Name: Tania Rajabally

Roll No: 43

UID: 2017130047

Batch: C

Course Code: CEL71 (AI and Soft Computing Lab)

Date:16/09/2020

**Experiment No 5**

**Designing An Expert System**

**Aim:** To implement BPN to classify IRIS dataset ( Iris Setosa, Iris Versicolour, or Iris Virginica.)

**Theory:**

A multilayer perceptron (MLP) is a class of feedforward artificial neural network (ANN). The term MLP is used ambiguously, sometimes loosely to any feedforward ANN, sometimes strictly to refer to networks composed of multiple layers of perceptrons (with threshold activation). Multilayer perceptrons are sometimes colloquially referred to as "vanilla" neural networks, especially when they have a single hidden layer.

An MLP consists of at least three layers of nodes: an input layer, a hidden layer and an output layer. Except for the input nodes, each node is a neuron that uses a nonlinear activation function. MLP utilizes a supervised learning technique called backpropagation for training. Its multiple layers and non-linear activation distinguish MLP from a linear perceptron. It can distinguish data that is not linearly separable.

The Backpropagation neural network is a multilayered, feedforward neural network and is by far the most extensively used. It is also considered one of the simplest and most general methods used for supervised training of multilayered neural networks. Backpropagation works by approximating the non-linear relationship between the input and the output by adjusting the weight values internally. It can further be generalized for the input that is not included in the training patterns (predictive abilities).

Generally, the Backpropagation network has two stages, training and testing. During the training phase, the network is "shown" sample inputs and the correct classifications. For example, the input might be an encoded picture of a face, and the output could be represented by a code that corresponds to the name of the person.

The operations of the Backpropagation neural networks can be divided into two steps: feedforward and Backpropagation. In the feedforward step, an input pattern is applied to the input layer and its effect propagates, layer by layer, through the network until an output is produced. The network's actual output value is then compared to the expected output, and an error signal is computed for each of the output nodes. Since all the hidden nodes have, to some degree, contributed to the errors evident in the output layer, the output error signals are transmitted backwards from the output layer to each node in the hidden layer that immediately contributed to the output layer. This process is then repeated, layer by layer, until each node in the network has received an error signal that describes its relative contribution to the overall error.

Once the error signal for each node has been determined, the errors are then used by the nodes to update the values for each connection weights until the network converges to a state that allows all the training patterns to be encoded. The Backpropagation algorithm looks for the minimum value of the error function in weight space using a technique called the delta rule or gradient descent[2]. The weights that minimize the error function is then considered to be a solution to the learning problem.

**Procedure:**

The weights in the network can be updated from the errors calculated for each training example and this is called online learning. It can result in fast but also chaotic changes to the network.

Alternatively, the errors can be saved up across all of the training examples and the network can be updated at the end. This is called batch learning and is often more stable.

Typically, because datasets are so large and because of computational efficiencies, the size of the batch, the number of examples the network is shown before an update is often reduced to a small number, such as tens or hundreds of examples.

The amount that weights are updated is controlled by a configuration parameters called the learning rate. It is also called the step size and controls the step or change made to network weight for a given error. Often small weight sizes are used such as 0.1 or 0.01 or smaller.

The update equation can be complemented with additional configuration terms that you can set.

Momentum is a term that incorporates the properties from the previous weight update to allow the weights to continue to change in the same direction even when there is less error being calculated.

Learning Rate Decay is used to decrease the learning rate over epochs to allow the network to make large changes to the weights at the beginning and smaller fine tuning changes later in the training schedule.

Once a neural network has been trained it can be used to make predictions.

You can make predictions on test or validation data in order to estimate the skill of the model on unseen data. You can also deploy it operationally and use it to make predictions continuously.

The network topology and the final set of weights is all that you need to save from the model. Predictions are made by providing the input to the network and performing a forward-pass allowing it to generate an output that you can use as a prediction.

