HERITAGE INSTITUTE OF TECHNOLOGY

Implementation of Multi-Layer Perceptron.

Authors

UPASANA DUTTA (1551114)

ARIJIT DUTTA (1551131)

Objective -

The objective of this project is to implement a Multi-Layer Perceptron using Python programming Language and use it for classification purpose.

Wine Data Set

The data has been taken from the **UCI Machine Learning Repository**.

Link - https://archive.ics.uci.edu/ml/datasets/wine

These data are the results of a chemical analysis of wines grown in the same region in Italy but derived from three different cultivars. The analysis determined the quantities of 13 constituents found in each of the three types of wines. All attributes are continuous.

There are 178 rows of data.

The attributes are -

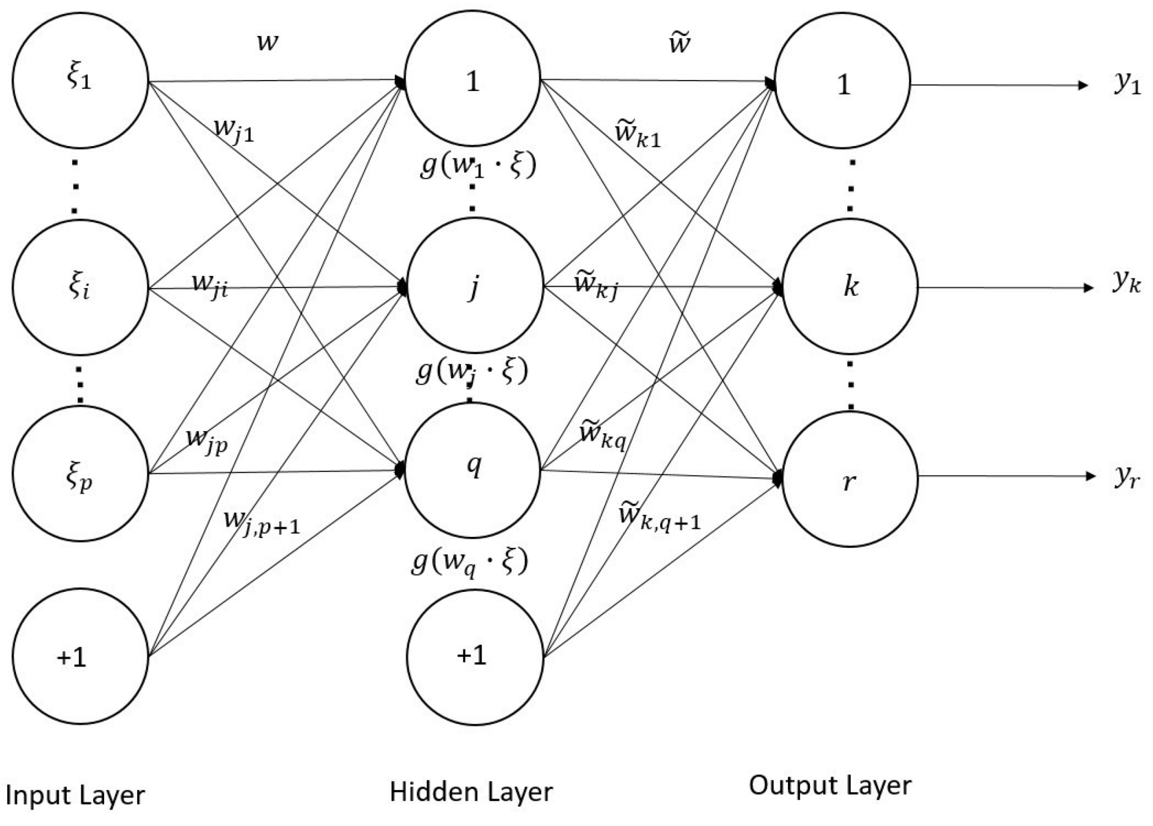
1) Alcohol
2) Malic acid
3) Ash
4) Alcalinity of ash
5) Magnesium
6) Total phenols
7) Flavanoids
8) Nonflavanoid phenols
9) Proanthocyanins
10) Color intensity
11) Hue
12) OD280/OD315 of diluted wines
13) Proline

Introduction -

An Artificial Neural Network (ANN) is a computational model that is inspired by the way biological neural networks in the human brain process information.

