Optical Character Recognition Using Artificial Neural Networks

Colby McKibbin
Colorado State University-Pueblo Honors Thesis
Spring 2015
Advisor: Dr. Jude DePalma

Abstract

Optical character recognition is a complicated task that requires heavy image processing followed by algorithms used to convert that data into a recognized character. While programs exist that already can perform character recognition, they require intensive processing that is not always necessary because they recognize a wide range of characters spanning numerous fonts. In applications where a specific character set in a specific font is defined, the processing requirements can be easily reduced by developing a OCR system tailored to those specifications. One way to do this is with artificial neural networks. This method has many tunable parameters, and in many cases the optimal settings may need to be determined through trial and error. This purpose of this project was to develop and train an artificial neural network on the Raspberry Pi microcontroller and determine the optimal layer settings to identify a specific set of characters. An additional goal was for the coding to be customizable and easy to use so it could be used for other recognition tasks as well beyond just character recognition.

Keywords: optical character recognition, python, Raspberry Pi, neural network

1: Indroduction

1.1 Problem Overview

The focus of this research project was to develop a method of optical character recognition (OCR) for a specific character size and font set. The specific characters used were those provided and defined by the 2015 IEEE Region 5 Robotics Competition. The characters were provided in the form of stickers to be placed on maze walls for robots to identify as they solved the maze. The characters were to be posted on only the red and blue walls in the maze. Capital letters A through Z were to be posted exclusively on the red walls; numeric values 0 through 9 were to be posted exclusively on the blue walls; and the "special characters" !, @, #, $, %, ^, &, *, (, and) were to be posted on both the red and the blue walls. Because this would be for a robotics competition and would occur while a robot was solving a maze, the OCR system had to be portable and was therefore developed entirely on a Raspberry Pi microcontroller. The competition also prevents the device from connecting to the internet and had therefore there was a limit to the processing resources available.

1.2 Problem Approach

A quick solution to the problem would be to use Google's Tesseract program which has a library available for the Raspberry Pi microcontroller which was used for the project. This program, however, has a wide range of fonts for which it can identify characters, and identifies the data passed to it from all character sets in a single process. This means it requires much more processing power than would be needed to identify the separate, unique set of characters that each wall color contained. In addition, there are some characters in the red wall character set and the blue wall character set that would be nearly impossible to differentiate without using a separate identification set for each. For example the numeric value "0" or the letter "O" could easily be misidentified as the other.

Due to the drawbacks of Google's Tesseract alternative method was developed for use on the Raspberry Pi. This method uses the Pybrain library for Python to create two artificial neural networks (ANNs) to accomplish the OCR task. This library provides a high amount of customization options for neural networks, so depending on the task the default parameters can sometimes require adjustment in order to optimize the results. In this case the classification problem presented was fairly simple as there were relatively few output classes available for the classification with high variation in the features of the classes, so many of the default options were appropriate. One key parameter that needed to be tuned, however, was the layer types for the hidden layer and the output layer of the neural network. In order to determine which layer settings provided the best solution to the OCR task, multiple settings were tested and compared.

1.3: A Brief Overview of ANNs

In general, an artificial neural network is a virtual network developed internally within a program used to convert a series of inputs to a series of outputs by running calculations on inputs at multiple layers. In the most basic neural network there is an input layer, a hidden layer, and an output layer. Each layer is comprised of multiple neurons. The number of input neurons is equal to the number of input data values, the number of output neurons is equal to the number of output values, and the number of hidden neurons is simply an arbitrary value. There is not a definitive formula for calculating the number if hidden neurons, but less the less linear data is the more hidden neurons will be required for processing. Determining the number is usually just a matter of trial and error or experience[9]. In a basic feedforward network each neuron in a layers is connected to each neuron in the layer immediately succeeding it, and there are no connections moving backward in the network[5]. A simple feedforward network with 2 input neurons, 3 hidden neurons, and 2 output neurons is shown in Figure 1.

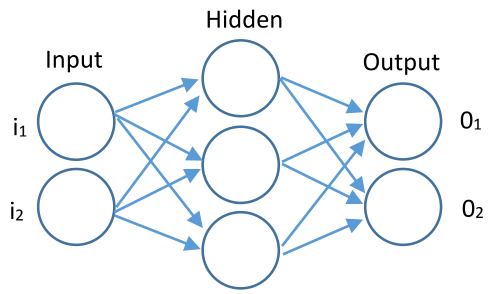


Figure 1. A Basic ANN Structure

Neural networks can also involve multiple hidden layers, but for the most part a single hidden layer is adequate. Each neuron in the hidden and output layers have a calculated activation level based on the values or activation levels of all neurons in the layer that is immediately preceding it. In order to calculate this activation value, weights are assigned and trained for each connection in the network. A bias is sometimes also factored in to the calculation, which is used to set a threshold value for activation[3]. For a basic network with an input layer, hidden layer, and output layer each neuron in the hidden layer assigns unique weight values to each input neuron to determine its own activation level; and the output neurons follow the same method using the hidden neurons. The calculations done by each neuron is shown in Equation 1 below.

$$y = f\left(\sum_{j=0}^{n} i_j * w_j + bias\right) \qquad \text{Equation 1}$$

Where n is the number of neurons in the preceding layer, i_j's are the values or activation levels of the neurons in the preceding layer, w_j's are the weights assigned to the neurons, and f(x) is the activation function.

Various activation functions can be used to process the input, weight, and bias values. Three of the most basic activation functions are the linear activation function (Equation 2), the logistic activation function (Equation 3), and the hyperbolic tangent activation function (Equation 4). These are just examples of basic activation functions, many others exist as well[2].

$$f_{linear}(x) = x \qquad \text{Equation 2}$$

$$f_{logistic}(x) = \frac{1}{1+e^{-x}} \qquad \text{Equation 3}$$

$$f_{tanh}(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \qquad \text{Equation 4}$$

The bias defined earlier shifts the graphs of these functions to the left and right which effectively adjusts the threshold value required for activation of the network. Figure 2 shows a graph of these activation functions.

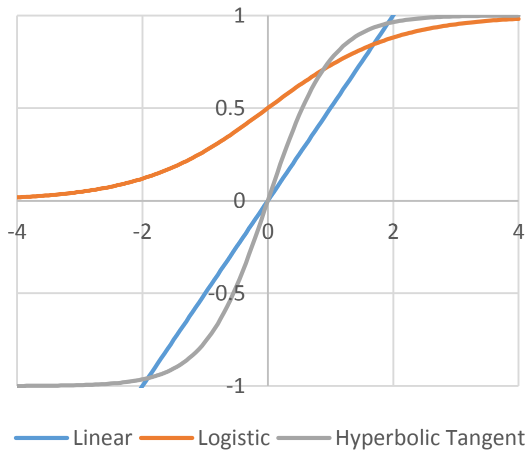


Figure 2. Graph of Activation Functions

Once the network is built by defining the neuron count in each layer, it can be trained. For the initial training run, arbitrary weight values are assigned throughout the network and the training data set is fed through the network. The error based on the expected results in the training data is then used to adjust the arbitrarily assigned weights. The updated weights are then used for further training. The most common and basic way to accomplish this training is through a backpropagation training algorithm which calculates the weight adjustment for each weight synapse by using the partial derivate of the error function with respect to the weight. This requires the derivative of the activation functions used, so for backpropagation training the activation function must be differentiable. Equations 5 and 6 show the basic equations used for the weight adjustment.

