Neural Network Regression

Md.Toufiq Zaman
md.toufiqzaman@gmail.com

***Abstract*—This paper describes a Neural Network Regression Algorithm and its Implementation in C# language at Microsoft.NET Core platform. In the Program a Neural Network model has been created which can predict the value of a Sine wave. Learning Api is also used in this project.**

Keywords— Neural network ,Regression ,LearningAPI

I. INTRODUCTION

Artificial Intelligence and Machine Learning is one of hot topic in today world and it's exploding. Everything today we are experiencing has behind power of machine learning algorithms. Neural network is a machine learning technique or algorithm that try to mimic the working of neuron in human brain for learning.

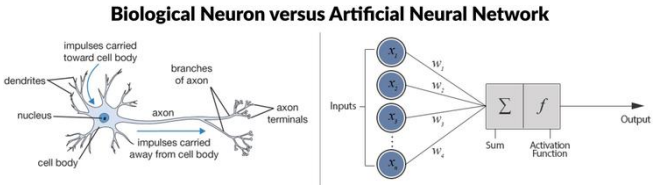


Fig. 1: Biological Neuron and Artificial Neural Network

Although neural networks are widely known for use in deep learning and modeling complex problems such as image recognition, they are easily adapted to regression problems. Any class of statistical models can be termed a neural network if they use adaptive weights and can approximate non-linear functions of their inputs. Thus neural network regression is suited to problems where a more traditional regression model cannot fit a solution.

Neural network regression is a supervised learning method, and therefore requires a tagged dataset, which includes a label column. Because a regression model predicts a numerical value, the label column must be a numerical data type.

II. USING NEURAL NETWORK WITH REGRESSION

The goal of a regression problem is to predict the value of a numeric variable (usually called the dependent variable) based on the values of one or more predictor variables (the independent variables), which can be either numeric or categorical. For example, if we want to predict the annual income of a person based on age, sex (male or female) and years of education.

The simplest form of regression is called linear regression (LR). An LR prediction equation might look like this: income = 17.53 + (5.11 * age) + (-2.02 * male) + (-1.32 * female) + (6.09 * education). Although LR is useful for some problems, in many situations it's not effective. But there are other common types of regression—polynomial regression, general linear model regression and neural network regression (NNR). Arguably, the last type of regression is the most powerful form of regression.

The most common type of neural network (NN) is one that predicts a categorical variable. For example, To predict a person's political inclination (conservative, moderate, liberal) based on factors such as age, income and sex. An NN classifier has n output nodes, where n is the number of values that the dependent variable can take. The values of the n output nodes sum to 1.0 and can be loosely interpreted as probabilities. So, for predicting political inclination, an NN classifier would have three output nodes. If the output node values were (0.24, 0.61, 0.15), the NN classifier is predicting "moderate" because the middle node has the largest probability.

In NN regression, the NN has a single output node that holds the predicted value of the dependent numeric variable. So, for the example that predicts annual income, there would be three input nodes (one for age, one for sex where male = -1 and female = +1, and one for years of education), and one output node (annual income).

Reasonable people can disagree about whether using neural networks for regression is overkill. The point is just to explain below how it can be done.

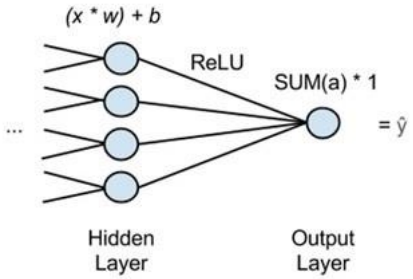


Fig. 2: Using Neural Network with Regression

In the diagram above, x stands for input, the features passed forward from the network's previous layer. Many x's will be fed into each node of the last hidden layer, and each x will be multiplied by a corresponding weight, w.

The sum of those products is added to a bias and fed into an activation function. The activation function is a rectified linear unit (ReLU), commonly used and highly useful because it doesn't saturate on shallow gradients as sigmoid activation functions do.

For each hidden node, ReLU outputs an activation, a, and the activations are summed going into the output node, which simply passes the activations' sum through.