- A single neuron - The basic unit of computation in a neural network is the **neuron**, often called a **node** or **unit**. It receives input from some other nodes, or from an external source and computes an output. Each input has an associated **weight**(w), which is assigned on the basis of its relative importance to other inputs. The node applies a function ***f*** to the weighted sum of its inputs and generates an output.

- Feedforward Neural Network - It contains multiple neurons (nodes) arranged in **layers**. Nodes from adjacent layers have **connections** or **edges** between them. All these connections have **weights** associated with them.



- Multi-layer Perceptron - A Multi Layer Perceptron has one or more hidden layers. While a single layer perceptron can only learn linear functions, a multi layer perceptron can also learn non – linear functions.

  - **Supervised Learning** - It is a supervised training scheme, which means it learns from labeled training data. The goal of learning is to assign correct weights for the edges between the layers. Given an input vector, these weights determine what the output vector is.

  - **Backpropagation Algorithm** - Initially all the edge weights are randomly assigned. For every input in the training dataset, the ANN is activated and its output is observed. This output is compared with the desired output that we already know, and the error is propagated back to the previous layer. This error is noted and the weights are adjusted accordingly.

Steps-by-step process -

1. Initialize the Neural Network.

- Each neuron has a set of input weights that need to be maintained - one weight for each input connection and an additional weight for the bias.

- We will need to store additional properties for a neuron during training, therefore we will use a dictionary to represent each neuron and store properties by names such as 'weights' for the weights (random number between 0 and 1).

- We will organize layers as lists of dictionaries and treat the whole network as an array of layers.

- We write a function named **initialize_network()** that creates a neural network for training. It accepts three parameters, the number of inputs, the number of neurons to have in the hidden layer and the number of outputs.
- Each neuron in the hidden layer has **n_inputs + 1** weights, one for each input and an additional one for the bias.
- Each neuron in the output layer has **n_hidden + 1** weights, one for each neuron in the hidden layer, and an additional one for the bias.

2. Forward Propagate

We can break forward propagation down into three parts:

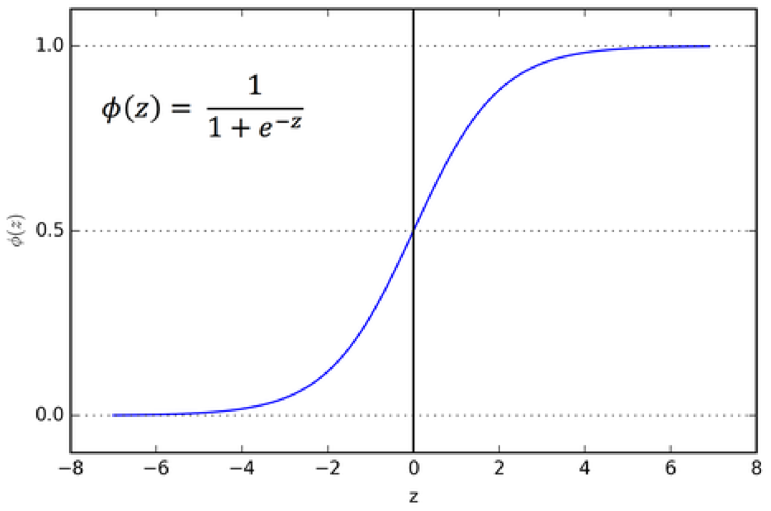
1.Neuron Activation.
2.Neuron Transfer.
3.Forward Propagation.

2.1. Neuron activation - calculated as the weighted sum of the inputs.

activation = sum(weight_i * input_i) + bias

2.2. Neuron Transfer – We will use Sigmoid function as our transfer function.

output = 1 / (1 + e^(-activation))



2.3. Forward Propagation - All of the outputs from one layer become inputs to the neurons on the next layer.

3. Back Propagate Error

This part is broken down into two sections.

1.Transfer Derivative.
2.Error Backpropagation.

3.1. Transfer Derivative

We are using the sigmoid transfer function, the derivative of which can be calculated as follows:
transfer_derivative = output * (1.0 – output)

3.2. Error Backpropagation

The error for a given neuron can be calculated as follows -

error = expected - output
delta = error * transfer_derivative(output)

This error signal calculation is used for neurons in the output layer.