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial o}\frac{\partial o}{\partial x}\frac{\partial x}{\partial w} \qquad \text{Equation 5}$$

Where E is the error function for the neuron in question, w is the weight for the current synapse, and o is the output of the activation function, and x is the input to the activation function or $x = \sum_{j=0}^{n}(i_j * w_j)$. The weight adjustment is then given by:

$$\Delta\mathrm{w} = -\alpha\frac{\partial E}{\partial w} \qquad \text{Equation 6}$$

Where α is a constant representing the desired training rate.

For a more in depth look at the calculations involved with a neural network, see D. Kriese's *A Brief Introduction to Neural Networks* [3] or Chapter 3 of Zhang and Gupta's *Neural Networks for RF and Microwave Design* [9]. Using the previous calculations, neural networks can be built, trained, and used for both regression and classification tasks.

In summary, a neural network is a method of converting a set of inputs to a set of outputs by passing them through one or more hidden layers. Each layer is comprised of multiple neurons. In a feedforward network, every neuron in a layer sends its value or activation level to every neuron in the succeeding layer. Each individual neuron assigns weights to each of its input neurons to determine its own activation level. The neurons in the succeeding layers repeat this process until the output layer is reached. During backpropagation training of a network, the error from the target value in the output layer is used to adjust the weights assigned throughout the network. The training is repeated until acceptably low error exists. A basic feedforward artificial neural network that is trained using backpropagation was used for classification in this project.

2: Methodologies

2.1 Overview

There were three main aspects to determining the optimal settings for the neural network. First, image data captured by the Raspberry Pi camera module had to be processed and converted into a binary array in order to be processed by the network. Next, the binary arrays were used to develop datasets for both training and testing the networks. This was done using a GUI that was developed in Python for use on the Raspberry Pi. Finally, the training dataset was used to build multiple neural networks using different layer settings, which were then tested on the test dataset to determine the optimal layer settings. Only the red character set was used to build and test the networks. The red sets contains more output classes and should therefore be more difficult to identify than the blue set, thus the optimal settings determined for identifying from the red character set would be sufficient for the identifying from the blue character set as well. A Raspberry Pi microcontroller was used to do all coding, processing, and production of data for results.

2.2 Image Processing

All of the image processing was done using the Open-CV library for Python. The camera captures images in full RGB color, so the first step in converting this into a binary array was converting it to a grayscale image to simplify the remaining steps of the processing. The grayscale image for used for processing the capture of the letter "G" is shown below in Figure 3.

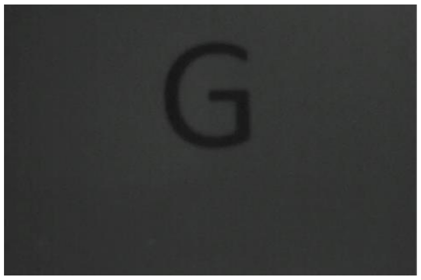


Figure 3. Grayscale Image Capture

Once the image has been converted to a grayscale image, the noise has to be removed. This is done using the blur function. The blurred image is shown in Figure 4.

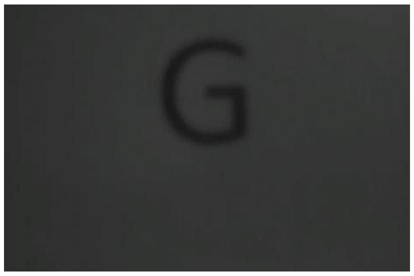


Figure 4. Blurred Image

With the noise eliminated, an adaptive threshold was applied to the image using the mean method to convert it to binary. This threshold method ensures that the threshold is applied properly regardless of non-uniform lighting or other such conditions. It uses the average pixel value of a cluster of pixels in the image to set a threshold value for that cluster. All pixel values below that threshold become black and the rest become white. This results in a white letter with black background as shown below in Figure 5.

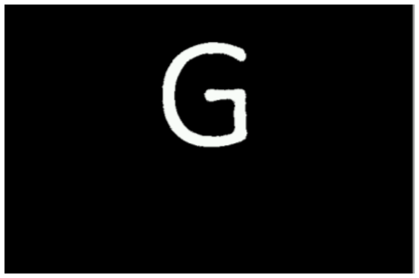


Figure 5. Image with Adaptive Threshold

The image is then dilated, or thickened, to reinforce the letter, and ensure there are no small gaps that may have been removed by the adaptive threshold. The dilated image is shown below in Figure 6.

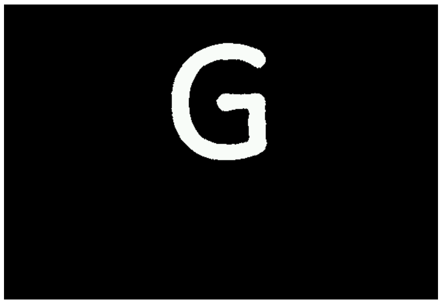


Figure 6. Dilated Image

The contours of the image are then identified, and the contour area that is the largest is assumed to be the character. This area becomes the region of interest. The height and width of this region are returned for calibration reference for the user. The user can input these values or use these values to determine more appropriate values to input at the beginning of codes that depend on the image processing. This ensures that future captures from the same position will adjust the region of interest to a minimum dimension. This is important because characters that are split in the middle such as "!" will identify the two separate contours. This results in a misidentification unless the region of interest for the largest contour is expanded to include the rest of the character. In addition, thin characters like an "I" would just appear as a solid black box unless the width is expanded.

After the current dimensions of the region of interest are found and returned, if they are not at least the size of the user-defined calibration values, the region is expanded to be at minimum the size of the calibration values. The updated region of interest is marked on the original camera capture by a red box as shown in Figure 7 below. This updated original image can be used in graphical user interfaces (GUIs) to display what the program is isolating as the letter.

