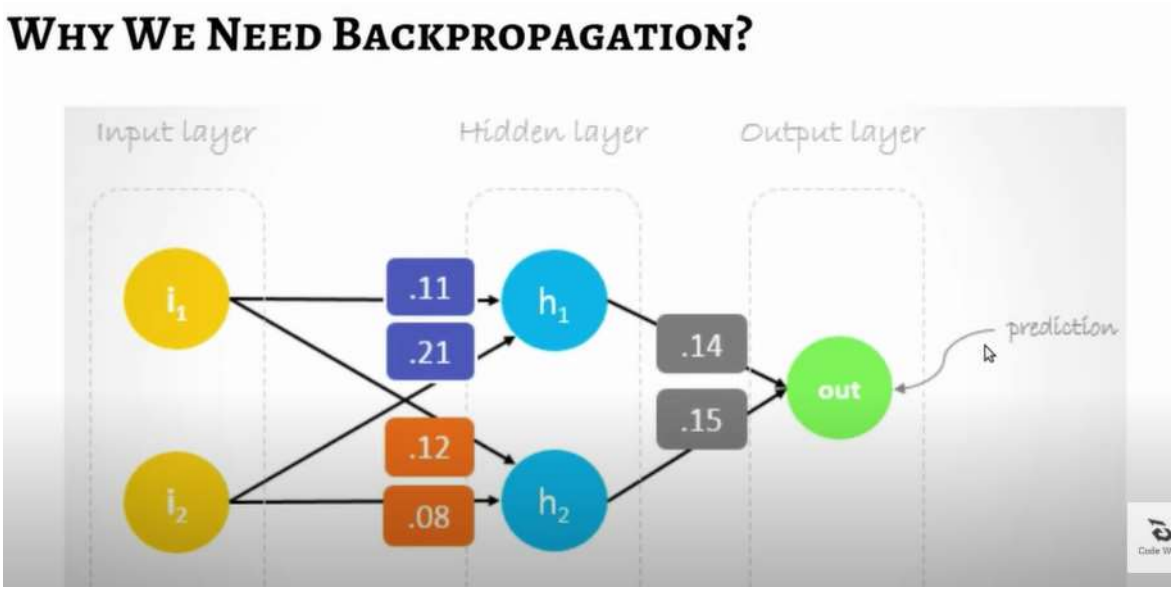
Aim – To design supervised NN model using BPN