That is, a neural network performing regression will have one output node, and that node will just multiply the sum of the previous layer's activations by 1. The result will be ŷ, "y hat", the network's estimate, the dependent variable that all x's map to.

To perform backpropagation and make the network learn, it is simply compared ŷ to the ground-truth value of y and adjust the weights and biases of the network until error is minimized, much as it would with a classifier. Root-means-squared-error (RMSE) could be the loss function.

III. Brief Details of Algorithm and Function

The goal of the program used in this project is to create an Neural Network model that can predict the value of the sine function. The graph of the sine function is shown in Fig 3. The sine function accepts a single real input value from negative infinity to positive infinity and returns a value between -1.0 and +1.0. The sine function returns 0 when x = 0.0, x = pi (~3.14), x = 2 * pi, x= 3 * pi, and so on. The sine function is a surprisingly difficult function to model.

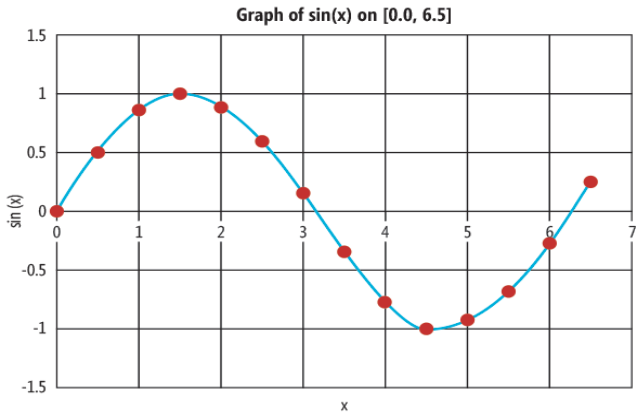


Fig. 3: The Sin(x) Function

The program creates a 1-12-1 NN, that is, an NN with one input node (for x), 12 hidden processing nodes (that effectively define the prediction equation), and one output node (the predicted sine of x). When working with NNs, there's always experimentation involved; the number of hidden nodes was determined by trial and error.

NN classifiers have two activation functions, one for the hidden nodes and one for the output nodes. The output node activation function for a classifier is almost always the softmax function because softmax produces values that sum to 1.0. The hidden node activation function for a classifier is usually either the logistic sigmoid function or the hyperbolic tangent function (abbreviated tanh). But in NN regression, there's a hidden node activation function, but no output node activation function. The program NN uses the tanh function for hidden node activation.

The output of an NN is determined by its input values and a set of constants called the weights and biases. Because biases are really just special kinds of weights, the term "weights" is sometimes used to refer to both. A neural network with i input nodes, j hidden nodes, and k output nodes has a total of (i * j) + j + (j * k) + k weights and biases. So the 1-12-1 NN has (1 * 12) + 12 + (12 * 1) + 1 = 37 weights and biases.

The process of determining the values of the weights and the biases is called training the model. The idea is to try different values of the weights and biases to determine where the computed output values of the NN closely match the known correct output values of the training data.

There are several different algorithms that can be used to train an NN. By far the most common approach is to use the back-propagation algorithm. Back propagation is an iterative process in which values of the weights and biases slowly change, so that the NN usually computes more accurate output values.

Back propagation uses two required parameters (maximum number of iterations and learning rate) and one optional parameter (the momentum rate). The maxEpochs parameter sets a limit on the number of algorithm iterations. The learnRate parameter controls how much the weight and bias values can change in each iteration. The momentum parameter speeds up training and also helps prevent the back-propagation algorithm from getting stuck at a poor solution.

When using the back-propagation algorithm for NN training, there are three variations that can be used. In batch back propagation, all training items are examined first and then all the weights and bias values are adjusted. In stochastic back propagation (also called online back propagation), after each training item is examined, all weights and bias values are adjusted. In mini-batch back propagation, all weights and bias values are adjusted after examining a specified fraction of the training items. This program uses the most common variant, stochastic back propagation.