**Code:**

**from random import seed**

**from random import randrange**

**from random import random**

**from csv import reader**

**from math import exp**

**def load\_csv(filename):**

**dataset = list()**

**with open(filename, 'r') as file:**

**csv\_reader = reader(file)**

**for row in csv\_reader:**

**if not row:**

**continue**

**dataset.append(row)**

**return dataset**

**from google.colab import drive**

**drive.mount('/content/gdrive')**

**%cd "/content/gdrive/My Drive/AISC"**

**filename = 'iris.csv'**

**dataset = load\_csv(filename)**

**def str\_column\_to\_float(dataset, column):**

**for row in dataset:**

**row[column] = float(row[column].strip())**

**for i in range(len(dataset[0])-1):**

**str\_column\_to\_float(dataset, i)**

**def str\_column\_to\_int(dataset, column):**

**class\_values = [row[column] for row in dataset]**

**unique = set(class\_values)**

**lookup = dict()**

**for i, value in enumerate(unique):**

**lookup[value] = i**

**for row in dataset:**

**row[column] = lookup[row[column]]**

**return lookup**

**str\_column\_to\_int(dataset, len(dataset[0])-1)**

**def dataset\_minmax(dataset):**

**minmax = list()**

**stats = [[min(column), max(column)] for column in zip(\*dataset)]**

**return stats**

**def normalize\_dataset(dataset, minmax):**

**for row in dataset:**

**for i in range(len(row)-1):**

**row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] - minmax[i][0])**

**minmax = dataset\_minmax(dataset)**

**normalize\_dataset(dataset, minmax)**

**n\_inputs = len(dataset[0]) - 1**

**n\_inputs**

**n\_outputs = len(set([row[-1] for