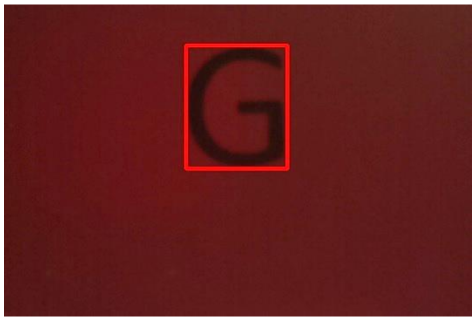


Figure 7. Original with **ROI**

Finally, the image data in the region of interest is resized to a 10 by 10 size and converted to 10 by 10 binary matrix, or a matrix comprised of entirely ones and zeros. The matrix for the processed "G" is printed to the LX Terminal for reference and shown below in Figure 8. It is clear that the ones accurately represent the shape of the "G".

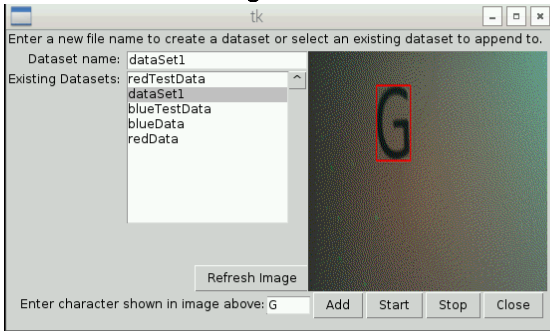
```
[[0 0 0 1 1 1 1 1 1 0]
 [0 0 1 1 1 0 0 1 1 1]
 [0 1 1 0 0 0 0 0 0 0]
 [1 1 0 0 0 0 0 0 0 0]
 [1 1 0 0 0 1 1 1 1 1]
 [1 1 0 0 0 1 1 1 1 1]
 [1 1 0 0 0 0 0 0 1 1]
 [0 1 1 0 0 0 0 0 1 1]
 [0 1 1 1 0 0 0 0 1 1]
 [0 0 0 1 1 1 1 1 1 0]]
```

Figure 8. Matrix from Image Processing

This 10 by 10 matrix is "squashed" to be a 1 by 100 array, and the image is now in a proper format to be used as an input to the neural networks. The code developed for image processing is included with comments in Appendix A-1. For more information regarding this coding for image processing, see the Open-CV documentation[1].

2.3 Building the Datasets

A GUI was built in Python to simplify the building of the data. In the program, the user can select and exist dataset to append to or create a new dataset. Once the dataset is created and named, the start button will be clicked and the program will collect the camera data. It then uses the previous image processing steps to isolate the character, convert it to an array, and display the original image with a red box drawn around the character it has identified within the image. From here the user can either input the character isolated and click the "Add" button, or click the "Refresh Image" button if the capture and character shown in the display are incorrect. When the "Add" button is clicked, the image data converted to an array is saved in the input data, and the entered character is saved in the output data. The code for the GUI is included in Appendix A-2 and a screenshot of the GUI is shown in Figure 9.



tk

Enter a new file name to create a dataset or select an existing dataset to append to.

Dataset name: dataSet1

Existing Datasets: redTestData, dataSet1, blueTestData, blueData, redData

Enter character shown in image above: G

Figure 9. GUI for Building Datasets

For both the red and the blue wall characters a training and testing dataset was created. For each set, the training set consisted of 9 camera captures of each character with the walls they were on held at slightly different angles each time to account for variation in their positions in a working environment. For the testing data, lists of 100 random integers representing the characters in the sets were generated. For the red testing dataset the random integers ranged from 1 to 36 as that is the number of characters in the red set; and for the blue set the random integers ranged from 1 to 20. These lists of random integers were then converted to their corresponding characters, and the testing datasets were then built based on those lists.

2.4 Training and Testing the Networks

Using the training dataset for the red character sets, neural networks were built using Pybrain. Each of the networks used a different combination of hidden layer type and output layer type. The hidden layer types used were the linear, logistic, and hyperbolic tangent; and the output layer types were SoftMax and hyperbolic tangent. The SoftMax layer is a layer type that uses the logistic curve to get a series of output values, and then converts those output values to probabilities by dividing each individual value by the sum of all values. Each network was trained for 20 epochs; which, due to the limited number of outputs and higher variation in the input data, should have been ample training. The code for building and training the red network is shown in Appendix A-3.

Once the networks were trained, the testing data was passed through the networks. For each data entry, the input item was passed through the network to obtain a series of outputs, the index max argument of the outputs was used to determine which character was identified, and that was compared to the actual character listed in the output array of the dataset to determine if the network was correct. The correct number of responses was then converted to a percentage to determine the accuracy of each network. The code for testing the networks is included in Appendix A-4. For additional information regarding the coding related to the neural networks see the Pybrain documentation[8].

3: Results and Discussion

3.1 Results

The training data was plotted for each of the training types. When the hyperbolic tangent function was used as the output layer type, the error returned during training was extremely high relative to the training done with the SoftMax layer, so the two output layer types were split onto two different graphs. The training data from each layer combination is shown below in Figure 10 and Figure 11, where MSE is the mean squared error determined when the feedforward operation was performed during training.

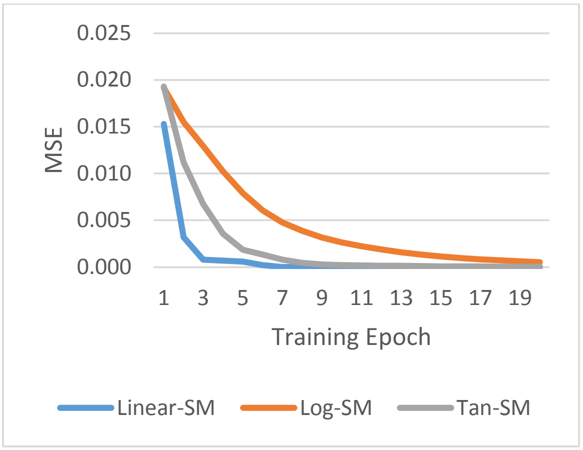


Figure 10. Training Data for SoftMax Output Layer

Hidden Layer Activation Function	Output Layer Activation Function	Accuracy
Linear	Soft Max	92%
Logistic	Soft Max	90%
Hyperbolic Tangent	Soft Max	75%
Linear	Hyperbolic Tangent	4%
Logistic	Hyperbolic Tangent	7%
Hyperbolic Tangent	Hyperbolic Tangent	5%

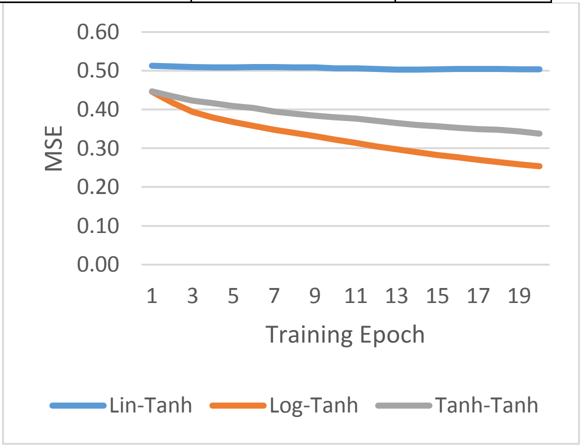


Figure 11. Training Graph for Hyperbolic Tangent Output Layer

The results of the testing of the trained networks are shown in Table 1 below.

