Optical Character Recognition

Name: George Saman

CWID: 802965368

Table of Contents

Abstract

Nowadays, companies using paper based systems are striving to improve their system's performance by digitalizing their information. A digital system provides a secure and easy way of sharing and retrieving information, which is one of the main issues in a paper based system. This problem becomes a burden as paper stacks up, since productivity is tied to the amount of time wasted searching for a relevant piece of information. Meaning as paper stacks up, productivity goes down.

Fortunately, Optical Character Recognition, OCR in short, is the key for solving this problem. OCR is a way of converting text written on a paper or an unsearchable pdf to a searchable, editable format. Thus, Companies utilizing this technology can digitalize their systems in an incredibly short time, relatively speaking when compared to retyping everything from scratch.

Introduction

Character recognition has gained much attention through the past decades, due to its importance in image processing, allowing a machine to read text. Started in 1957 by Frank Rosenblatt, Charles Wightman and others, where they successfully designed the first neurocomputer which could detect characters using a neural network and what they so called “a high-resolution camera”. Their neural network was trained using a technique called “**The Perceptron Learning Rule”**, only capable of training a single layer network. Due to this limitation, the training method used here is the **“Back Propagation”** algorithm, capable of training multi-layer networks.

Instead of using a camera or an optical scanner to input a text to the neural network, a PNG image generated from an unsearchable pdf text is used, eliminating the noise that could arise due to dimmed lighting or other kinds of distortion.

The OCR implemented in this project detects monospaced characters, in other words, all characters have a fixed width and height. The reason behind this, is that the efficiency increases greatly when using a monospaced font. An example of such a font is called “OCR A Extended”, which is what this OCR is trained on.

Review of the project

For purpose of ease of understanding, the training method and how the network works is described first, then its architecture is described next.

> ***NOTE:*** *The purpose of this documentation is to describe how the network works. Thus, back propagation algorithm explanation is not described here. Although, a brief explanation of how the program implements it is introduced in the code section.*

1. Training Method

The neural network contains 3 layers, an input layer (input image pixels serve as the input to it), a hidden layer and an output layer. Each neuron in the output layer categorizes the input set to a classified output. Meaning, for a character 'a' to be trained, only the neuron which classifies character 'a' fires a value close to 1, the rest fires a value close to 0. The accuracy of the neural network is defined as how close the actual is to the target output, which is improved by decreasing the error defined per this formula:

$$\textbf{Total Error} = \tfrac{1}{2}\sum (out_z - target_z)^2$$ where z is the output neuron.

This approach of decreasing the mean squared error is called **"The Gradient Descent"**, used in conjunction with a training algorithm called **"Back Propagation"**, adjusting the network's weights in such a way that the total error converges to a global minimum. The weights and biases are initialized to random numbers between -0.5 and 0.5. If the weights were initialized such that they lie between 0 and 1, the neural network will never converge. This is because the input to all hidden neurons will be higher than the active input range of the sigmoid function, resulting in an output always 1.

The network is designed and then trained over a set of inputs "X" such that for each element "E" in set "X", only neuron "Z" that is classifying element "E" fires the highest value among all other output neurons. This training algorithm is described in the pseudo code below:

```
Initialize weights and biases ()
while Total Error > Target Error
        for each element E in set X
                target output = {1 for neuron Z classifying E; 0 otherwise}
                feed forward through network ()
                calculate total error ()
                back propagate through net and adjust weights ()
loop until criteria is met
```

After the network is trained on all elements in "X" it can be used for recognition.

The number of output neurons depends on the number of characters to be classified. Thus, If X characters are to be classified then X neurons are present. As for the number of inputs, the network has M*N inputs, where M is the width and N is the height of the character's image. The hidden layer has 100 neurons.

Since this neural network can classify inputs only if their quantity is fixed, the image must pass a pre-processing phase to allow such input uniformity.