row in dataset]))**

**n\_outputs**

**l\_rate = 0.3**

**n\_epoch = 500**

**n\_hidden = 5**

**def initialize\_network(n\_inputs, n\_hidden, n\_outputs):**

**network = list()**

**hidden\_layer = [{'weights':[random() for i in range(n\_inputs + 1)]} for i in range(n\_hidden)]**

**network.append(hidden\_layer)**

**output\_layer = [{'weights':[random() for i in range(n\_hidden + 1)]} for i in range(n\_outputs)]**

**network.append(output\_layer)**

**return network**

**network = initialize\_network(n\_inputs, n\_hidden, n\_outputs)**

**def activate(weights, inputs):**

**activation = weights[-1]**

**for i in range(len(weights)-1):**

**activation += weights[i] \* inputs[i]**

**return activation**

**def transfer(activation):**

**return 1.0 / (1.0 + exp(-activation))**

**def forward\_propagate(network, row):**

**inputs = row**

**for layer in network:**

**new\_inputs = []**

**for neuron in layer:**

**activation = activate(neuron['weights'], inputs)**

**neuron['output'] = transfer(activation)**

**new\_inputs.append(neuron['output'])**

**inputs = new\_inputs**

**return inputs**

**def transfer\_derivative(output):**

**return output \* (1.0 - output)**

**def backward\_propagate\_error(network, expected):**

**for i in reversed(range(len(network))):**

**layer = network[i]**

**errors = list()**

**if i != len(network)-1:**

**for j in range(len(layer)):**

**error = 0.0**

**for neuron in network[i + 1]:**

**error += (neuron['weights'][j] \* neuron['delta'])**

**errors.append(error)**

**else:**

**for j in range(len(layer)):**

**neuron = layer[j]**

