***Assignment 2***

Name: Joshua Page

Student Number: 224046136

Date: 2024/09/02

I declare that this is my own, original work.

Signature: JPage

*Implementation Details*

Stopping Conditions: Max number of training iterations reached. However, prevention of overfitting was investigated, in which case training would stop to prevent overfitting, see additional techniques below.

Initial Weights: Random values between 0 and 0.1. However, the appropriate initialisation of weights was also investigated, see additional techniques below.

Training Set size: 1 710

Test Set size: 190

Values for investigated: 0.1, 0.01, 0.001

Activation functions: Sigmoid

Number of hidden neurons: 35

The dataset of steel plates has twenty-seven attributes to describe each plate, based on these attributes, each steel plate is classified as a fault between 1 and 7 categories. Thus, my neural network has twenty-seven input neurons, one for each of the inputs provided in the data set. For the hidden layer, there are thirty-five hidden neurons, and the output layer has seven output neurons (for each classification). In summary, my Feedforward Neural Network consists of one input layer with twenty-seven neurons, one hidden layer with thirty-five hidden neurons, and one output layer with seven output neurons.

*Results*

(These are the initial results prior to investigating techniques, and are the best in terms of test accuracy)

Number of iterations (typically): 20 000

Best  value: 0.001

Sum Squared Error (SSE) on Training Set: 176.96169745341334

SSE on Test Set: 40.953952363786016

Number correctly classified on Training Set: 1298 correct out of 1520 (85.39% accuracy)

Number correctly classified on Test Set: 129 correct out of 190 (67.89% accuracy)

A graph showing different colored lines

Description automatically generated

Figure 1: SSE vs iterations for various values of 

Although Figure 1 indicates that Ƞ = 0.001 did not have the lowest SSE after training, Ƞ = 0.001 had the best results in terms of test accuracy. For this reason, I chose to investigate the additional techniques with a learning rate of Ƞ = 0.001.

*Investigated Techniques*

Additional Techniques Investigated:

|  |  |  |  |
| --- | --- | --- | --- |
| **Technique** | **SSE on training set** | **SSE on test set** | **Test Accuracy (%)** |
| Appropriate weight initialisation | 175.0350064462353 | 37.15319385467177 | 72.105% |
| Prevention of overfitting | 223.6059792902008 | 33.54332117071792 | 75.263% |
| Noise Injection | 170.38035864501404 | 38.574732949126705 | 73.684% |
| All | 205.93456077231787 | 33.24664928872811 | 76.315% |

A graph of a function

Description automatically generated

Figure 2: SSE vs iterations for the techniques for learning rate  = 0.001

Results on patterns provided without category:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 4 | 11 | 7 | 21 | 3 | 31 | 3 |
| 2 | 4 | 12 | 3 | 22 | 7 | 32 | 3 |
| 3 | 3 | 13 | 7 | 23 | 2 | 33 | 7 |
| 4 | 6 | 14 | 1 | 24 | 1 | 34 | 7 |
| 5 | 6 | 15 | 3 | 25 | 7 | 35 | 5 |
| 6 | 7 | 16 | 7 | 26 | 3 | 36 | 7 |
| 7 | 7 | 17 | 2 | 27 | 1 | 37 | 7 |
| 8 | 3 | 18 | 6 | 28 | 3 | 38 | 2 |
| 9 | 1 | 19 | 6 | 29 | 7 | 39 | 6 |
| 10 | 1 | 20 | 2 | 30 | 2 | 40 | 6 |

*Observations*

The Feedforward Neural Network did not perform as well as I initially anticipated. I thought the model would perform far better than assignment one, which utilised a single neuron to predict salaries, and I expected near-perfect results in terms of test accuracy. The fact that the model would only need to classify each pattern into a fault category between 1 and 7 further strengthened this assumption, which was proven wrong throughout implementation of this assignment.

When classifying each fault category for the last forty patterns, I noticed that the model tended to favour classifying patterns as a fault category of 6 or 7 and seldom predicted anything else (especially with higher learning rates). This bias seemed strange to me, so I investigated the dataset and found that there were far more patterns that classified faults as a 6 or 7, providing good results during training, but poor results for test accuracy.

Understanding this, I realised the neural network’s performance was heavily influenced by the imbalance, leading to biased predictions. To address this, I chose techniques such as preventing overfitting so that the model would not merely memorise the patterns and their classifications, but instead perform better when classifying the test data.