Table 1. Testing Results of Layer Types

3.2 Discussion

The best results by far occurred when the SoftMax layer type was used for the output layer. When output layer was a hyperbolic tangent layer, the accuracy of the network was under 10% regardless of the hidden layer type. At first this might seem like a result of lack of training, however it is clear by the training data that the training would have converged at a high error value regardless. When the SoftMax layer type was use the linear hidden layer type had the best results, with the logistic hidden layer type coming in a close second. The hyperbolic tangent performed poorly here as well.

Although one study found a hyperbolic tangent-hyperbolic tangent hidden and output layer type combination to be the most accurate[4], that was likely because it was applied to a regression situation. The cause for the poor performance of the networks with hyperbolic tangent activation functions and the high performance of those with linear layers can be seen explained by looking at the curves developed for the activation functions in Figure n. The input data is extremely varied, but highly specific to each class because each character is highly unique. As such the values being passed to the activation function are likely very varied as well. Because the hyperbolic tangent function approaches its asymptotes of -1 and 1 very quickly, when it is run on highly varied input data multiple very close or approximately equal to those asymptotes will occur. This means when the network is trying to classify the data it sees multiple max arguments and choses the wrong class. The linear function however produces highly varied outputs for highly varied inputs because it has no asymptotes. The logistic function does not approach its asymptotes very quickly so it also gives varied output values. As such it is much more clear which value is the correct classification when a hyperbolic tangent function is not present. In the case of regression, data is being analyzed at a continuous level so it is acceptable to have multiple outputs at the same activation level. For example, if the network is predicting the temperature based on previous temperatures it would be reasonable to assume that it would predict multiple possible temperature values.

In this classification case, the linear hidden layer type combined with the SoftMax output layer type yielded the best results. The training curve converged to the lowest value and it yielded the best accuracy when tested.

4: Conclusions

4.1 Implementation of the Results

As stated previously, the optimal layer structure appeared to be a linear layer for the hidden layer and a softmax layer for the output layer. The network for identifying the characters on the wall was then retrained for the finalized red network, and a new blue network was trained using the blue wall data alongside the finalized red network with the same layer settings. The code for training both networks is included in Appendix A-5. The training curves for the two final networks are shown below in Figure 12.

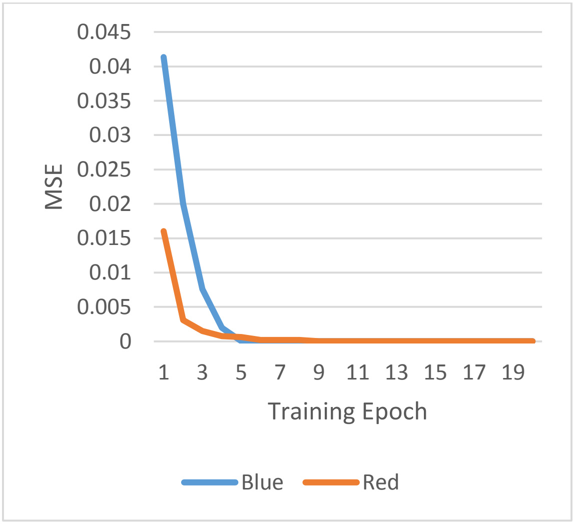
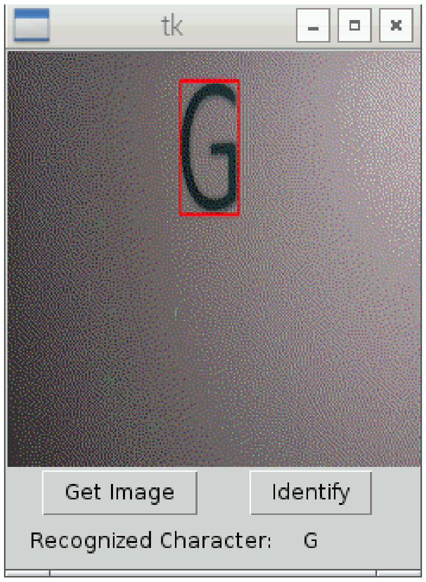


Figure 12. Final Training Curves

The networks were then retested using the test datasets. Once again, he code for testing the datasets is included in Appendix A-4. This time the red network once again yielded an accuracy of 92% and the blue network came in at an equally impressive 90% accuracy.

In order to ensure that the performance of the final networks could be easily demonstrated, a simple GUI was made that would display the isolated character seen by the Pi Camera which would then be identified. The code for this GUI is included in Appendix A-6. As seen in the code, code only identifies the characters on the red walls, however it could be easily modified to address the blue walls or a different custom dataset by changing the loaded network. The GUI is shown accurately identifying a letter “G” in Figure 13.



Recognized Character: G

Figure 13. OCR Application

4.2 Other Applications

Due to the nature of neural networks, the customizability of Pybrain, and the simplicity of the GUIs developed, the codes provided in Appendix A could easily be used for other applications. By simply changing the image processing techniques used, the program could be changed to identify shapes or edges. This could be useful in planet exploration because features such as mountains or plateaus could be identified as a rover explores a planet.

Artificial neural networks are also used can be used for facial recognition, which many Raspberry Pi users utilize.

There are also uses beyond just classification. As an example of a regression application, temperature data can be used to predict upcoming temperatures using neural networks. One study even used neural networks to perform land surveying via satellite images[6].

4.3 Conclusions

The task of developing an OCR tool on a Raspberry Pi to identify a specific set of characters posted on a maze wall was completed with 92% and 90% accuracy for each of the provided character sets. In the competition rules defined by the 2015 IEEE Region 5 Robotics Competition, this would have been sufficient to provide a team with the tools necessary to complete the optional character recognition task.

In addition, the tools and codes developed for the task were highly adaptable so their use could be easily expanded. All of the codes in question have been provided in Appendix A.

5: References

[1] Bradski, G. "The OpenCV Library." *Dr. Dobb's Journal of Software Tools* (2000).

[2] Dasgupta, Bhaskar and Georg Schnitger. "The Power of Approximating: a Comparison of Activation Functions." *Advances in Neural Information Processing Systems 5* (1993):615-622. Morgan Kaufmann. Web. 12 Apr.2015.

