

LAB-F: WRITE A PROGRAM TO IMPLEMENT THE TRAINING ALGORITHM OF BACK PROPAGATION IN NEURAL NETWORK.

THEORY:

Back propagation training algorithm is a supervised learning algorithm for multilayer feed forward neural network. Since it is a supervised learning algorithm, both input and target output vectors are provided for training the network. The error data at the output layer is calculated using network output and target output. Then the error is back propagated to intermediate layers, allowing incoming weights to these layers to be updated [1]. This algorithm is based on the error correction learning rule. Basically, the error back-propagation process consists of two passes through the different layers of the network: a forward pass and a backward pass. In the forward pass, an input vector is applied to the network, and its effect propagates through the network, layer by layer [4]. Finally, a set of outputs is produced as the actual response of the network. During the forward pass the synaptic weights of network are all fixed. During the backward pass, on the other hand, the synaptic weights are all adjusted in accordance with the error correction rule. The actual response of the network is subtracted from a desired target response to produce an error signal. This error signal is then propagated backward through the network, against direction of synaptic connections - hence the name “error back-propagation”. The synaptic weights are adjusted so as to make the actual response of the network move closer to the desired response.

ALGORITHM:

Step 0: Initialize the weights to small random values

Step 1: Feed the training sample through the network and determine the final output

Step 2: Compute the error for each output unit, for unit k it is:

$$\delta_k = (t_k - y_k)f'(y_{in_k})$$

Actual output
Required output
Derivative of f

Step 3: Calculate the weight correction term for each output unit, for unit k it is:

$$\Delta w_{jk} = \alpha \delta_k z_j$$

Hidden layer signal
A small constant

Step 4: Propagate the delta terms (errors) back through the weights of the hidden units where the delta input for the j^{th} hidden unit is:

$$\delta_{in_j} = \sum_{k=1}^m \delta_k w_{jk}$$

The delta term for j^{th} hidden unit is: $\delta_j = \delta_{in_j} f'(z_{in_j})$

Step 5: Calculate the weight correction term for the hidden units: $\Delta w_{ij} = \alpha \delta_j x_i$

Step 6: Update the weights: $w_{ik}(\text{new}) = w_{ik}(\text{old}) + \Delta w_{ik}$

Step 7: Test for stopping (maximum cycles, small changes, etc)

Note: There are a number of options in the design of a backprop system;

- Initial weights – best to set the initial weights (and all other free parameters) to random numbers inside a small range of values (say -0.5 to 0.5)
- Number of cycles – tend to be quite large for backprop systems
- Number of neurons in the hidden layer – as few as possible

PROGRAM:

```
from math import exp
from random import seed
from random import random

# Initialize a network
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

# Calculate neuron activation for an input
def activate(weights, inputs):
    activation = weights[-1]
    for i in range(len(weights)-1):
        activation += weights[i] * inputs[i]
    return activation

# Transfer neuron activation
def transfer(activation):
    return 1.0 / (1.0 + exp(-activation))

# Forward propagate input to a network output
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

# Calculate the derivative of an neuron output
def transfer_derivative(output):
    return output * (1.0 - output)
```

```

# Backpropagate error and store in neurons
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(neuron['output'] - expected[j])
                for j in range(len(layer)):
                    neuron = layer[j]
                    neuron['delta'] = errors[j] * transfer_derivative(neuron['output'])

# Update network weights with error
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']

# Train a network for a fixed number of epochs
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))

# Test training backprop algorithm
seed(1)
dataset = [[6.785454836,1.559537003,0],
[8.465489372,2.362125076,0],
[2.396561688,7.400293529,0],
[2.38807019,1.850220317,0],
[1.06407232,3.005305973,0],
[4.627531214,5.759262235,1],
[5.332441248,2.088626775,1],
[2.922596716,-1.77106367,1],

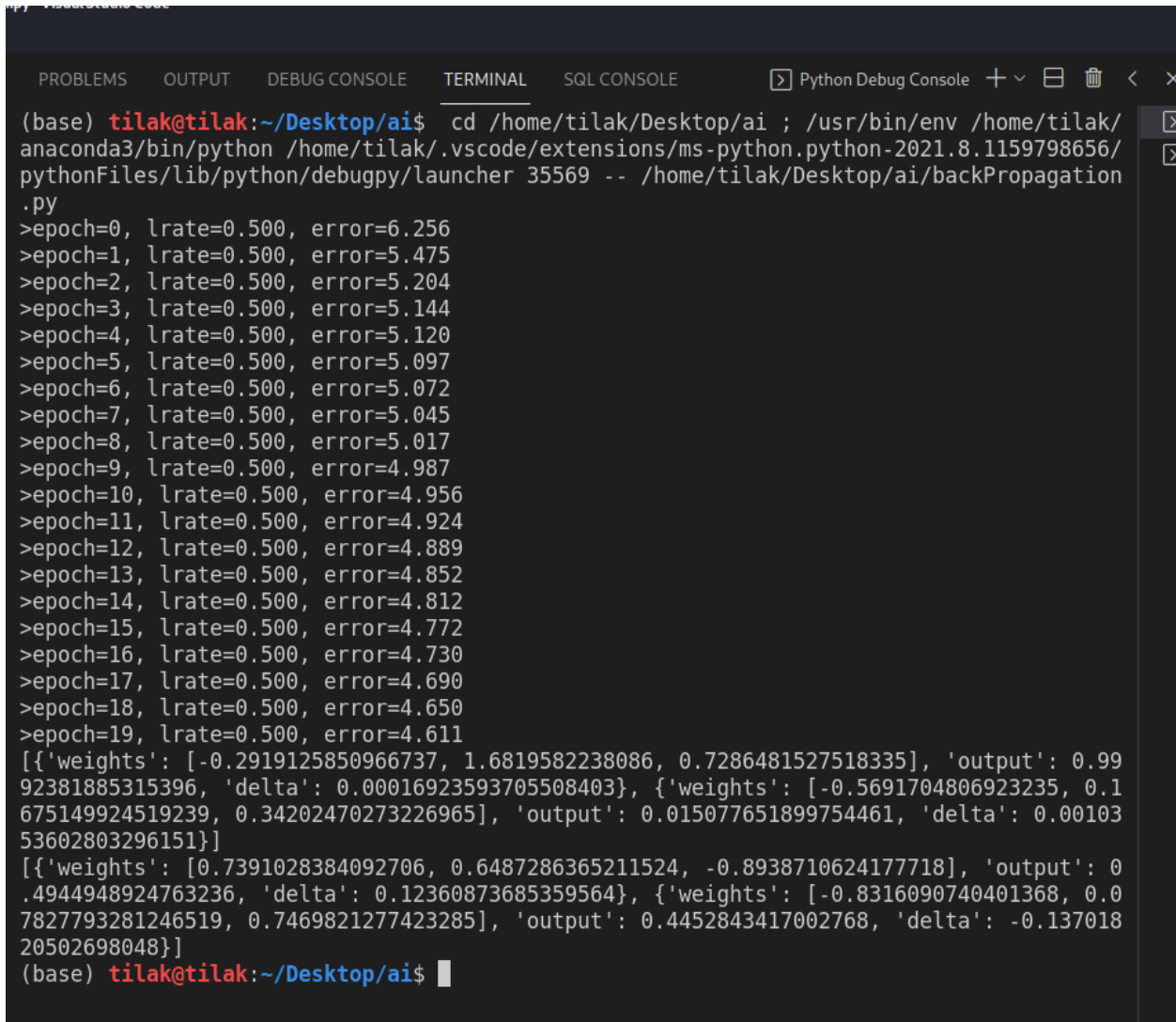
```

```

[9.675418651,-0.242068655,1],
[9.678756466,5.508563011,1]]
n_inputs = len(dataset[0]) - 1
n_outputs = len(set([row[-1] for row in dataset]))
network = initialize_network(n_inputs, 2, n_outputs)
train_network(network, dataset, 0.5, 20, n_outputs)
for layer in network:
    print(layer)