**errors.append(expected[j] - neuron['output'])**

**for j in range(len(layer)):**

**neuron = layer[j]**

**neuron['delta'] = errors[j] \* transfer\_derivative(neuron['output'])**

**def update\_weights(network, row, l\_rate):**

**for i in range(len(network)):**

**inputs = row[:-1]**

**if i != 0:**

**inputs = [neuron['output'] for neuron in network[i - 1]]**

**for neuron in network[i]:**

**for j in range(len(inputs)):**

**neuron['weights'][j] += l\_rate \* neuron['delta'] \* inputs[j]**

**neuron['weights'][-1] += l\_rate \* neuron['delta']**

**def train\_network(network, train, l\_rate, n\_epoch, n\_outputs):**

**for epoch in range(n\_epoch):**

**sum\_error = 0**

**for row in train:**

**outputs = forward\_propagate(network, row)**

**expected = [0 for i in range(n\_outputs)]**

**expected[row[-1]] = 1**

**sum\_error += sum([(expected[i]-outputs[i])\*\*2 for i in range(len(expected))])**

**backward\_propagate\_error(network, expected)**

**update\_weights(network, row, l\_rate)**

**print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l\_rate, sum\_error))**

**train\_network(network, dataset, l\_rate, n\_epoch, n\_outputs)**

**for layer in network:**

**print(layer)**

**def predict(network, row):**

**outputs = forward\_propagate(network, row)**

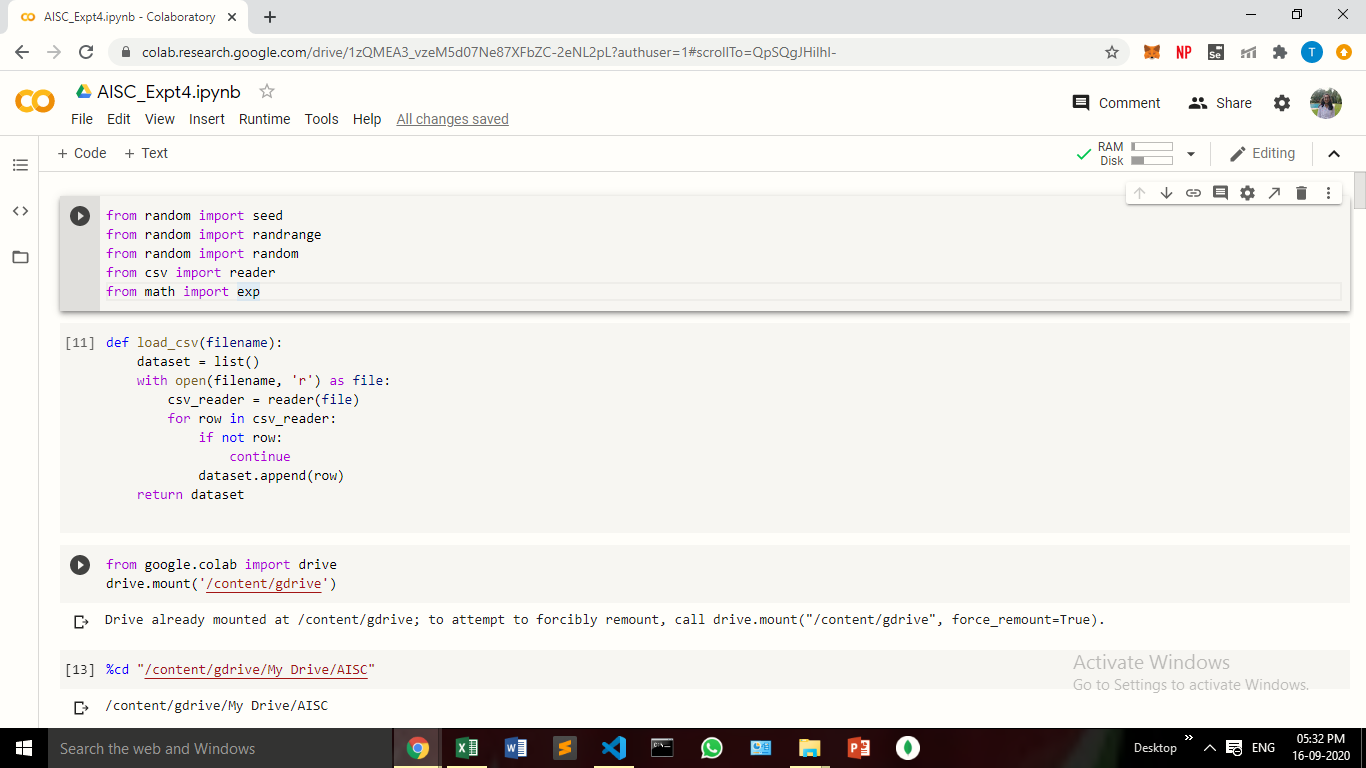
**return outputs.index(max(outputs))**

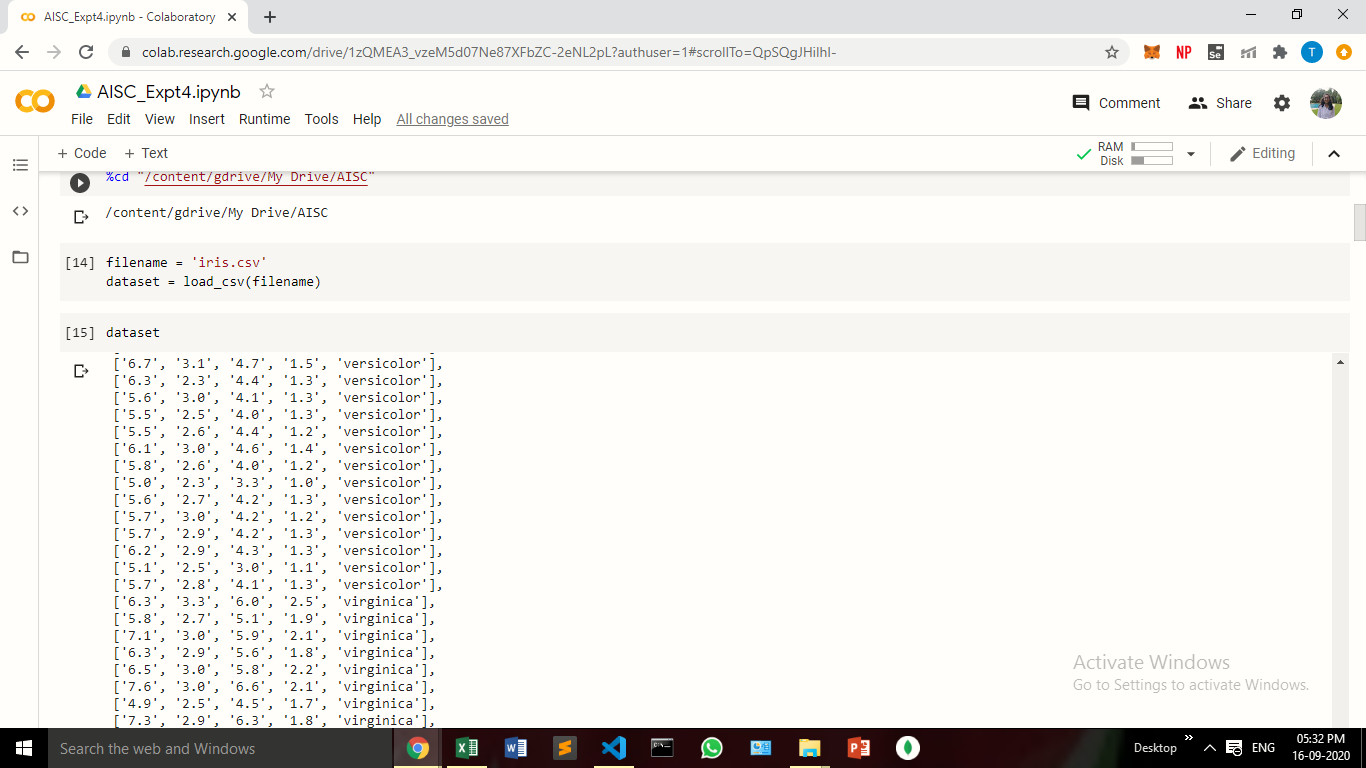
**new = dataset[2]**

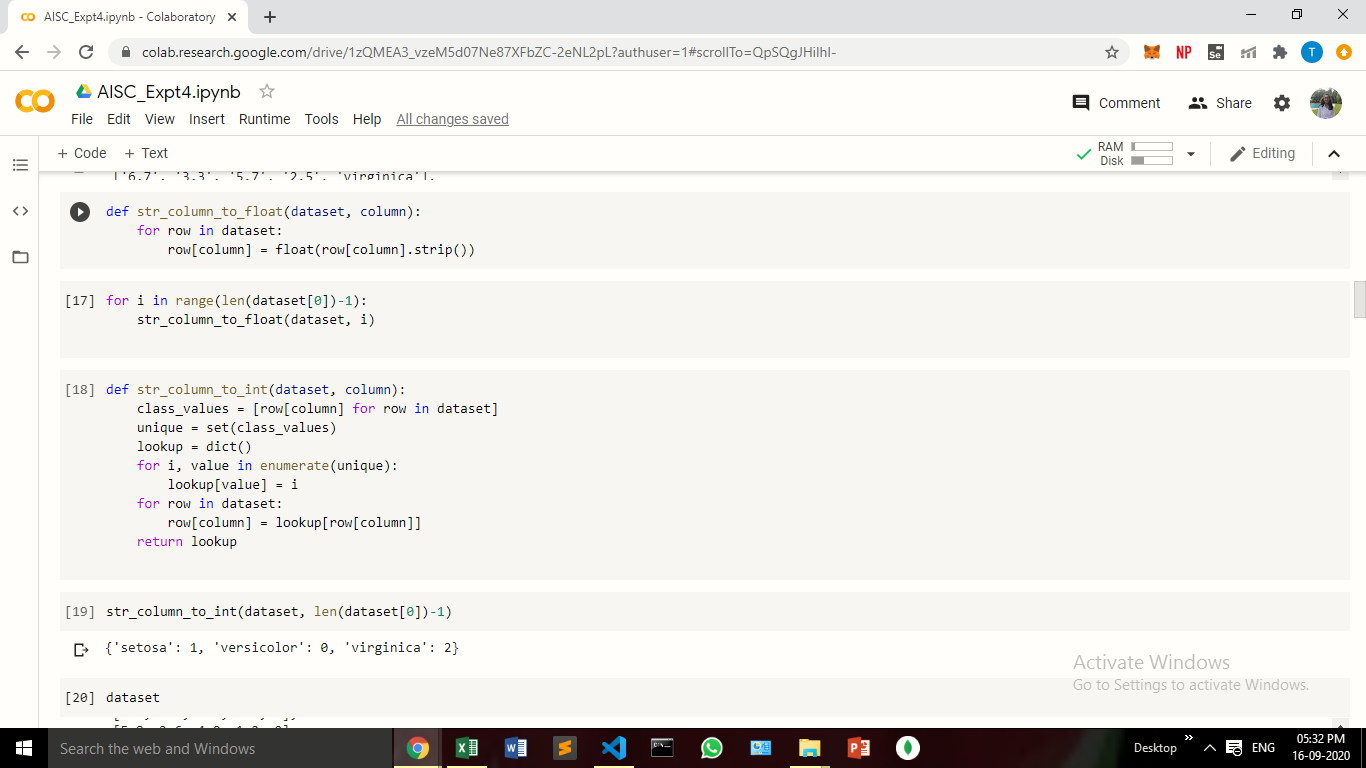
**prediction = predict(network, new)**

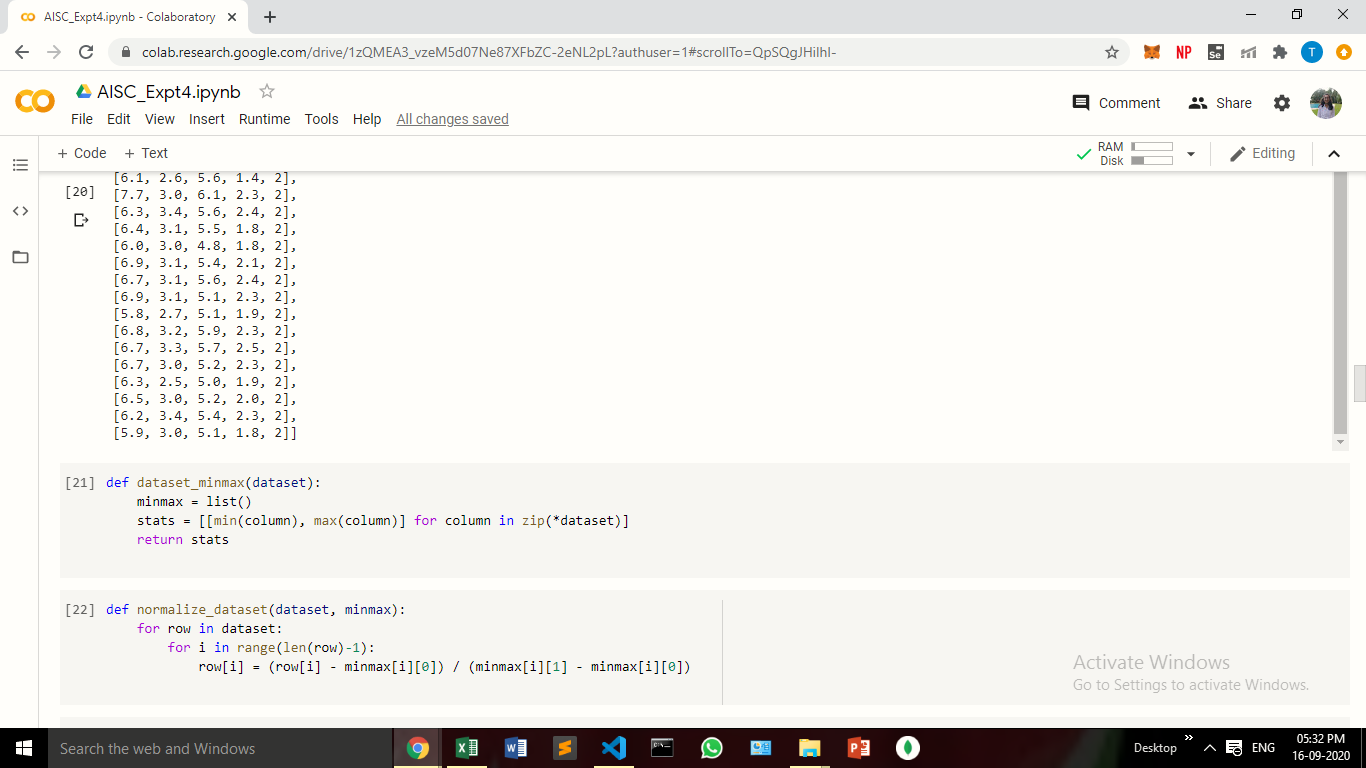
**print('Expected=%d, Got=%d' % (new[-1], prediction))**

