

Title: Implementation of Back Propagation Algorithm.

Objectives:

1. Applying multi-layer perceptron
2. Solve XOR problem and similar problems
3. Using the power of neural network to perfectly predict for supervised data

Methodology:

Firstly, let's look at the model of Back Propagation algorithm:

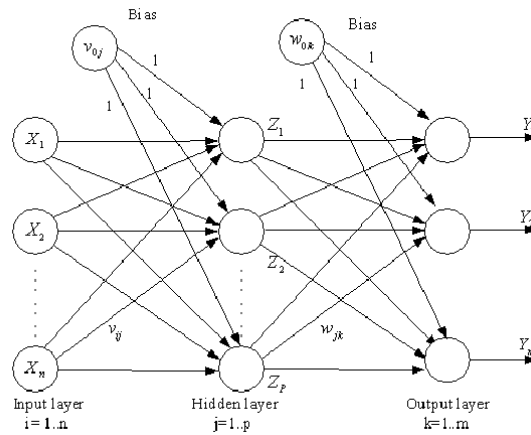


Figure: Back Propagation Algorithm Model

The first step is initializing the weight vectors W_{ij} and W_{jk} and the threshold values for each PE (processing element) with minimum random numbers.

In second step, the network provides the input patterns and the desired respective output patterns.

In third step, the input patterns are connected to the hidden PEs through the weights W_{ij} . In the hidden layer, each PE computed the weighted sum according to the equation,

$$net_{aj} = \sum W_{ij} O_{ai}$$

where O_{ai} is the input of unit i for pattern number a . The threshold of each PE was then added to its weighted sum to obtain the activation $activ_j$ of that PE i.e.

$$activ_j = net_{aj} + uh_j$$

where uh_j is the hidden threshold weights for j^{th} PEs. This activation determined whether the output of the respective PE was either 1 or 0 (fires or not) by using a sigmoid function,

$$O_{aj} = 1/(a + e^{-k_1 * activ_j})$$

Where k_1 is called the spread factors, these O_{aj} were then served as the input to the output computation. Signal O_{aj} were then fan out to the output layer according to the relation,

$$net_{ak} = \sum W_{jk} O_{aj}$$

And the output threshold weight uo_k for k^{th} output PEs was added to it to find out the activation $activo_k$

$$activo_k = net_{ak} + uo_k$$

The actual output O_{ak} was computed using the same sigmoid function which was

$$O_{ak} = 1/(a + e^{-k_2 * activo_k})$$

Here another spread factor k_2 has been employed for the output units.

In the second stages, after computing the feed-forward propagation, an error was computed by comparing the output O_{ak} with the respective target t_{ak} , i.e.

$$\delta_{ak} = t_{ak} - O_{ak}$$

This error was then used to adjust the weights vector W_{jk} using the equation,

$$\begin{aligned} \Delta W_{jk} &= \eta_2 k_2 \delta_{ak} O_{aj} O_{ak} (1 - O_{ak}) \\ W_{jk} &= W_{jk} + \Delta W_{jk} \end{aligned}$$

For the threshold weight of the output PE, similar equation was employed,

$$\begin{aligned} \Delta uo_k &= \eta_2 k_2 \delta_{ak} O_{ak} (1 - O_{ak}) \\ U_{ok} &= U_{ok} + \Delta U_{ok} \end{aligned}$$

In the next step, this error and the adjusted weight vector W_{jk} were feedback to the hidden layer adjust the weight vector W_{ij} and threshold weight uh_i . In this layer, the weight vector W_{ij} was computed by using equation,

$$\begin{aligned} \Delta W_{ij} &= \eta_1 k_1 O_{ai} O_{aj} (1 - O_{aj}) \sum \delta_{ak} W_{jk} \\ W_{ij} &= W_{ij} + \Delta W_{ij} \end{aligned}$$

