0.617021,0.998790,0.269592,1.62
0.000345,0.200000,0.105205,-1,0.119136,0.652807,0.353244,0.374205,0.173913,0.055344,0.244681,0.988224,0.078918,3.54
0.003506,0.000000,0.346041,-1,0.327160,0.450853,0.826982,0.260892,0.130435,0.223282,0.617021,0.984366,0.458057,1.78
```

```
0.111482,0.000000,0.646628,-1,0.730453,0.515424,0.964985,0.097155,1.000000,0.914122,0.808511,0.978869,0.405905,1.26
0.064346,0.000000,0.646628,-1,0.302469,0.670627,0.763131,0.207422,1.000000,0.914122,0.808511,0.995915,0.145695,2.50
0.088041,0.000000,0.646628,-1,0.555556,0.507377,0.643666,0.166756,1.000000,0.914122,0.808511,1.000000,0.317053,2.14
0.073423,0.000000,0.646628,1,0.506173,0.662004,0.974253,0.006620,1.000000,0.914122,0.808511,0.987770,0.033940,5.00
0.001214,0.000000,0.296921,-1,0.277778,0.616976,0.704428,0.156999,0.173913,0.375954,0.882979,0.996672,0.163907,2.65
0.054984,0.000000,0.646628,-1,0.506173,0.269975,1.000000,0.018451,1.000000,0.914122,0.808511,0.946089,0.042219,5.00
0.001168,0.250000,0.171188,-1,0.139918,0.554704,0.668383,0.554329,0.304348,0.185115,0.755319,1.000000,0.137969,2.22
0.007387,0.200000,0.128666,-1,0.539095,0.722744,1.000000,0.069565,0.173913,0.146947,0.042553,0.965682,0.167219,3.60
0.000327,0.340000,0.206378,-1,0.098765,0.580379,0.386200,0.396666,0.260870,0.270992,0.372340,0.997100,0.214404,2.20
0.006803,0.200000,0.128666,-1,0.539095,0.985438,0.865088,0.061054,0.173913,0.146947,0.042553,0.981845,0.093543,5.00
0.003071,0.000000,0.338343,-1,0.411523,0.453152,0.408857,0.113859,0.217391,0.389313,0.702128,1.000000,0.327263,2.45
0.002457,0.125000,0.271628,-1,0.286008,0.539567,0.941298,0.474416,0.173913,0.236641,0.276596,0.988956,0.516556,1.50
0.004826,0.000000,0.371334,1,0.213992,0.341636,1.000000,0.249652,0.130435,0.171756,0.638298,1.000000,0.589404,2.00
0.119877,0.000000,0.646628,-1,0.730453,0.555279,0.946447,0.078049,1.000000,0.914122,0.808511,0.107771,0.613962,1.18
0.121703,0.000000,0.646628,-1,0.604938,0.617168,0.905252,0.062736,1.000000,0.914122,0.808511,0.053583,0.663907,0.75
0.001053,0.340000,0.206378,-1,0.098765,0.655490,0.152420,0.396666,0.260870,0.270992,0.372340,0.983686,0.086369,3.31
0.000908,0.000000,0.453446,-1,0.106996,0.494156,0.441813,0.269249,0.173913,0.402672,0.648936,0.974936,0.235651,2.08
0.002957,0.000000,0.346041,-1,0.327160,0.518299,0.822863,0.193982,0.130435,0.223282,0.617021,0.991149,0.170254,2.16
0.000998,0.000000,0.453446,-1,0.106996,0.522131,0.433574,0.306723,0.173913,0.402672,0.648936,0.965530,0.198951,2.14
0.220331,0.000000,0.646628,-1,0.588477,0.718912,0.978373,0.016978,1.000000,0.914122,0.808511,1.000000,0.323124,1.50
0.162015,0.000000,0.646628,-1,0.730453,0.555662,0.930999,0.079386,1.000000,0.914122,0.808511,0.068511,0.450331,0.96
0.026552,0.000000,0.700880,-1,1.000000,0.261544,0.955716,0.030118,0.173913,0.412214,0.223404,0.986913,0.767108,1.46
0.001180,0.000000,1.000000,-1,0.460905,0.464074,0.830072,0.089143,0.130435,1.000000,0.797872,1.000000,0.320640,2.01
0.000174,0.825000,0.057551,-1,0.061728,0.775819,0.131823,0.467441,0.043478,0.307252,0.223404,0.996167,0.038079,4.23
0.002426,0.000000,0.350073,-1,0.333333,0.484959,0.952626,0.128982,0.217391,0.467557,0.553191,1.000000,0.423841,1.87
0.107782,0.000000,0.646628,-1,0.633745,0.544740,1.000000,0.046322,1.000000,0.914122,0.808511,0.947577,0.512693,1.21
0.031607,0.000000,0.646628,-1,0.302469,0.421728,0.385170,0.269958,1.000000,0.914122,0.808511,0.989964,0.239790,2.18
0.041220,0.000000,0.646628,-1,0.407407,0.527112,0.504634,0.260264,1.000000,0.914122,0.808511,0.979121,0.244205,2.12
0.001519,0.000000,0.236437,-1,0.129630,0.499713,0.038105,0.417509,0.086957,0.087786,0.563830,0.965883,0.112583,2.53
0.001086,0.300000,0.163856,-1,0.088477,0.535926,0.514933,0.537051,0.217391,0.215649,0.425532,0.939104,0.261865,2.22
0.003281,0.000000,0.210411,-1,0.244856,0.790381,0.145211,0.204194,0.304348,0.229008,0.510638,0.951107,0.060430,4.67
0.011775,0.000000,0.281525,-1,0.314815,0.454876,0.271885,0.306359,0.130435,0.229008,0.893617,0.974658,0.133830,2.31
0.006981,0.000000,0.281525,-1,0.314815,0.435524,0.552008,0.306359,0.130435,0.229008,0.893617,0.996772,0.185982,1.99
0.049634,0.000000,0.646628,-1,0.409465,0.467906,0.943357,0.128282,1.000000,0.914122,0.808511,0.834560,0.540563,1.91
0.003954,0.000000,0.210411,1,0.251029,0.649550,0.881565,0.157508,0.304348,0.229008,0.510638,0.986888,0.220199,2.67
0.103189,0.000000,0.646628,-1,0.648148,0.378425,1.000000,0.040993,1.000000,0.914122,0.808511,1.000000,0.603477,1.13
0.001315,0.000000,0.350073,-1,0.333333,0.450661,0.926880,0.111286,0.217391,0.467557,0.553191,0.995083,0.399558,1.88
0.000612,0.000000,0.420455,-1,0.386831,0.654340,0.907312,0.094381,0.000000,0.164122,0.893617,1.000000,0.107892,2.39
0.079174,0.000000,0.646628,-1,0.471193,0.487066,0.846550,0.081132,1.000000,0.914122,0.808511,0.005547,0.594923,1.34
0.000279,0.000000,0.144062,-1,0.117284,0.470013,0.469619,0.625995,0.086957,0.314885,0.659574,0.971607,0.242826,1.75
0.002423,0.200000,0.238270,1,0.162551,0.791339,0.503605,0.294347,0.086957,0.068702,0.638298,0.984543,0.133830,3.52
0.000430,0.250000,0.161290,-1,0.084362,0.585936,0.695160,0.388391,0.130435,0.179389,0.680851,0.996798,0.151490,2.39
0.000421,0.800000,0.106672,-1,0.026749,0.426518,0.290422,0.498495,0.130435,0.286260,0.372340,1.000000,0.234823,1.94
0.000314,0.000000,0.173387,-1,0.267490,0.442230,0.446962,0.372969,0.173913,0.070611,0.808511,1.000000,0.222682,1.95
0.001462,0.000000,0.371334,-1,0.213992,0.446446,0.199794,0.256054,0.130435,0.171756,0.638298,1.000000,0.252208,2.26
0.114945,0.000000,0.646628,-1,0.471193,0.502778,0.966014,0.094654,1.000000,0.914122,0.808511,0.956629,0.449779,1.46
0.001161,0.000000,0.420455,-1,0.386831,0.619467,0.889804,0.114514,0.000000,0.164122,0.893617,0.991301,0.131071,2.20
0.004894,0.000000,0.210411,-1,0.244856,0.573098,0.190525,0.204194,0.304348,0.229008,0.510638,0.958243,0.056015,3.15
0.002335,0.220000,0.197947,-1,0.094650,0.551255,0.061792,0.569897,0.260870,0.272901,0.691489,0.949997,0.051325,2.48
0.001928,0.000000,0.338343,-1,0.411523,0.384748,0.727085,0.115514,0.217391,0.389313,0.702128,0.997151,0.368929,1.75
0.002046,0.000000,0.236437,-1,0.129630,0.426327,0.313079,0.361084,0.086957,0.087786,0.563830,1.000000,0.342715,2.00
0.001514,0.000000,0.493402,-1,0.106996,0.427093,0.567456,0.471988,0.130435,0.194656,0.361702,1.000000,0.389349,2.03
0.027461,0.000000,0.700880,-1,0.452675,0.544357,0.950566,0.103020,0.173913,0.412214,0.223404,0.831409,0.264625,2.23
0.000323,0.800000,0.164589,-1,0.053498,0.632305,0.257467,0.362566,0.130435,0.110687,0.702128,1.000000,0.044150,2.85
0.001287,0.000000,0.089076,-1,0.123457,0.863767,0.752832,0.215115,0.043478,0.169847,0.574468,1.000000,0.068433,3.87
0.153668,0.000000,0.646628,-1,0.730453,0.454876,0.875386,0.062836,1.000000,0.914122,0.808511,0.173055,0.891004,0.84
0.103698,0.000000,0.646628,-1,0.506173,0.508718,1.000000,0.003592,1.000000,0.914122,0.808511,0.922462,0.215232,5.00
0.000717,0.000000,0.492302,-1,0.339506,0.590343,0.846550,0.208377,0.173913,0.169847,0.404255,0.989611,0.219647,2.87
0.000627,0.400000,0.028959,-1,0.090535,0.561219,0.427394,0.696787,0.000000,0.282443,0.755319,1.000000,0.117274,2.29
0.466707,0.000000,0.646628,-1,0.633745,0.377467,0.849640,0.043449,1.000000,0.914122,0.808511,0.829946,0.707781,0.85
0.058396,0.000000,0.646628,1,0.792181,0.491665,0.829042,0.144868,1.000000,0.914122,0.808511,0.996293,0.269040,2.27