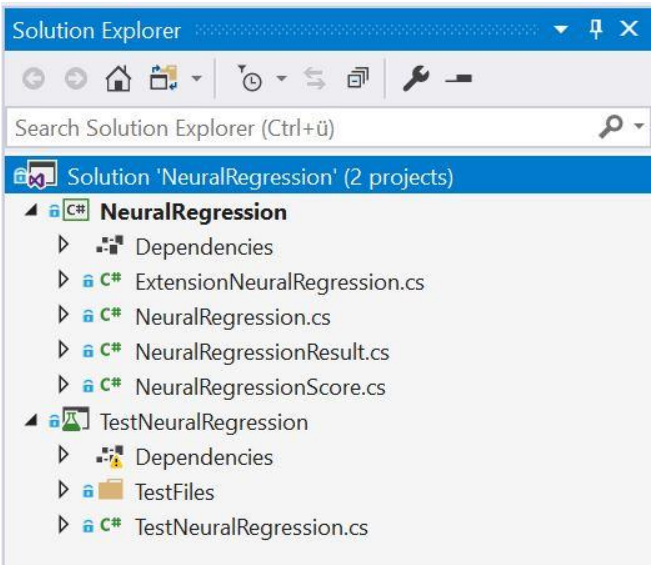
IV. Implementation of the Program

A. *Structure of the Program*

As a general rule when dealing with neural networks, the more training data it have, the better. For modeling the sine

function for x values between 0 and 2 * pi, it is needed at least 80 items to get good results. The choice of a seed value of 1 for the random number object was arbitrary. The training data is stored in an array-of-arrays-style matrix. I have also provided the training data from outside in excel file.

The Neural Network Regression algorithm is built on Microsoft.NET Core platform and is based on LearningApi (version – 1.2.3) project.
The **NeuralRegression** solution contains implementation and unit test projects, namely : **NeuralRegression** and **TestNeuralRegression**



B. Implementation overview

The main class is NeuralRegression which implements IAlgorithm interface from the LearningApi.

The neural network is created by these statements:

```
public NeuralRegression(double learningRate, int iterations)
{
    int numInput = 1;
    int numHidden = 12;
    int numOutput = 1;
    m_Alpha = learningRate;
    m_Iterations = iterations;
```

There's only one input node because the target sine function accepts only a single value. For most neural network regression problems there will be several input nodes, one for each of the predictor-independent variables. In most neural network regression problems there's only a single output node, but it's possible to predict two or more numeric values.

C. Computing Output Values

Most of the key differences between an NN designed for classification and one designed for regression occur in the methods that compute output and train the model. The definition of class NeuralNetwork method ComputeOutputs begins with:

```
public double[] ComputeOutputs(double[] xValues)
{
    double[] hSums = new double[numHidden]; // hidden nodes sums scratch array
    double[] oSums = new double[numOutput]; // output nodes sums
```

The method accepts an array that holds the values of the predictor-independent variables. Local variables hSums and oSums are scratch arrays that hold preliminary (before activation) values of the hidden and output nodes. Next, the independent variable values are copied into the neural network's input nodes:

```
for (int i = 0; i < numInput; ++i) // copy x-values to inputs
    this.inputs[i] = xValues[i];
```

Then the preliminary values of the hidden nodes are calculated by multiplying each input value by its corresponding input-to-hidden weight, and accumulating:

```
for (int j = 0; j < numHidden; ++j)  // compute i-h sum of weights * inputs
    for (int i = 0; i < numInput; ++i)
        hSums[j] += this.inputs[i] * this.ihWeights[i][j]; // note +=

for (int j = 0; j < numHidden; ++j)  // add biases to input-to-hidden sums
    hSums[j] += this.hBiases[j];
```

The values of the hidden nodes are determined by applying the hidden node activation function to each preliminary sum:

```
for (int j = 0; j < numHidden; ++j)   // apply activation
    this.hiddens[j] = HyperTan(hSums[j]); // hard-coded
```

Next, the preliminary values of the output nodes are calculated by multiplying each hidden node value by its corresponding hidden-to-output weight, and accumulating:

```
for (int k = 0; k < numOutput; ++k)   // compute h-o sum of weights * hOutputs
    for (int j = 0; j < numHidden; ++j)
        oSums[k] += hiddens[j] * hoWeights[j][k];
```

Then the hidden node bias values are added:

```
for (int k = 0; k < numOutput; ++k)
    oSums[k] += oBiases[k];
```

Computing output node values for a regression network is exactly the same as computing output node values for a classifier network. For a regression network no activation function is applied. Therefore, method ComputeOutputs concludes by simply copying the values in the oSums scratch array directly to the output nodes:

```
Array.Copy(oSums, this.outputs, outputs.Length);  // copy without activation

double[] retResult = new double[numOutput]; // could define a GetOutputs
Array.Copy(this.outputs, retResult, retResult.Length);
return retResult;
```

For convenience, the values in the output nodes are also copied to a local return array so they can be easily accessed without calling a GetOutputs method of some sort.