2. Pre-Processing:

As described in the previous section, the number of inputs to the neural network must be fixed. Thus, to allow such uniformity, the image is resized to a fixed width and height. This can be done by following these steps:

1. Binarization, a method of converting the image into black and white. (i.e. black = 1, white = 0)
2. Cropping, detect the character's borders of the image (i.e. left, right, top and bottom borders).
3. Normalization, Resize to desired width and height.

These steps are illustrated in figure 1.

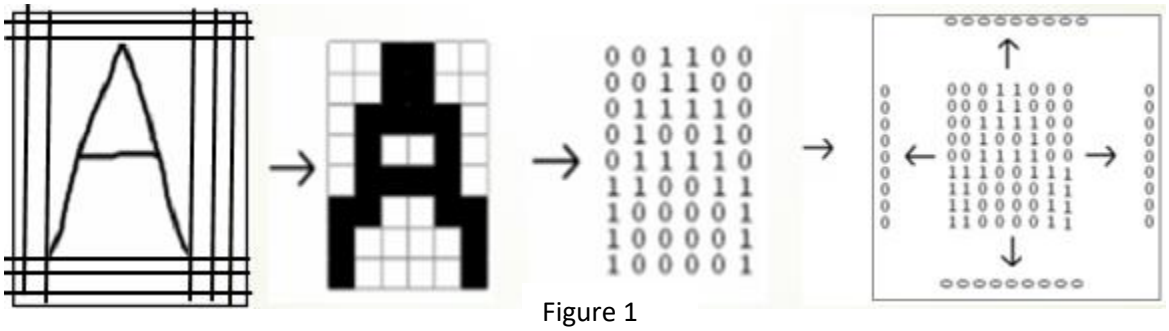

Figure 1

Note: Programs to crop, normalize, extract paragraphs, lines, words and characters are written by me.

Figure 1 shows the binarized input being searched for borders, converted to a matrix format and then resized. The output of the pre-processing stage is then inputted to the neural network.

3. Network Architecture

The overall architecture of the neural network is show in figure 2. The input to this system is '***I'***, block ***'P'*** represents the pre-processing phase which outputs ***'PI'*** (pre-processed input). ***'PI'*** serves as the input to the network, this input is a 18X16 matrix which is multiplied by weights connecting inputs to neurons in the hidden layer. Thus, the weight matrix between the input and the hidden layer is of size 18 X 16 X 100, where 100 is the number of neurons in the hidden layer.

The activation function used for both hidden and output layer's neurons is the **logistic sigmoid** function. Meaning, their outputs are in the range of 0 to 1. The hidden neurons outputs are fed as an input to the output layer. These inputs are then multiplied by weights connecting hidden neurons to output neurons. The output neuron with the maximum output value signifies greatest probability. Thus, all neurons will fire a value around 0 signifying low probability except the one classifying the input character will fire a value around 1.