For the threshold weights of the hidden PEs, similar equation was employed,

$$\begin{aligned} \Delta uh_j &= \eta_1 k_1 O_{aj} (1 - O_{aj}) \sum \delta_{ak} W_{jk} \\ uh_j &= uh_j + \Delta uh_j \end{aligned}$$

Calculating Errors:

After getting the output from the output layer, we calculate the error according to the targeted output in the following error calculating formula,

$$Error_a = 0.5 \sum (t_{ak} - o_{ak})^2$$

Now let's look at the flowchart for back propagation algorithm:

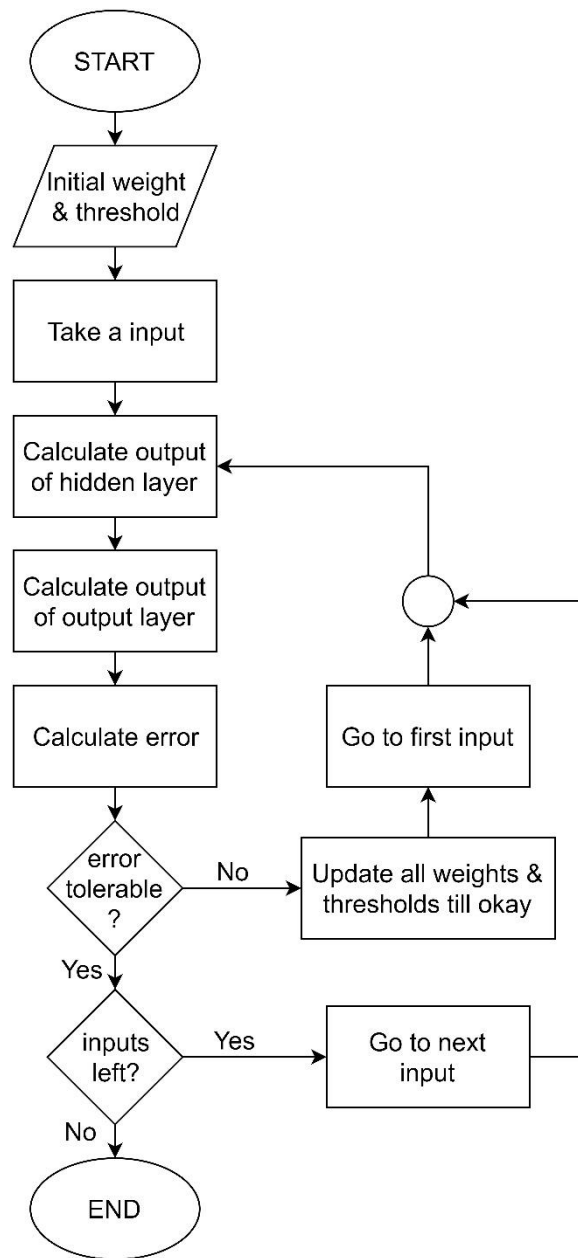


Figure: Flowchart for Back Propagation Algorithm

Unlike perceptron learning algorithm, back propagation algorithm handles more than one neuron at a time and hence it can solve XOR problem and most other problems that needs more than one decision boundary as well.

Implementation:

To implement the Back-Propagation Algorithm:

1. Firstly, generate the dataset
2. Split the dataset into train and test set
3. Apply algorithm to train, test and calculate accuracy

So, here there are 10 input lines. Hence, the number of possible inputs is 1024. The number of output line is 2. Hence, there are 4 possible classes. The number of hidden nodes is set as 5. The two constants and two etas are specified as 0.5 each. 20% data has been kept in train set and 80% data has been kept in test set as training needs a lot of time if more samples are there in train set.

After training, the kernel is then used for predicting the class of the samples from test set. Then the accuracy is calculated by determining how many right and wrong predictions have been made. Train log and test log has been generated for better understanding.

So, let's look at the codes and the corresponding outputs now.