When training an NN classifier using the back-propagation algorithm, the calculus derivatives of the two activation functions are used. For the hidden nodes the code looks like:

```
// 3. compute hidden nodes signals
for (int j = 0; j < numHidden; ++j)
{
    double sum = 0.0; // need sums of output signals times hidden-to-output weights
    for (int k = 0; k < numOutput; ++k)
    {
        sum += oSignals[k] * hoWeights[j][k];
    }
    double derivative = (1 + hiddens[j]) * (1 - hiddens[j]); // for tanh
    hSignals[j] = sum * derivative;
}
```

Then it updated the weights and biases of input to Hidden w eights, hidden biases, Hidden to update weights and Output biases. The code is given below:

```
// 1. update input-to-hidden weights
for (int i = 0; i < numInput; ++i)
{
    for (int j = 0; j < numHidden; ++j)
    {
        double delta = ihGrads[i][j] * learnRate;
        ihWeights[i][j] += delta;
        ihWeights[i][j] += ihPrevWeightsDelta[i][j] * momentum;
        ihPrevWeightsDelta[i][j] = delta; // save for next time
    }
}

// 2. update hidden biases
for (int j = 0; j < numHidden; ++j)
{
    double delta = hbGrads[j] * learnRate;
    hBiases[j] += delta;
    hBiases[j] += hPrevBiasesDelta[j] * momentum;
    hPrevBiasesDelta[j] = delta;
}
// 3. update hidden-to-output weights
for (int j = 0; j < numHidden; ++j)
{
    for (int k = 0; k < numOutput; ++k)
    {
        double delta = hoGrads[j][k] * learnRate;
        hoWeights[j][k] += delta;
        hoWeights[j][k] += hoPrevWeightsDelta[j][k] * momentum;
        hoPrevWeightsDelta[j][k] = delta;
    }
}

// 4. update output node biases
for (int k = 0; k < numOutput; ++k)
{
    double delta = obGrads[k] * learnRate;
    oBiases[k] += delta;
    oBiases[k] += oPrevBiasesDelta[k] * momentum;
    oPrevBiasesDelta[k] = delta;
}
```

V. UNIT TEST RESULT

Here trainData() is used to take the training data and also trained the algorithm. The training data as follows:

```
/// <summary>
/// Sample data generate randomely
/// </summary>
/// <returns></returns>
private double[][] getData()
{
    int numItems = 80;

    double[][] trainData = new double[numItems][];
    Random rnd = new Random(1);

    for (int i = 0; i < numItems; ++i)
    {
        double x = 6.4 * rnd.NextDouble(); // [0 to 2PI]
        double sx = Math.Sin(x);
        trainData[i] = new double[] { x, sx };
    }

    return trainData;
}
```