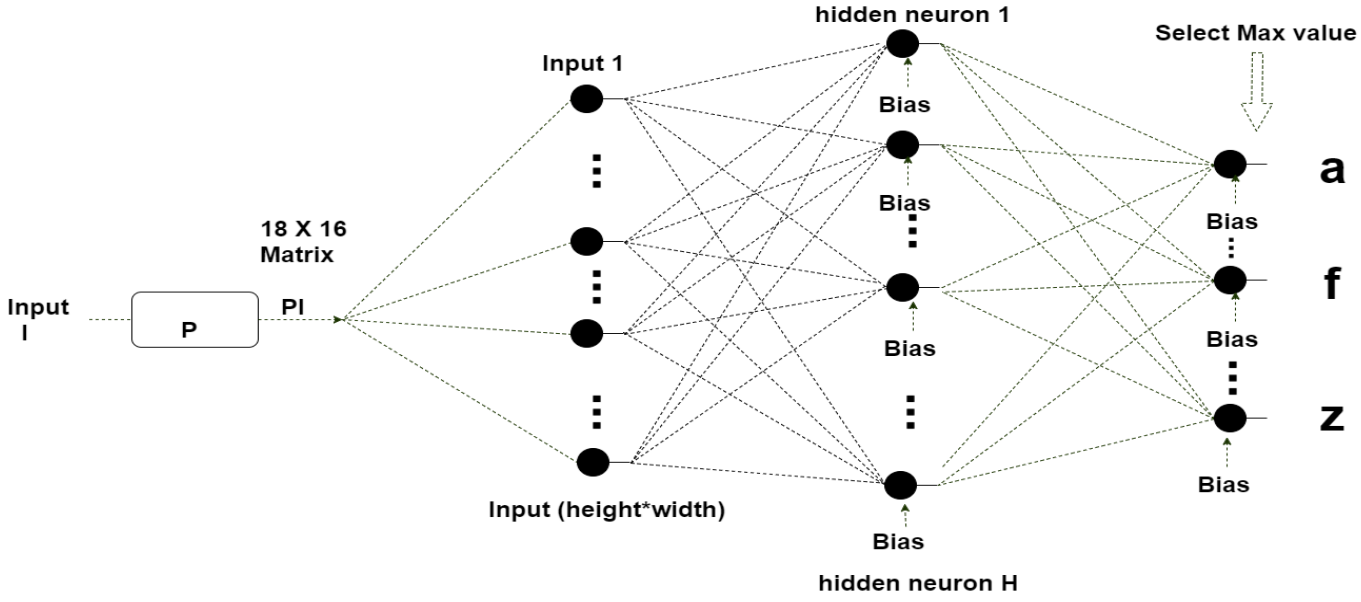

Figure 2

4. Character recognition and document scanning

The document to be digitalized is first searched for paragraphs, then lines from paragraphs, words from lines and eventually characters from words, these characters are the input to the neural network. The network will output a digital character matching the character in the image.

A paragraph is defined as a combination of black pixels with a white gap of size 60 pixels following it (This is the newline gap for a font size 20). Once the paragraph is found, a line is found by first detecting a horizontal black pixel and that defines the top, then the last encountered black horizontal pixel defines the bottom. Once the line is found, characters are found for by first detecting a vertical black pixel and that defines the left, then the last encountered black vertical pixel defines the right. This is repeated until no more encountered vertical black pixel is found. (i.e. no more characters in line). Once all characters are found, a word is defined as a combination of characters followed by a white gap of size 20 pixels. This is illustrated in figure 3.

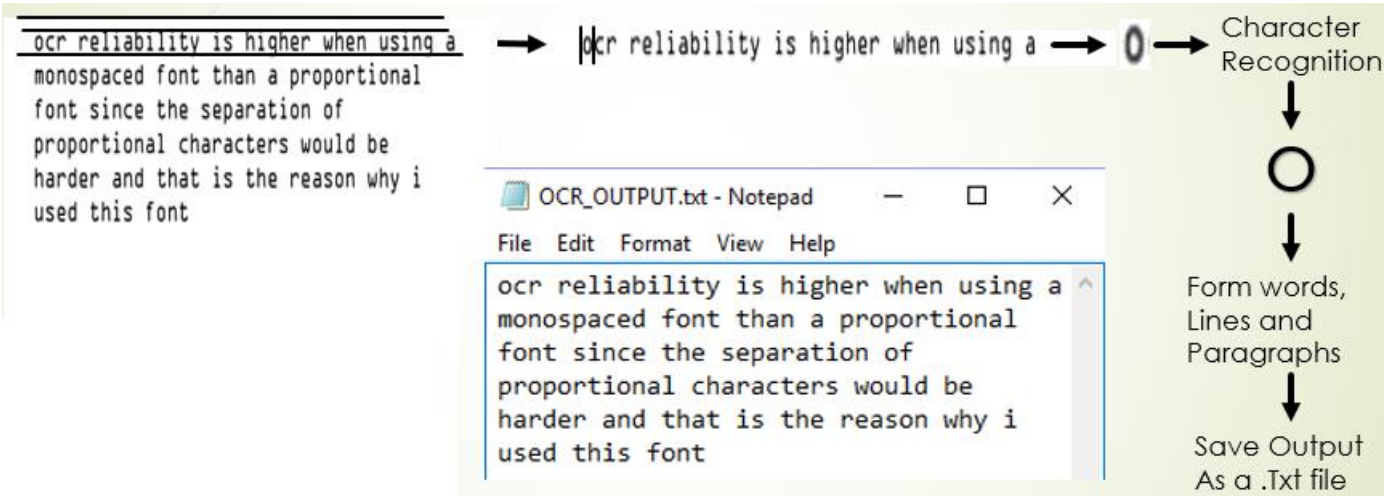

Figure 3

Once the document is scanned for characters, each character passes the pre-processing phase, then inputted to the neural network to be recognized.

