Soft Computing Methods and Applications

Lab Exercise and Assignment 12 (2020)

Develop an MLP application system that can deal with .cal data set.

(1) Analyze the data structure requirements of an MLP to design a class for the MLP neural network system with the following capabilities:

(a) can read in a .cal data set file.

(b) can configure an NN based on user's specification of hidden neurons and the read-in data set.

(c) can normalize values of each data field of the training and testing data sets.

(d) can randomly shuffle the data instances in the data sets to generate different sets of training data and testing data.

(e) can perform an epoch of data training and report the root mean square of the error.

(f) can test the trained NN using the testing data and report the correctness based on the classification confusing table.

(g) can perform a simple forward computation using raw input vector (normalized by your code) and return raw output vector (converting back).

(2) Add graphics display for user to visualize the structure of the NN and the process of training.

(3) Prepare a folder named as <your ID><your name>Ass12 to put your source code in it. Compress it as an rar file; submit the rar file to course web site.

Appendix: sample code snippets

```
namespace MultiLayerPercetronNeuralNetwork
{
    class BbackPropagationMLP
    {
        float[ ][ ] x;              // neuron values
        float[ ][ ][ ] w;           // weights
        float[ ][ ] e;              // epsilon; partial derivative of error with respect to net value.

        int[ ] n;                           // numbers of neuron on layers
        int inputDimension;     // dimension of input vector
        int inputNumber;            // number of instances on the data set
        int numberOfTrainningVectors;       // number of instances that are serving as training data
        float[,] originalInputs;        // original instances of input vectors (without normalization)
        float[,] inputs;                    // normalized input vectors
        float[ ] inputMax;              // upper bounds on all components of input vectors
        float[ ] inputMin;              // lower bounds on all components of input vectors
        int inputWidth;                 // dimension in width for a two-dimensional input vector

        int targetDimension;            // dimension of target vector
        float[,] originalTargets;   // original instances of target vectors (without normalization)
        float[,] targets;                   // normalized target vectors
        float[ ] targetMax;             // upper bounds on all components of target vectors
        float[ ] targetMin;             // lower bounds on all components of target vectors.

        int[ ] vectorIndices;           // array of shuffled indices of data instances; the front portion is training vectors;
        //the rear portion is testing vectors
        float rootMeanSquareError = 0.0f;       // root mean square of error for an epoch of data training
        int layerNumber;                                // number of neuron layer (including the input layer)

        Random randomizer = new Random( );

        float learningRate = 0.999f;        // learning rate, specified by the user
        /// <summary>
        /// The factor of reducing the eta epoch by epoch. That is
        /// eta <-- LearningRate  * eta
        /// </summary>
        public float LearningRate
        {
            get { return learningRate; }
            set { learningRate = value; }
        }

        float eta;                          // step size that specify the update amount on each weight
        float initialEta = 0.7f;        // initial step size, specified by the user
```

```
/// <summary>
///  Initialize  variable of the eta (can be regarded as step size).
/// </summary>
public float InitialEta
{
    set { initialEta = value; }
    get { return initialEta; }
}

/// <summary>
///  Current root mean square after an epoch training.
/// </summary>
public float RootMeanSquareError
{
    get { return rootMeanSquareError; }          //set { rootMeanSquare = value; }
}


/// <summary>
///  Read in the data set from the given file stream. Configure the portions of training
///  and testing data subsets. Original data are stored, bounds on each component of
///  input vector and target vector are founds, and normalized data set is prepared.
/// </summary>
/// <param name="sr">file stream</param>
/// <param name="trainingRatio">portion of trainning data</param>
public void ReadInDataSet( StreamReader sr, float trainingRatio )
{
    char[ ] separators = new char[ ] { ',', ' ' };
    string s = sr.ReadLine( );
    string[ ] items = s.Split( separators, StringSplitOptions.RemoveEmptyEntries );

    inputNumber = Convert.ToInt32( items[0] );
    inputDimension = Convert.ToInt32( items[1] );
    targetDimension = Convert.ToInt32( items[2] );
    inputWidth = Convert.ToInt32( items[3] );
    …
}


/// <summary>
///  Configure the topology of the NN with the user specified numbers of hidden
///  neuorns and layers.
/// </summary>
/// <param name="hiddenNeuronNumbers">list of numbers of neurons of hidden layers</param>
public void ConfigureNeuralNetwork( int[ ] hiddenNeuronNumbers )
{
```

```
    layerNumber = hiddenNeuronNumbers.Length + 2;
    n = new int[layerNumber];
    n[0] = inputDimension + 1;
    n[layerNumber - 1] = targetDimension + 1;
    …
}


/// <summary>
/// Randomly shuffle the orders of the data in the data set.
/// </summary>
private void RandomizeIndices( )
{
    …
}


/// <summary>
/// Randomly set values of weights between [-1,1] and randomly shuffle the orders of all
/// the datum in the data set. Reset value of initial eta and root mean square to 0.0.
/// </summary>
public void ResetWeightsAndInitialCondition( )
{
    …
}


/// <summary>
///  Sequentially loop through each training datum of the training data whose indices are
///  randomly shuffled in vectorIndices[] array, to perform on-line training of the NN.
/// </summary>
public void TrainAnEpoch( )
{
    float v;
    float errorSquareSum = 0.0f;
    float sumation = 0.0f;
    int layerNumberMinusOne = layerNumber - 1;

    /// forward computing for all neuro values.
    …

        /// compute the epsilon values for neurons on the output layer
    …

        /// backward computing for the epsilon values
    …
```

```
            /// update weights for all weights by using epsilon and neuron values.
        …

        /// update step size of the updating amount
        …
    }



    /// <summary>
    ///  Compute the output vector for an input vector. Both vectors are in the raw
    ///  format. The input vector is subject to scaling first before forward computing.
    ///  Output vector is then scaled back to raw format for recognition.
    /// </summary>
    /// <param name="input">input vector in raw format</param>
    /// <returns>output vector in raw format</returns>
    public float[ ] ComputeResults( float[ ] input )
    {
        float[ ] results = null;
        float v;
        results = new float[targetDimension];
        …
        return results;
    }

    /// <summary>
    /// If the data set is a classification data set, test the data to generate confusing table.
    /// The index of the largest component of the target vector is the targeted class id.
    /// The index of the largest component of the computed output vector is the resulting class id.
    /// If both the targeted class id and the resulting class id are the same, then the test
    /// data is correctly classified.
    /// </summary>
    /// <param name="confusingTable">generated confusing table</param>
    /// <returns>the ratio between the number of correctly classified testing data and the total number of testing
data.</returns>
    public float TestingClassification( out int[,] confusingTable )
    {
        confusingTable = new int[targetDimension, targetDimension];

        int successedCount = 0;

        float v;
        …

        return ( float )successedCount / ( float )( inputNumber - numberOfTrainningVectors );
    }
```

```
        }
}
```