A. *Test 1: TestByData*

Learning rate has been considered as 0.005 and maxepochs as 10000. As the algorithm is trained with data set and it provides error and weights value due to call the function of api.UseNeuralRegression (0.005, 10000).The following output is measured

```
// calling neural regression using learning rate 0.005 and maxepochs 10000
api.UseNeuralRegression(0.005,10000);

var score = api.Run() as NeuralRegressionScore;

//Checking Weights value
Assert.Equal(-0.15044, Math.Round(score.Weights[0], 5));
Assert.Equal(-0.25534, Math.Round(score.Weights[1], 5));
Assert.Equal(-0.14371, Math.Round(score.Weights[2], 5));
Assert.Equal(0.14475, Math.Round(score.Weights[3], 5));

//Errors after every 2000 iteration.
Assert.Equal(0.01126, Math.Round(score.Errors[0], 5));
Assert.Equal(0.00461, Math.Round(score.Errors[1], 5));
Assert.Equal(0.00341, Math.Round(score.Errors[2], 5));
Assert.Equal(0.00439, Math.Round(score.Errors[3], 5));

//Sample input for Predict function
double[][] predictors = new double[4][];
predictors[0] = new double[] { Math.PI };
predictors[1] = new double[] { Math.PI / 2 };
predictors[2] = new double[] { 3 * Math.PI / 2.0 };
predictors[3] = new double[] { 6 * Math.PI };

var result = api.Algorithm.Predict(predictors, api.Context) as NeuralRegressionResult;

//Checking predict values
Assert.Equal(-0.04453, Math.Round(result.PredictedValues[0][0],5));
Assert.Equal(0.97319, Math.Round(result.PredictedValues[1][0],5));
Assert.Equal(-1.01134, Math.Round(result.PredictedValues[2][0],5));
Assert.Equal(5.0182, Math.Round(result.PredictedValues[3][0],5));
```

B. *Test 2: TestWithMaxIteration*

In this test the training data is used same as the last test but maxepochs has been changed to 80000. Due to the changing Maxepochs the following output is measured.

```
// calling neural regression using learning rate 0.005 and maxepochs 80000
api.UseNeuralRegression(0.005, 80000);

var score = api.Run() as NeuralRegressionScore;

//Checking Weights value
Assert.Equal(-0.19048, Math.Round(score.Weights[0], 5));
Assert.Equal(-0.62763, Math.Round(score.Weights[1], 5));
Assert.Equal(-0.14277, Math.Round(score.Weights[2], 5));
Assert.Equal(0.15139, Math.Round(score.Weights[3], 5));

//Errors after every 2000 iteration.
Assert.Equal(0.00136, Math.Round(score.Errors[0], 5));
Assert.Equal(0.00054, Math.Round(score.Errors[1], 5));
Assert.Equal(0.00072, Math.Round(score.Errors[2], 5));
Assert.Equal(0.00033, Math.Round(score.Errors[3], 5));
```

```
//Sample input for Predict function
double[][] predictors = new double[4][];
predictors[0] = new double[] { Math.PI };
predictors[1] = new double[] { Math.PI / 2 };
predictors[2] = new double[] { 3 * Math.PI / 2.0 };
predictors[3] = new double[] { 6 * Math.PI };

var result = api.Algorithm.Predict(predictors, api.Context) as NeuralRegressionResult;

//Checking the predicted values of Sine function
Assert.Equal(0.0005, Math.Round(result.PredictedValues[0][0], 5));
Assert.Equal(0.99374, Math.Round(result.PredictedValues[1][0], 5));
Assert.Equal(-0.99345, Math.Round(result.PredictedValues[2][0], 5));
Assert.Equal(5.945, Math.Round(result.PredictedValues[3][0], 5));
```

C. *Test 3: TestWithCSVData*

This time the training data is used by the directory path. The algorithm is trained with directory path data. The learning rate and iteration are same. The Output is measured as below:

```
,
/// <summary>
/// Takes sample data from CSV file and perform Neural Regression on that dataset.
/// </summary>
[Fact]
public void TestWithCSVData()
{
    var path = System.IO.Path.Combine(Directory.GetCurrentDirectory(), @"TestFiles\SampleData.csv");

    DataDescriptor desc = getDescriptor(); ;

    LearningApi api = new LearningApi(desc);

    api.UseCsvDataProvider(path, ',', false,1);


// Use mapper for data, which will extract (map) required columns
api.UseDefaultDataMapper();

api.UseNeuralRegression(0.005, 10000);

var score = api.Run() as NeuralRegressionScore;


//Checking Weights value
Assert.Equal(0.00658, Math.Round(score.Weights[0], 5));
Assert.Equal(-1.14974, Math.Round(score.Weights[1], 5));
Assert.Equal(0.10792, Math.Round(score.Weights[2], 5));
Assert.Equal(0.02510, Math.Round(score.Weights[3], 5));

//Errors after every 2000 iteration.
Assert.Equal(0.32654, Math.Round(score.Errors[0], 5));
Assert.Equal(0.32539, Math.Round(score.Errors[1], 5));
Assert.Equal(0.32095, Math.Round(score.Errors[2], 5));
Assert.Equal(0.32113, Math.Round(score.Errors[3], 5));
//Sample input for Predict function
double[][] predictors = new double[4][];
predictors[0] = new double[] { Math.PI };
predictors[1] = new double[] { Math.PI / 2 };
predictors[2] = new double[] { 3 * Math.PI / 2.0 };
predictors[3] = new double[] { 6 * Math.PI };

var result = api.Algorithm.Predict(predictors, api.Context) as NeuralRegressionResult;

//Checking the predicted values of Sine function
Assert.Equal(1.05398, Math.Round(result.PredictedValues[0][0], 5));
Assert.Equal(0.64755, Math.Round(result.PredictedValues[1][0], 5));
Assert.Equal(2.32493, Math.Round(result.PredictedValues[2][0], 5));
Assert.Equal(4.32282, Math.Round(result.PredictedValues[3][0], 5));
```