*Observations for Appropriate Weight Initialisation*

This technique initialises the weights between , which definitely impacted the performance of the neural network. This technique regularly ensured that the training SSE was lower after the first training iteration, especially with lower learning rates, which reduced training time and improved the performance of the model. However, when not utilised in conjunction with the prevention of overfitting, it would result in overfitting more quickly, providing an increased SSE in the test set and worse test accuracy.

*Observations for Noise Injection*

This technique added a small noise amount to each of the inputs for every pattern, which optimistically would provide a smoother search space, increase accuracy, and reduce training time. Noise injection did assist in terms of increasing accuracy, which can be seen by the decreased SSE and increased test accuracy in comparison to the results provided in the results prior to applying additional techniques.

However, the amount of noise being injected was an important factor. Adding a noise amount that was too high would negatively impact the model, and the best level of noise to add was a random amount between -0.05 and 0.05, aiming to introduce balanced noise to add variability without distorting the data.

*Observations for Prevention of Overfitting*

This technique utilised both a training set and a training validation set to train the neural network. When overfitting is detected, training is stopped. Overfitting is detected utilising the following formula from the course notes, where is the current iteration’s training validation error, is the average training validation error, and is the standard deviation of training validation errors:

This technique did improve the performance of my neural network, as it would typically prevent overfitting, and regularly ensure better results, in terms of test accuracy. Additionally, training required fewer iterations. However, the training SSE remained higher than with any of the other investigated techniques, except when all techniques were used together, which included prevention of overfitting

*Observations Applying All Techniques*

I chose to investigate noise injection, prevention of overfitting, and appropriate weight initialisation all being applied during training, to determine whether these techniques work better separately, or work well together. Utilising all these techniques in conjunction, the neural network achieved a test accuracy of 76.315%, which is the best I had obtained. This implied that utilising further enhancements to the model would provide even better test accuracy results, as long as all the techniques performed well in conjunction and complimented one another (such as prevention of overfitting and appropriate weight initialisation).

*Conclusion*

This assignment significantly enhanced my understanding of neural networks. Previously, I was extremely unaware of the impact the dataset itself would have; a dataset that is not balanced, meaning it may favour certain classifications (such as classifying 6s and 7s in this dataset) would encourage the model to favour these classifications when predicting.

My most surprising finding was the importance of learning rate choices. Previously, I thought that a high learning rate would be better, and provide the best accuracy, as the model is then learning more. However, that was not the case at all. Although a higher learning rate would typically obtain a lower SSE quickly, it would never be able to find the lower SSEs, whereas the smaller learning rates would. Typically, the higher learning rates would also fluctuate, as can be seen by Ƞ = 0.001 in Figure 1.

Unfortunately, I was impacted by the power outages in Summerstrand, and I was unable to train my model with lower learning rates for longer than 4 hours at a time, meaning that the model would have not had enough time to train until the power would cycle on or off, halting my training completely. I would have liked to investigate lower learning rates and see their impact on the model.

Additionally, I would have liked to investigate oversampling, as a technique to address the unbalanced dataset. Optimistically, this technique would have provided a more balanced dataset, encouraging the model to learn equally among all possible classifications. However, this was not one of the available techniques: prevention of overfitting, noise injection, appropriate weight initialization, dynamic learning rate, momentum, and network architecture, so I opted to not investigate oversampling.

Lastly, this neural network was an enjoyable challenge and greatly contributed to my understanding of neural networks. Going forward, I hope to use this knowledge and insight for further assignments.

*Code*

using System;

using System.Diagnostics;

using System.Collections.Generic;

using System.IO;

using System.Linq;

namespace Assignment\_2

{

public class Program

{

static double[,] z;

static int[] t;

static HashSet<int> y;

public static void Main()

{

LoadRawData("Plates.csv", out z, out t, out y);

int inputSize = z.GetLength(1);

double[,] z\_val;

int[] t\_val;

LoadValidationData(out z\_val, out t\_val, "TrainValidation.csv");

FFNNModel model = new(

z: z,

t: t,

input\_size: inputSize,

hidden\_neurons: 35,

output\_size: y.Count(),

learning\_rate: 0.001,

epochs: 20000,

train: true,

z\_val: z\_val,

t\_val: t\_val

);