Code-1: Dataset Generation

```
"""
Code: Back Propagation Algorithm
Title: Dataset Generator
Author: Azmain Yakin Srizon
"""

from numpy import binary_repr

filename = 'Final-Data.csv'
log = open(filename, 'w')

data = []

# defining input and output nodes
input_line = 10
output_line = 2

# calculating all possible inputs
for i in range (0,int(pow(2,input_line))):
    tmp = list(binary_repr(i,input_line))
    data.append(tmp)

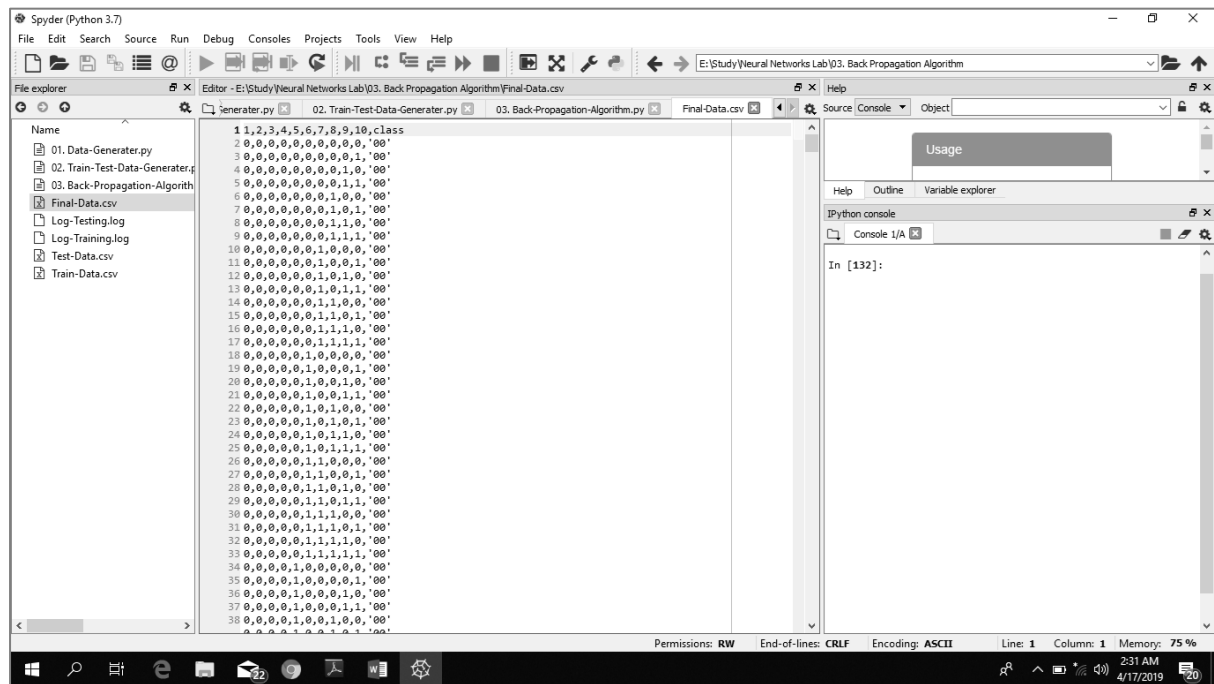
# creating column names
for i in range(0,input_line):
    log.write(str(i+1)+',')
log.write('class\n')

# creating dataset
for i in data:
    for j in i:
        log.write('' + str(j) + ',')
    log.write('')
    for j in range(0,output_line):
        if int(i[j])!=0:
            log.write('1') # class A
        else:
            log.write(str(i[j])) # class B
    log.write('' + '\n')
```

```
log.close() # closing csv
```