D. *Test 4: TestWithMaxLearningRate*

Now under TestWithMaxLearningRate test, the value of learning rate has been increased to .01 form .005 but the value of iteration rate kept the same as 10000. From the result we can see that both the error and weight value have been changed though I use the same excel value as input. The following results are found.

```
/// <summary>
/// Performs the Neural Regression with maximum learning rate
/// </summary>
[Fact]
public void TestWithMaxLearningRate()
{
    var path = System.IO.Path.Combine(Directory.GetCurrentDirectory(), @"TestFiles\SampleData.csv");

DataDescriptor desc = getDescriptor(); ;

LearningApi api = new LearningApi(desc);

api.UseCsvDataProvider(path, ',', false, 1);

// Use mapper for data, which will extract (map) required columns
api.UseDefaultDataMapper();

api.UseNeuralRegression(0.01, 10000);

var score = api.Run() as NeuralRegressionScore;

//Checking Weights value
Assert.Equal(0.00612, Math.Round(score.Weights[0], 5));
Assert.Equal(-1.62082, Math.Round(score.Weights[1], 5));
Assert.Equal(0.51902, Math.Round(score.Weights[2], 5));
Assert.Equal(0.02292, Math.Round(score.Weights[3], 5));

//Errors after every 2000 iteration.
Assert.Equal(0.32526, Math.Round(score.Errors[0], 5));
Assert.Equal(0.32625, Math.Round(score.Errors[1], 5));
Assert.Equal(0.31942, Math.Round(score.Errors[2], 5));
Assert.Equal(0.32239, Math.Round(score.Errors[3], 5));


//Sample input for Predict function
double[][] predictors = new double[4][];
predictors[0] = new double[] { Math.PI };
predictors[1] = new double[] { Math.PI / 2 };
predictors[2] = new double[] { 3 * Math.PI / 2.0 };
predictors[3] = new double[] { 6 * Math.PI };

var result = api.Algorithm.Predict(predictors, api.Context) as NeuralRegressionResult;

//Checking the predicted values of Sine function
Assert.Equal(1.10758, Math.Round(result.PredictedValues[0][0], 5));
Assert.Equal(0.64127, Math.Round(result.PredictedValues[1][0], 5));
Assert.Equal(2.35681, Math.Round(result.PredictedValues[2][0], 5));
Assert.Equal(4.25913, Math.Round(result.PredictedValues[3][0], 5));
```