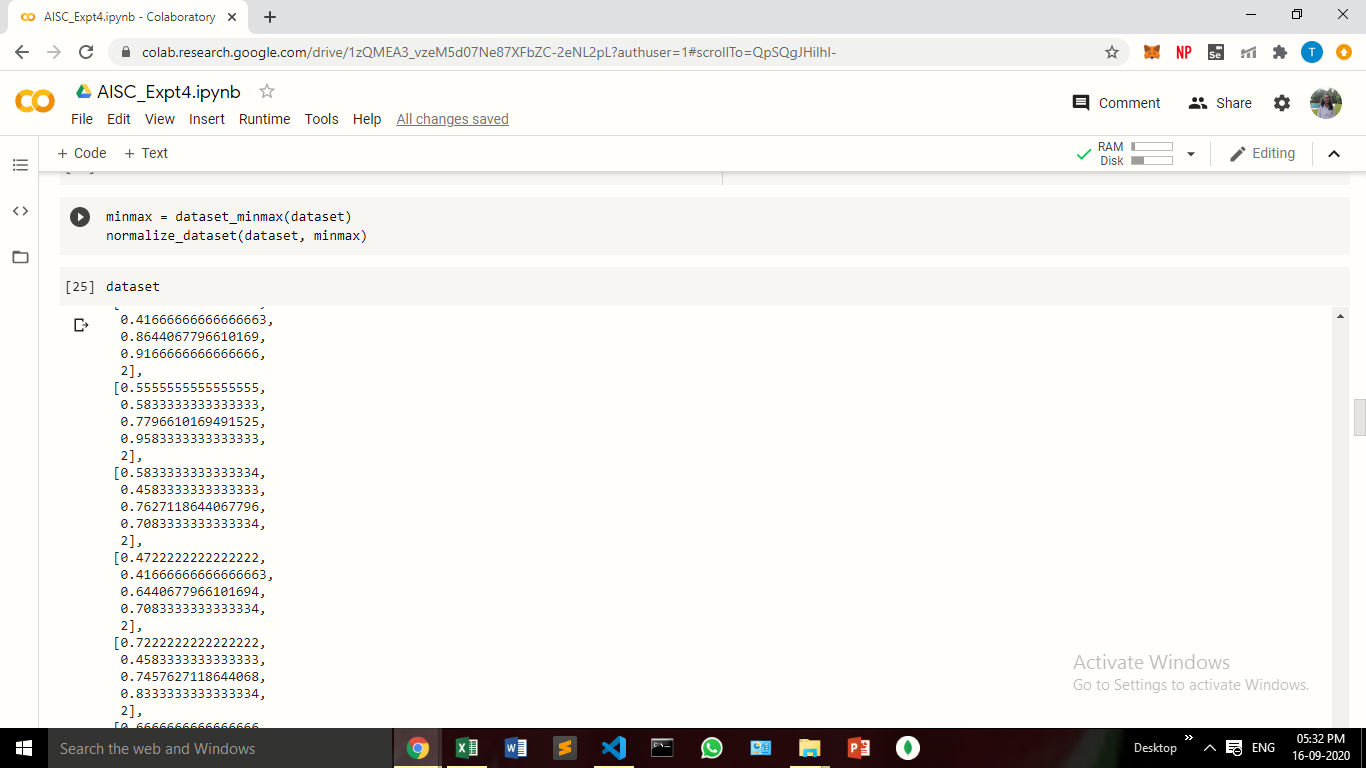
**Output:**

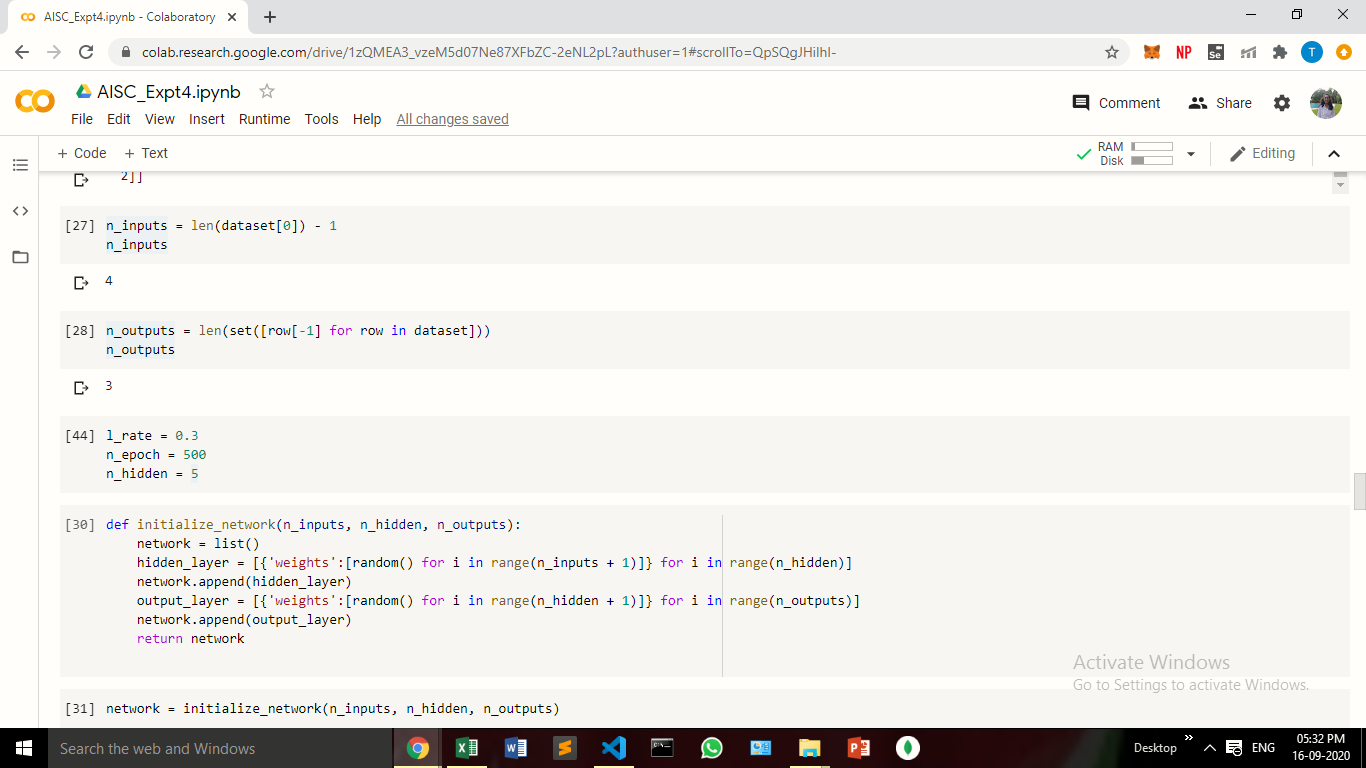
****

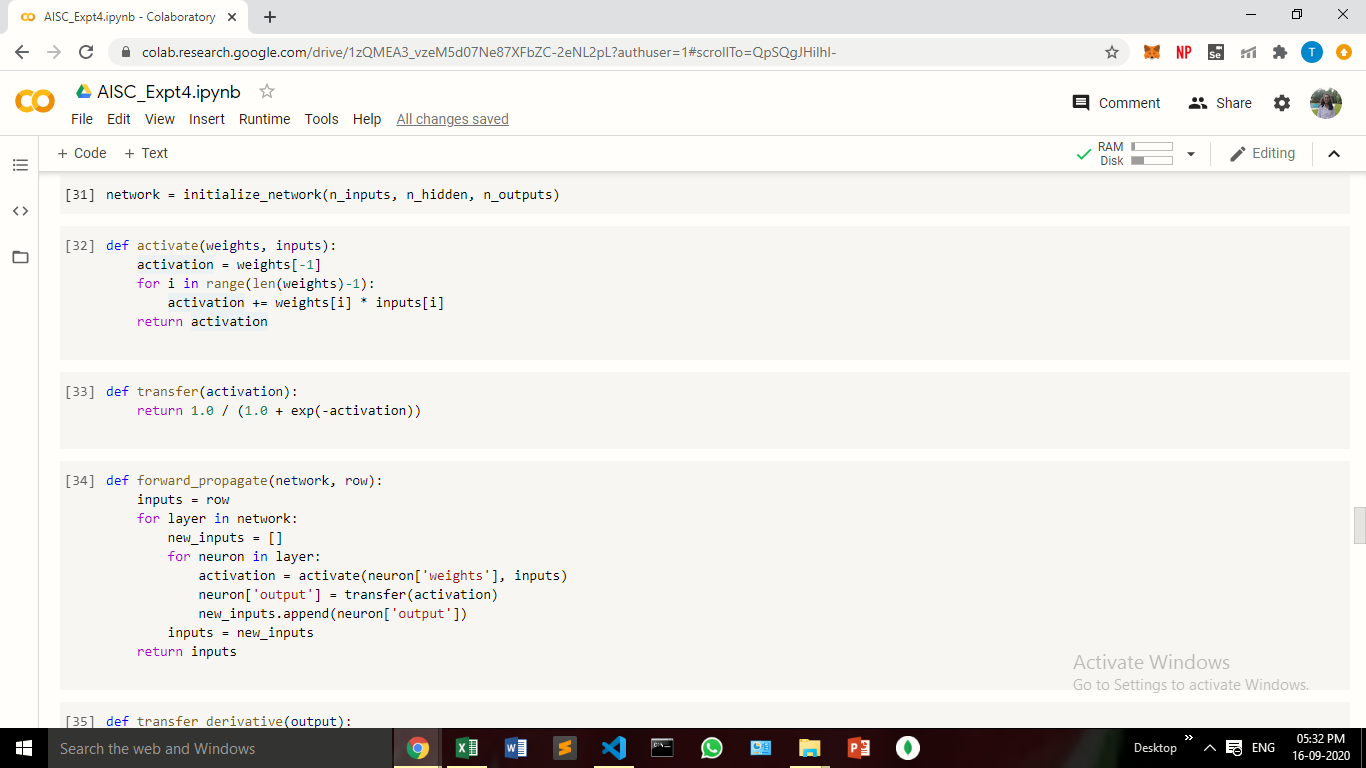