Experimentation

Parameters that could be changed are:

1. Learning Rate.
2. Number of neurons in the hidden layer.
3. Momentum

Experimenting the neural network performance with the adjustment of the learning rate is shown in figure 4. Figure 4.a, b, c shows learning rates of 0.1, 0.5 and 1 respectively, the activation function used is sigmoid and number of hidden neurons is 55. All experiments are done with a target mean square error of 0.0035. **Note:** best results were obtained at a total target error of 0.001.

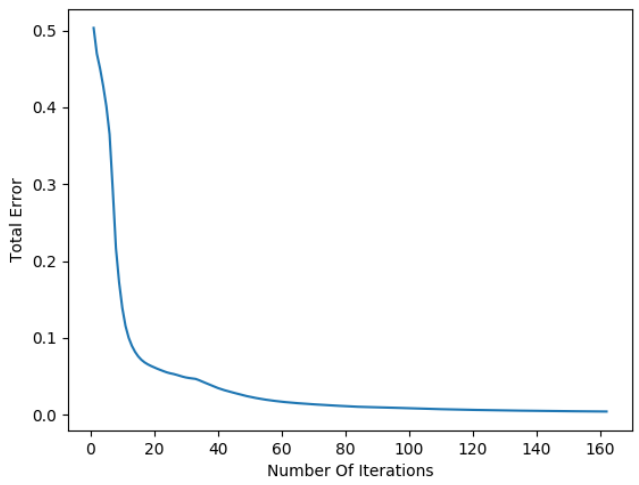

Figure 4.a. Training took 161 epochs.

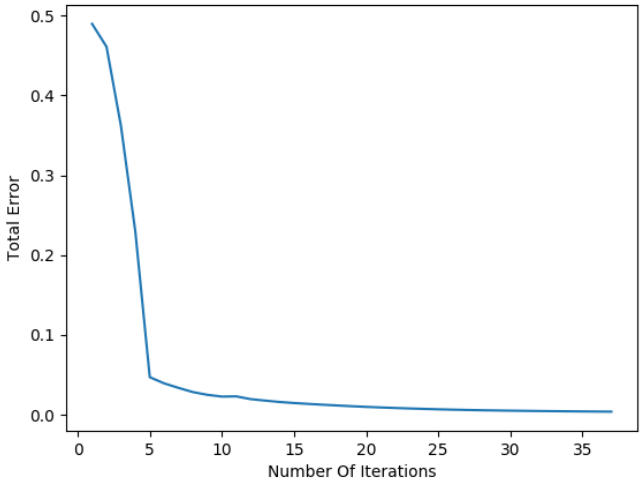

Figure 4.b. Training took 38 epochs.

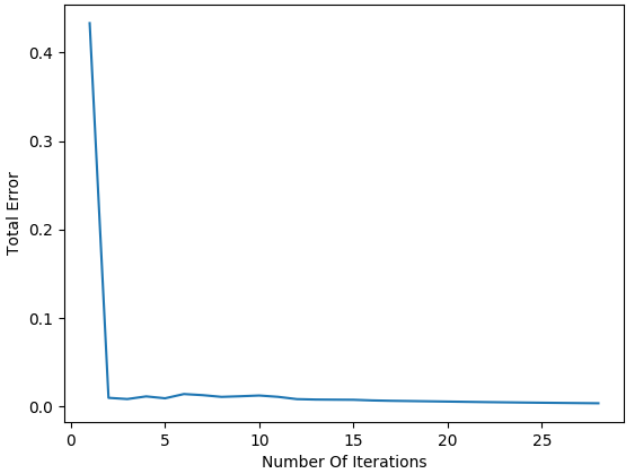

Figure 4.c. Training took 27 epochs.