Console.WriteLine("\nComparing against training data...");

ComparePredictions(model);

LoadValidationData(out z\_val, out t\_val, "TestValidation.csv");

Console.WriteLine("\nComparing against test data...");

ComparePredictions(model, z\_val, t\_val);

LoadEvalData("Evaluation.csv", out z);

DisplayPredictions(model);

}

private static void ComparePredictions(FFNNModel model, double[,] z\_data = null, int[] t\_data = null)

{

z\_data ??= z;

t\_data ??= t;

int correct = 0;

int wrong = 0;

int[] o = new int[t\_data.Length];

Dictionary<int, int> correctPredictionsPerClass = new Dictionary<int, int>();

Dictionary<int, int> totalClassificationsPerClass = new Dictionary<int, int>();

for (int i = 1; i <= 7; i++)

{

correctPredictionsPerClass[i] = 0;

totalClassificationsPerClass[i] = 0;

}

for (int i = 0; i < z\_data.GetLength(0); i++)

{

double[] predictedOutput = model.Predict(GetCurrentPattern(z\_data, i));

int predictedClass = Array.IndexOf(predictedOutput, predictedOutput.Max()) + 1;

o[i] = predictedClass;

int actualClass = t\_data[i];

totalClassificationsPerClass[actualClass]++;

if (predictedClass == actualClass)

{

correct++;

correctPredictionsPerClass[actualClass]++;

}

else

{

wrong++;

}

}

double percentage = (double)correct / (correct + wrong) \* 100;

Console.WriteLine($"Correct: {correct} | Wrong: {wrong} | Accuracy: {percentage}%");

Console.WriteLine($"SSE: {model.SSE(z\_data, t\_data)}");

Console.WriteLine("Correct predictions and accuracy per class:");

foreach (var kvp in correctPredictionsPerClass)

{

int classLabel = kvp.Key;

int correctPredictions = kvp.Value;

int totalClassifications = totalClassificationsPerClass[classLabel];

double classAccuracy = (double)correctPredictions / totalClassifications \* 100;

Console.WriteLine($"Class {classLabel}: {correctPredictions} / {totalClassifications} ({classAccuracy}%)");

}

}

private static void DisplayPredictions(FFNNModel model)

{

Console.WriteLine("\nPredicted values: ");

for (int i = 0; i < z.GetLength(0); i++)

{

double[] predictedOutput = model.Predict(GetCurrentPattern(z, i));

int predictedClass = Array.IndexOf(predictedOutput, predictedOutput.Max()) + 1;

Console.WriteLine($"Predicted: {predictedClass}");

}

}

public static double[] GetCurrentPattern(double[,] z, int p)

{

double[] z\_p = new double[z.GetLength(1)];

for (int i = 0; i < z.GetLength(1); i++)

{

z\_p[i] = z[p, i];

}

return z\_p;

}

public static void LoadValidationData(out double[,] z, out int[] t, String filePath)

{

string[] lines = File.ReadAllLines(filePath);

int numLines = lines.Length;

int numInputs = lines[0].Split(',').Length - 1;

z = new double[numLines, numInputs];

t = new int[numLines];

y = new HashSet<int>();

int[] classificationCounts = new int[7];

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

{

string[] values = lines[i].Split(',');

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

{

z[i, j] = double.Parse(values[j]);

}

t[i] = int.Parse(values[numInputs]);

y.Add(t[i]);

classificationCounts[t[i] - 1]++;

}

NormaliseData(z);

}

public static void LoadRawData(string filePath, out double[,] z, out int[] t, out HashSet<int> y, double noiseLevel = 0.1)

{

string[] lines = File.ReadAllLines(filePath);

int numLines = lines.Length;

int numInputs = lines[0].Split(',').Length - 1;

z = new double[numLines, numInputs];

t = new int[numLines];

y = new HashSet<int>();

Random rand = new Random();

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

{

string[] values = lines[i].Split(',');

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

{

z[i, j] = double.Parse(values[j]);

//z[i, j] += (rand.NextDouble() - 0.5) \* noiseLevel;

}

t[i] = int.Parse(values[numInputs]);

y.Add(t[i]);

}

NormaliseData(z);

}

private static void LoadEvalData(string filePath, out double[,] z)

{

string[] lines = File.ReadAllLines(filePath);

int numLines = lines.Length;

int numInputs = lines[0].Split(',').Length;

z = new double[numLines, numInputs];