****

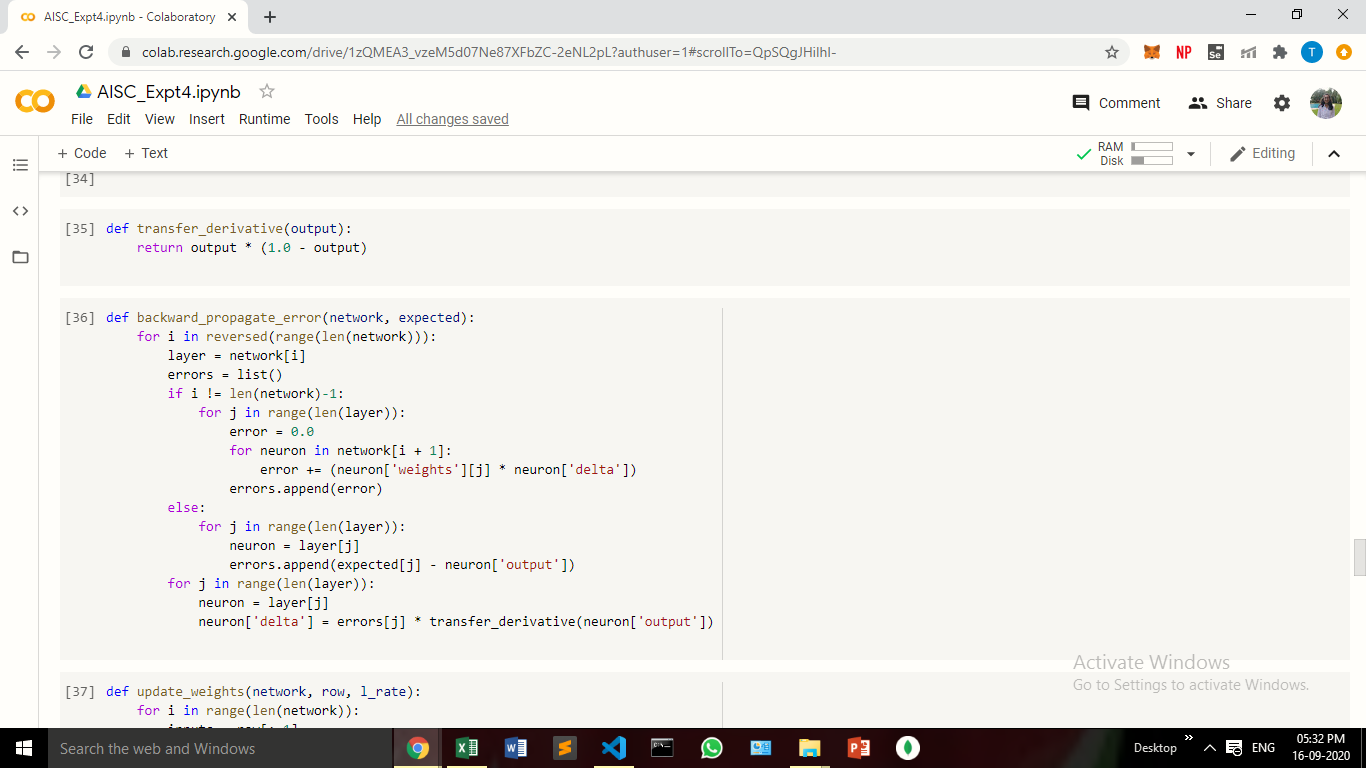
****

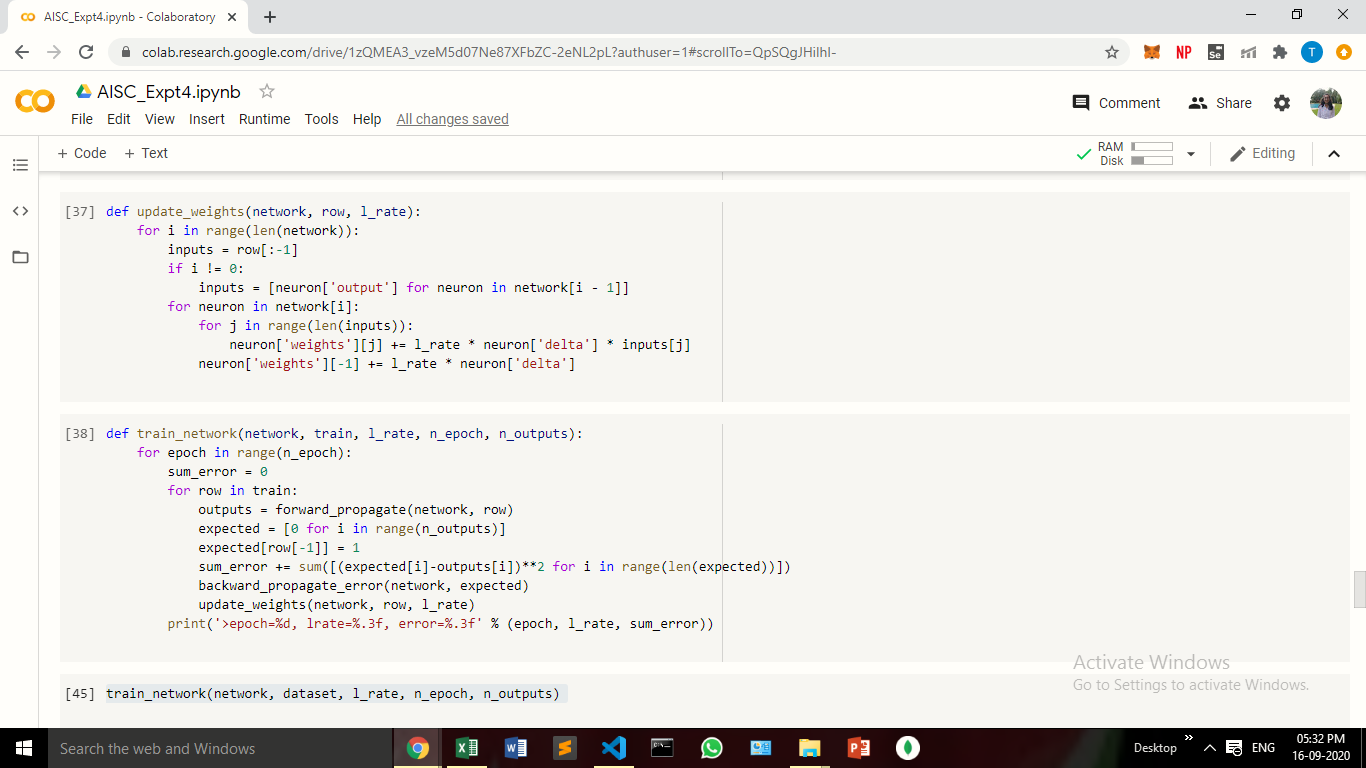