As it can be seen from the above figures, the higher the learning rate, the faster the convergence to the global minimum.

When the number of hidden neurons is decreased, the convergence takes more epochs, even though the network will converge, it's character recognition performance is decreased. The best recognition was found when the number of hidden neurons were 100. This can be shown in figure 5, where learning rate is set to 0.5 and target error is 0.0035. The number of training iterations is 416 for 10 neurons compared to 38 for 55 neurons, 27 for 100 and 10 for 150. The recognition performance at 150 hidden neurons was lower than that of 55. Meaning, increasing the number of hidden neurons beyond a limit will give bad results.

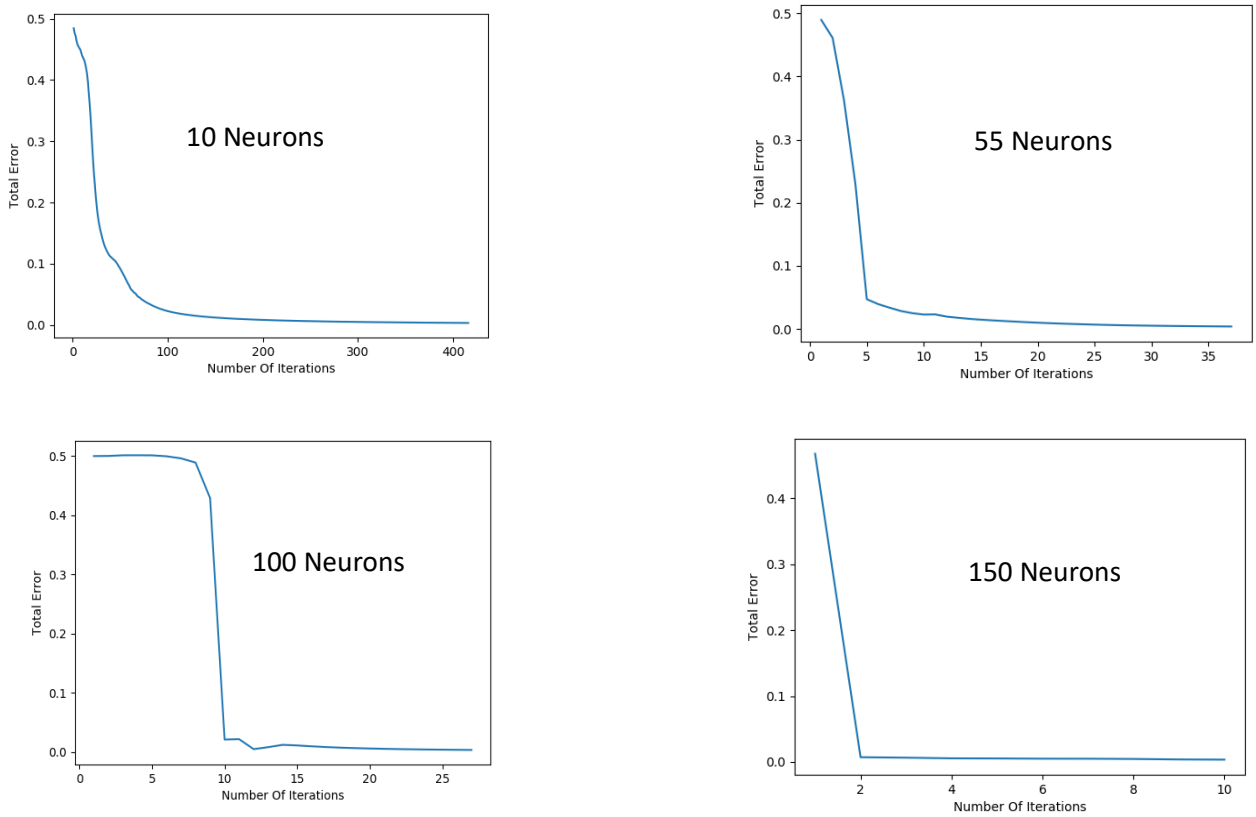

Figure 5

As a last parameter, the momentum is varied from 1 to 0.1 while keeping all other parameters the same, the result is that the network did not have enough momentum to converge to a global minimum instead it converged at a local minimum. The same is observed with a momentum of 0.5 and 0.95.