0.001840,0.125000,0.271628,-1,0.286008,0.468097,0.854789,0.496731,0.173913,0.236641,0.276596,0.974305,0.424117,1.89
0.001363,0.300000,0.163856,-1,0.088477,0.542633,0.050463,0.537051,0.217391,0.215649,0.425532,0.944047,0.095475,2.37
0.000903,0.450000,0.109238,-1,0.106996,0.693045,0.240989,0.486519,0.173913,0.402672,0.276596,0.983837,0.031457,3.64
0.052405,0.000000,0.646628,-1,0.674897,0.462732,0.875386,0.131946,1.000000,0.914122,0.808511,0.025619,0.476821,1.27
0.002511,0.000000,0.371334,-1,0.213992,0.529795,0.510814,0.293292,0.130435,0.171756,0.638298,0.994881,0.254967,2.44
0.000498,0.000000,0.147727,-1,0.131687,0.541866,0.464470,0.331894,0.086957,0.114504,0.627660,1.000000,0.217715,2.39
0.001990,0.000000,1.000000,-1,0.460905,0.355049,0.982492,0.056907,0.130435,1.000000,0.797872,0.866736,0.613687,0.70
0.000525,0.000000,0.108138,-1,0.213992,0.674075,0.619979,0.207777,0.043478,0.158397,0.553191,0.997882,0.109547,2.87
0.001054,0.000000,0.073314,-1,0.211934,0.575206,0.954686,0.156171,0.086957,0.011450,0.553191,1.000000,0.108996,3.25
0.026672,0.000000,0.700880,-1,1.000000,0.492240,1.000000,0.026326,0.173913,0.412214,0.223404,0.435196,0.719371,1.38
0.111950,0.000000,0.646628,-1,0.730453,0.560261,1.000000,0.077185,1.000000,0.914122,0.808511,0.974356,0.472406,1.54
0.000856,0.300000,0.163856,-1,0.088477,0.559494,0.160659,0.460157,0.217391,0.215649,0.425532,0.955898,0.127759,2.37
0.000580,0.125000,0.205645,-1,0.049383,0.443955,0.190525,0.488174,0.130435,0.301527,0.670213,0.998260,0.175773,2.20
0.000816,0.000000,0.453446,-1,0.106996,0.519640,0.031926,0.283889,0.173913,0.402672,0.648936,0.995007,0.139349,2.41
0.000283,0.950000,0.037023,-1,0.037037,0.654148,0.127703,0.593240,0.086957,0.410305,0.468085,1.000000,0.078091,3.49
0.005470,0.000000,0.346041,-1,0.327160,0.589002,0.819773,0.198956,0.130435,0.223282,0.617021,1.000000,0.077539,2.28
0.005932,0.200000,0.128666,-1,0.539095,0.758574,0.890834,0.091862,0.173913,0.146947,0.042553,0.978491,0.152594,4.31
0.000921,0.125000,0.271628,-1,0.286008,0.469630,0.656025,0.402923,0.173913,0.236641,0.276596,0.996722,0.295254,2.29
0.051136,0.000000,0.646628,-1,0.685185,0.000000,0.875386,0.043976,1.000000,0.914122,0.808511,0.893590,0.148731,2.75
0.000656,0.700000,0.065249,-1,0.030864,0.533436,0.177137,0.609099,0.173913,0.326336,0.234043,0.927732,0.089404,2.25
0.086161,0.000000,0.646628,-1,0.633745,0.418854,0.988671,0.045813,1.000000,0.914122,0.808511,0.990418,0.501932,0.85
0.124781,0.000000,0.646628,-1,0.582305,0.257712,1.000000,0.004056,1.000000,0.914122,0.808511,1.000000,0.911700,1.38
0.000974,0.000000,0.923387,-1,0.403292,0.459858,0.926880,0.087052,0.043478,0.001908,0.691489,0.952569,0.447020,2.05
0.000700,0.000000,0.089076,-1,0.123457,0.738647,0.613800,0.215115,0.043478,0.169847,0.574468,1.000000,0.123068,3.32
0.000620,0.000000,0.173387,-1,0.267490,0.461199,0.572606,0.334876,0.173913,0.070611,0.808511,1.000000,0.208609,1.87
0.011196,0.000000,0.281525,-1,0.314815,0.596474,0.869207,0.282752,0.130435,0.229008,0.893617,0.957966,0.282837,2.10
0.022523,0.000000,0.700880,-1,0.452675,0.836942,0.960865,0.083323,0.173913,0.412214,0.223404,0.930405,0.054360,5.00
0.001122,0.300000,0.163856,-1,0.088477,0.485534,0.640577,0.473452,0.217391,0.215649,0.425532,0.994251,0.294426,2.01
0.001941,0.000000,0.338343,-1,0.411523,0.404100,0.266735,0.151770,0.217391,0.389313,0.702128,0.990897,0.437914,2.31
0.001713,0.000000,0.379399,-1,0.057613,0.459858,0.150360,0.378079,0.130435,0.225191,0.702128,0.949670,0.224890,2.17
0.000676,0.000000,0.131598,-1,0.257202,0.632113,0.736354,0.162382,0.173913,0.208015,0.425532,0.985804,0.143212,2.99
0.000704,0.000000,0.923387,-1,0.403292,0.442422,0.687951,0.102583,0.043478,0.001908,0.691489,0.980458,0.348786,2.20
```

0.001556,0.000000,0.350073,-1,0.333333,0.415788,0.641607,0.148187,0.217391,0.467557,0.553191,0.986384,0.327815,1.93
0.003749,0.220000,0.197947,-1,0.094650,0.488025,0.329557,0.629805,0.260870,0.272901,0.691489,0.983055,0.205022,2.43
0.000155,0.950000,0.081378,-1,0.063992,0.857061,0.298661,0.362684,0.130435,0.070611,0.223404,0.983988,0.031733,5.00
0.000489,0.210000,0.189883,-1,0.111111,0.466948,0.190525,0.516973,0.130435,0.106870,0.446809,1.000000,0.184879,2.34
0.000669,0.000000,0.073314,-1,0.211934,0.805518,0.828012,0.146532,0.086957,0.011450,0.553191,0.996621,0.160872,3.98
0.000328,0.250000,0.161290,-1,0.084362,0.499329,0.451081,0.388391,0.130435,0.179389,0.680851,0.984215,0.159492,2.29
0.002261,0.000000,1.000000,-1,0.460905,0.293543,0.979403,0.063018,0.130435,1.000000,0.797872,0.802133,0.771247,0.81
0.015530,0.000000,0.281525,-1,0.314815,0.457751,0.814624,0.260110,0.130435,0.229008,0.893617,0.585708,0.716887,1.32
0.063903,0.000000,0.646628,-1,0.407407,0.489174,0.791967,0.219726,1.000000,0.914122,0.808511,0.989359,0.365618,1.91
0.161036,0.000000,0.646628,-1,0.471193,0.511209,0.876416,0.074712,1.000000,0.914122,0.808511,0.965757,0.314018,2.14
0.000210,0.000000,0.052419,-1,0.273663,0.570799,0.584964,0.467159,0.000000,0.448473,0.351064,0.982500,0.190949,1.65
0.082734,0.000000,0.646628,-1,0.604938,0.504311,0.774459,0.073293,1.000000,0.914122,0.808511,0.243104,0.546082,1.10
0.025785,0.000000,0.700880,-1,0.452675,0.528454,0.959835,0.088243,0.173913,0.412214,0.223404,0.748323,0.258554,2.38
0.000948,0.450000,0.109238,-1,0.106996,0.649550,0.191555,0.486519,0.173913,0.402672,0.276596,0.951536,0.092991,3.70
0.000293,0.000000,0.063050,-1,0.150206,0.658555,0.441813,0.448545,0.086957,0.066794,0.648936,0.994276,0.033389,3.34
0.000650,0.000000,0.201613,-1,0.234568,0.454493,0.672503,0.202848,0.173913,0.175573,0.702128,1.000000,0.219371,1.89
0.008617,0.000000,0.281525,-1,0.314815,0.562177,0.942327,0.302367,0.130435,0.229008,0.893617,0.977407,0.305464,1.84
0.000349,0.000000,0.173387,-1,0.267490,0.526729,0.366632,0.484573,0.173913,0.070611,0.808511,0.981088,0.138521,2.07
0.054016,0.000000,0.646628,-1,0.674897,0.601648,0.897013,0.133483,1.000000,0.914122,0.808511,0.642771,0.405353,1.64
0.001619,0.000000,0.923387,-1,0.403292,0.439739,0.969104,0.074094,0.043478,0.001908,0.691489,0.932952,0.653422,1.73
0.000102,0.900000,0.120601,-1,0.018519,0.745928,0.322348,0.473452,0.086957,0.108779,0.351064,0.973372,0.038079,4.40
0.009506,0.000000,0.281525,-1,0.314815,0.460625,0.888774,0.262138,0.130435,0.229008,0.893617,0.988981,0.333885,1.96
0.025937,0.000000,0.700880,-1,0.452675,0.444338,0.972194,0.114496,0.173913,0.412214,0.223404,0.877024,0.284216,1.91
0.151915,0.000000,0.646628,-1,0.506173,0.057865,1.000000,0.034646,1.000000,0.914122,0.808511,0.330576,0.320088,2.31
0.013999,0.000000,0.281525,-1,0.314815,0.384940,0.980433,0.242641,0.130435,0.229008,0.893617,0.948737,0.532285,1.36
0.041482,0.000000,0.646628,-1,0.685185,0.268634,0.911432,0.056625,1.000000,0.914122,0.808511,0.796081,0.338576,2.19
0.000983,0.125000,0.271628,-1,0.286008,0.446062,0.371782,0.392956,0.173913,0.236641,0.276596,0.983862,0.385762,2.17
0.000853,0.220000,0.197947,-1,0.094650,0.650699,0.040165,0.707208,0.260870,0.272901,0.691489,0.972742,0.049669,2.96
0.015820,0.000000,0.700880,1,1.000000,0.492048,0.958805,0.056361,0.173913,0.412214,0.223404,0.808664,0.369481,1.70
0.001343,0.450000,0.109238,-1,0.106996,0.573865,0.269825,0.312552,0.173913,0.402672,0.276596,0.964547,0.078091,2.98
0.020540,0.000000,0.700880,1,0.452675,0.812608,0.981462,0.082851,0.173913,0.412214,0.223404,0.981618,0.005243,5.00
0.174996,0.000000,0.646628,-1,0.401235,0.453152,0.701339,0.161755,1.000000,0.914122,0.808511,0.928993,0.452539,1.91