****

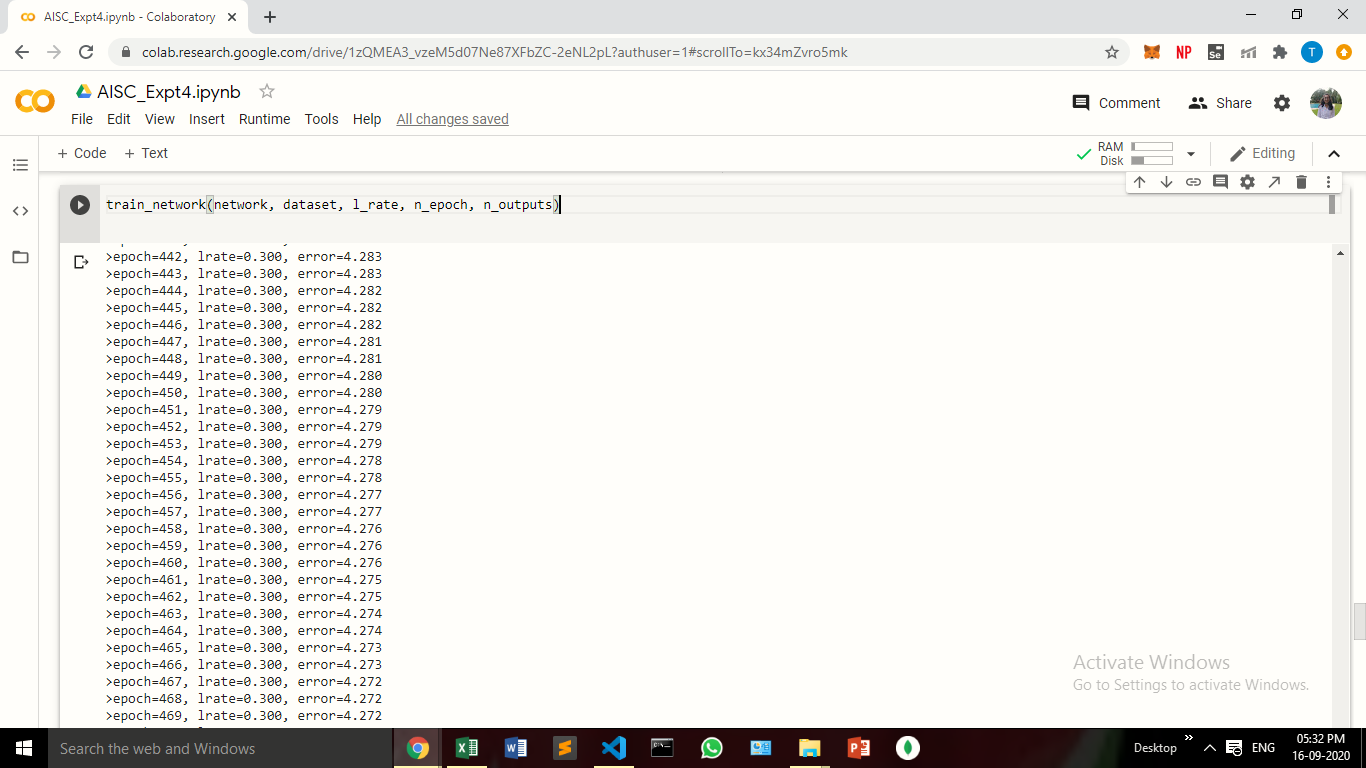
****

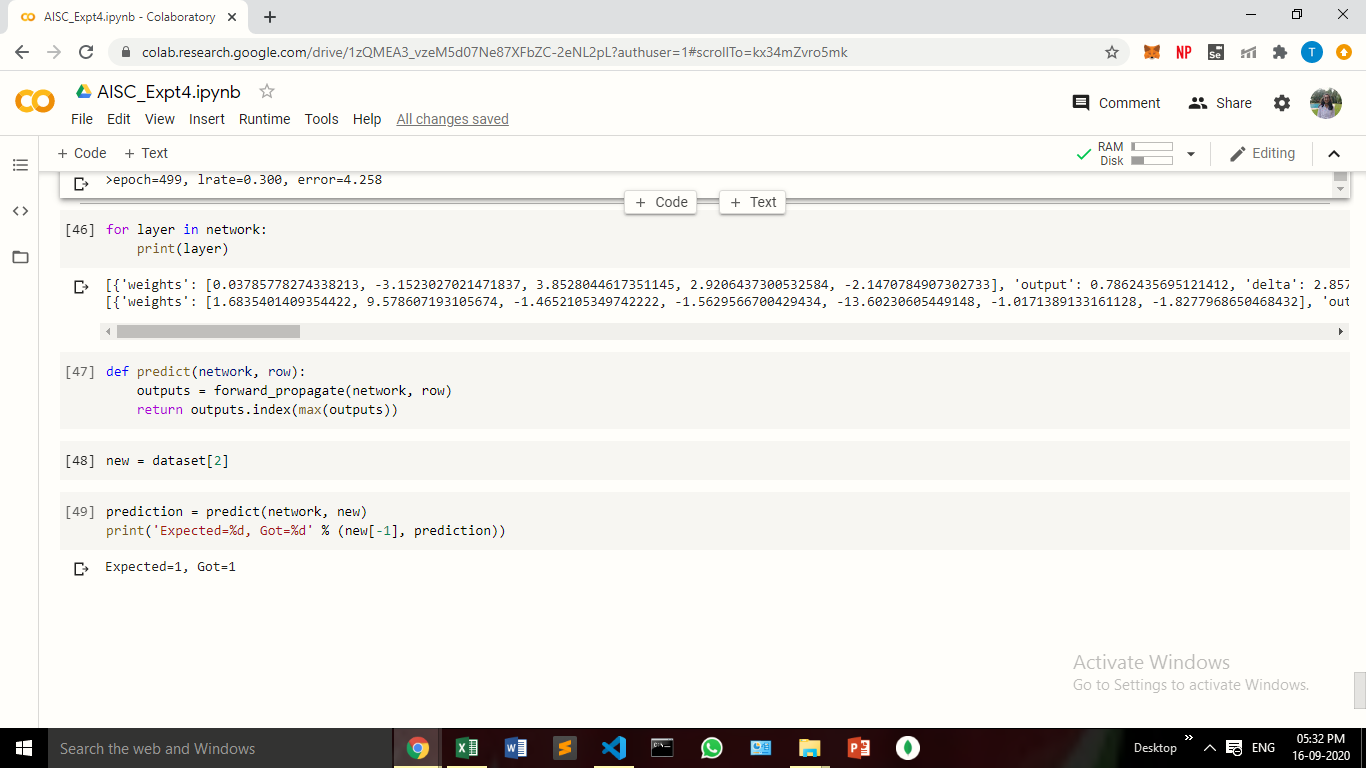
****

****

****

****

****

****

**Conclusion:**

In this experiment, we have implemented a multilayer perceptron learning algorithm on the IRIS dataset .