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

{

string[] values = lines[i].Split(',');

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

{

z[i, j] = double.Parse(values[j]);

}

}

NormaliseData(z);

}

private static void NormaliseData(double[,] z)

{

int numLines = z.GetLength(0);

int numInputs = z.GetLength(1);

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

{

double min = double.MaxValue;

double max = double.MinValue;

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

{

if (z[i, j] < min)

min = z[i, j];

if (z[i, j] > max)

max = z[i, j];

}

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

{

z[i, j] = (z[i, j] - min) / (max - min);

}

}

}

}

}

namespace Assignment\_2

{

public class FFNNModel

{

private double[,] w; // Weights for output layer neurons

private double[,] v; // Weights for hidden layer neurons

private int J; // Number of hidden layer neurons

private int K; // Number of output layer neurons

private int I; // Number of input units

private double learning\_rate;

public FFNNModel(double[,] z, int[] t, int input\_size, int hidden\_neurons, int output\_size, double learning\_rate, int epochs, bool train, double[,] z\_val = null, int[] t\_val = null)

{

this.I = input\_size;

this.J = hidden\_neurons;

this.K = output\_size;

this.learning\_rate = learning\_rate;

w = InitWeights(K, J + 1, -1 / Math.Sqrt(I), 1 / Math.Sqrt(I));

v = InitWeights(J, I + 1, -1 / Math.Sqrt(I), 1 / Math.Sqrt(I));

if (!train)

{

LoadWeights("outputWeights.csv", w);

LoadWeights("hiddenWeights.csv", v);

}

else GradientDescent(z, t, epochs, z\_val, t\_val);

}

private double[,] InitWeights(int rows, int cols, double lb, double ub)

{

Random rand = new Random();

double[,] weights = new double[rows, cols];

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

{

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

{

weights[i, j] = lb + rand.NextDouble() \* (ub - lb);

//weights[i, j] = rand.NextDouble() \* 0.1;

}

}

return weights;

}

public double SSE(double[,] z, int[] t)

{

double sum = 0;

for (int p = 0; p < z.GetLength(0); p++)

{

double[] z\_p = GetCurrentPattern(z, p);

double[] y = CalculateHiddenLayerOutputs(z\_p);

double[] o = CalculateOutputLayerOutputs(y);

double[] target = new double[o.Length];

target[t[p] - 1] = 1.0;

for (int k = 0; k < K; k++)

{

sum += Math.Pow(target[k] - o[k], 2);

}

}

return 0.5 \* sum;

}

private double[] CalculateHiddenLayerOutputs(double[] z\_p)

{

double[] y = new double[this.J];

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

{

double net\_yj = 0;

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

{

net\_yj += v[j, i] \* z\_p[i];

}

net\_yj += v[j, I];

y[j] = Sigmoid(net\_yj);

}

return y;

}

private double[] CalculateOutputLayerOutputs(double[] y)

{

double[] o = new double[this.K];

for (int k = 0; k < K; k++)

{

double net\_ok = 0;

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

{

net\_ok += w[k, j] \* y[j];

}

net\_ok += w[k, J];

o[k] = Sigmoid(net\_ok);

}

return o;

}

public void GradientDescent(double[,] z, int[] t, int max\_epoch, double[,] z\_val, int[] t\_val)

{

Console.WriteLine("Training model...");

int cur\_epoch = 0;

double[] sse\_history = new double[max\_epoch];

List<double> val\_errors = new List<double>();

while (cur\_epoch < max\_epoch)

{

int misclassifications = 0;

double total\_error = 0;

for (int p = 0; p < z.GetLength(0); p++)

{

double[] z\_p = GetCurrentPattern(z, p);

double[] y = CalculateHiddenLayerOutputs(z\_p);

double[] o = CalculateOutputLayerOutputs(y);

int predictedClass = Array.IndexOf(o, o.Max()) + 1;

if (predictedClass != t[p]) misclassifications++;

double[] target = new double[o.Length];

target[t[p] - 1] = 1.0;

for (int k = 0; k < K; k++)

{

double error = target[k] - o[k];

total\_error += Math.Pow(error, 2);

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

{

w[k, j] -= learning\_rate \* -2 \* error \* o[k] \* (1 - o[k]) \* y[j];