0.004076,0.220000,0.197947,-1,0.094650,0.900172,0.056643,0.707208,0.260870,0.272901,0.691489,1.000000,0.049945,4.28
0.000332,0.800000,0.106672,-1,0.026749,0.522897,0.153450,0.498495,0.130435,0.286260,0.372340,1.000000,0.081126,2.35
0.008813,0.000000,0.346041,-1,0.327160,0.490707,0.513903,0.137375,0.130435,0.223282,0.617021,1.000000,0.117274,2.21
0.002399,0.000000,0.236437,-1,0.129630,0.391071,0.608651,0.450863,0.086957,0.087786,0.563830,1.000000,0.399283,1.94
0.423240,0.000000,0.646628,-1,0.604938,0.506036,0.780639,0.066682,1.000000,0.914122,0.808511,0.046649,0.352925,1.09
0.065929,0.000000,0.646628,-1,0.633745,0.544932,0.958805,0.049759,1.000000,0.914122,0.808511,1.000000,0.486755,1.25
0.001062,0.000000,0.350073,-1,0.333333,0.604330,0.810505,0.140758,0.217391,0.467557,0.553191,0.996697,0.232616,2.28
0.000618,0.400000,0.218109,1,0.127572,0.625599,0.254377,0.339477,0.130435,0.127863,0.531915,0.991301,0.067053,3.31
0.135348,0.000000,0.646628,-1,0.471193,0.399885,0.872297,0.074712,1.000000,0.914122,0.808511,0.734354,0.341336,2.08
0.291495,0.000000,0.646628,-1,0.604938,0.333972,0.887745,0.047095,1.000000,0.914122,0.808511,0.320339,0.687362,1.04
0.000814,0.450000,0.109238,-1,0.106996,0.617168,0.393409,0.241795,0.173913,0.402672,0.276596,0.992360,0.136589,3.20
0.207844,0.000000,0.646628,-1,0.582305,0.110558,1.000000,0.000673,1.000000,0.914122,0.808511,1.000000,1.000000,1.38
0.000121,0.900000,0.057185,-1,0.051440,0.606821,0.341916,1.000000,0.173913,0.000000,0.468085,0.968632,0.076435,3.01
0.000453,0.800000,0.038856,-1,0.039095,0.679441,0.347065,0.561922,0.043478,0.270992,0.000000,0.892607,0.189845,3.03
0.098750,0.000000,0.646628,-1,0.409465,0.383982,0.697219,0.084924,1.000000,0.914122,0.808511,0.008397,0.425773,1.17
0.001855,0.000000,0.350073,-1,0.333333,0.453535,0.878476,0.121261,0.217391,0.467557,0.553191,0.868904,0.387141,1.83
0.000547,0.000000,0.173387,-1,0.267490,0.464457,0.437693,0.334876,0.173913,0.070611,0.808511,1.000000,0.221026,1.90
0.001856,0.000000,0.236437,-1,0.129630,0.406400,0.318229,0.361084,0.086957,0.087786,0.563830,1.000000,0.233996,1.93
0.000412,0.280000,0.534457,-1,0.162551,0.515041,0.766220,0.226009,0.130435,0.158397,0.595745,1.000000,0.244481,2.06
0.104786,0.000000,0.646628,-1,0.674897,0.502778,0.986612,0.102938,1.000000,0.914122,0.808511,1.000000,0.452539,1.41
0.826435,0.000000,0.646628,-1,0.604938,0.459092,1.000000,0.061199,1.000000,0.914122,0.808511,0.040673,0.521247,0.88
0.004224,0.000000,0.210411,-1,0.244856,0.858210,0.860968,0.189699,0.304348,0.229008,0.510638,0.975995,0.038631,3.76
0.007007,0.000000,0.281525,-1,0.314815,0.457559,0.606591,0.325355,0.130435,0.229008,0.893617,1.000000,0.180188,2.04
0.001970,0.000000,0.253666,-1,0.222222,0.539375,0.529351,0.310160,0.173913,0.190840,0.744681,1.000000,0.141832,2.31
0.000391,0.250000,0.161290,-1,0.084362,0.606630,0.315139,0.388391,0.130435,0.179389,0.680851,1.000000,0.098234,2.80
0.000416,0.210000,0.189883,-1,0.111111,0.489366,0.618950,0.516973,0.130435,0.106870,0.446809,0.992612,0.212472,2.05
0.048808,0.000000,0.646628,-1,0.401235,0.499329,0.835221,0.173122,1.000000,0.914122,0.808511,1.000000,0.401766,1.99
0.005775,0.200000,0.128666,-1,0.539095,0.926806,0.912461,0.105384,0.173913,0.146947,0.042553,0.974684,0.115342,4.88
0.074729,0.000000,0.646628,-1,0.674897,0.528071,0.824923,0.145932,1.000000,0.914122,0.808511,1.000000,0.338300,1.95
0.001498,0.000000,0.296921,-1,0.277778,0.499329,0.897013,0.117433,0.173913,0.375954,0.882979,0.989384,0.292494,2.01
0.039662,0.000000,0.700880,1,1.000000,0.496455,0.820803,0.056007,0.173913,0.412214,0.223404,0.221116,0.366722,1.56
0.002245,0.220000,0.197947,-1,0.094650,0.389347,0.757981,0.620657,0.260870,0.272901,0.691489,0.938449,0.297185,1.76
0.002500,0.000000,0.296921,-1,0.277778,0.544932,0.849640,0.144141,0.173913,0.375954,0.882979,0.177720,0.245585,1.86
0.232774,0.000000,0.646628,-1,0.563786,0.110558,1.000000,0.004410,1.000000,0.914122,0.808511,0.932725,0.596302,1.19
0.000863,0.000000,0.073314,-1,0.211934,0.391454,0.894954,0.168984,0.086957,0.011450,0.553191,0.985123,0.338024,2.64
0.000090,1.000000,0.031525,-1,0.053498,0.623683,0.387230,0.654293,0.173913,0.131679,0.265957,0.989914,0.061258,3.16
0.000969,0.300000,0.163856,-1,0.088477,0.583445,0.404737,0.460157,0.217391,0.215649,0.425532,0.966917,0.155629,2.33
0.000921,0.000000,0.379399,-1,0.057613,0.547231,0.038105,0.378079,0.130435,0.225191,0.702128,0.966791,0.137693,2.42
0.278694,0.000000,0.646628,-1,0.633745,0.342594,0.958805,0.052124,1.000000,0.914122,0.808511,1.000000,0.497792,0.83
0.014242,0.000000,0.700880,1,0.452675,0.515233,0.923790,0.060817,0.173913,0.412214,0.223404,0.853800,0.104029,2.70
0.007481,0.000000,0.281525,-1,0.314815,0.431500,0.900103,0.323036,0.130435,0.229008,0.893617,0.949518,0.360927,1.66
0.004448,0.000000,0.346041,-1,0.327160,0.540525,0.662204,0.218507,0.130435,0.223282,0.617021,0.995739,0.238135,2.31
0.001301,0.000000,0.089076,-1,0.123457,0.587086,0.565396,0.215115,0.043478,0.169847,0.574468,0.901861,0.135762,2.84
0.000870,0.450000,0.109238,-1,0.106996,0.694386,0.370752,0.312552,0.173913,0.402672,0.276596,1.000000,0.100993,3.49
0.001008,0.400000,0.218109,-1,0.127572,0.630964,0.410917,0.285326,0.130435,0.127863,0.531915,1.000000,0.034492,3.20
0.002303,0.125000,0.271628,-1,0.286008,0.396628,1.000000,0.450354,0.173913,0.236641,0.276596,0.974104,0.778146,1.65
0.000352,0.800000,0.038856,-1,0.039095,0.711439,0.364573,0.561922,0.043478,0.270992,0.000000,0.988149,0.134934,3.46
0.003340,0.000000,0.253666,-1,0.222222,0.527112,0.267765,0.389773,0.173913,0.190840,0.744681,1.000000,0.121965,2.30
0.000823,0.600000,0.045088,-1,0.053498,0.578272,0.339856,0.871218,0.130435,0.427481,0.606383,0.934137,0.103753,2.41
0.003291,0.200000,0.238270,-1,0.162551,0.439739,0.403708,0.300030,0.086957,0.068702,0.638298,0.979197,0.310982,2.11
0.002451,0.000000,0.338343,-1,0.411523,0.472504,0.790937,0.124453,0.217391,0.389313,0.702128,1.000000,0.347682,1.68
0.000531,0.210000,0.189883,-1,0.111111,0.565242,0.187436,0.516973,0.130435,0.106870,0.446809,1.000000,0.097958,2.50
0.000851,0.000000,0.493402,-1,0.106996,0.469055,0.405767,0.397667,0.130435,0.194656,0.361702,1.000000,0.239238,2.17
0.001217,0.200000,0.238270,-1,0.162551,0.570416,0.574665,0.253517,0.086957,0.068702,0.638298,0.995108,0.165563,2.44
0.104896,0.000000,0.646628,-1,0.604938,0.540142,0.954686,0.076258,1.000000,0.914122,0.808511,0.152302,0.616722,0.95
0.008951,0.000000,0.281525,-1,0.314815,0.363096,0.347065,0.242514,0.130435,0.229008,0.893617,0.727899,0.274834,2.02
0.000961,0.000000,0.131598,-1,0.257202,0.547040,0.836251,0.137921,0.173913,0.208015,0.425532,0.996470,0.201711,2.36
0.007310,0.200000,0.128666,-1,0.539095,0.628664,1.000000,0.080123,0.173913,0.146947,0.042553,0.987468,0.142660,3.01
0.000554,0.700000,0.065249,-1,0.030864,0.666794,0.073120,0.609099,0.173913,0.326336,0.234043,0.936154,0.083057,2.90
0.035487,0.000000,0.646628,-1,0.555556,0.421153,0.466529,0.176131,1.000000,0.914122,0.808511,0.842403,0.342163,1.99
0.001626,0.000000,0.350073,-1,0.333333,0.471355,0.820803,0.147114,0.217391,0.467557,0.553191,0.993973,0.236479,1.92

```
0.037257,0.000000,0.700880,1,1.000000,0.352941,1.000000,0.017459,0.173913,0.412214,0.223404,1.000000,0.692329,1.34
0.001249,0.125000,0.271628,-1,0.286008,0.469055,0.823893,0.463503,0.173913,0.236641,0.276596,1.000000,0.318433,1.89
0.110343,0.000000,0.646628,-1,0.588477,0.619467,0.987642,0.020769,1.000000,0.914122,0.808511,1.000000,0.538355,1.33