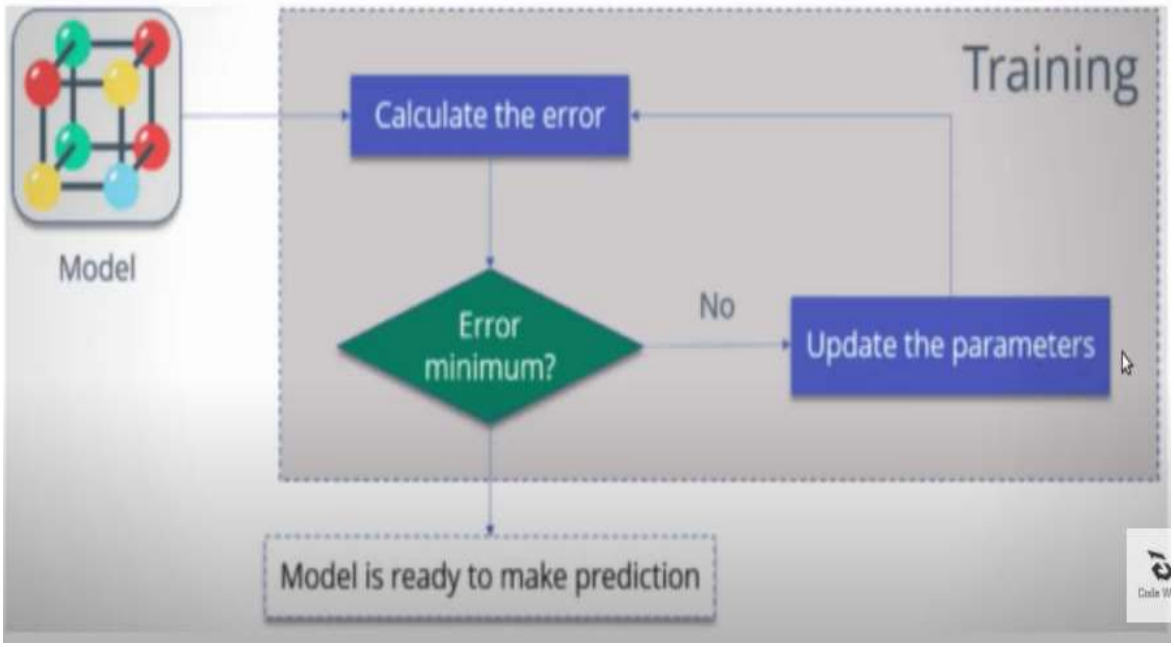
Theory –

Back-Propagation

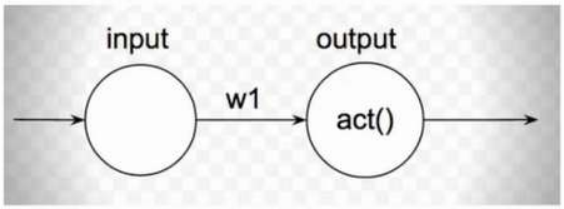
Back-propagation is the essence of neural net training. It is the method of fine-tuning the weights of a neural net based on the error rate obtained in the previous epoch (i.e., iteration). Proper tuning of the weights allows you to reduce error rates and to make the model reliable by increasing its generalization.

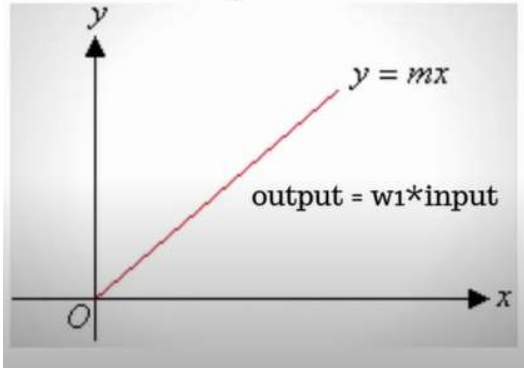
Backpropagation is a short form for "backward propagation of errors." It is a standard method of training artificial neural networks. This method helps to calculate the gradient of a loss function with respects to all the weights in the network.