[3] David Kriesel. *A Brief Introduction to Neural Networks*. 2007. Web. 11 Apr. 2015. Available at http://www.drkiesel.com

[4] Karlık, B., & Olgaç, A. V. "Performance analysis of various activation functions in generalized MLP architectures of neural networks." *International Journal of Artificial Intelligence and Expert Systems 1(4)* (2011): 111–122.

[5] Leverington, David. "A Basic Introduction to Feedforward Backpropagation Neural Networks." *Neural Network Basics.* Texas Tech University - Department of Geosciences, 2009. Web. 10 Apr. 2015.

[6] Özkan, Coşkun, and Filiz Sunar Erbek. "The Comparison of Activation Functions for Multispectral Landsat TM Image Classification." *Photogrammetric Engineering & Remote Sensing 69.11* (2003): 1225-234. Web. 12 Apr. 2015.

[7] Python Software Foundation. *Python Language Reference, version 2.7*. Available at http://www.python.org

[8] Schaul, Tom et al. "Pybrain." *Journal of Machine Learning Research 11* (2010): 743-746.

[9] Zhang, Q. J., and K. C. Gupta. "Chapter 3: Neural Network Structures." *Neural Networks for RF and Microwave Design*. Boston: Artech House, 2000. 61-102. Web. 11 Apr. 2015.

Appendix A – Python Codes

See Python documentation[7], Open-CV documentation[1], and Pybrain documentation[8] for code reference.

Appendix A-1: Code for Image Processing

```
import numpy as np
import cv2
import os, sys
import picamera
import time
import Image
from array import *


#Sets max area to zero to ensure it is calculated appropriately later
contMaxArea=0

#Sets calibrated values for ensuring minimum h and w are appropriate
Wcal = 100
Hcal = 100

#Sets up pi camera and captures image
camera = picamera.PiCamera()

#Gets and saves new camera capture
if os.path.exists('/home/pi/Python_scripts/capture.jpg'):
   os.remove('/home/pi/Python_scripts/capture.jpg')
camera.capture('capture.jpg')

#Converts im to cv2 image
im = cv2.imread('/home/pi/Python_scripts/capture.jpg')

#Crops cv2 image (can change if necessary)
im = im[184:584,212:812]

#Converts image to grayscale and displays it until key press
gray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)
cv2.imshow('norm',gray)
cv2.waitKey(0)

#Blurs the image to remove noise and shows the result untile key press
#May need to adjust size (25,25) depending on noise size and desired removal
#Values must be positive and odd
blur=cv2.GaussianBlur(gray,(25,25),0)
```

```
cv2.imshow('blur',blur)
cv2.waitKey(0)

#Applies adaptive threshold to convert to B&W and displays image until keypress
#May need to adjust size by adjusting value that is currently 31
#VAlues must be positive and odd
adthreshMean=cv2.adaptiveThreshold(blur,255,cv2.ADAPTIVE_THRESH_MEAN_C,1,31,2)
cv2.imshow('adthreshMean',adthreshMean)
cv2.waitKey(0)

#Sets up kernal for dilating (change size by changing (3,3) to positive odd values)
kernel = cv2.getStructuringElement(cv2.MORPH_CROSS,(3,3))

#Dilates the image to remove holes and displays image until keypress
dilated = cv2.dilate(adthreshMean,kernel,iterations = 3)
cv2.imshow('dilated',dilated)
cv2.waitKey(0)

#Finds all contours in dilated, B&W image to determine character location
contours,hierarchy = cv2.findContours(dilated,cv2.RETR_EXTERNAL,cv2.CHAIN_APPROX_NONE)


#For each contour item
for contour in contours:

  #[x,y,w,h] values determined by bounding rect around contour
  [x,y,w,h]=cv2.boundingRect(contour)

  #If size is to small it can be ignored
  if h<40 or w<20:
    continue

  #Determine if contour has max size because max contour size will be character
  if (w*h)>contMaxArea:
      xmax=x
      ymax=y
      wmax=w
      hmax=h


#Prints the heigth and width of max contour area to LX terminal for calibration
print wmax
print hmax

#Sets bounding region to max width and height to ensure character is seen appropriately
if wmax<Wcal:
  wmax = wmax+(Wcal-wmax)
if hmax<Hcal:
    hmax = hmax+(Hcal-hmax)


#Determines ROI based on max width and height
roi = dilated[ymax:ymax+hmax,xmax:xmax+wmax]

#Draws ROI on image and saves image with bounding box to character.jpg
cv2.rectangle(im,(xmax,ymax),(xmax+wmax,ymax+hmax),(0,0,255),3)
cv2.imwrite('character.jpg',im)

#Setus up image for displaying in a GUI
bigImage=Image.open('character.jpg')
gifimage=bigImage.resize((240,240),Image.ANTIALIAS)
gifimage.save('character.gif')

#Resizes image to 10x10 square and set all values >1 to 1 so array is binary
roismall = cv2.resize(roi, (10,10))
roismall[roismall>1]=1

#Prints 10x10 matrix of 1's and 0's to LX Terminal for verification
print(roismall)

#Reshapes 10x10 to 1x100 so it can be used in neural network
charAsArray = roismall.reshape((1,100))
```