0.004512,0.000000,0.210411,1,0.251029,0.498755,0.910402,0.174449,0.304348,0.229008,0.510638,0.995814,0.544426,2.17
0.043199,0.000000,0.646628,1,0.792181,0.543016,0.907312,0.125090,1.000000,0.914122,0.808511,0.985980,0.318433,2.17
0.003183,0.000000,0.371334,-1,0.213992,0.354666,0.071061,0.223508,0.130435,0.171756,0.638298,0.879041,0.767660,2.37
0.012874,0.000000,0.281525,-1,0.314815,0.410040,0.948507,0.241668,0.130435,0.229008,0.893617,0.903853,0.458609,1.31
0.000272,0.550000,0.121701,-1,0.203704,0.634796,0.259526,0.485209,0.173913,0.349237,0.531915,0.977482,0.079470,3.12
0.001508,0.000000,0.371334,-1,0.213992,0.539184,0.302781,0.256054,0.130435,0.171756,0.638298,0.972036,0.211093,2.81
0.059422,0.000000,0.646628,-1,0.648148,0.477103,0.819773,0.094408,1.000000,0.914122,0.808511,0.953301,0.469923,2.32
0.089763,0.000000,0.646628,-1,0.648148,0.375359,1.000000,0.036692,1.000000,0.914122,0.808511,1.000000,0.629967,1.23
0.430994,0.000000,0.646628,-1,0.633745,0.362522,1.000000,0.032736,1.000000,0.914122,0.808511,1.000000,0.796358,0.50
0.001829,0.000000,0.923387,-1,0.403292,0.464648,0.880536,0.078504,0.043478,0.001908,0.691489,0.970044,0.360927,2.14
0.001607,0.250000,0.171188,-1,0.139918,0.417705,0.651905,0.554320,0.304348,0.185115,0.755319,0.995486,0.315121,1.87
0.000564,0.000000,0.453446,-1,0.106996,0.511784,0.523172,0.353236,0.173913,0.402672,0.648936,0.973524,0.292770,2.12
0.514104,0.000000,0.646628,-1,0.633745,0.183560,1.000000,0.048068,1.000000,0.914122,0.808511,0.221771,0.972682,0.70
0.031173,0.000000,0.700880,-1,1.000000,0.257137,0.977343,0.019669,0.173913,0.412214,0.223404,1.000000,0.760486,1.18
0.001106,0.400000,0.218109,1,0.127572,0.710098,0.474768,0.332603,0.130435,0.127863,0.531915,0.980710,0.119205,3.32
0.071786,0.000000,0.646628,-1,0.409465,0.498371,0.973223,0.097882,1.000000,0.914122,0.808511,0.762620,0.617274,1.33
0.000148,0.175000,0.033724,-1,0.063992,0.678866,0.582904,0.735962,0.086957,0.055344,0.638298,0.990771,0.174393,3.30
0.005780,0.000000,0.210411,1,0.251029,0.588235,0.757981,0.274477,0.304348,0.229008,0.510638,0.978693,0.215508,2.51
0.000307,0.750000,0.091276,-1,0.088477,0.663537,0.132853,0.388428,0.086957,0.124046,0.606383,0.996772,0.006898,3.49
0.000533,0.000000,0.493402,-1,0.106996,0.572523,0.495366,0.439287,0.130435,0.194656,0.361702,0.989788,0.156181,2.71
0.065389,0.000000,0.646628,-1,0.302469,0.513700,0.636457,0.208659,1.000000,0.914122,0.808511,1.000000,0.248620,2.30
0.000091,0.600000,0.090543,-1,0.032922,0.583062,0.163749,0.462858,0.000000,0.148855,0.319149,0.949065,0.073124,2.91
0.006838,0.000000,0.210411,-1,0.251029,0.585744,0.802266,0.194828,0.304348,0.229008,0.510638,1.000000,0.161976,3.01
0.045983,0.000000,0.700880,-1,1.000000,0.365396,1.000000,0.025662,0.173913,0.412214,0.223404,1.000000,0.681291,1.56
0.076372,0.000000,0.646628,-1,0.674897,0.482851,0.839341,0.144395,1.000000,0.914122,0.808511,1.000000,0.357892,2.00
0.008362,0.000000,0.281525,-1,0.314815,0.452769,0.939238,0.297357,0.130435,0.229008,0.893617,0.993520,0.402042,1.56
0.000733,0.000000,0.147727,-1,0.131687,0.490515,0.555098,0.238067,0.086957,0.114504,0.627660,0.995587,0.185155,2.22
0.057141,0.000000,0.646628,-1,0.674897,0.524238,0.915551,0.112632,1.000000,0.914122,0.808511,0.970220,0.428808,1.61
0.000150,0.800000,0.047654,-1,0.000000,0.511401,0.294542,0.723804,0.000000,0.103053,0.595745,0.860558,0.309051,2.01
```

Data Listing 5-2: Boston_test.txt (tabs replaced with commas to save space)

```
0.025133,0.000000,0.700880,-1,0.452675,0.439356,0.915551,0.117524,0.173913,0.412214,0.223404,0.995486,0.273455,2.27
0.105477,0.000000,0.646628,-1,0.730453,0.395861,0.937178,0.062527,1.000000,0.914122,0.808511,1.000000,0.583609,1.28
0.000948,0.200000,0.238270,1,0.162551,0.452002,0.603502,0.253517,0.086957,0.068702,0.638298,0.985980,0.328918,2.07
0.001554,0.125000,0.271628,-1,0.286008,0.500287,0.959835,0.438387,0.173913,0.236641,0.276596,1.000000,0.480684,2.71
0.006004,0.200000,0.128666,-1,0.390947,0.748994,0.511843,0.158445,0.173913,0.146947,0.042553,0.983358,0.039459,4.35
0.000926,0.210000,0.189883,-1,0.111111,0.460241,0.440783,0.516973,0.130435,0.106870,0.446809,0.996621,0.323400,1.97
0.006000,0.200000,0.128666,-1,0.539095,0.697835,0.812564,0.089343,0.173913,0.146947,0.042553,0.989662,0.216887,3.38
0.092102,0.000000,0.646628,-1,0.674897,0.455068,0.797116,0.150006,1.000000,0.914122,0.808511,0.008019,0.419702,1.35
0.001036,0.000000,0.923387,-1,0.403292,0.444146,0.956746,0.079722,0.043478,0.001908,0.691489,0.955822,0.437362,1.88
0.062659,0.000000,0.646628,-1,0.674897,0.550872,0.875386,0.107867,1.000000,0.914122,0.808511,0.251828,0.399834,1.43
0.095378,0.000000,0.646628,-1,0.409465,0.534010,0.856849,0.083942,1.000000,0.914122,0.808511,0.209617,0.439018,1.45
0.005682,0.000000,0.210411,-1,0.251029,0.727534,0.707518,0.274477,0.304348,0.229008,0.510638,0.982778,0.082781,3.15
0.150090,0.000000,0.646628,-1,0.633745,0.445679,0.945417,0.059335,1.000000,0.914122,0.808511,1.000000,0.403422,1.27
0.225678,0.000000,0.646628,-1,0.648148,0.154627,0.909372,0.028181,1.000000,0.914122,0.808511,0.719930,0.797461,0.88
0.097903,0.000000,0.646628,-1,0.633745,0.557578,0.987642,0.054206,1.000000,0.914122,0.808511,0.987594,0.424669,1.31
0.053344,0.000000,0.646628,-1,0.674897,0.567925,0.860968,0.118779,1.000000,0.914122,0.808511,0.127591,0.452539,1.41
0.001528,0.000000,0.350073,-1,0.333333,0.515999,0.837281,0.102474,0.217391,0.467557,0.553191,0.979424,0.240618,1.85
0.002079,0.220000,0.197947,-1,0.094650,0.391646,0.693100,0.620657,0.260870,0.272901,0.691489,0.980407,0.461645,1.85
0.001420,0.000000,0.296921,-1,0.277778,0.438781,0.966014,0.088870,0.173913,0.375954,0.882979,0.992814,0.406733,1.95
0.000412,0.800000,0.053152,-1,0.057613,0.402759,0.195675,0.859888,0.130435,0.280534,1.000000,0.964446,0.174393,1.82
0.001890,0.000000,0.371334,1,0.213992,0.459667,0.918641,0.249843,0.130435,0.171756,0.638298,0.990796,0.428808,2.17
0.247778,0.000000,0.646628,-1,0.730453,0.432458,0.921730,0.066983,1.000000,0.914122,0.808511,0.986258,0.562362,1.05
0.002737,0.000000,0.785557,-1,0.491770,0.439931,0.981462,0.049014,0.130435,0.477099,0.914894,0.987745,0.540563,1.33
0.011039,0.000000,0.281525,-1,0.314815,0.431500,1.000000,0.269676,0.130435,0.229008,0.893617,0.994049,0.500828,1.45
0.322013,0.000000,0.646628,-1,0.436214,0.305422,1.000000,0.041812,1.000000,0.914122,0.808511,0.531166,0.506347,1.63
0.063658,0.000000,0.646628,1,0.506173,0.598199,0.967044,0.020651,1.000000,0.914122,0.808511,0.945610,0.055188,5.00
0.265729,0.000000,0.646628,-1,0.588477,0.540142,0.960865,0.023325,1.000000,0.914122,0.808511,1.000000,0.605960,1.31
0.000705,0.000000,0.063050,-1,0.150206,0.687105,0.528321,0.448545,0.086957,0.066794,0.648936,1.000000,0.099338,3.62
0.065357,0.000000,0.646628,-1,0.674897,0.565626,0.895984,0.152043,1.000000,0.914122,0.808511,0.992234,0.236203,2.02
0.000703,0.000000,0.073314,-1,0.211934,0.494922,0.610711,0.133519,0.086957,0.011450,0.553191,1.000000,0.213024,3.62
0.000849,0.000000,0.089076,-1,0.123457,0.816057,0.350154,0.215115,0.043478,0.169847,0.574468,0.991502,0.050773,4.38
0.000445,0.525000,0.178152,-1,0.041152,0.527687,0.439753,0.562668,0.217391,0.202290,0.425532,1.000000,0.161976,2.23
0.027429,0.000000,0.700880,-1,1.000000,0.327841,0.938208,0.055179,0.173913,0.412214,0.223404,0.222679,0.397627,1.31
0.003856,0.000000,0.346041,-1,0.327160,0.461966,0.760041,0.179405,0.130435,0.223282,0.617021,0.998336,0.227373,2.03