E. *Test 5: TestWithMinIteration*

In this test, the iteration number has been changed to 8000 and the learning rate has kept the same value as .005. from the output we can observe that though I have used the same excel value as input and same learning rate but for changing the iteration value the value of weight and error has been changed. The following results are found.

```
/// <summary>
/// Performs the Neural Regression with minimum iteration rate
/// </summary>
[Fact]
public void TestWithMinimumIteration()
{
    var path = System.IO.Path.Combine(Directory.GetCurrentDirectory(), @"TestFiles\SampleData.csv");


DataDescriptor desc = getDescriptor(); ;

LearningApi api = new LearningApi(desc);

api.UseCsvDataProvider(path, ',', false, 1);

// Use mapper for data, which will extract (map) required columns
api.UseDefaultDataMapper();

api.UseNeuralRegression(0.005, 8000);

var score = api.Run() as NeuralRegressionScore;

//Checking Weights value
Assert.Equal(0.00699, Math.Round(score.Weights[0], 5));
Assert.Equal(-1.07739, Math.Round(score.Weights[1], 5));
Assert.Equal(0.11352, Math.Round(score.Weights[2], 5));
Assert.Equal(0.02667, Math.Round(score.Weights[3], 5));

//Errors after every 2000 iteration.
Assert.Equal(0.35317, Math.Round(score.Errors[0], 5));
Assert.Equal(0.32448, Math.Round(score.Errors[1], 5));
Assert.Equal(0.33701, Math.Round(score.Errors[2], 5));
Assert.Equal(0.32060, Math.Round(score.Errors[3], 5));
```

```
//Sample input for Predict function
double[][] predictors = new double[4][];
predictors[0] = new double[] { Math.PI };
predictors[1] = new double[] { Math.PI / 2 };
predictors[2] = new double[] { 3 * Math.PI / 2.0 };
predictors[3] = new double[] { 6 * Math.PI };

var result = api.Algorithm.Predict(predictors, api.Context) as NeuralRegressionResult;

//Checking the predicted values of Sine function
Assert.Equal(1.06164, Math.Round(result.PredictedValues[0][0], 5));
Assert.Equal(0.64727, Math.Round(result.PredictedValues[1][0], 5));
Assert.Equal(2.34583, Math.Round(result.PredictedValues[2][0], 5));
Assert.Equal(4.33913, Math.Round(result.PredictedValues[3][0], 5));
```

*F. **Test Summary***

Here, 05 (five) unit tests are performed based on training data set acquisition, learning rate and iteration. In 02 Unit test inputs are given by randomly generated data and for other 03 Unit tests inputs are given from excel sheet. Considering these parameters, algorithm provides different predicted values. To use the module, *api.UseNeuralRegression()* method have to followed. It has two parameters- learning rate and iteration. The function calculates errors and weights and also the predicted values for given test dataset. From the result we can observe that, for changing the parameters of learning rates and iteration the values of error and weights are changed.

VI. DISCUSSION

The computing world has a lot to gain from neural networks. Their ability to learn by example makes them very flexible and powerful. Furthermore there is no need to devise an algorithm in order to perform a specific task; i.e. there is no need to understand the internal mechanisms of that task. They are also very well suited for real time systems because of their fast response and computational times which are due to their parallel architecture.

Neural networks also contribute to other areas of research such as neurology and psychology. They are regularly used to model parts of living organisms and to investigate the internal mechanisms of the brain.

Perhaps the most exciting aspect of neural networks is the possibility that someday 'conscious' networks might be produced. There is a number of scientists arguing that consciousness is a 'mechanical' property and that 'conscious' neural networks are a realistic possibility.

ACKNOWLEDGEMENT

I humbly thank Frankfurt University of Applied Sciences and the professors to have guided us students and also paved way for hands on experience to learn and work with ML algorithms on Microsoft.NETCore platform and LearningApi.

References

[1] Symon haykin, “Neural Networks,” A comprehensive Foundation, 2nd Edition, 2005.

[2] Neural Network regression [online], Available: https://msdn.microsoft.com/en-us/magazine/mt683800.aspx [Accessed 2019]

[3] Getting Started with Neural Network for Regression [online], Available: https://medium.com/@rajatgupta310198/getting-started-with-neural-network-for-regression-and-tensorflow-58ad3bd75223 [Accessed 2019]

[4] A beginneers Guide to Neural Network and Deep Learning [online], Available: https://skymind.ai/wiki/neural-network [Accessed 2019]

[5] Christos Stergiou and Dimitrios Siganos, “Neural Networks”, vol4, cs11

[6] O. Source, "UniversityOfAppliedSciencesFrankfurt/LearningApi: Machine Learning foundation on top of .NET Core," 2019. [Online]. Available: https://github.com/UniversityOfAppliedSciencesFrankfurt/LearningApi. [Accessed 2019].