Output-1: Dataset Generation

Here is a snapshot of the dataset:



```
1 1,2,3,4,5,6,7,8,9,10,class
2 0,0,0,0,0,0,0,0,0,0,'00'
3 0,0,0,0,0,0,0,0,0,1,'00'
4 0,0,0,0,0,0,0,0,1,0,'00'
5 0,0,0,0,0,0,0,0,1,1,'00'
6 0,0,0,0,0,0,0,1,0,0,'00'
7 0,0,0,0,0,0,0,1,0,1,'00'
8 0,0,0,0,0,0,0,1,1,0,'00'
9 0,0,0,0,0,0,0,1,1,1,'00'
10 0,0,0,0,0,0,1,0,0,0,'00'
11 0,0,0,0,0,0,1,0,0,1,'00'
12 0,0,0,0,0,0,1,0,1,0,'00'
13 0,0,0,0,0,0,1,0,1,1,'00'
14 0,0,0,0,0,0,1,1,0,0,'00'
15 0,0,0,0,0,0,1,1,0,1,'00'
16 0,0,0,0,0,0,1,1,1,0,'00'
17 0,0,0,0,0,0,1,1,1,1,'00'
18 0,0,0,0,0,0,1,0,0,0,'00'
19 0,0,0,0,0,0,1,0,0,1,'00'
20 0,0,0,0,0,0,1,0,1,0,'00'
21 0,0,0,0,0,0,1,0,1,1,'00'
22 0,0,0,0,0,1,0,1,0,0,'00'
23 0,0,0,0,0,1,0,1,0,1,'00'
24 0,0,0,0,0,1,0,1,1,0,'00'
25 0,0,0,0,0,1,0,1,1,1,'00'
26 0,0,0,0,0,1,1,0,0,0,'00'
27 0,0,0,0,0,1,1,0,0,1,'00'
28 0,0,0,0,0,1,1,0,1,0,'00'
29 0,0,0,0,0,1,1,0,1,1,'00'
30 0,0,0,0,0,1,1,1,0,0,'00'
31 0,0,0,0,0,1,1,1,0,1,'00'
32 0,0,0,0,0,1,1,1,1,0,'00'
33 0,0,0,0,0,1,1,1,1,1,'00'
34 0,0,0,0,1,0,0,0,0,0,'00'
35 0,0,0,0,1,0,0,0,0,1,'00'
36 0,0,0,0,1,0,0,0,1,0,'00'
37 0,0,0,0,1,0,0,0,1,1,'00'
38 0,0,0,0,1,0,0,1,0,0,'00'
```

Code-2: Train-Test Set Generation

```
"""
Code: Back Propagation Algorithm
Title: Train-Test Dataset Generator
Author: Azmain Yakin Srizon
"""

import pandas
import random

train_percentage = 0.2

filename = 'Train-Data.csv'
train_file = open(filename, 'w') # creating train csv

filename = 'Test-Data.csv'
test_file = open(filename, 'w') # creating test csv

df = pandas.read_csv('Final-Data.csv') # reading dataset
train_elements = int(train_percentage*len(df))
test_elements = len(df)-train_elements
train = []
test = []

# take random rows and enter in train set
tmp = train_elements
while(tmp!=0):
    a = random.randint(0,len(df)-1)
    if a not in train:
        train.append(a)
        tmp=tmp-1

# Take the remaining samples in test set
for i in range(0,len(df)):
    if i not in train:
        test.append(i)

# calculating features
a = df.size
b = len(df)
features = int(a/b)

# creatig column names
for i in range(0,features-1):
    train_file.write(str(i+1)+',')
train_file.write('class\n')

# creating rows one by one
for i in train:
    for j in range(0,features-1):
        train_file.write(str(df[str(j+1)][i]))
        train_file.write(',')
    train_file.write(str(df['class'][i]))
    train_file.write('\n')

train_file.close() # closing train file

# creating column names for test
for i in range(0,features-1):
    test_file.write(str(i+1)+',')
test_file.write('class\n')

# creating rows one by one for test
for i in test:
    for j in range(0,features-1):
```

```

        test_file.write(str(df[str(j+1)][i]))
        test_file.write(',')
    test_file.write(str(df['class'][i]))
    test_file.write('\n')

```

```
test_file.close() # closing test file
```

Output-2: Train-Test Set Generation

Here are the snaps of train and test sets.