0.003587,0.000000,0.785557,-1,0.491770,0.549914,0.987642,0.062099,0.130435,0.477099,0.914894,1.000000,0.376932,1.80
0.111389,0.000000,0.646628,-1,0.633745,0.438973,0.771370,0.033719,1.000000,0.914122,0.808511,0.851884,0.779249,0.63
0.000579,0.000000,0.073314,-1,0.211934,0.655106,0.571576,0.154534,0.086957,0.011450,0.553191,1.000000,0.091336,3.72
0.001408,0.000000,0.350073,-1,0.333333,0.501054,0.716787,0.145541,0.217391,0.467557,0.553191,0.990922,0.284492,2.12
0.000539,0.000000,0.131598,-1,0.257202,0.527687,0.726056,0.198956,0.173913,0.208015,0.425532,0.996722,0.125828,2.46
0.017002,0.000000,0.700880,1,0.452675,0.922399,0.937178,0.093881,0.173913,0.412214,0.223404,0.978693,0.043874,5.00
0.097094,0.000000,0.646628,-1,0.633745,0.504311,0.923790,0.060162,1.000000,0.914122,0.808511,1.000000,0.370861,1.38
0.002892,0.000000,0.296921,-1,0.277778,0.511209,0.909372,0.128718,0.173913,0.375954,0.882979,0.985703,0.381347,1.94
0.008760,0.200000,0.128666,-1,0.539095,0.661621,0.841401,0.091235,0.173913,0.146947,0.042553,0.967648,0.360375,3.07
0.003555,0.000000,0.785557,-1,0.491770,0.456218,0.933059,0.076140,0.130435,0.477099,0.914894,0.952973,0.418598,1.74
0.020164,0.000000,0.700880,-1,0.452675,0.443763,0.785788,0.117879,0.173913,0.412214,0.223404,0.573125,0.287252,2.38
0.001456,0.000000,0.371334,1,0.213992,0.479594,0.578785,0.282771,0.130435,0.171756,0.638298,0.960714,0.356788,2.44
0.211360,0.000000,0.646628,-1,0.436214,0.204445,1.000000,0.038584,1.000000,0.914122,0.808511,0.071789,0.900662,1.79
0.000483,0.330000,0.063050,-1,0.179012,0.630006,0.694130,0.186698,0.260870,0.066794,0.617021,1.000000,0.160044,2.82
0.001222,0.000000,0.089076,-1,0.123457,0.498563,0.686921,0.215115,0.043478,0.169847,0.574468,0.987216,0.265177,2.14
0.054281,0.000000,0.646628,-1,0.407407,0.449128,0.518023,0.183934,1.000000,0.914122,0.808511,0.978113,0.268212,2.06
```

```
0.001600,0.000000,0.296921,-1,0.277778,0.606630,0.792997,0.149879,0.173913,0.375954,0.882979,0.994604,0.212196,2.75
0.001202,0.300000,0.163856,-1,0.088477,0.639203,0.529351,0.473452,0.217391,0.215649,0.425532,0.985753,0.266280,2.20
0.001368,0.000000,0.296921,-1,0.277778,0.558153,0.970134,0.118515,0.173913,0.375954,0.882979,0.995814,0.290839,1.98
0.000264,0.000000,0.063050,-1,0.150206,0.549722,0.574665,0.448545,0.086957,0.066794,0.648936,0.992990,0.096026,2.87
0.004564,0.000000,0.210411,-1,0.244856,0.690171,0.792997,0.189699,0.304348,0.229008,0.510638,0.937415,0.127759,3.16
0.026129,0.000000,0.700880,-1,1.000000,0.311362,0.936148,0.036374,0.173913,0.412214,0.223404,0.899365,0.733720,1.78
0.000959,0.000000,0.379399,-1,0.057613,0.479785,0.050463,0.378079,0.130435,0.225191,0.702128,0.984896,0.104581,2.28
0.000331,0.200000,0.105205,-1,0.119136,0.816057,0.634398,0.324191,0.173913,0.055344,0.244681,0.975818,0.056015,4.54
0.001853,0.000000,0.296921,-1,0.277778,0.435907,0.916581,0.098337,0.173913,0.375954,0.882979,0.996898,0.467163,1.95
0.001024,0.000000,0.201613,-1,0.234568,0.436865,0.602472,0.204449,0.173913,0.175573,0.702128,0.951233,0.267108,2.00
0.000678,0.000000,0.131598,-1,0.257202,0.571949,0.311020,0.182115,0.173913,0.208015,0.425532,0.985022,0.099338,2.94
0.000215,0.550000,0.121701,-1,0.203704,0.600690,0.550978,0.418527,0.173913,0.349237,0.531915,1.000000,0.150386,2.39
0.001720,0.000000,0.236437,-1,0.129630,0.507760,0.037075,0.417509,0.086957,0.087786,0.563830,0.993847,0.157561,2.47
0.007099,0.000000,0.281525,-1,0.314815,0.485725,0.840371,0.303022,0.130435,0.229008,0.893617,0.957436,0.235375,1.82
0.000075,0.350000,0.038856,-1,0.117284,0.705116,0.477858,0.537270,0.000000,0.185115,0.308511,0.994553,0.103753,3.27
0.000830,0.000000,0.201613,-1,0.234568,0.438590,0.397528,0.255036,0.173913,0.175573,0.702128,1.000000,0.194260,2.10
0.013682,0.000000,0.700880,-1,0.452675,0.648017,0.973223,0.067992,0.173913,0.412214,0.223404,0.915603,0.078918,4.13
0.012639,0.000000,0.281525,-1,0.314815,0.412340,0.939238,0.282207,0.130435,0.229008,0.893617,0.907383,0.575883,1.27
0.001506,0.220000,0.197947,-1,0.094650,0.560644,0.104016,0.569897,0.260870,0.272901,0.691489,0.998437,0.115066,2.44
0.000462,0.000000,0.420455,-1,0.386831,0.473079,0.802266,0.125072,0.000000,0.164122,0.893617,1.000000,0.169702,1.19
0.109226,0.000000,0.646628,-1,0.730453,0.545124,0.971164,0.085069,1.000000,0.914122,0.808511,0.972414,0.490894,1.71
0.052159,0.000000,0.646628,-1,0.471193,0.655106,0.666323,0.127609,1.000000,0.914122,0.808511,0.943971,0.274007,2.98
0.000319,0.350000,0.205279,-1,0.108848,0.473271,0.210093,0.501150,0.000000,0.223282,0.457447,0.912628,0.168322,1.94
0.008087,0.000000,0.281525,-1,0.314815,0.415022,0.685891,0.242514,0.130435,0.229008,0.893617,0.984997,0.263521,1.82
0.002085,0.000000,0.253666,-1,0.222222,0.549914,0.121524,0.389773,0.173913,0.190840,0.744681,0.991881,0.092439,2.46
0.015156,0.000000,0.281525,-1,0.314815,0.481127,1.000000,0.276933,0.130435,0.229008,0.893617,0.949140,0.312086,1.45
0.253915,0.000000,0.646628,-1,0.648148,0.275723,0.891864,0.035355,1.000000,0.914122,0.808511,1.000000,0.834989,0.74
0.002617,0.000000,0.338343,-1,0.411523,0.470971,0.642636,0.116351,0.217391,0.389313,0.702128,1.000000,0.308775,2.12
0.008489,0.200000,0.128666,-1,0.539095,0.383024,0.616890,0.077922,0.173913,0.146947,0.042553,0.988653,0.240618,2.28
0.002138,0.220000,0.197947,-1,0.094650,0.510634,0.785788,0.629805,0.260870,0.272901,0.691489,0.947652,0.232340,2.05
0.000441,0.000000,0.492302,1,0.339506,0.445871,0.546859,0.180278,0.173913,0.169847,0.404255,0.989662,0.325055,2.33
0.000252,0.280000,0.534457,-1,0.162551,0.507760,0.267765,0.230638,0.130435,0.158397,0.595745,0.998563,0.123620,2.50
0.010672,0.000000,0.281525,-1,0.314815,0.476336,0.884655,0.302249,0.130435,0.229008,0.893617,0.771748,0.429084,1.48
0.010901,0.000000,0.785557,-1,0.491770,0.420770,0.983522,0.110613,0.130435,0.477099,0.914894,0.661758,0.429912,1.56
0.281441,0.000000,0.646628,-1,0.633745,0.464840,1.000000,0.041757,1.000000,0.914122,0.808511,1.000000,0.690949,0.56
0.000371,0.000000,0.108138,-1,0.213992,0.544932,0.731205,0.178459,0.043478,0.158397,0.553191,0.991553,0.178532,2.20
0.001777,0.220000,0.197947,-1,0.094650,0.550297,0.475798,0.608981,0.260870,0.272901,0.691489,0.944047,0.214956,2.45
0.063618,0.000000,0.646628,-1,0.730453,0.509293,1.000000,0.079586,1.000000,0.914122,0.808511,0.996949,0.410044,1.84
0.000952,0.000000,0.073314,-1,0.211934,0.688638,0.919670,0.142858,0.086957,0.011450,0.553191,0.992990,0.085265,3.79
0.000082,0.750000,0.129765,-1,0.051440,0.445871,0.460350,0.562895,0.086957,0.538168,0.904255,1.000000,0.360651,1.89
0.000401,0.280000,0.534457,-1,0.162551,0.552021,0.522142,0.230638,0.130435,0.158397,0.595745,0.995234,0.177428,2.29
0.042359,0.000000,0.646628,-1,0.555556,0.458134,0.842430,0.158399,1.000000,0.914122,0.808511,0.054693,0.425497,1.90
0.000734,0.000000,0.923387,-1,0.403292,0.468097,0.836251,0.097100,0.043478,0.001908,0.691489,0.951510,0.346026,2.03
0.001070,0.000000,0.453446,-1,0.106996,0.520789,0.737384,0.265766,0.173913,0.402672,0.648936,0.941399,0.282561,2.00
0.156312,0.000000,0.646628,-1,0.674897,0.507185,0.948507,0.099355,1.000000,0.914122,0.808511,0.252938,0.370861,1.17
0.013337,0.000000,0.785557,-1,0.491770,0.529795,0.976313,0.103793,0.130435,0.477099,0.914894,1.000000,0.290563,1.96
0.002506,0.000000,0.236437,-1,0.129630,0.473079,0.850669,0.414644,0.086957,0.087786,0.563830,0.989510,0.471026,1.66
0.006395,0.000000,0.210411,-1,0.251029,0.915118,0.725026,0.246324,0.304348,0.229008,0.510638,0.972288,0.020419,4.17