Another approach would be increasing the learning rate for the network to converge. A learning rate of 1 and 2 were experimented with a momentum of 0.95, both did not converge.

Conclusion

Based on this OCR method described in previous sections, the system can detect 26 monospaced characters (a to z), it can also be trained to recognized more characters by subjecting the neural network to more training sets. This is one of the future goals of this project, enhancing the performance of the OCR by training it on numbers, capital letters, periods, commas, question and exclamation marks.

The OCR cannot detect images or fonts other than 20. This can be done by adding a program to detect the font size by measuring the width of the first detected character. This is also a future work.

Also, as a future work, another way to enhance the OCR performance is by using a dictionary, referring to it whenever a word is detected, checking if it is an English word or not, this makes sure all characters are recognized correctly.

References

1. https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

2. Principles of Training Multi-Layer Neural Network Using Backpropagation Mariusz Bernacki Przemysław Włodarczyk

3. Multi-Font/Size Character Recognition and Document Scanning, Ritesh Kapoor, Sonia Gupta, C.M. Sharma.

4. Neural Network Based English Alphanumeric Character Recognition, Md Fazul kader and Kaushik deb.

5. Visual Character Recognition using Artificial Neural Networks, Shashank Araokar.

6. Neural Network Design, 2nd edition, Hagan, Demuth, Beale.

Code

Code such as preprocessing, cropping paragraphs, lines, words and characters are not described here.

USER MANUAL:

With the provided code files, open OCR.py, there is a **parameters** section at the top.

Learning rate, momentum, target error, width and height of the image, number of hidden neurons, number of training samples are all input parameters to the neural network which can be adjusted from here.

The last parameter is the documentLocation which tells the neural network where to find the document to be recognized.