Code-3: Main Algorithm

```
"""
Code: Back Propagation Algorithm
Title: Main Algorithm
Author: Azmain Yakin Srizon
"""

# importing necessary libraries
import pandas
import random
import math

# reading training data
df = pandas.read_csv('Train-Data.csv')

# defining input, hidden and output lines
input_line = int(df.size/len(df))-1
output_line = len(list(df['class'][0]))-2
hidden_line = 5

# defining parameters
k1 = 0.5
k2 = 0.5
eta1 = 0.5
eta2 = 0.5

weight_ij = []
weight_jk = []

threshold_j = []
threshold_k = []

output_i = []
output_j = []
output_k = []

deltaw_ij = []
deltaw_jk = []
deltaw_j = []
deltaw_k = []

# initializing weights from input to hidden
for i in range(0,input_line):
    tmp = []
    for j in range(0, hidden_line):
        tmp.append(0.1*random.randint(1,10))
    weight_ij.append(tmp)

# initializing thresholds of hidden nodes
for j in range(0,hidden_line):
    threshold_j.append(0.1*random.randint(1,10))

# initializing weights from hidden to output
for j in range(0,hidden_line):
    tmp = []
    for k in range(0, output_line):
        tmp.append(0.1*random.randint(1,10))
    weight_jk.append(tmp)

# initializing thresholds of output nodes
for k in range(0,output_line):
    threshold_k.append(0.1*random.randint(1,10))

# Training...
steps = 0
```



```

inputs = 0
error_handled = False # used for error handling

filename = 'Log-Training.log'
train_log = open(filename, 'w') # open train log

# writing in train log - the general logs
train_log.write('Number of Input Nodes: ' + str(input_line) + '\n')
train_log.write('Number of Hidden Nodes: ' + str(hidden_line) + '\n')
train_log.write('Number of Output Nodes: ' + str(output_line) + '\n')

train_log.write('Initial Weights: \n')
train_log.write('Weight from i to j: ' + str(weight_ij) + '\n')
train_log.write('Weight from j to k: ' + str(weight_jk) + '\n')

train_log.write('Initial Thresholds: \n')
train_log.write('Threshold of j layer: ' + str(threshold_j) + '\n')
train_log.write('Threshold of k layer: ' + str(threshold_k) + '\n')
train_log.write('\n\n')

while inputs < len(df): # for each input
    steps = steps + 1 # calculating the steps

    # taking present input
    input_i = []
    for i in range(0, input_line):
        input_i.append(df[str(i+1)][inputs])

    # taking present output
    output_i = []
    for i in input_i:
        output_i.append(i)

    # calculating net_j
    net_j = []
    for j in range(0, hidden_line):
        tmp = 0.0
        for i in range(0, input_line):
            tmp = tmp + weight_ij[i][j]*output_i[i]
        net_j.append(tmp)

    # calculating activ_j
    activ_j = []
    for j in range(0, hidden_line):
        activ_j.append(net_j[j]+threshold_j[j])

    # calculating output_j
    output_j = []
    for j in range(0, hidden_line):
        output_j.append(1.0/(1.0+math.exp(-k1*activ_j[j])))

    # calculating net_k
    net_k = []
    for k in range(0, output_line):
        tmp = 0.0
        for j in range(0, hidden_line):
            tmp = tmp + weight_jk[j][k]*output_j[j]
        net_k.append(tmp)

    # calculating activ_k
    activ_k = []
    for k in range(0, output_line):
        activ_k.append(net_k[k]+threshold_k[k])

    # calculating output_k
    output_k = []
    for k in range(0, output_line):
        output_k.append(1.0/(1.0+math.exp(-k2*activ_k[k])))