0.000494,0.350000,0.205279,-1,0.108848,0.410998,0.262616,0.501150,0.000000,0.223282,0.457447,0.992738,0.295254,1.71
0.000471,0.800000,0.116569,-1,0.014403,0.488025,0.299691,0.735726,0.000000,0.244275,0.404255,0.989889,0.133554,2.19
```

Data Listing 6-1: Cleveland_train.txt (tabs replaced with commas to save space)

```
0.5208,1,-1,-1,-1,0.1860,0.2580,-1,1,0,0.4198,1,0.4516,0,1,0.3333,-1,-1,1
0.6458,-1,0,0,1,0.3023,0.1187,1,1,0,0.1908,-1,0.0000,1,0,0.0000,1,0,0
0.3958,1,0,1,0,0.4186,0.2717,-1,-1,-1,0.8321,-1,0.0323,0,1,0.0000,1,0,0
0.6875,-1,-1,-1,-1,0.5349,0.6119,-1,-1,-1,0.6565,-1,0.1935,0,1,0.0000,1,0,0
0.8750,-1,0,1,0,0.7674,0.4018,-1,1,0,0.6947,-1,0.0645,1,0,0.6667,1,0,0
0.4167,-1,0,1,0,0.4651,0.3311,-1,1,0,0.6947,-1,0.0000,0,1,0.0000,1,0,0
0.4167,-1,-1,-1,-1,0.4186,0.3265,-1,1,0,0.7023,-1,0.0000,1,0,0.0000,1,0,0
0.6875,-1,-1,-1,-1,0.5116,0.3836,1,1,0,0.2672,-1,0.3065,0,1,1.0000,1,0,1
0.5000,1,-1,-1,-1,0.5581,0.2283,-1,-1,-1,0.3053,1,0.0000,1,0,0.0000,-1,-1,0
0.6875,1,0,1,0,0.3953,0.1872,1,-1,-1,0.5267,-1,0.0000,1,0,0.0000,1,0,0
0.4167,1,0,1,0,0.4186,0.3196,-1,1,0,0.7634,-1,0.0968,1,0,0.0000,1,0,0
0.5833,1,0,0,1,0.6512,0.0000,1,1,0,0.7786,-1,0.0323,1,0,0.3333,-1,-1,0
0.6042,1,0,0,1,0.5349,0.1941,1,-1,-1,0.7176,-1,0.0000,1,0,0.0000,1,0,0
0.9792,-1,0,0,1,0.5349,0.1621,-1,0,1,0.3435,-1,0.1774,0,1,0.0000,1,0,0
0.2500,1,-1,-1,-1,0.1860,0.1050,-1,-1,-1,0.6641,-1,0.0000,1,0,0.0000,-1,-1,1
0.5417,1,-1,-1,-1,0.7674,0.3721,-1,-1,-1,0.5649,1,0.1290,0,1,0.3333,-1,-1,1
0.2917,1,-1,-1,-1,0.2442,0.4041,-1,1,0,0.8397,-1,0.1935,0,1,0.0000,1,0,0
0.9375,-1,0,1,0,0.3023,0.3265,-1,-1,-1,0.3817,1,0.0323,1,0,0.3333,1,0,0
0.4375,1,0,0,1,0.5349,0.2443,-1,1,0,0.7023,-1,0.0968,0,1,0.3333,-1,-1,1
0.2708,-1,-1,-1,-1,0.0930,0.3174,-1,-1,-1,0.3893,-1,0.0968,0,1,0.0000,1,0,0
0.2917,-1,0,0,1,0.3256,0.1986,-1,1,0,0.7176,-1,0.0323,0,1,0.0000,1,0,0
```

```
0.3333,-1,0,1,0,0.2093,0.0776,-1,1,0,0.5115,-1,0.0000,0,1,0.0000,1,0,0
0.0000,1,0,1,0,0.4186,0.1781,-1,-1,-1,1.0000,-1,0.0000,1,0,0.0000,1,0,0
0.7083,-1,-1,-1,-1,0.1628,0.3265,-1,1,0,0.7481,1,0.2903,0,1,0.6667,1,0,1
0.3542,-1,-1,-1,-1,0.5116,0.2671,-1,-1,-1,0.6183,1,0.0000,0,1,0.0000,1,0,0
0.1875,1,1,0,0,0.3023,0.2397,-1,1,0,0.8473,1,0.6129,0,1,0.0000,-1,-1,1
0.3125,1,0,0,1,0.4186,0.2443,-1,1,0,0.8244,1,0.0645,1,0,0.0000,1,0,0
0.6042,1,0,0,1,0.2093,0.2374,-1,-1,-1,0.7176,-1,0.4032,0,1,0.3333,-1,-1,1
0.3542,-1,0,0,1,0.5581,0.1164,-1,-1,-1,0.6794,1,0.2258,-1,-1,0.0000,1,0,0
0.5625,1,1,0,0,0.3023,0.1530,-1,-1,-1,0.6947,-1,0.3065,0,1,0.0000,-1,-1,0
0.6042,1,-1,-1,-1,0.2326,0.4384,-1,0,1,0.5267,-1,0.7097,-1,-1,1.0000,0,1,1
0.4792,1,0,1,0,0.3023,0.4543,-1,1,0,0.7710,-1,0.0323,1,0,0.0000,1,0,0
0.3333,-1,-1,-1,-1,0.5116,0.2511,-1,-1,-1,0.6183,1,0.0323,0,1,0.0000,1,0,0
0.6875,1,0,1,0,0.3023,0.3539,-1,-1,-1,0.2443,-1,0.2258,0,1,0.3333,-1,-1,1
0.4375,-1,0,0,1,0.3023,0.2123,-1,1,0,0.6641,-1,0.2581,0,1,0.0000,1,0,0
0.4375,-1,-1,-1,-1,0.1860,0.2922,-1,-1,-1,0.6718,-1,0.0000,1,0,0.0000,1,0,0
0.5833,-1,-1,-1,-1,0.3953,0.4041,-1,-1,-1,0.6718,-1,0.0000,1,0,0.3333,1,0,0
0.2292,1,-1,-1,-1,0.6744,0.2215,-1,1,0,0.8397,-1,0.0000,1,0,0.0000,-1,-1,1
0.3750,1,0,0,1,0.4186,0.2900,-1,1,0,0.8244,-1,0.0000,1,0,0.0000,1,0,0
0.5833,1,-1,-1,-1,0.1860,0.4772,-1,1,0,0.5496,1,0.4839,0,1,0.3333,-1,-1,1
0.5625,-1,0,1,0,0.5349,0.3836,-1,-1,-1,0.6260,-1,0.2097,0,1,0.0000,1,0,0
0.5208,1,-1,-1,-1,0.3256,0.3653,-1,-1,-1,0.3435,1,0.5161,0,1,0.6667,1,0,1
0.5208,1,0,0,1,0.3023,0.3014,-1,-1,-1,0.5802,-1,0.0645,0,1,0.0000,-1,-1,0
0.5833,-1,-1,-1,-1,0.3023,0.5205,-1,1,0,0.7023,1,0.0968,1,0,0.0000,1,0,0
0.8125,1,-1,-1,-1,0.5814,0.1530,1,1,0,0.5344,-1,0.5484,0,1,0.6667,-1,-1,1
0.7292,-1,0,0,1,0.5349,0.4269,-1,1,0,0.4733,-1,0.0323,1,0,0.0000,-1,-1,0
0.6042,1,-1,-1,-1,0.6512,0.3288,-1,-1,-1,0.3053,1,0.1290,1,0,0.0000,-1,-1,1
0.6250,1,-1,-1,-1,0.8837,0.4566,-1,-1,-1,0.5267,1,0.5484,-1,-1,0.0000,-1,-1,1
0.2917,1,-1,-1,-1,0.3023,0.1164,-1,-1,-1,0.3740,1,0.4032,0,1,0.0000,-1,-1,1
0.7917,1,-1,-1,-1,0.3023,0.2534,-1,1,0,0.0000,-1,0.1613,0,1,0.0000,1,0,1
0.6250,1,1,0,0,0.7674,0.3356,-1,-1,-1,0.4122,-1,0.0000,1,0,0.0000,1,0,1
0.5625,1,-1,-1,-1,0.4186,0.3584,1,-1,-1,0.2443,1,0.2581,-1,-1,0.0000,-1,-1,1
0.5417,1,0,1,0,0.4186,0.3105,-1,1,0,0.6412,-1,0.0000,1,0,0.0000,1,0,0
0.7083,-1,0,1,0,0.5349,0.1575,-1,1,0,0.8244,-1,0.0000,1,0,0.6667,1,0,0
0.7292,1,0,0,1,0.5349,0.4772,-1,1,0,0.6641,-1,0.0000,1,0,0.0000,1,0,1
0.4167,1,0,0,1,0.2791,0.0525,-1,-1,-1,0.4198,-1,0.1290,1,0,1.0000,1,0,1
0.4583,1,-1,-1,-1,0.5349,0.3927,-1,1,0,0.3893,1,0.6774,0,1,1.0000,-1,-1,1
0.5833,-1,-1,-1,-1,0.5349,0.2626,-1,1,0,0.3969,1,0.0323,0,1,0.0000,-1,-1,1
0.5208,-1,0,0,1,0.1628,0.3219,-1,-1,-1,0.7328,-1,0.0000,1,0,0.0000,1,0,0
0.2917,1,-1,-1,-1,0.6512,0.2763,-1,1,0,0.7634,-1,0.2419,1,0,0.0000,1,0,0
0.3958,-1,0,0,1,0.4186,0.3402,-1,1,0,0.5191,-1,0.0323,1,0,0.0000,1,0,0
0.6875,-1,0,0,1,0.4186,0.3128,-1,1,0,0.1985,-1,0.1935,0,1,0.3333,-1,-1,1
0.2083,-1,0,0,1,0.0000,0.1667,-1,1,0,0.8244,-1,0.0000,1,0,0.0000,1,0,0
0.4167,1,0,0,1,0.3023,0.1416,-1,1,0,0.5191,-1,0.3226,0,1,1.0000,-1,-1,1
0.5208,-1,0,0,1,0.7674,0.1712,-1,1,0,0.7023,-1,0.0000,1,0,0.3333,1,0,0
0.3125,1,-1,-1,-1,0.3023,0.0982,-1,1,0,0.5573,1,0.4516,-1,-1,0.0000,0,1,1
0.5417,1,-1,-1,-1,0.5349,0.2078,-1,1,0,0.3053,1,0.9032,-1,-1,0.0000,-1,-1,1
0.4792,1,1,0,0,0.2791,0.1370,-1,-1,-1,0.9084,-1,0.0000,0,1,0.0000,0,1,0
0.6250,-1,-1,-1,-1,0.9302,0.2808,-1,1,0,0.5496,1,0.0000,0,1,0.0000,1,0,1
0.3333,1,-1,-1,-1,0.5581,0.4178,-1,-1,-1,0.5802,1,0.0000,0,1,1.0000,-1,-1,1
0.4583,1,0,0,1,0.0000,0.2306,-1,1,0,0.6336,1,0.0000,1,0,0.3333,-1,-1,0
0.5417,-1,-1,-1,-1,0.3953,0.1804,-1,0,1,0.4504,1,0.3226,0,1,0.3333,-1,-1,1
0.5833,1,-1,-1,-1,0.5349,0.1507,-1,1,0,0.5878,-1,0.0645,0,1,0.0000,0,1,0
0.6250,1,0,1,0,0.5349,0.2169,-1,1,0,0.7099,1,0.0000,1,0,0.0000,1,0,0
0.5208,1,0,1,0,0.1628,0.4178,-1,1,0,0.6489,-1,0.0000,1,0,0.0000,-1,-1,0
0.5833,1,-1,-1,-1,0.8256,0.3721,1,-1,-1,0.4046,-1,0.1613,0,1,1.0000,-1,-1,1
0.2500,1,0,1,0,0.4767,0.1758,-1,1,0,0.4656,-1,0.0000,0,1,0.0000,0,1,0
0.2917,-1,-1,-1,-1,0.4419,0.4909,1,-1,-1,0.4962,1,0.4839,0,1,0.0000,-1,-1,1
0.5625,1,-1,-1,-1,0.4419,0.1324,-1,-1,-1,0.2595,1,0.3387,0,1,0.3333,0,1,1
0.3125,1,0,1,0,0.4186,0.2123,-1,-1,-1,0.8931,-1,0.0000,1,0,0.0000,1,0,0
0.6250,1,-1,-1,-1,0.5116,0.3311,-1,-1,-1,0.8473,-1,0.0000,1,0,0.0000,1,0,0
0.2500,1,0,1,0,0.3023,0.0708,-1,1,0,0.8473,-1,0.0000,1,0,0.0000,1,0,0
0.8333,1,0,0,1,0.5349,0.2922,-1,-1,-1,0.5725,-1,0.3226,0,1,1.0000,-1,-1,1
```

```
0.6667,1,-1,-1,-1,0.6279,0.1758,-1,1,0,0.6870,-1,0.0000,1,0,0.3333,-1,-1,1
0.8125,1,0,0,1,0.2791,0.3447,-1,1,0,0.6107,-1,0.1613,1,0,0.3333,-1,-1,0
0.6458,1,0,0,1,0.5349,0.1347,-1,-1,-1,0.6412,-1,0.4839,0,1,0.0000,1,0,1
0.6458,1,-1,-1,-1,0.3605,0.3014,-1,-1,-1,0.5344,1,0.4516,0,1,0.3333,-1,-1,1
0.6250,1,0,0,1,0.6512,0.1963,1,1,0,0.6565,-1,0.2581,1,0,0.0000,1,0,0
0.2500,-1,0,1,0,0.4186,0.1781,-1,-1,-1,0.7710,-1,0.2258,1,0,0.0000,1,0,0
0.5417,-1,0,1,0,0.4419,0.4932,-1,1,0,0.7252,-1,0.1935,1,0,0.0000,1,0,0
0.6042,-1,-1,-1,-1,0.8837,0.2260,1,-1,-1,0.5725,1,0.4516,0,1,0.6667,0,1,1
0.2500,-1,0,0,1,0.2093,0.3242,-1,-1,-1,0.7710,1,0.0000,1,0,0.0000,1,0,0
0.4792,1,1,0,0,0.6744,0.3927,1,1,0,0.8168,-1,0.1935,0,1,0.0000,-1,-1,0
0.6458,-1,-1,-1,-1,0.7442,0.4087,-1,-1,-1,0.6870,-1,0.0000,1,0,0.0000,1,0,1
0.3958,1,-1,-1,-1,0.3488,0.3379,-1,-1,-1,0.7252,-1,0.0806,0,1,0.0000,-1,-1,1
0.2708,1,-1,-1,-1,0.5349,0.2283,-1,1,0,0.8168,-1,0.0000,1,0,0.0000,1,0,0
0.8333,-1,1,0,0,0.5349,0.2580,-1,1,0,0.6107,-1,0.2903,1,0,0.6667,1,0,0