Appendix A-2: Code for Dataset Building GUI

```
#!/usr/bin/python

import os,sys
import Tkinter
from Tkinter import *
import picamera
import time
import tkMessageBox
import numpy as np
import cv2
import Image
from array import *


#Sets up empty array for storing input arrays
samples = np.empty((0,100))

#Sets up Pi Camera and captures and displays initial image
camera=picamera.PiCamera()
```

```
camera.capture('capture.jpg')
bigIm = Image.open('capture.jpg')
smallIm = bigIm.resize((240,240),Image.ANTIALIAS)
smallIm.save('capture.gif')

#Gets list of existing datasets
existingDatasets = os.listdir('/home/pi/Datasets/')

#Sets initial states for interface
go=False
waitingForChar=False

#Sets up arrays and variables for loading/storing data
inArrays = np.empty((1,100))
outChars = np.empty((1,1),dtype=str)
filepath = ''

#Calibrated values for standard width and height for characters
#Used to ensure short characters are used to their relative size
#and split characters are captured in their entirety
calW = 100
calH = 100


#Code for building data that loops throughout
#Looping ensures image and status are up to date
def dataBuilding():

   #Retrieves global states
   global go
   global waitingForChar

   #If state is go and ready to collect new data capture a camera
   #image, convert it to an array, wait for a character input, and
   #then store both the array as the input and character as output
   if go and waitingForChar==False:
      camera.capture('capture.jpg')
      imtoArray()
      camFile = PhotoImage(file = 'character.gif')
      charImage.config(image=camFile)
      charImage.image = camFile
      waitingForChar=True

   #Code loops after half a second (500 ms)
   top.after(500, dataBuilding)

#Code to convert image to array
#Commented out in imtoArray.py file
```

```
def imtoArray():
    global charAsArray
    contMaxArea=0
    im = cv2.imread('/home/pi/Python_scripts/capture.jpg')
    im = im[0:584,0:1024]
    gray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)
    blur=cv2.GaussianBlur(gray,(25,25),0)
    thresh=cv2.adaptiveThreshold(blur,255,cv2.ADAPTIVE_THRESH_MEAN_C,1,31,2)
    kernel = cv2.getStructuringElement(cv2.MORPH_CROSS,(3,3))
    dilated = cv2.dilate(thresh,kernel,iterations = 3)
    contours,hierarchy = cv2.findContours(dilated,cv2.RETR_EXTERNAL,cv2.CHAIN_APPROX_NONE)

    for contour in contours:
        [x,y,w,h]=cv2.boundingRect(contour)

        if h<40 or w<20:
            continue

        if (w*h)>contMaxArea:
            xmax=x
            ymax=y
            wmax=w
            hmax=h

    if wmax<calW:
        wmax = wmax+(calW-wmax)
    if hmax<calH:
        hmax = hmax+(calH-hmax)

    roi = dilated[ymax:ymax+hmax,xmax:xmax+wmax]
    cv2.rectangle(im,(xmax,ymax),(xmax+wmax,ymax+hmax),(0,0,255),3)
    cv2.imwrite('character.jpg',im)
    bigImage=Image.open('character.jpg')
    gifimage=bigImage.resize((240,240),Image.ANTIALIAS)
    gifimage.save('character.gif')
    roismall = cv2.resize(roi, (10,10))
    roismall[roismall>1]=1
    print(roismall)
    charAsArray = roismall.reshape((1,100))


#Code that runs when Start button is clicked
def OK():

    #Loads global states, arrays, and filepath
    global go
    global inArrays
    global outChars
    global filepath

    #If already running return error message
    if go==True:
        tkMessageBox.showinfo("Warning","Already running. Hit stop to select new dataset.")

    #Otherwise determines if new data needs to be created or old needs loaded
    else:
        go=True
        datasetname = fnameEntry.get()
        filepath = '/home/pi/Datasets/' + datasetname
        if not os.path.exists(filepath):
            os.makedirs(filepath)
            existingDatasets = os.listdir('/home/pi/Datasets/')
            fexistLB.insert(len(existingDatasets), datasetname)
        else:
            inArrays = np.loadtxt(filepath+'/inputArrays.data')
            outChars = np.genfromtxt(filepath+'/outputChars.data', comments = '+',dtype=str).reshape(-1,1)


#Code that runs when stop button clicked
def Stop():
    #Loads global states, arrays, and filepath
    global go
    global waitingForChar
    global inArrays
    global outChars
    global filepath

    #Resets states so code for building datasets stops running
    go=False
    waitingForChar=False

    #Clears the entry box
    charEntry.delete(0,END)

    #Saves current arrays to their appropriate input and output files
    np.savetxt(filepath+'/inputArrays.data',inArrays)
    np.savetxt(filepath+'/outputChars.data',outChars, fmt='%s')

    #Resets arrays
    inArrays = np.empty((1,100))
    outChars = np.empty((1,1),dtype=str)

    #Prints status to LX terminal
    print('Stopped')
```

```
#Code that runs when Add button clicked
def Add():

   #Loads global states and arrays
   global go
   global waitingForChar
   global charAsArray
   global inArrays
   global outChars

   #If ready for character entry
   if waitingForChar==True:

      #Gets current character entry
      enteredCharacter = charEntry.get()

      #Ensures character is valid and gives error if it is not
      #Can change to allow full phrases if necessary
      if enteredCharacter=='' or enteredCharacter==' ':
         tkMessageBox.showinfo("Warning","Please enter a valid character.")
      elif len(enteredCharacter)>1:
         tkMessageBox.showinfo("Warning","Please enter a single character.")

      #If character is valid:
      else:
         #Add charAsArray and entered char to datasets
         inArrays = np.append(inArrays,charAsArray,0)
         outChars = np.append(outChars,np.array([[enteredCharacter]]),0)

         #Sets states to appropriate levels
         waitingForChar=False
         go=True

         #Resets entry box
         charEntry.delete(0,END)

   #Error message if not ready for character
   else:
      tkMessageBox.showinfo("Warning","Not ready for character.")


#Code that runs when refresh button clicked
def Refresh():

   #Gets global states
   global go
   global waitingForChar
```

```
    #If building datasets is running
    if go==True:

        #Sets waiting for char to false so new image is captured
        waitingForChar=False

    #If not running give error message
    else:
        tkMessageBox.showinfo("Warning","Must be running to refresh. Please hit Start button.")


#Code that runs when Close button clicked
def closeProgram():
    #Load global arrays and filepath
    global inArrays
    global outChars
    global filepath

    #Saves current arrays to appropriate input and output files
    np.savetxt(filepath+'/inputArrays.data',inArrays)
    np.savetxt(filepath+'/outputChars.data',outChars, fmt='%s')

    #Closes the window
    top.destroy()



#Code that runs when ^ button next to dataset list is clicked
def selectListItem():

    #Sets the current dataset to be selected item
    curIndex = fexistLB.curselection()
    listItem = fexistLB.get(curIndex)
    fnameEntry.delete(0,END)
    fnameEntry.insert(0,listItem)



#Code for setting up GUI
#See tkinter documentation for details
top = Tk()

top.geometry('550x300')

FnameLabel = Label(top, text = "Dataset name:")
FnameLabel.place(x=20,y=20)
```

```
descLabel = Label(top, text = "Enter a new file name to create a dataset or select an existing dataset to
append to.")
descLabel.place(x=0, y=0)

existingLabel = Label(top, text = "Existing Datasets:")
existingLabel.place(x=0, y=40)

Refbutton = Tkinter.Button(top, text="Refresh Image", command = Refresh)
Refbutton.place(x=188,y=234)

doneButton = Tkinter.Button(top, text="Close", command = closeProgram)
doneButton.place(x=478,y=261)

Stopbutton = Tkinter.Button(top, text="Stop", command = Stop)
Stopbutton.place(x=420,y=261)

OKbutton = Tkinter.Button(top, text="Start", command = OK)
OKbutton.place(x=360,y=261)

Addbutton = Tkinter.Button(top, text="Add", command = Add)
Addbutton.place(x=307,y=261)

LBbutton = Tkinter.Button(top, text="^", command = selectListItem)
LBbutton.place(x=282, y=40, width=20, height=20)

fnameEntry = Entry(top)
fnameEntry.place(x=120, y=20, width=183)

charEntry = Entry(top, width=5)
charEntry.place(x=260,y=265)

charLabel = Label(top, text = "Enter character shown in image above:")
charLabel.place(x=12,y=265)

fexistLB = Listbox(top, selectmode=SINGLE)
for fileIndex in range(len(existingDatasets)):
  fexistLB.insert(fileIndex, existingDatasets[fileIndex])
fexistLB.place(x=120, y=40)

camFile = PhotoImage(file = 'capture.gif')
charImage = Label(top, image=camFile)
charImage.place(x=302, y=20)

top.after(1000, dataBuilding)
top.mainloop()
```