For example, documentLocation = 'paragraphs/ocr1.png'

```
########################################
# George Saman                         #
# Character Recognition using ANN      #
# with backpropagation algorithm       #
# Email: georgesaman@csu.fullerton.edu #
########################################

from PIL import Image as im
import numpy as np
import matplotlib.pyplot as plt

# files i wrote
import CropAndNormalize as cAn
import LineExtraction as lN

########################################Letters To Train
letters = ['a','b','c','d','e','f','g','h','i',             # Letters to learn
           'j','k','l','m','n','o','p','q','r',
```

```
            's','t','u','v','w','x','y','z']

############################################################


#-------------------------------------------------------------------------------------
#------------------------------Weights initialization---------------------------------
#-------------------------------------------------------------------------------------

def initializeWeights(width,height,numberOfHiddenNeurons):
# input to hidden layer weights
    Wi_h = np.random.random(size=(numberOfHiddenNeurons,height,width))-0.5
# hidden to output weights
    Wh_o = np.random.random(size=(26,numberOfHiddenNeurons))-0.5
# hidden layer biases
       Bh   = np.random.random(numberOfHiddenNeurons) - 0.5
# output layer biases
       Bo   = np.random.random(26) - 0.5

    return Wi_h, Wh_o, Bh, Bo


#-------------------------------------------------------------------------------------
#------------------------------Threshold Function-------------------------------------
#-------------------------------------------------------------------------------------
# sigmoid function
def logistic(summation):
       out = 1 / (1 + np.exp(-summation))
       return out

#-------------------------------------------------------------------------------------
#------------------------------FEED FORWARD THROUGH NETWORK---------------------------
#-------------------------------------------------------------------------------------
# feed forward through net
```

```
def feedForward(normalized, Wi_h, Wh_o, Bh, Bo):
    n_h                                 = 0
    [numberOfHiddenNeurons,height,width] = Wi_h.shape
    #-----------------------------feed Forward, From input layer to hidden
    outputOfHiddenNeurons               = []
    netInputForHiddenNeurons            = []
# forward pass from input to hidden
    for hiddenNeuron in range (0,numberOfHiddenNeurons):
# calculating net activation input
             for i in range (0,height):
                 for j in range (0,width):
                    WxP = Wi_h[hiddenNeuron,i,j] * normalized[i,j]
                    # The overall sum of W's X P's
                    n_h   = n_h + WxP
# total input = WP+Bias
             n_h                    = n_h + Bh[hiddenNeuron]
# calculate and save hidden neurons output
             outputOfHiddenNeurons.append(logistic(n_h))
# save total net input
        netInputForHiddenNeurons.append(n_h)
        n_h                   = 0


    #---------------------------feed forward, from hidden to output
# out of hidden layer multiplied by weights from hidden to output layer
    outHiddenXweightsH_O      = outputOfHiddenNeurons * Wh_o
# this is a 26 X 10 matrix, each row contains the weights connecting hidden neurons to
specific out neuron.
# sum of all (Wh_o weights X hidden neuron outputs), each row is the total input for
each output neuron
    netInputForOutNeurons     = np.sum(outHiddenXweightsH_O, axis= 1)
    outputOfOutNeurons        = []


# Find and Calculate output of out neurons
    for outputNeuron in range(0,26):
       # the input to the kth output neuron
       totalInputForNeuron   = netInputForOutNeurons[outputNeuron] + Bo[outputNeuron]
```

```
        # get the output and save it
        outputOfOutNeurons.append(logistic(totalInputForNeuron))
    return outputOfOutNeurons, outputOfHiddenNeurons        # return outputs

#-------------------------------------------------------------------------------
#-----------------------------Calculate Error At Output Neurons-----------------
#-------------------------------------------------------------------------------

def calculateErrorAtOutput(outputOfOutNeurons, targetOutput):
    outputError =[]
# Calculating the error at the output
    for outputNeuron in range(0,26):
        # error = out - target
        outputNeuronError     = outputOfOutNeurons[outputNeuron] -
        targetOutput[outputNeuron]
        # save error for all outputs

        outputError.append(outputNeuronError)
        return outputError

#-------------------------------------------------------------------------------
#-----------------------------BACK PROPAGATE AND ADJUST WEIGHTS ----------------
#-------------------------------------------------------------------------------

def backPropagate(Wi_h, Wh_o, Bh, Bo, normalized, outputError, outputOfOutNeurons,
outputOfHiddenNeurons, learningRate, momentum):
    # save old weights for b.prop from hidden to input
    oldWh_o    = np.array(Wh_o[:,:])
    oldWi_h    = np.array(Wi_h[:,:])

    [numberOfHiddenNeurons,height,width] = Wi_h.shape
    #-----------------Back Propagating from output to hidden and adjusting weights