```

```

# reading targets
target = []
a = df['class'][inputs]
a = list(a)
for i in a:
    if i!="":
        target.append(int(i))

# calculating errors
error = 0.0
for k in range(0,output_line):
    error = error + 0.5*(target[k]-output_k[k])*(target[k]-output_k[k])

# calculating one by one error
delta_k = []
for k in range(0,output_line):
    delta_k.append(target[k]-output_k[k])

# if error happens
if error>0.2:
    # update weight among hidden and output
    deltaw_jk = []
    for j in range(0,hidden_line):
        tmp = []
        for k in range(0,output_line):
            tmp.append(0.0)
        deltaw_jk.append(tmp)

    for j in range(0,hidden_line):
        for k in range(0,output_line):
            deltaw_jk[j][k] = eta2*k2*delta_k[k]*output_j[j]*output_k[k]*(1-
output_k[k])

    for j in range(0,hidden_line):
        for k in range(0,output_line):
            weight_jk[j][k] = weight_jk[j][k] + deltaw_jk[j][k]

    # update thresholds of output nodes
    deltau_k = []
    for k in range(0,output_line):
        deltau_k.append(eta2*k2*delta_k[k]*output_k[k]*(1-output_k[k]))

    for k in range(0,output_line):
        threshold_k[k] = threshold_k[k] + deltau_k[k]

    # update weights among input and hidden
    deltaw_ij = []
    for i in range(0,input_line):
        tmp = []
        for j in range(0,hidden_line):
            tmp.append(0.0)
        deltaw_ij.append(tmp)

    for i in range(0,input_line):
        for j in range(0,hidden_line):
            tmp = 0.0
            for k in range(0,output_line):
                tmp = tmp + delta_k[k]*weight_jk[j][k]
            deltaw_ij[i][j] = eta1*k1*output_i[i]*output_j[j]*(1-
output_j[j])*tmp

    for i in range(0,input_line):
        for j in range(0,hidden_line):
            weight_ij[i][j] = weight_ij[i][j] + deltaw_ij[i][j]

    # update thresholds of hidden nodes
    deltau_j = []

```

```

        for j in range(0,hidden_line):
            tmp = 0.0
            for k in range(0,output_line):
                tmp = tmp + delta_k[k]*weight_jk[j][k]
                deltau_j.append(eta1*k1*output_j[j]*(1-output_j[j])*tmp)

        for j in range(0,hidden_line):
            threshold_j[j] = threshold_j[j] + deltau_j[j]

        error_handled = True # error handling

    else:
        if error_handled == True:
            error_handled = False
            if inputs!=0:
                inputs = 0 # go to first step
            else:
                inputs = inputs + 1 # go to next sample
        else:
            inputs = inputs + 1 # go to next sample

    # write down every parameters
    train_log.write('Step No: ' + str(steps) + '\n')
    train_log.write('Current Input: ' + str(input_i) + '\n')
    train_log.write('Net of j layer: ' + str(net_j) + '\n')
    train_log.write('Activ of j layer: ' + str(activ_j) + '\n')
    train_log.write('Output of j layer: ' + str(output_j) + '\n')
    train_log.write('Net of k layer: ' + str(net_k) + '\n')
    train_log.write('Activ of k layer: ' + str(activ_k) + '\n')
    train_log.write('Output of k layer: ' + str(output_k) + '\n')
    train_log.write('Delta_W from j to k layer: ' + str(deltaw_jk) + '\n')
    train_log.write('Updated Weight from j to k layer: ' + str(weight_jk) + '\n')
    train_log.write('Delta_U of k layer: ' + str(deltau_k) + '\n')
    train_log.write('Updated Threshold of k layer: ' + str(threshold_k) + '\n')
    train_log.write('Delta_W from i to j layer: ' + str(deltaw_ij) + '\n')
    train_log.write('Updated Weight from i to j layer: ' + str(weight_ij) + '\n')
    train_log.write('Delta_U of j layer: ' + str(deltau_j) + '\n')
    train_log.write('Updated Threshold of j layer: ' + str(threshold_j) + '\n')
    train_log.write('Target Output: ' + str(target) + '\n')
    train_log.write('Delta of k layer: ' + str(delta_k) + '\n')
    train_log.write('Current Error: ' + str(error) + '\n\n\n')

train_log.close() # closing train log

# Testing...
df = pandas.read_csv('Test-Data.csv') # reading test data

filename = 'Log-Testing.log'
test_log = open(filename, 'w') # creating testing log

# write initial parameters value in test log
test_log.write('Adjusted Weight from i to j layer: ' + str(weight_ij) + '\n')
test_log.write('Adjusted Threshold of j layer: ' + str(threshold_j) + '\n')
test_log.write('Adjusted Weight from j to k layer: ' + str(weight_jk) + '\n')
test_log.write('Adjusted Threshold of k layer: ' + str(threshold_k) + '\n\n')

errors = []
right = 0
wrong = 0
inputs = 0

# apply the same algorithm again without error handling
while inputs<len(df):
    steps = steps + 1

    input_i = []
    for i in range(0,input_line):