0.4583,-1,0,0,1,0.5349,0.4155,-1,-1,-1,0.5420,-1,0.2419,1,0,0.3333,1,0,0
0.4375,1,0,0,1,0.4070,0.1598,-1,1,0,0.7023,-1,0.0000,1,0,0.0000,1,0,0
0.3958,1,0,0,1,0.3488,0.2945,1,1,0,0.7939,-1,0.0000,1,0,0.6667,1,0,0
0.4583,-1,-1,-1,-1,0.4186,0.4087,-1,1,0,0.5420,1,0.1935,0,1,0.0000,-1,-1,1
0.8750,-1,-1,-1,-1,0.2093,0.0525,-1,1,0,0.4122,-1,0.2581,0,1,0.0000,1,0,0
1.0000,1,-1,-1,-1,0.3605,0.4064,-1,-1,-1,0.6947,1,0.0000,1,0,1.0000,1,0,1
0.7292,1,1,0,0,0.8837,0.2306,-1,-1,-1,0.6412,-1,0.0968,0,1,0.0000,-1,-1,0
0.7500,-1,0,0,1,0.5349,0.6644,1,-1,-1,0.6565,-1,0.1290,1,0,0.3333,1,0,0
0.6250,1,1,0,0,0.4651,0.1781,-1,1,0,0.6947,-1,0.1290,1,0,0.6667,1,0,1
0.5000,1,0,0,1,0.4186,0.1621,1,-1,-1,0.6183,-1,0.1935,-1,-1,0.0000,1,0,0
0.7292,1,1,0,0,0.1860,0.1941,-1,-1,-1,0.5573,1,0.2903,0,1,0.0000,1,0,0
0.6458,1,-1,-1,-1,0.5349,0.3813,-1,-1,-1,0.7557,-1,0.1935,0,1,0.6667,-1,-1,1
0.6875,-1,-1,-1,-1,0.3488,0.1895,-1,1,0,0.7023,-1,0.0000,1,0,0.0000,1,0,0
0.5833,1,-1,-1,-1,0.4419,0.1849,-1,1,0,0.7405,1,0.0000,1,0,0.0000,-1,-1,0
0.8750,-1,0,0,1,0.1860,0.3174,1,-1,-1,0.4504,-1,0.0000,1,0,0.3333,1,0,0
0.6667,1,1,0,0,0.4651,0.2466,-1,1,0,0.5649,-1,0.4194,0,1,0.6667,1,0,1
0.5208,1,-1,-1,-1,0.5349,0.2580,-1,1,0,0.6794,-1,0.1935,1,0,0.0000,1,0,0
0.5833,1,-1,-1,-1,0.4186,0.0114,-1,1,0,0.3359,1,0.1935,0,1,0.3333,-1,-1,1
0.4792,1,0,1,0,0.3953,0.1804,1,1,0,0.8626,-1,0.0000,1,0,0.0000,1,0,0
0.3125,1,0,1,0,0.3023,0.2146,-1,1,0,0.7557,-1,0.0000,1,0,0.0000,1,0,0
0.6250,1,-1,-1,-1,0.4767,0.2466,-1,1,0,0.6870,-1,0.0806,0,1,0.0000,-1,-1,0
0.5417,-1,-1,-1,-1,1.0000,0.4589,-1,0,1,0.3511,1,0.5484,0,1,0.0000,1,0,1
0.3542,1,0,0,1,0.6512,0.2397,-1,1,0,0.5802,-1,0.5806,0,1,0.0000,1,0,1
0.2083,-1,0,0,1,0.5116,0.2146,-1,1,0,0.6183,-1,0.0000,0,1,0.0000,1,0,0
0.3125,1,-1,-1,-1,0.2093,0.3744,-1,-1,-1,0.6260,-1,0.0000,1,0,0.3333,1,0,1
0.4583,1,1,0,0,0.3605,0.1986,-1,-1,-1,0.4122,1,0.2258,1,0,0.3333,1,0,0
0.6042,-1,0,1,0,0.4884,0.4406,1,-1,-1,0.6183,-1,0.0000,1,0,0.6667,1,0,1
0.4583,1,-1,-1,-1,0.5349,0.3950,-1,1,0,0.7786,1,0.2581,1,0,0.0000,-1,-1,1
0.6042,1,0,0,1,0.4419,0.2237,-1,-1,-1,0.7786,-1,0.5161,1,0,0.6667,-1,-1,1
0.4583,-1,0,0,1,0.4186,0.2968,-1,-1,-1,0.5954,-1,0.0806,1,0,0.0000,1,0,0
0.7917,-1,-1,-1,-1,0.1395,0.2215,-1,1,0,0.5420,-1,0.0484,1,0,0.6667,1,0,0
0.3125,1,0,1,0,0.3023,0.3128,-1,1,0,0.7786,-1,0.0000,1,0,0.0000,-1,-1,0
0.6875,1,0,0,1,0.4186,0.2397,-1,1,0,0.5725,-1,0.2903,0,1,1.0000,-1,-1,0
0.6667,1,-1,-1,-1,0.5116,0.0913,-1,-1,-1,0.4122,1,0.5806,0,1,0.3333,1,0,1
0.8333,1,1,0,0,0.7674,0.2466,1,-1,-1,0.4580,-1,0.0161,0,1,0.3333,1,0,0
0.6875,-1,-1,-1,-1,0.5349,0.3242,-1,-1,-1,0.6794,-1,0.5806,-1,-1,0.6667,1,0,1
0.5208,1,-1,-1,-1,0.3023,0.1416,-1,1,0,0.3206,-1,0.2258,0,1,0.3333,-1,-1,1
0.3333,1,0,1,0,0.3953,0.4155,-1,-1,-1,0.7557,-1,0.0000,1,0,0.0000,1,0,0
0.8542,1,-1,-1,-1,0.5930,0.1096,-1,1,0,0.4122,1,0.4194,-1,-1,0.0000,-1,-1,1
0.4792,1,-1,-1,-1,0.3605,0.1963,-1,1,0,0.7405,-1,0.1613,1,0,0.6667,-1,-1,1
0.5000,-1,-1,-1,-1,0.4186,0.3151,-1,-1,-1,0.5496,-1,0.0645,0,1,0.0000,1,0,0
0.1250,1,-1,-1,-1,0.3023,0.1644,-1,1,0,0.4504,1,0.2581,0,1,0.0000,-1,-1,1
0.3333,1,-1,-1,-1,0.2442,0.3059,-1,-1,-1,0.8702,-1,0.0000,1,0,0.0000,1,0,0
0.7917,1,-1,-1,-1,0.0698,0.3950,-1,-1,-1,0.4122,1,0.1452,0,1,0.6667,1,0,1
0.3333,1,1,0,0,0.1860,0.3151,-1,1,0,0.4656,-1,0.1935,0,1,0.0000,-1,-1,1
0.2708,1,-1,-1,-1,0.4884,0.4315,-1,1,0,0.4122,1,0.2903,0,1,0.0000,0,1,1
0.7083,-1,0,0,1,0.4767,0.2877,-1,-1,-1,0.7710,-1,0.0000,1,0,0.0000,1,0,0
0.6042,1,0,0,1,0.1279,0.2603,-1,-1,-1,0.6336,1,0.0968,0,1,0.0000,-1,-1,0
```

```
0.5833,1,0,1,0,0.6977,0.2420,-1,-1,-1,0.7099,-1,0.0000,1,0,0.3333,1,0,1
0.5417,1,-1,-1,-1,0.4419,0.5183,-1,1,0,0.4656,1,0.1935,0,1,0.3333,-1,-1,1
0.7500,-1,0,0,1,0.7093,0.3265,-1,1,0,0.5878,-1,0.1290,1,0,0.0000,1,0,0
0.7708,1,-1,-1,-1,0.3023,0.4018,-1,-1,-1,0.6107,-1,0.0645,0,1,0.0000,1,0,0
0.7708,1,-1,-1,-1,0.7674,0.2329,-1,-1,-1,0.5115,-1,0.3710,1,0,0.0000,0,1,0
0.2708,1,0,0,1,0.3023,0.2603,1,1,0,0.9389,-1,0.1290,-1,-1,0.0000,-1,-1,0
0.7917,-1,0,0,1,0.6744,0.3447,-1,1,0,0.7710,-1,0.0000,1,0,0.3333,1,0,0
0.7083,1,-1,-1,-1,0.4186,0.4658,1,-1,-1,0.4656,1,0.2903,1,0,1.0000,-1,-1,1
0.2500,1,0,0,1,0.4186,0.2009,-1,-1,-1,0.7405,-1,0.3226,0,1,0.0000,1,0,0
0.3750,1,0,0,1,0.5116,0.2991,-1,-1,-1,0.6489,-1,0.0000,1,0,0.0000,1,0,0
0.6042,-1,1,0,0,0.6512,0.3584,1,-1,-1,0.6947,-1,0.1613,1,0,0.0000,1,0,0
0.4583,1,-1,-1,-1,0.5349,0.3082,-1,-1,-1,0.8779,1,0.0000,1,0,0.0000,1,0,0
0.7292,1,-1,-1,-1,0.5930,0.1963,-1,-1,-1,0.4656,-1,0.3226,0,1,0.6667,0,1,1
0.5208,1,0,0,1,0.3605,0.3356,-1,-1,-1,0.6183,-1,0.0806,-1,-1,0.3333,1,0,0
0.3542,1,-1,-1,-1,0.3023,0.2808,-1,-1,-1,0.5573,-1,0.1290,1,0,0.0000,-1,-1,1
0.3125,1,-1,-1,-1,0.1860,0.1621,-1,-1,-1,0.8092,-1,0.0000,1,0,0.3333,1,0,1
0.7708,-1,-1,-1,-1,0.9767,0.2329,1,1,0,0.7176,1,0.1613,0,1,0.6667,-1,-1,1
0.5417,-1,0,1,0,0.4767,0.2831,-1,-1,-1,0.6870,-1,0.2258,0,1,0.0000,1,0,0
0.2292,1,-1,-1,-1,0.1860,0.0936,-1,-1,-1,0.3282,1,0.3226,0,1,0.0000,-1,-1,1
0.3958,1,-1,-1,-1,0.3256,0.2192,-1,-1,-1,0.8779,-1,0.0000,1,0,0.0000,1,0,0
0.2500,1,0,1,0,0.1860,0.2489,-1,1,0,0.6260,-1,0.0000,1,0,0.0000,1,0,0
0.6875,-1,-1,-1,-1,0.7674,0.0868,-1,-1,-1,0.5649,-1,1.0000,-1,-1,1.0000,-1,-1,1
0.7917,-1,0,0,1,0.2442,1.0000,-1,-1,-1,0.6794,-1,0.2581,0,1,0.0000,-1,-1,0
0.3125,1,0,0,1,0.3023,0.2283,-1,1,0,0.7481,-1,0.0000,1,0,0.0000,1,0,0
0.1667,1,0,0,1,0.4186,0.2831,-1,1,0,0.8855,-1,0.5645,-1,-1,0.0000,1,0,0
0.6250,1,1,0,0,0.9767,0.3288,-1,-1,-1,0.5649,-1,0.6774,-1,-1,0.0000,-1,-1,0
0.4583,1,0,0,1,0.1860,0.1119,-1,1,0,0.3969,-1,0.0968,1,0,0.0000,1,0,0
0.3542,-1,0,1,0,0.1279,0.1781,-1,1,0,0.7710,-1,0.0000,1,0,0.0000,1,0,0
0.3750,1,-1,-1,-1,0.1860,0.3402,-1,-1,-1,0.3588,1,0.1613,0,1,0.3333,1,0,1
0.4583,-1,0,0,1,0.3023,0.3858,-1,-1,-1,0.6565,-1,0.0968,1,0,0.0000,1,0,0
0.6042,-1,-1,-1,-1,0.0698,0.2785,-1,-1,-1,0.3893,-1,0.1613,0,1,0.0000,1,0,0
0.5208,-1,0,0,1,0.4767,0.4064,1,1,0,0.7557,-1,0.0000,1,0,0.0000,1,0,0
0.3542,1,0,1,0,0.0814,0.1621,1,1,0,0.6489,-1,0.0000,1,0,0.0000,-1,-1,0
0.3333,-1,0,1,0,0.4186,0.2466,-1,-1,-1,0.7939,-1,0.0968,0,1,0.0000,1,0,0
0.6458,1,-1,-1,-1,0.4186,0.1826,-1,-1,-1,0.4656,1,0.3871,0,1,0.6667,-1,-1,1
0.6458,-1,-1,-1,-1,0.6512,0.3014,-1,-1,-1,0.6565,-1,0.4194,0,1,0.6667,-1,-1,1
0.8542,1,0,1,0,0.7209,0.2717,-1,-1,-1,0.5496,-1,0.0000,1,0,0.0000,1,0,0
0.6667,1,-1,-1,-1,0.3023,0.3059,-1,1,0,0.5267,1,0.5806,0,1,0.3333,-1,-1,1
0.2917,1,0,0,1,0.4186,0.4315,-1,1,0,0.6947,-1,0.3065,1,0,0.3333,1,0,0
0.6042,-1,-1,-1,-1,0.4186,0.1621,-1,1,0,0.4580,-1,0.0968,0,1,0.0000,1,0,0
0.3125,1,0,0,1,0.5349,0.2489,-1,-1,-1,0.8321,-1,0.0000,1,0,0.0000,1,0,0
0.6042,1,-1,-1,-1,0.3953,0.2055,-1,-1,-1,0.4580,1,0.3548,0,1,1.0000,-1,-1,1
0.3750,1,0,0,1,0.1628,0.2671,-1,1,0,0.6183,-1,0.0000,1,0,0.0000,1,0,1
0.6667,1,0,0,1,0.6512,0.2671,1,1,0,0.5038,1,0.1613,0,1,0.0000,1,0,0
0.7083,1,-1,-1,-1,0.5349,0.1393,-1,-1,-1,0.5573,1,0.6452,1,0,0.6667,-1,-1,1
0.1250,1,-1,-1,-1,0.3721,0.3562,-1,-1,-1,0.6489,1,0.0000,1,0,0.0000,-1,-1,1
0.3958,1,0,1,0,0.1860,0.2352,-1,1,0,0.7405,-1,0.1613,-1,-1,0.0000,-1,-1,1
0.7917,1,-1,-1,-1,0.3605,0.2922,1,1,0,0.7023,-1,0.0323,0,1,0.6667,-1,-1,1
0.8125,-1,0,0,1,0.3023,0.1941,-1,-1,-1,0.3359,-1,0.2419,0,1,0.0000,1,0,0
0.6042,1,-1,-1,-1,0.6047,0.2100,-1,1,0,0.2595,-1,0.3226,0,1,0.3333,-1,-1,1
0.7292,-1,-1,-1,-1,0.4186,0.4041,-1,1,0,0.3893,-1,0.3226,0,1,0.6667,1,0,0
0.7708,1,0,1,0,0.7674,0.2740,-1,1,0,0.3740,1,0.0000,0,1,1.0000,0,1,1
0.3333,1,-1,-1,-1,0.1163,0.1872,-1,-1,-1,0.5878,1,0.4839,0,1,0.0000,1,0,0