    for outputNeuron in range(0,26):
        for hiddenNeuron in range(0,numberOfHiddenNeurons):
```

```
 # calculating the adjustment which is learning rate * error at current output neuron
* sigmoid derivative *  output of current hidden neuron)

            adjustment                          = (learningRate *
outputError[outputNeuron] * outputOfOutNeurons[outputNeuron] * (1 -
outputOfOutNeurons[outputNeuron]) * outputOfHiddenNeurons[hiddenNeuron])


# adjusting weights per this formula, Wnew = momentum* Wold - adjustment

       Wh_o[outputNeuron, hiddenNeuron] = (momentum * Wh_o[outputNeuron,
hiddenNeuron]) - adjustment

 #-------Back Propagating from hidden to input and adjusting weights

    for hiddenNeuron in range(0,numberOfHiddenNeurons):

        deltaTotalError_hiddenNeuron = 0

        for outputNeuron in range(0,26):

   # Calculate delta error at each output neuron with respect to current hidden neuron

            deltaErrorOutputNeuron_hiddenNeuron = outputError[outputNeuron] *
outputOfOutNeurons[outputNeuron] * (1-outputOfOutNeurons[outputNeuron]) *
oldWh_o[outputNeuron,hiddenNeuron]

# delta total error with respect to current hidden neuron

       deltaTotalError_hiddenNeuron = deltaTotalError_hiddenNeuron +
deltaErrorOutputNeuron_hiddenNeuron

  # loop over all input weights connecting to current hidden neuron

        for i in range (0,height):

            for j in range (0,width):

# delta Total Error with respect to weight to be adjusted. this weight is connecting
input to current hidden layer

                deltaTotalError_inputTohiddenNeuronWeight =
deltaTotalError_hiddenNeuron * outputOfHiddenNeurons[hiddenNeuron] *(1 -
outputOfHiddenNeurons[hiddenNeuron]) * normalized[i,j]

# Adjust Weights

        Wi_h[hiddenNeuron,i,j] = (momentum * Wi_h[hiddenNeuron,i,j]) - (learningRate *
deltaTotalError_inputTohiddenNeuronWeight)


    return Wi_h, Wh_o


#-------------------------------------------------------------------------------

#------------------------------TRAIN NETWORK ------------------------------------

#-------------------------------------------------------------------------------
```

```
def trainNet(Wi_h, Wh_o, Bh,
Bo,height,width,numberOfTrainingSamples,learningRate,momentum,targetError):

    iteration  = 0

    totalError = 1

    errorList  = []     # to save all total error generated

    y_axis     = []     # for plotting the error minimization at the end

# loop until criteria is met

    while totalError > targetError:

# loop for all letters to be trained

        for letterToTrain in range(0,26):

# target output is all zeros except the one to be trained

             targetOutput                = np.zeros(26)
             targetOutput[letterToTrain] = 1

            for n in range (0,numberOfTrainingSamples):

#---------------Cropping and Normalizing the image to have a uniform input to the ANN


             # training sample image file name

             trainingSample      = 'samples/%s%d.png' %(letters[letterToTrain],n)

             character_in        = im.open(trainingSample)    # load Image

# Convert to BW, 1->black

              blackAndWhite       = cAn.convertToBW(character_in)
              toggledBW           = cAn.toggleOnesAndZeros(blackAndWhite)

# Crop Image to get character only  and the resize

              croppedBW           = cAn.crop(toggledBW)
              normalized          = cAn.normalize(croppedBW,width,height)

#----------------------- end of pre processing phase

                # feed forward

                outputOfOutNeurons, outputOfHiddenNeurons = feedForward(normalized,
Wi_h, Wh_o, Bh, Bo)

# calculate error at output neurons

               outputError         = calculateErrorAtOutput(outputOfOutNeurons,
       targetOutput)

# backpropage and adjust weights

                Wi_h, Wh_o          = backPropagate(Wi_h, Wh_o, Bh, Bo, normalized,
outputError, outputOfOutNeurons, outputOfHiddenNeurons, learningRate,momentum)


#----------------------Calculate the mean squared error
```

```
        totalError = 0
        for x in range(0,26):
            squared     = 0.5 * outputError[x]**2
            totalError  = totalError + squared

        print('Total Error = %f' %totalError)
        iteration = iteration + 1
        errorList.append(totalError)
        y_axis.append(iteration)

    #------------------Plot Total Error vs Iteration
    print('Total Number of iterations %d' %iteration)
    plt.plot(y_axis, errorList)
    plt.ylabel('Total Error')
    plt.xlabel('Number Of Iterations')
    plt.show()

    return (Wi_h, Wh_o, Bh, Bo)


#-------------------------------------------------------------------------------------
#------------------------------Recognize Character-----------------------------------
#-------------------------------------------------------------------------------------
 # Returns the character recognized

def recognizeCharacter(inputNormalized,Wi_h,Wh_o,Bh,Bo):
   # feed forward
        outputOfOutNeurons, outputOfHiddenNeurons = feedForward(inputNormalized, Wi_h,
Wh_o, Bh, Bo)
   #character recognized is neuron with highest output
        maxOut                                    = np.argmax(outputOfOutNeurons)
    return letters[maxOut]
```