```

```

        input_i.append(df[str(i+1)][inputs])

output_i = []
for i in input_i:
    output_i.append(i)

net_j = []
for j in range(0,hidden_line):
    tmp = 0.0
    for i in range(0,input_line):
        tmp = tmp + weight_ij[i][j]*output_i[i]
    net_j.append(tmp)

activ_j = []
for j in range(0,hidden_line):
    activ_j.append(net_j[j]+threshold_j[j])

output_j = []
for j in range(0,hidden_line):
    output_j.append(1.0/(1.0+math.exp(-k1*activ_j[j])))

net_k = []
for k in range(0, output_line):
    tmp = 0.0
    for j in range(0, hidden_line):
        tmp = tmp + weight_jk[j][k]*output_j[j]
    net_k.append(tmp)

activ_k = []
for k in range(0,output_line):
    activ_k.append(net_k[k]+threshold_k[k])

output_k = []
for k in range(0,output_line):
    output_k.append(1.0/(1.0+math.exp(-k2*activ_k[k])))

target = []
a = df['class'][inputs]
a = list(a)
for i in a:
    if i!="":
        target.append(int(i))

error = 0.0
for k in range(0,output_line):
    error = error + 0.5*(target[k]-output_k[k])*(target[k]-output_k[k])
errors.append(error)

flag = True
for k in range(0,output_line):
    if int(target[k])!=int(round(output_k[k])): # rounding the output and
matching
        flag = False
        break

if flag == True:
    right = right + 1 # calculating correct predictions
else:
    wrong = wrong + 1 # calculating wrong predictions

inputs = inputs + 1

# writing all parameters values
test_log.write('Current Input: ' + str(input_i) + '\n')
test_log.write('Target Output: ' + str(target) + '\n')
test_log.write('Calculated Output: ' + str(output_k) + '\n')
test_log.write('Error: ' + str(error) + '\n')
if flag == True:

```

```

        test_log.write('Correctness: ' + 'Correct' + '\n')
    else:
        test_log.write('Correctness: ' + 'Not Correct' + '\n')
    test_log.write('\n')

test_log.write('\n\n')
test_log.write('Number of Right Prediction: ' + str(right) + '\n')
test_log.write('Number of Wrong Prediction: ' + str(wrong) + '\n')
test_log.write('Accuracy: ' + str(float(right/(right+wrong)*100.0)) + '\n')

avg = 0.0
for i in errors:
    avg = avg + i
avg = avg/(right+wrong) # calculating average error

test_log.write('Average Error: ' + str(avg) + '\n') # writing average error
test_log.close() # closing test log
print('Right: ' + str(right)) # printing number of rights
print('Wrong: ' + str(wrong)) # printing number of wrongs

```

Output-3: Main Algorithm

Right: 819
Wrong: 1

And here are the snaps of train and test logs:

The top screenshot shows the 'Log-Training.log' file in the Spyder IDE. It displays the initialization of the neural network, including the number of input nodes (18), hidden nodes (5), and output nodes (2). It also shows the initial weights, thresholds, and the results of 10 steps of training, including current inputs, net values, and calculated outputs.