The error signal for a neuron in the hidden layer is calculated as the weighted error of each neuron in the output layer -

error = (weight_k * delta_j)

delta = error * transfer_derivative(output)

Where **delta_j** is the error signal from the **j**th neuron in the output layer, **weight_k** is the weight that connects the **k**th neuron to the current neuron and output is the output for the current neuron.

4. Train Network

This part is broken down into two sections:

1.Update Weights.
2.Train Network.

4.1. Update Weights

Network weights are updated as follows -

w(t+1)= w(t) + learning_rate * delta(t) * input(t)

For bias term -

bias(t+1) = bias(t) + learning_rate * delta(t)

Learning rate controls how much to change the weight to correct for the error. For example, a value of 0.1 will update the weight 10% of the amount that it possibly could be updated. Small learning rates are preferred that cause slower learning over a large number of training iterations. This increases the likelihood of the network finding a good set of weights across all layers rather than the fastest set of weights that minimize error.

4.2. Train Network

This involves -

- Multiple iterations of exposing a training dataset to the network
- For each row of data forward propagating the inputs
- Backpropagating the error
- Updating the network weights

The network is updated using stochastic gradient descent - looping for a fixed number of epochs and within each epoch updating the network for each row in the training dataset.

The expected number of output values is used to transform class values in the training data into a one hot encoding. That is a binary vector with one column for each class value to match the output of the network. This is required to calculate the error for the output layer.

The sum squared error between the expected output and the network output is accumulated each epoch and printed. This is helpful to create a trace of how much the network is learning and improving each epoch.

i) <u>Normalisation</u>

- We find out the min value and the max value of each of the attributes from the dataset.
- Then, we normalise each value using the formula -

value = (value - min) / (max – min)

This normalised value will always lie between 0 and 1.

ii) <u>Cross-validation method for evaluation</u>

To evaluate the algorithm we will use Cross-validation method.

The general procedure is as follows -

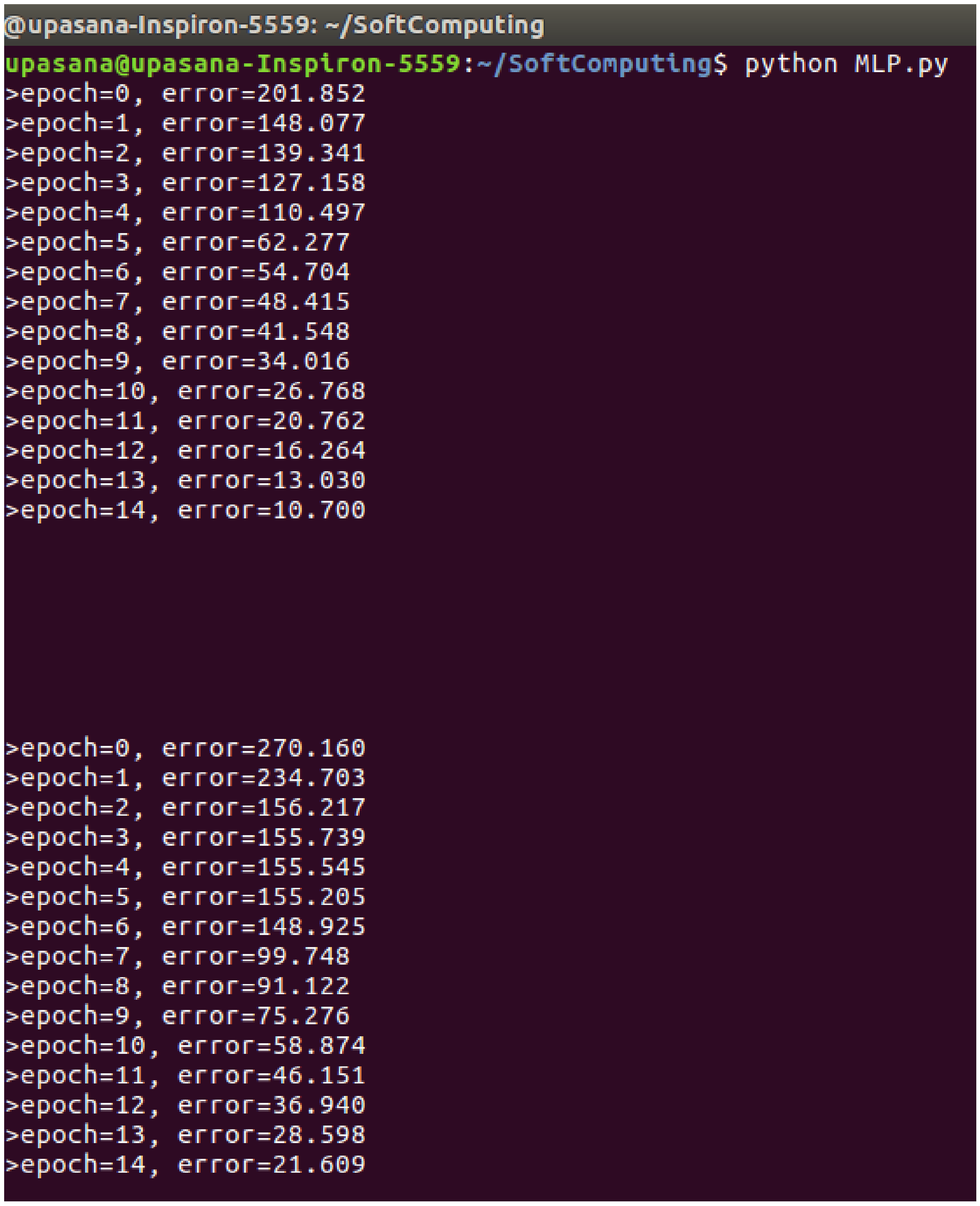
1. Split the dataset into k random groups
2. For each unique group:

- ➢ Take the group as the test data set
- ➢ Take the remaining groups as a training data set
- ➢ Fit a model on the training set and evaluate it on the test set
- ➢ Retain the evaluation score and discard the model
- ➢ Summarize the skill of the model using the sample of model evaluation scores

<u>Accuracy</u>

The accuracy of our algorithm will be the mean accuracy of all the runs done by taking different random sets of training and testing data.

Results



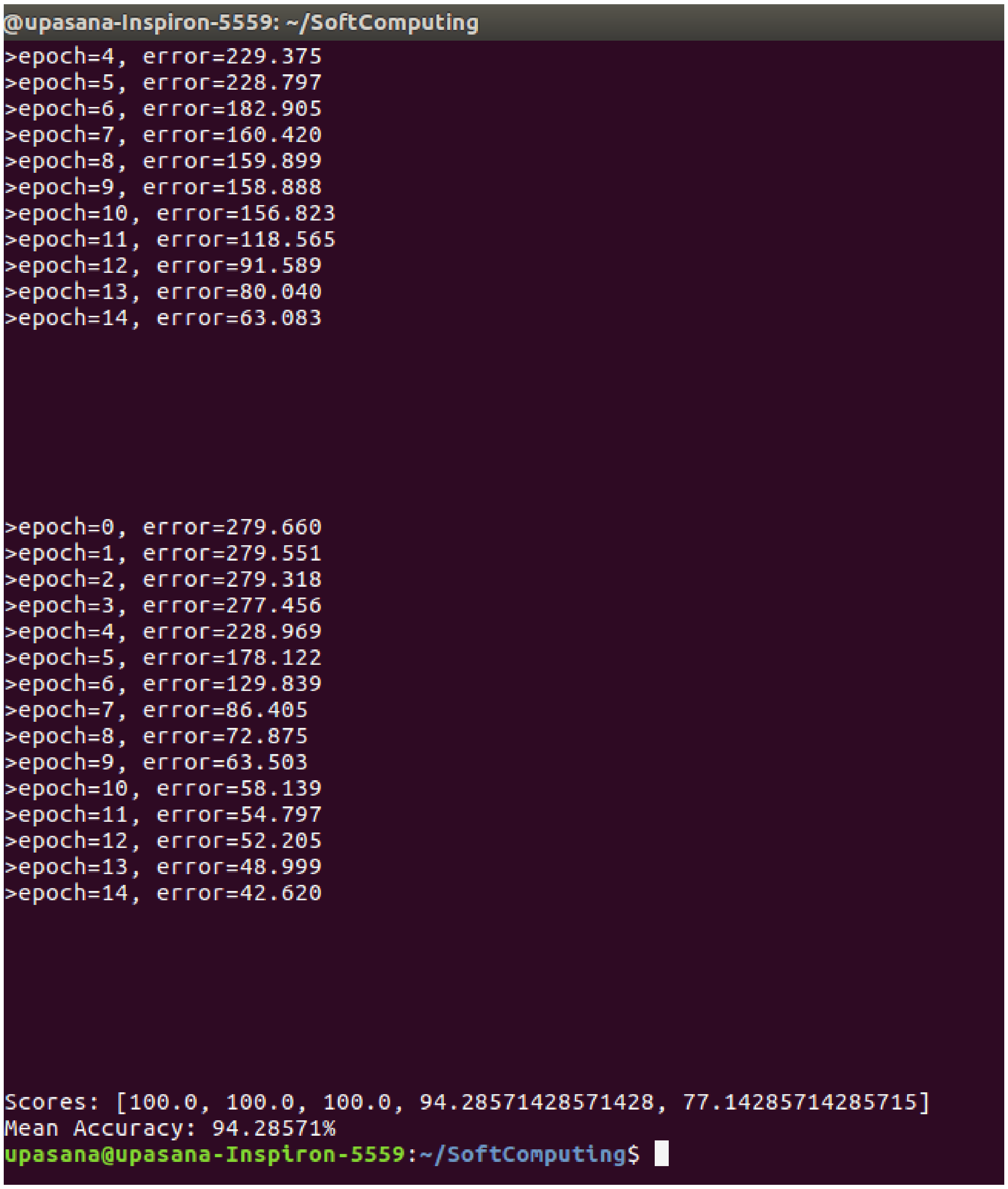
```
@upasana-Inspiron-5559: ~/SoftComputing
upasana@upasana-Inspiron-5559:~/SoftComputing$ python MLP.py
>epoch=0, error=201.852
>epoch=1, error=148.077
>epoch=2, error=139.341
>epoch=3, error=127.158
>epoch=4, error=110.497
>epoch=5, error=62.277
>epoch=6, error=54.704
>epoch=7, error=48.415
>epoch=8, error=41.548
>epoch=9, error=34.016
>epoch=10, error=26.768
>epoch=11, error=20.762
>epoch=12, error=16.264
>epoch=13, error=13.030
>epoch=14, error=10.700







>epoch=0, error=270.160
>epoch=1, error=234.703
>epoch=2, error=156.217
>epoch=3, error=155.739
>epoch=4, error=155.545
>epoch=5, error=155.205
>epoch=6, error=148.925
>epoch=7, error=99.748
>epoch=8, error=91.122
>epoch=9, error=75.276
>epoch=10, error=58.874
>epoch=11, error=46.151
>epoch=12, error=36.940