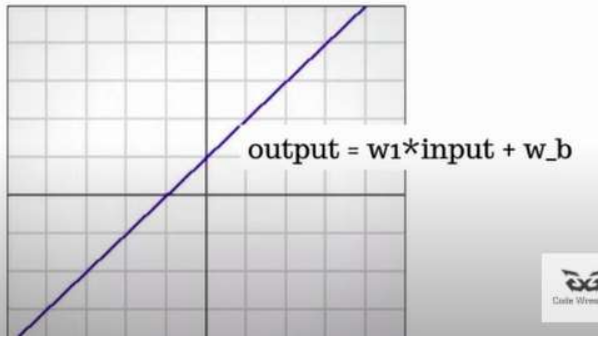


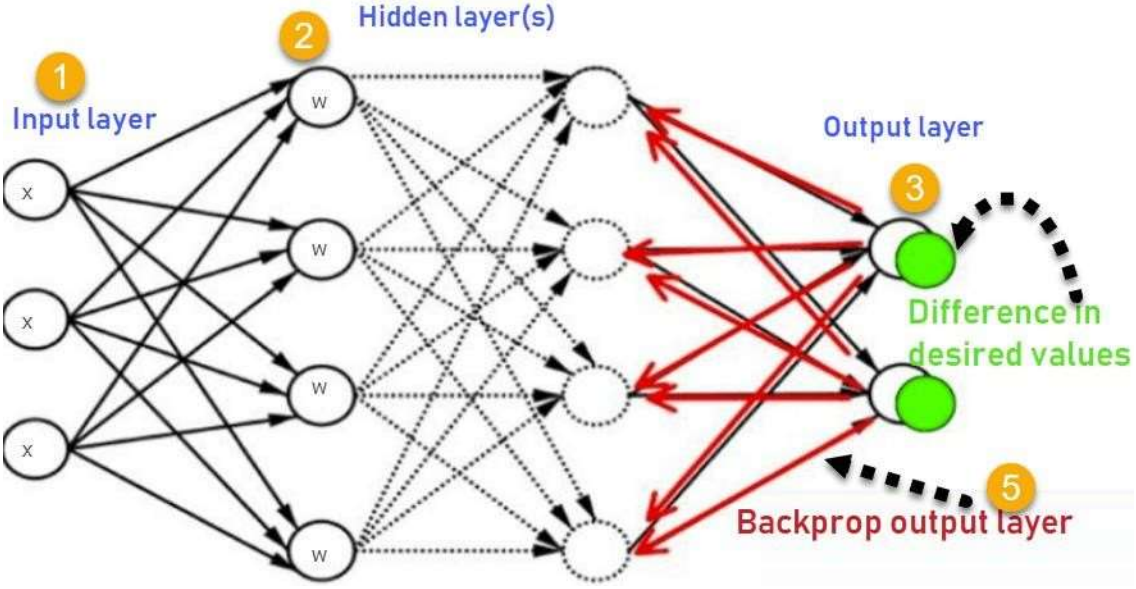


- y = w1 * x
- Output of bias is **w_b**.
- w_b = 1 * b1









FORWARD PASS

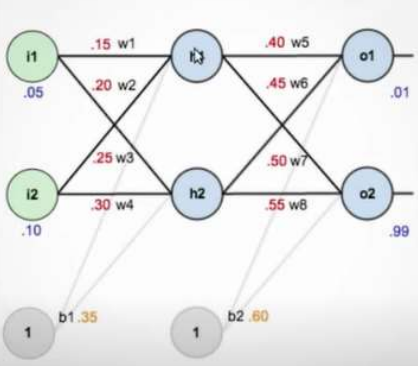
$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

$$net_{h1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$$

Squash it using logistic function to get output

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}} = \frac{1}{1+e^{-0.3775}} = 0.593269992$$

$$out_{h2} = 0.596884378$$



The backpropagation works on these simple steps

1. Inputs X, arrive through the preconnected path
2. Input is modelled using real weights W. The weights are usually randomly selected.
3. Calculate the output for every neuron from the input layer, to the hidden layers, to the output layer.
4. Calculate the error in the outputs - $Error_B$= Actual Output – Desired Output
5. Travel back from the output layer to the hidden layer to adjust the weights such that the error is decreased.

Keep repeating the process until the desired output is achieved

Advantages:

- Backpropagation is fast, simple, and easy to program
- It has no parameters to tune apart from the numbers of input
- It is a flexible method as it does not require prior knowledge about the network
- It is a standard method that generally works well
- It does not need any special mention of the features of the function to be learned.

Types of Backpropagation Networks

Two Types of Backpropagation Networks are:

- Static Backpropagation
  - It is one kind of backpropagation network which produces a mapping of a static input for static output. It is useful to solve static classification issues like optical character recognition.
- Recurrent Backpropagation
  - Recurrent backpropagation is fed forward until a fixed value is achieved. After that, the error is computed and propagated backward.

The main difference between both methods is: that the mapping is rapid in static backpropagation while it is non static in recurrent backpropagation.

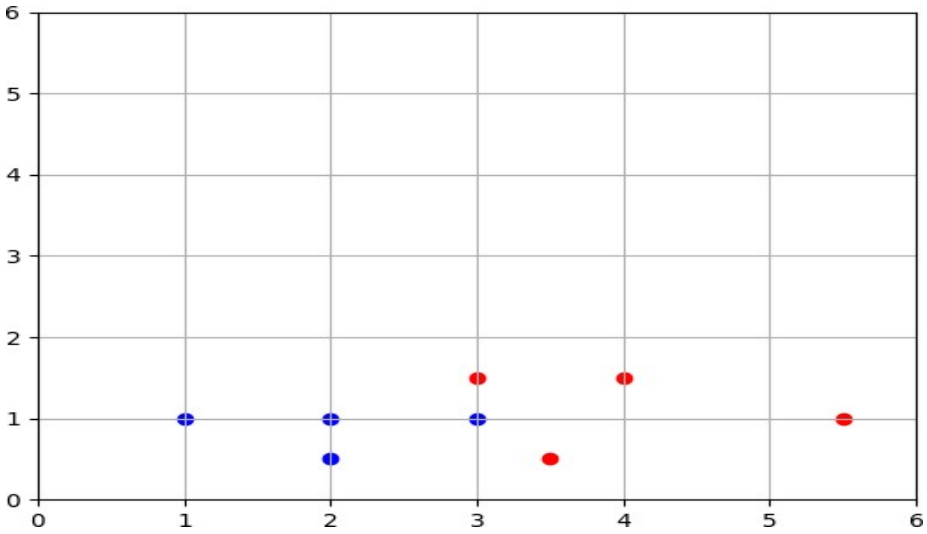
Code –

```
from matplotlib import pyplot as plt
import numpy as np


def sigmoid(x):
    return 1 / (1 + np.exp(-x))


def sigmoid_p(x):
    return sigmoid(x) * (1 - sigmoid(x))
```