```

OUTPUT:



```

(base) tilak@tilak:~/Desktop/ai$ cd /home/tilak/Desktop/ai ; /usr/bin/env /home/tilak/anaconda3/bin/python /home/tilak/.vscode/extensions/ms-python.python-2021.8.1159798656/pythonFiles/lib/python/debugpy/launcher 35569 -- /home/tilak/Desktop/ai/backPropagation.py
>epoch=0, lrate=0.500, error=6.256
>epoch=1, lrate=0.500, error=5.475
>epoch=2, lrate=0.500, error=5.204
>epoch=3, lrate=0.500, error=5.144
>epoch=4, lrate=0.500, error=5.120
>epoch=5, lrate=0.500, error=5.097
>epoch=6, lrate=0.500, error=5.072
>epoch=7, lrate=0.500, error=5.045
>epoch=8, lrate=0.500, error=5.017
>epoch=9, lrate=0.500, error=4.987
>epoch=10, lrate=0.500, error=4.956
>epoch=11, lrate=0.500, error=4.924
>epoch=12, lrate=0.500, error=4.889
>epoch=13, lrate=0.500, error=4.852
>epoch=14, lrate=0.500, error=4.812
>epoch=15, lrate=0.500, error=4.772
>epoch=16, lrate=0.500, error=4.730
>epoch=17, lrate=0.500, error=4.690
>epoch=18, lrate=0.500, error=4.650
>epoch=19, lrate=0.500, error=4.611
[{'weights': [-0.2919125850966737, 1.6819582238086, 0.7286481527518335], 'output': 0.9992381885315396, 'delta': 0.00016923593705508403}, {'weights': [-0.5691704806923235, 0.1675149924519239, 0.34202470273226965], 'output': 0.015077651899754461, 'delta': 0.0010353602803296151}]
[{'weights': [0.7391028384092706, 0.6487286365211524, -0.8938710624177718], 'output': 0.4944948924763236, 'delta': 0.12360873685359564}, {'weights': [-0.8316090740401368, 0.07827793281246519, 0.7469821277423285], 'output': 0.4452843417002768, 'delta': -0.13701820502698048}]
(base) tilak@tilak:~/Desktop/ai$

```

CONCLUSION:

NN is an interconnected network that resembles human brain. The most important characteristic of NN is its ability to learn. When presented with training set (form of supervised learning) where input and output values are known, NN model could be created to help with classifying new data. Results that are achieved by using NN are encouraging, especially in some fields like pattern recognition. NN is getting more and more attention in last two decades. BP algorithm is most popular algorithm used in NN. It is one of the main reasons why NN are becoming so popular.