>epoch=13, error=28.598
>epoch=14, error=21.609
```



```
@upasana-Inspiron-5559: ~/SoftComputing
>epoch=4, error=229.375
>epoch=5, error=228.797
>epoch=6, error=182.905
>epoch=7, error=160.420
>epoch=8, error=159.899
>epoch=9, error=158.888
>epoch=10, error=156.823
>epoch=11, error=118.565
>epoch=12, error=91.589
>epoch=13, error=80.040
>epoch=14, error=63.083







>epoch=0, error=279.660
>epoch=1, error=279.551
>epoch=2, error=279.318
>epoch=3, error=277.456
>epoch=4, error=228.969
>epoch=5, error=178.122
>epoch=6, error=129.839
>epoch=7, error=86.405
>epoch=8, error=72.875
>epoch=9, error=63.503
>epoch=10, error=58.139
>epoch=11, error=54.797
>epoch=12, error=52.205
>epoch=13, error=48.999
>epoch=14, error=42.620







Scores: [100.0, 100.0, 100.0, 94.28571428571428, 77.14285714285715]
Mean Accuracy: 94.28571%
upasana@upasana-Inspiron-5559:~/SoftComputing$
```

Observation

- The accuracy is 94.28% .
- With each epoch, the error of the model reduces.
- Increasing the number of epochs increases the accuracy of the model.
- We get the best results when number of epochs are high and the learning rate is low. This increases the likelihood of the network finding a good set of weights across all layers rather than the fastest set of weights that minimize error.