Appendix A-3: Code for Training Red Network Only

```
from pybrain.datasets import ClassificationDataSet
from pybrain.tools.shortcuts import buildNetwork
from pybrain.supervised.trainers import BackpropTrainer
from pybrain.structure.modules import SoftmaxLayer
from pybrain.structure.modules import TanhLayer
from pybrain.structure.modules import SigmoidLayer
from pybrain.structure.modules import LinearLayer
from pybrain.tools.customxml.networkwriter import NetworkWriter
from pybrain.tools.customxml.networkreader import NetworkReader
import numpy as np
import os,sys
import time


#Set class labels and get input arrays and output characters
redClassLabels =
['A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z','!','@','#','$','%','^','&','*
','(',')']
redInputs = np.loadtxt('/home/pi/Datasets/redData/inputArraysBinary.data', dtype = int, skiprows = 1)
redChars = np.genfromtxt('/home/pi/Datasets/redData/outputChars.data',dtype=str, comments = '+',
skiprows=1)



#Convert red output characters to integers based on index in redClassLabels
redOutputs = np.empty((1,1), dtype=int)
for r in np.nditer(redChars):
   redCharInd =  redClassLabels.index(r)
   redOutputs = np.append(redOutputs,np.array([[redCharInd]]),0)
redOutputs = np.delete(redOutputs,0,0)



#Build red datasets
redDS = ClassificationDataSet(100,1,nb_classes=len(redClassLabels))
assert(redInputs.shape[0] == redOutputs.shape[0])
redDS.setField('input',redInputs)
redDS.setField('target',redOutputs)
redDS._convertToOneOfMany()



#Build network and setup trainer
redNet = buildNetwork(100,68,len(redClassLabels),bias=True,hiddenclass=SigmoidLayer,
outclass=TanhLayer)
```

```
redTrainer = BackpropTrainer(redNet,redDS)



#train for 20 epochs and store error each time
errVals = np.empty((1,1))
for x in range(20):
    errR = redTrainer.train()
    #errB = blueTrainer.trainEpochs(1)
    print errR
    errVals = np.append(errVals,np.array([[errR]]),0)


#Save error arrays and network
np.savetxt('/home/pi/NetworkTrainErrVals/sigtanh.data',errVals)
NetworkWriter.writeToFile(redNet,'/home/pi/ANNs/sigtanh.xml')
```

Appendix A-4: Code for Testing Red and Blue Networks

```
from pybrain.tools.customxml.networkreader import NetworkReader
import numpy as np



#Load networks
redNet = NetworkReader.readFrom('/home/pi/ANNs/redNet.xml')
blueNet = NetworkReader.readFrom('/home/pi/ANNs/blueNet.xml')

#Setup class labels for test sets
redClassLabels =
['A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z','!','@','#','$','%','^','&','*
','(',')']
blueClassLabels = ['0','1','2','3','4','5','6','7','8','9','!','@','#','$','%','^','&','*','(',')']

#Load testing data and convert output chars to values using class labels
redTestInputs = np.loadtxt('/home/pi/Datasets/redTestData/inputArrays.data', dtype = int, skiprows = 1)
redTestChars = np.genfromtxt('/home/pi/Datasets/redTestData/outputChars.data',dtype=str, comments
= '+', skiprows=1)

redOutputs = np.empty((1,1), dtype=int)

for r in np.nditer(redTestChars):
    redCharInd =  redClassLabels.index(r)
    redOutputs = np.append(redOutputs,np.array([[redCharInd]]),0)
redOutputs = np.delete(redOutputs,0,0)
```

```
blueTestInputs = np.loadtxt('/home/pi/Datasets/blueTestData/inputArrays.data', dtype = int, skiprows =
1)
blueTestChars = np.genfromtxt('/home/pi/Datasets/blueTestData/outputChars.data',dtype=str,
comments = '+', skiprows=1)

blueOutputs = np.empty((1,1), dtype=int)

for r in np.nditer(blueTestChars):
    blueCharInd =  blueClassLabels.index(r)
    blueOutputs = np.append(blueOutputs,np.array([[blueCharInd]]),0)
blueOutputs = np.delete(blueOutputs,0,0)


#Sets initial correct to 0 and tests networks on test data
Redcorrect = 0
Bluecorrect = 0

for rInd in range(len(redTestInputs)):
    outputArray = redNet.activate(redTestInputs[rInd])
    indMax = outputArray.argmax()
    if indMax == redOutputs[rInd]:
        Redcorrect = Redcorrect+1

for bInd in range(len(blueTestInputs)):
    outputArray = blueNet.activate(blueTestInputs[bInd])
    indMax = outputArray.argmax()
    if indMax == blueOutputs[bInd]:
        Bluecorrect = Bluecorrect+1

#Returns number of correct ID's
print Redcorrect, Bluecorrect
```