}

w[k, J] -= learning\_rate \* -2 \* error \* o[k] \* (1 - o[k]);

}

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

{

double delta\_v\_bias = 0;

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

{

double delta\_v = 0;

for (int k = 0; k < K; k++)

{

double error = target[k] - o[k];

delta\_v += -2 \* error \* o[k] \* (1 - o[k]) \* w[k, j] \* y[j] \* (1 - y[j]) \* z\_p[i];

delta\_v\_bias += -2 \* error \* o[k] \* (1 - o[k]) \* w[k, j] \* y[j] \* (1 - y[j]);

}

v[j, i] -= learning\_rate \* delta\_v;

}

v[j, I] -= learning\_rate \* delta\_v\_bias;

}

}

double current\_error = (0.5 \* total\_error);

double val\_error = Validate(z\_val, t\_val);

val\_errors.Add(val\_error);

double val\_mean = val\_errors.Average();

double std\_val\_error = Math.Sqrt(val\_errors.Average(v => Math.Pow(v - val\_mean, 2) / val\_errors.Count));

if (val\_error > (val\_mean + std\_val\_error))

{

Console.WriteLine("Overfitting on validation SSE, breaking early...");

break;

}

if (cur\_epoch % 10 == 0)

{

Console.WriteLine($"Epoch: {cur\_epoch} | Learning Rate: {learning\_rate} | Training SSE: {current\_error} | Validation SSE: {val\_error} | Stopping value {val\_mean + std\_val\_error}");

}

sse\_history[cur\_epoch] = 0.5 \* total\_error;

cur\_epoch++;

}

SaveWeights("outputWeights.csv", w);

SaveWeights("hiddenWeights.csv", v);

SaveSSE("sseOverEpochs.csv", sse\_history);

}

private double Validate(double[,] z\_val, int[] t\_val)

{

double total\_error = 0;

for (int p = 0; p < z\_val.GetLength(0); p++)

{

double[] z\_p = GetCurrentPattern(z\_val, p);

double[] y = CalculateHiddenLayerOutputs(z\_p);

double[] o = CalculateOutputLayerOutputs(y);

double[] target = new double[o.Length];

target[t\_val[p] - 1] = 1.0;

for (int k = 0; k < K; k++)

{

double error = target[k] - o[k];

total\_error += Math.Pow(error, 2);

}

}

return (0.5 \* total\_error);

}

private void SaveSSE(string fileName, double[] sse\_history)

{

using (StreamWriter streamWriter = new StreamWriter(fileName))

{

for (int i = 0; i < sse\_history.Length; i++)

{

streamWriter.Write(sse\_history[i]);

if (i != sse\_history.Length - 1)

{

streamWriter.Write(",");

}

}

}

Console.WriteLine($"Saved {fileName} successfully!");

}

public double[] Predict(double[] z\_p)

{

double[] y = CalculateHiddenLayerOutputs(z\_p);

return CalculateOutputLayerOutputs(y);

}

private double Sigmoid(double x) { return 1.0 / (1.0 + Math.Exp(-x)); }

private double[] GetCurrentPattern(double[,] z, int p)

{

double[] z\_p = new double[z.GetLength(1)];

for (int i = 0; i < z.GetLength(1); i++)

{

z\_p[i] = z[p, i];

}

return z\_p;

}

private void SaveWeights(string fileName, double[,] weights)

{

using (StreamWriter streamWriter = new StreamWriter(fileName))

{

for (int i = 0; i < weights.GetLength(0); i++)

{

for (int j = 0; j < weights.GetLength(1); j++)

{

streamWriter.Write(weights[i, j]);

if (j != weights.GetLength(1) - 1)

{

streamWriter.Write(",");

}

}

streamWriter.WriteLine();

}

}

Console.WriteLine($"Saved {fileName} successfully!");

}

private void LoadWeights(string fileName, double[,] weights)

{

Console.WriteLine($"Loading {fileName}...");

using (StreamReader streamReader = new StreamReader(fileName))

{

int rows = weights.GetLength(0);

int cols = weights.GetLength(1);

string line;

int i = 0;

while ((line = streamReader.ReadLine()) != null && i < rows)

{

string[] values = line.Split(new[] { ',' }, StringSplitOptions.RemoveEmptyEntries);

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

{

weights[i, j] = double.Parse(values[j]);

}

i++;

}

}

}

}

}