0.5625,1,0,0,1,0.4186,0.2968,1,-1,-1,0.5420,1,0.0968,0,1,0.3333,0,1,1
0.5833,1,-1,-1,-1,0.1860,0.1712,-1,1,0,0.4198,1,0.2419,0,1,0.0000,0,1,0
0.6042,1,-1,-1,-1,0.3953,0.3037,-1,-1,-1,0.4504,1,0.4839,0,1,0.6667,-1,-1,1
0.2500,1,0,0,1,0.2093,0.2831,-1,1,0,0.8244,-1,0.0000,1,0,0.0000,1,0,0
0.5208,1,0,0,1,0.6512,0.2420,-1,-1,-1,0.7176,-1,0.2581,1,0,0.0000,-1,-1,0
0.5000,1,0,0,1,0.4186,0.2740,1,-1,-1,0.7786,-1,0.0000,1,0,1.0000,1,0,0
0.5000,-1,-1,-1,-1,0.5116,0.2466,-1,-1,-1,0.6794,-1,0.0000,1,0,0.0000,1,0,0
0.2500,-1,0,1,0,0.3721,0.4110,-1,1,0,0.7023,-1,0.0000,1,0,0.0000,1,0,0
0.1667,-1,0,0,1,0.3023,0.2032,-1,1,0,0.7557,-1,0.0000,1,0,0.0000,1,0,0
```

```
0.3125,-1,0,0,1,0.2791,0.2648,-1,1,0,0.5954,-1,0.0484,0,1,0.3333,1,0,0
0.5625,1,-1,-1,-1,0.3605,0.2808,1,-1,-1,0.5573,1,0.1935,0,1,0.3333,1,0,1
0.7083,1,-1,-1,-1,0.4186,0.2922,-1,-1,-1,0.5802,-1,0.2258,0,1,0.3333,-1,-1,1
0.7292,-1,-1,-1,-1,1.0000,0.4543,-1,1,0,0.6336,1,0.0000,1,0,0.0000,1,0,0
0.6250,1,-1,-1,-1,0.5349,0.1164,-1,1,0,0.6947,1,0.0000,1,0,0.3333,-1,-1,1
0.4375,1,-1,-1,-1,0.6512,0.2671,-1,-1,-1,0.4351,-1,0.4194,0,1,0.0000,-1,-1,1
0.3125,-1,0,0,1,0.1628,0.0342,-1,1,0,0.7939,-1,0.0968,0,1,0.0000,1,0,0
0.4583,1,0,0,1,0.3605,0.2717,1,-1,-1,0.7252,-1,0.3871,0,1,0.0000,1,0,0
0.6250,1,-1,-1,-1,0.8140,0.1142,1,-1,-1,0.1450,-1,0.1613,0,1,0.6667,0,1,1
0.8125,1,0,0,1,1.0000,0.3379,1,-1,-1,0.6031,1,0.2581,0,1,0.0000,-1,-1,1
0.4792,1,-1,-1,-1,0.3953,0.2945,-1,1,0,0.6870,1,0.0000,1,0,0.3333,-1,-1,1
0.5000,1,-1,-1,-1,0.3372,0.3562,-1,1,0,0.1832,1,0.3226,0,1,0.6667,-1,-1,1
0.6875,1,-1,-1,-1,0.3023,0.3219,-1,1,0,0.2137,1,0.2903,0,1,0.6667,-1,-1,1
0.5208,1,-1,-1,-1,0.3488,0.3196,-1,-1,-1,0.2901,1,0.3548,0,1,0.3333,-1,-1,1
0.6458,-1,0,0,1,0.0930,0.4384,-1,1,0,0.6794,-1,0.0000,1,0,0.3333,1,0,0
0.2500,-1,0,1,0,0.1279,0.1644,-1,1,0,0.7405,-1,0.0000,1,0,0.3333,1,0,0
0.5833,1,-1,-1,-1,0.6512,0.3425,-1,-1,-1,0.3130,1,0.0968,0,1,0.3333,0,1,1
0.2708,1,0,0,1,0.4186,0.1233,-1,1,0,0.6031,-1,0.0000,1,0,0.0000,1,0,0
0.5833,-1,0,1,0,0.4186,0.2511,-1,-1,-1,0.7863,-1,0.0000,0,1,0.3333,1,0,1
0.6042,-1,0,0,1,0.3023,0.4886,-1,1,0,0.7710,-1,0.0000,1,0,0.0000,1,0,0
0.5833,1,0,1,0,0.3488,0.3082,-1,1,0,0.5344,-1,0.0484,1,0,0.0000,-1,-1,1
0.6042,1,-1,-1,-1,0.3605,0.3973,-1,-1,-1,0.7634,-1,0.0000,1,0,0.6667,-1,-1,1
0.5208,-1,0,0,1,0.1860,0.2009,-1,1,0,0.6641,-1,0.2581,0,1,0.0000,1,0,0
0.6667,1,-1,-1,-1,0.5349,0.1849,-1,-1,-1,0.5115,1,0.3065,1,0,0.3333,-1,-1,1
0.4792,-1,0,0,1,0.4884,0.1598,-1,-1,-1,0.7481,-1,0.0161,0,1,0.0000,1,0,0
0.6042,1,0,1,0,0.3023,0.3607,-1,-1,-1,0.6794,-1,0.2903,0,1,0.0000,1,0,1
0.5625,-1,-1,-1,-1,0.4651,0.6461,-1,-1,-1,0.6031,1,0.3065,0,1,0.6667,-1,-1,1
0.5208,1,-1,-1,-1,0.1860,0.1826,-1,-1,-1,0.2824,1,0.0000,0,1,0.3333,1,0,1
0.6667,-1,-1,-1,-1,0.4186,0.4658,-1,-1,-1,0.7481,-1,0.0000,1,0,0.0000,1,0,1
0.7500,1,-1,-1,-1,0.4767,0.2922,-1,-1,-1,0.4275,-1,0.4516,0,1,0.3333,-1,-1,1
```

Data Listing 6-2: Cleveland_test.txt (tabs replaced with commas to save space)

```
0.8542,1,-1,-1,-1,0.4186,0.4475,-1,-1,-1,0.2901,-1,0.3871,0,1,1.0000,1,0,1
0.6458,1,-1,-1,-1,0.4186,0.2900,-1,1,0,0.5573,1,0.2258,1,0,0.3333,-1,-1,1
0.2292,1,1,0,0,0.5349,0.1667,-1,1,0,0.8168,1,0.2258,1,0,0.0000,-1,-1,0
0.2708,1,1,0,0,0.6279,0.2694,-1,-1,-1,0.8168,-1,0.1290,1,0,0.6667,1,0,0
0.2708,1,0,1,0,0.3023,0.3858,-1,1,0,0.6947,-1,0.0000,1,0,0.0000,1,0,0
0.7708,1,-1,-1,-1,0.2093,0.1963,-1,-1,-1,0.4656,1,0.0161,1,0,0.3333,1,0,1
0.5625,1,0,1,0,0.4186,0.2169,-1,-1,-1,0.7023,-1,0.0000,1,0,0.0000,-1,-1,0
0.3958,1,-1,-1,-1,0.4186,0.2968,1,-1,-1,0.6031,1,0.0000,1,0,0.6667,-1,-1,1
0.2917,1,-1,-1,-1,0.1860,0.1941,-1,1,0,0.6870,-1,0.0000,1,0,0.0000,-1,-1,0
0.1042,-1,0,1,0,0.2791,0.1918,-1,1,0,0.9237,-1,0.1129,1,0,0.0000,1,0,0
0.5833,1,0,0,1,0.6512,0.0959,-1,1,0,0.7863,-1,0.2581,1,0,0.0000,1,0,0
0.3542,1,-1,-1,-1,0.5349,0.4224,-1,1,0,0.3740,1,0.2903,0,1,0.6667,-1,-1,1
0.6250,1,-1,-1,-1,0.1860,0.2580,-1,-1,-1,0.5420,1,0.1935,0,1,0.3333,-1,-1,1
0.5625,1,0,1,0,0.3023,0.2603,-1,1,0,0.7481,-1,0.0000,-1,-1,0.0000,1,0,0
0.5625,-1,-1,-1,-1,1.2326,0.3699,1,-1,-1,0.4733,1,0.6452,-1,-1,0.6667,-1,-1,1
0.1042,1,1,0,0,0.2791,0.1279,-1,-1,-1,0.7863,-1,0.0000,1,0,0.0000,1,0,0
0.4792,1,0,1,0,0.4651,0.1712,-1,1,0,0.6641,-1,0.1290,1,0,0.3333,1,0,0
0.5208,1,0,1,0,1.1395,0.3584,-1,-1,-1,0.9466,-1,0.0000,1,0,0.3333,-1,-1,1
0.5625,1,0,1,0,0.3023,0.2511,-1,1,0,0.8168,-1,0.1290,1,0,0.0000,1,0,0
0.6458,1,-1,-1,-1,0.2674,0.2374,1,1,0,0.6794,1,0.2258,1,0,0.6667,-1,-1,1
0.7917,1,0,0,1,0.6744,0.1963,-1,-1,-1,0.6031,-1,0.1290,0,1,0.0000,-1,-1,1
0.2708,-1,0,0,1,0.3023,0.1895,-1,1,0,0.7786,-1,0.0000,0,1,0.0000,1,0,0
0.6458,1,-1,-1,-1,0.5930,0.3562,-1,-1,-1,0.5420,1,0.4516,0,1,0.6667,-1,-1,1
0.3750,1,-1,-1,-1,0.2093,0.1781,-1,1,0,0.5496,-1,0.0161,1,0,0.0000,1,0,0
0.6250,1,1,0,0,0.8837,0.3699,-1,-1,-1,0.6718,-1,0.0323,0,1,0.0000,-1,-1,1
```

```
0.6667,-1,-1,-1,-1,0.5930,0.4132,-1,-1,-1,0.5725,1,0.1613,0,1,0.0000,-1,-1,1
0.7292,1,0,0,1,0.3605,0.4178,-1,1,0,0.4580,1,0.2903,0,1,0.0000,-1,-1,1
0.4792,1,-1,-1,-1,0.2093,0.2374,-1,1,0,0.6794,-1,0.0000,1,0,0.3333,1,0,1
0.7292,1,-1,-1,-1,0.3953,0.3128,-1,1,0,0.2595,1,0.0323,0,1,0.3333,-1,-1,0
0.4792,1,-1,-1,-1,0.1628,0.2443,1,1,0,0.5802,-1,0.0161,1,0,1.0000,-1,-1,0
0.1250,1,0,1,0,0.3256,0.1507,-1,1,0,0.7863,-1,0.0000,1,0,0.0000,1,0,0
0.7292,1,-1,-1,-1,0.3023,0.2740,-1,-1,-1,0.1908,1,0.3548,-1,-1,0.3333,1,0,1
0.5000,1,-1,-1,-1,0.5349,0.1758,1,-1,-1,0.6412,1,0.5000,-1,-1,0.0000,-1,-1,1
0.1250,-1,-1,-1,-1,0.5116,0.1301,-1,1,0,0.8473,-1,0.2258,1,0,0.0000,1,0,0
0.7083,1,1,0,0,0.5930,0.2443,1,-1,-1,0.6031,-1,0.3710,-1,-1,0.0000,0,1,0
0.7917,1,-1,-1,-1,0.3023,0.2352,-1,-1,-1,0.4427,1,0.4194,0,1,0.6667,-1,-1,1
0.2083,1,-1,-1,-1,0.2791,0.2123,-1,1,0,0.5267,-1,0.1935,0,1,0.0000,-1,-1,1
0.4375,1,-1,-1,-1,0.5814,0.1689,-1,-1,-1,0.4198,1,0.1452,0,1,0.0000,-1,-1,1
0.2083,1,0,0,1,0.5349,0.4452,-1,-1,-1,0.8473,-1,0.0000,1,0,0.0000,1,0,0
0.7500,-1,-1,-1,-1,0.6512,0.2260,-1,-1,-1,0.3282,-1,0.1613,0,1,1.0000,-1,-1,1
0.7708,-1,1,0,0,0.6512,0.2283,-1,1,0,0.3282,-1,0.4194,-1,-1,0.0000,1,0,0
0.7500,1,-1,-1,-1,0.1860,0.2785,-1,-1,-1,0.6641,-1,0.0968,1,0,0.6667,0,1,1
0.8542,1,0,0,1,0.7674,0.3265,-1,1,0,0.3130,1,0.4677,0,1,0.3333,-1,-1,1
0.6250,1,0,0,1,0.3721,0.2100,1,1,0,0.4809,-1,0.3548,0,1,0.3333,0,1,1
0.7500,-1,0,0,1,0.7674,0.5342,-1,-1,-1,0.6107,-1,0.1290,1,0,0.0000,1,0,0
0.7917,1,-1,-1,-1,0.7674,0.3653,-1,-1,-1,0.2824,1,0.2419,0,1,1.0000,1,0,1
0.7708,-1,0,0,1,0.6047,0.3470,-1,-1,-1,0.6183,-1,0.0000,0,1,0.3333,1,0,0
0.5833,1,-1,-1,-1,0.6744,0.3379,-1,1,0,0.1298,1,0.1935,0,1,0.3333,-1,-1,1
0.6042,1,-1,-1,-1,0.0698,0.2466,-1,1,0,0.6489,-1,0.0161,1,0,0.3333,-1,-1,1
0.5208,-1,0,1,0,0.4419,0.3699,1,-1,-1,0.6718,1,0.0000,1,0,0.3333,1,0,0
0.5833,1,0,0,1,0.3953,0.2352,-1,-1,-1,0.6031,-1,0.0645,0,1,0.3333,-1,-1,1
0.7083,-1,-1,-1,-1,0.3488,0.1621,-1,1,0,0.4962,1,0.0000,0,1,0.0000,1,0,1
0.6458,-1,1,0,0,0.6512,0.2603,-1,1,0,0.7634,-1,0.1452,1,0,0.0000,1,0,0
0.7500,1,1,0,0,0.5116,0.3562,1,-1,-1,0.7863,-1,0.2258,0,1,0.3333,1,0,1
0.4792,1,0,0,1,0.9070,0.1667,1,1,0,0.6947,-1,0.0806,1,0,0.0000,-1,-1,0
0.7083,-1,-1,-1,-1,0.6512,0.6416,-1,-1,-1,0.6336,-1,0.6452,0,1,1.0000,-1,-1,1
0.4375,-1,0,1,0,0.3023,0.2694,-1,1,0,0.6947,-1,0.1774,1,0,0.0000,1,0,0
0.7500,1,-1,-1,-1,0.3023,0.1164,-1,1,0,0.5267,-1,0.0645,1,0,0.0000,-1,-1,0
0.4583,1,0,0,1,0.0698,0.2192,-1,1,0,0.5496,1,0.1935,0,1,0.0000,1,0,0
0.6875,-1,-1,-1,-1,0.6512,0.2694,-1,1,0,0.6336,1,0.2258,0,1,0.0000,1,0,1
```