The bottom screenshot shows the 'Log-Testing.log' file in the Spyder IDE. It displays the results of 10 test cases, including the current input, target output, calculated output, error, and correctness status. The results show that the network correctly classified 819 out of 820 test cases, with 1 incorrect classification.

Performance Analysis:

Back Propagation Algorithm is not that easy to understand or tune. There are a lot of parameter and the performance of the algorithm depends on a lot of parameters as well. In this analysis section, these parameters will be described in short.

Let's start with the number of nodes present in input, hidden and output layer. If the number of nodes increases in the input layer then we need a less percentage of samples in train data. Hence, a great concept is that whatever the number of samples are in total the percentage of training always remains almost same. That's why Back Propagation Algorithm shows high accuracy.

If the number of nodes in hidden layer increases, it indicates that the hidden layer is adding more decision boundaries to channel to output layer and hence, increasing the accuracy. For example, if there are 5 nodes in hidden layer, it indicates that the output is being divided into 5 parts while going to the output layer and hence many neurons are working together to take a decision. When more than one neuron tries to take a decision, the decision automatically becomes better.

If the number of output node increase, it actually does even matter. The reason is there are thresholds also in output nodes. Hence more output nodes denote more thresholds. That means, it'll need more time for weight adjustment.

Now coming to the gain factors. If the values of the two etas are higher then there can be some problems like local minima problem or local solutions will be obtained. But the speed will be super-fast. On the other hand, if the value of etas is lower, then the process will be much slower and more steps will be required for weight adaptation. However, local minima problems will not occur on regular basis in this path.

The value of k_1 and k_2 has a mass effect on activation function. Notice that activation function has high impact on calculating the outputs of hidden and output layer which is pretty important in case of prediction. Hence, the values are important either. If the values are higher then the activation function will need more steps to give desired outputs for weight adaptation but the process will be, as usual, faster. On the other hand, if k_1 and k_2 has a small value, then the process will be much slower. However, the impact will be better than before.

If you notice carefully, we've taken only 20% data in train set and 80% data on test set which is a huge jump from the traditional approach. In other algorithms we use 80% or 70% data in train set and 20% or 30% data in test set but in neural net we use more data in test set and less data in train set. However, Back Propagation Algorithm is even better in this manner. If the input lines are 10, then the number of possible inputs is 1024. 20% of 1024 is 104. That means only 104 samples are needed for predicting 820 test samples. Now look at the number of right and wrong predictions.

Only one sample has been predicted wrong. That means, the accuracy is approximately 100%. However, if we run the program multiple times, the result may vary. Most of the time it gives an accuracy of 100%. The reason behind different result is that each time different values are being assigned as weights among input and hidden layer, hidden and

output layer and thresholds of hidden and output layers. That's why the results may vary as well.

Experimental analysis showed that when input lines are 10, hidden nodes are 5 and output nodes are 2, then for 20% data in train most of the times accuracy is 100%. For 30% data in train the accuracy is always 100% and for higher percentage of samples in train, the accuracy remains same. Hence, we can conclude that Back Propagation Algorithm need less inputs than other algorithms to predict perfectly.

Back Propagation Algorithm does have some limitations:

1. Very complex algorithms
2. If the inputs are not binary, weight initialization process is hard
3. There are a lot of parameters
4. Take a lot of time for training
5. Sometimes the values don't get adjusted at all

Apart from all these limitations, Back Propagation Algorithm is the best output generating algorithm in neural net for most of the supervised datasets existing in real world.