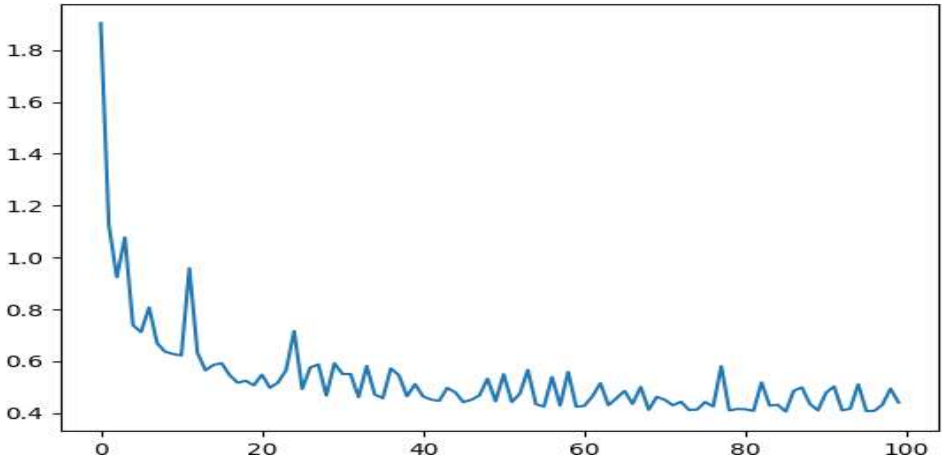
```
# data[i][0] = length, data[i][1] = height, data [i][2] = color - 1 = red &
# 0 = blue
data = [[3, 1.5, 1],
[2, 1, 0],
[4, 1.5, 1],
[3, 1, 0],
[3.5, 0.5, 1],
[2, 0.5, 0],
[5.5, 1, 1],
[1, 1, 0]]
find_color_flower = [4.5, 1]
# red flower
plt.axis([0, 6, 0, 6])
plt.grid()
for i in range(len(data)):
    point = data[i]
    color = "r"
    if point[2] == 0:
        color = "b"
    plt.scatter(point[0], point[1], c = color)
plt.show()

w1 = np.random.randn()
w2 = np.random.randn()
b = np.random.randn()
learning_rate = 0.1
costs = []

for i in range(10000):
    random_index = np.random.randint(len(data))
    point = data[random_index]
    z = point[0] * w1 + point[1] * w2 + b
    prediction = sigmoid(z)
    target = point[2]
    cost = np.square(prediction - target)
    if i % 100 == 0:
        cost_sum = 0
        for j in range(len(data)):
            point = data[j]
            z = point[0] * w1 + point[1] * w2 + b
            p_prediction = sigmoid(z)
            target = point[2]
            cost_sum += np.square(p_prediction - target)
        costs.append(cost_sum)
    dxCost_pred = 2 * (prediction - target)
    dxPred_dz = sigmoid_p(z)
    dz_dw1 = point[0]
    dz_dw2 = point[1]
    dz_db = 1
    dxCost_dw1 = dxCost_pred * dxPred_dz * dz_dw1
    dxCost_dw2 = dxCost_pred * dxPred_dz * dz_dw2
```

```
    dxCost_db = dxCost_pred * dz_db
    w1 = w1 - learning_rate * dxCost_dw1
    w2 = w2 - learning_rate * dxCost_dw2
    b = b - learning_rate * dxCost_db
plt.plot(costs)
plt.show()

z = find_color_flower[0] * w1 + find_color_flower[1] * w2 + b
pred = sigmoid(z)
if pred > 0.5:
    print("RED")
else:
    print("BLUE")
```

Output –



The above plot diagram represents the flowers with respect to their length and height of their petals. The color of flower is indicated by the dot.



The above diagram shows the cost of error of the program. As we can see it is decreasing.

```
C:\Users\nick_pc\Desktop\CI PRACS>bpn.py
RED
```

This output is printed on the console and it predicts themissing flower as red.

Conclusion – In this program I learn how a Backpropagation Neural Network works. How to train a model on the neural network and predict the output for new given values. I was able to adjust the weights and bias to make the prediction more desirable.