Appendix A-5: Code for Training Red and Blue Networks

```
from pybrain.datasets import ClassificationDataSet
from pybrain.tools.shortcuts import buildNetwork
from pybrain.supervised.trainers import BackpropTrainer
from pybrain.structure.modules import SoftmaxLayer
from pybrain.structure.modules import LinearLayer
from pybrain.tools.customxml.networkwriter import NetworkWriter
from pybrain.tools.customxml.networkreader import NetworkReader
import numpy as np
import os,sys
import time
```

```
#Sets up class labels
redClassLabels =
['A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z','!','@','#','$','%','^','&','*
','(',')']
blueClassLabels = ['0','1','2','3','4','5','6','7','8','9','!','@','#','$','%','^','&','*','(',')']

#Loads input and output data for red and blue datasets
#Converts output chars to numberic values using corresponding index in class labels
redInputs = np.loadtxt('/home/pi/Datasets/redData/inputArraysBinary.data', dtype = int, skiprows = 1)
redChars = np.genfromtxt('/home/pi/Datasets/redData/outputChars.data',dtype=str, comments = '+',
skiprows=1)
redOutputs = np.empty((1,1), dtype=int)

for r in np.nditer(redChars):
    redCharInd =  redClassLabels.index(r)
    redOutputs = np.append(redOutputs,np.array([[redCharInd]]),0)
redOutputs = np.delete(redOutputs,0,0)


blueInputs = np.loadtxt('/home/pi/Datasets/blueData/inputArraysBinary.data', skiprows = 1)
blueChars = np.genfromtxt('/home/pi/Datasets/blueData/outputChars.data',dtype=str, comments = '+',
skiprows=1)
blueOutputs = np.empty((1,1),dtype=int)

for b in np.nditer(blueChars):
    blueCharInd =  blueClassLabels.index(b)
    blueOutputs = np.append(blueOutputs,np.array([[blueCharInd]]),0)
blueOutputs = np.delete(blueOutputs,0,0)


#Build red and blue classification datasets using input data
#convertToOneOfMany is helpful with classification
#See Pybrain documentation for details
redDS = ClassificationDataSet(100,1,nb_classes=len(redClassLabels))
assert(redInputs.shape[0] == redOutputs.shape[0])
redDS.setField('input',redInputs)
redDS.setField('target',redOutputs)
redDS._convertToOneOfMany()

blueDS = ClassificationDataSet(100,1,nb_classes=len(blueClassLabels))
assert(blueInputs.shape[0] == blueOutputs.shape[0])
blueDS.setField('input',blueInputs)
blueDS.setField('target',blueOutputs)
blueDS._convertToOneOfMany()


#Build Red and Blue networks and set up trainers
```

```
redNet = buildNetwork(100,68,len(redClassLabels),bias=True,hiddenclass=LinearLayer,
outclass=SoftmaxLayer)
redTrainer = BackpropTrainer(redNet,redDS)

blueNet =
buildNetwork(100,55,len(blueClassLabels),bias=True,hiddenclass=LinearLayer,outclass=SoftmaxLayer)
blueTrainer = BackpropTrainer(blueNet,blueDS)

#Sets up arrays for storing training data
errValsR = np.empty((1,1))
errValsB = np.empty((1,1))

#train for 20 epochs and print and store error each time
for x in range(20):
    errR = redTrainer.train()
    errB = blueTrainer.train()
    print errR, errB
    errValsR = np.append(errValsR,np.array([[errR]]),0)
    errValsB = np.append(errValsB,np.array([[errB]]),0)

#Saves training data for each dataset
np.savetxt('/home/pi/NetworkTrainErrVals/redTraining.data',errValsR)
np.savetxt('/home/pi/NetworkTrainErrVals/blueTraining.data',errValsB)

#Saves each network
NetworkWriter.writeToFile(redNet,'/home/pi/ANNs/redNet.xml')
NetworkWriter.writeToFile(blueNet,'/home/pi/ANNs/blueNet.xml')
```

Appendix A-6: GUI for Running OCR in Real Time

```
#!/usr/bin/python

import os,sys
import Tkinter
from Tkinter import *
import picamera
import time
import tkMessageBox
import numpy as np
import cv2
import Image
from array import *
from pybrain.tools.customxml.networkreader import NetworkReader


#Loads items for red network
```

```
redNet = NetworkReader.readFrom('/home/pi/ANNs/lin.xml')
redClassLabels =
['A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z','!','@','#','$','%','^','&','*
','(',')']


#Sets calibrated h and w minimum values
Hcal = 100
Wcal = 100


#sets up camera and gets initial image to display
camera=picamera.PiCamera()
camera.capture('capture.jpg',use_video_port=True)
bigIm = Image.open('capture.jpg')
smallIm = bigIm.resize((240,240),Image.ANTIALIAS)
smallIm.save('capture.gif')


#sets default state
haveIm=False


#imtoArray code
#For comments see imtoArray.py file
def imtoArray():
   global charAsArray
   contMaxArea=0
   im = cv2.imread('/home/pi/Python_scripts/capture.jpg')
   im = im[0:584,0:1024]
   gray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)
   blur=cv2.GaussianBlur(gray,(25,25),0)
   thresh=cv2.adaptiveThreshold(blur,255,cv2.ADAPTIVE_THRESH_MEAN_C,1,31,2)
   kernel = cv2.getStructuringElement(cv2.MORPH_CROSS,(3,3))
   dilated = cv2.dilate(thresh,kernel,iterations = 3)
   contours,hierarchy = cv2.findContours(dilated,cv2.RETR_EXTERNAL,cv2.CHAIN_APPROX_NONE)

   for contour in contours:
      [x,y,w,h]=cv2.boundingRect(contour)

      if h<40 or w<20:
         continue

      if (w*h)>contMaxArea:
         xmax=x
         ymax=y
         wmax=w
         hmax=h
```

```
    if wmax<Wcal:
        wmax = wmax+(Wcal-wmax)
    if hmax<Hcal:
        hmax = hmax+(Hcal-hmax)

    print hmax
    print wmax
    roi = dilated[ymax:ymax+hmax,xmax:xmax+wmax]
    cv2.rectangle(im,(xmax,ymax),(xmax+wmax,ymax+hmax),(0,0,255),3)
    cv2.imwrite('character.jpg',im)
    bigImage=Image.open('character.jpg')
    gifimage=bigImage.resize((240,240),Image.ANTIALIAS)
    gifimage.save('character.gif')
    roismall = cv2.resize(roi, (10,10))
    roismall[roismall>1]=1
    print(roismall)
    charAsArray = roismall.reshape((1,100))
    charAsArray = np.array(charAsArray).squeeze()


#If Identify button clicked
def OK():

    #Loads global state and char array
    global haveIm
    global charAsArray

    #If state indicates image is ready
    if haveIm==True:
        #Run the charAsArray through net and identify char
        charOutArray = redNet.activate(charAsArray)
        maxCharProb = max(charOutArray)
        maxCharInd = charOutArray.argmax()
        char = redClassLabels[maxCharInd]
        print char
        recChar.config(text=char)
        recChar.text=char

    #Otherwise give error
    else:
        tkMessageBox.showinfo("No image for OCR. Please click Get Image")


#Code that runs when Get Image is clicked
def GetIm():
    #Uses global state
    global haveIm
    #Gets image and updates display
    camera.capture('capture.jpg',use_video_port=True)
    imtoArray()
    camFile = PhotoImage(file = 'character.gif')
    charImage.config(image=camFile)
    charImage.image = camFile

    #Updates state
    haveIm=True


#Code for GUI
#See tkinter documentation for details

top = Tk()

top.geometry('240x300')

Refbutton = Tkinter.Button(top, text="Get Image", command = GetIm)
Refbutton.place(x=20,y=242)

OKbutton = Tkinter.Button(top, text="Identify", command = OK)
OKbutton.place(x=140,y=242)

charLabel = Label(top, text = "Recognized Character: ")
charLabel.place(x=12,y=275)

recChar = Label(top, text = "N/A")
recChar.place(x=170, y=275)

camFile = PhotoImage(file = 'capture.gif')
charImage = Label(top, image=camFile)
charImage.place(x=0, y=0)

top.mainloop()
```