Project Report

Bengali digits recognition from an image using image preprocessing

(EE 769)

By

Sudip Walter Thomas(193100063)

Prajual Pillai(193100069)

Saurabh Mandoakar(193100081)

M.Tech (Manufacturing)

Department of Mechanical Engineering

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY

May 2020

Background and inspiration:

About Research paper : According to A. Choudhury and J. Mukherjee [1] an approach is discussed towards extraction of Bengali digits from an image. The detailed process of digit extraction is explained through graphs as shown below. We followed the paper's approach till image binarisation.

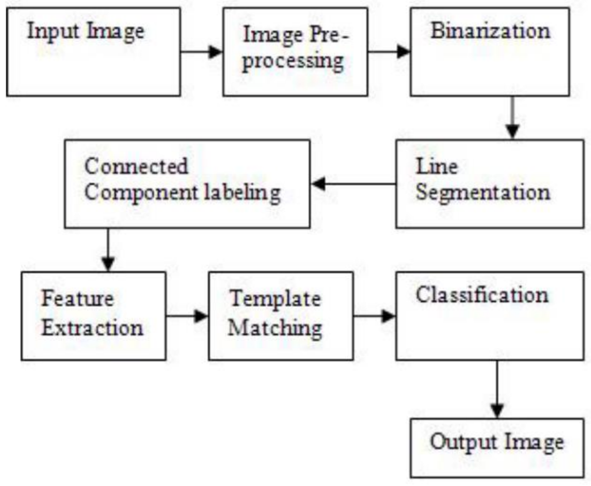


Fig: Approach in research paper[1]

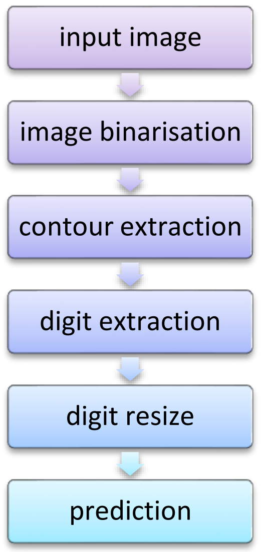


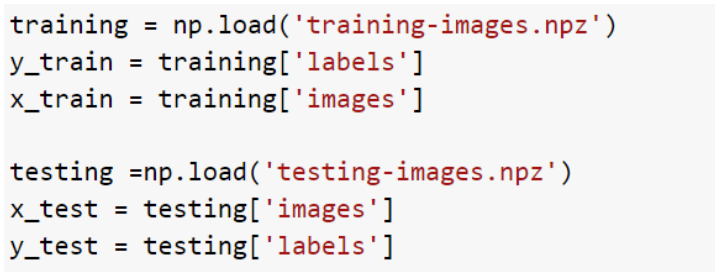
Fig: Approach in code

Training Data:

For Bengali dataset we have numpy's compressed array (**.NPZ)** file formats for training and testing images data. A .npz file is a compressed version of .npy file formats. The dataset consists of image data in form of arrays and the corresponding image labels.

Explanation of code: Step 1:

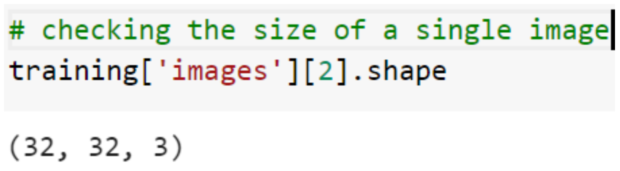
Defining training and testing datasets



```
training = np.load('training-images.npz')
y_train = training['labels']
x_train = training['images']

testing =np.load('testing-images.npz')
x_test = testing['images']
y_test = testing['labels']
```

Visualization

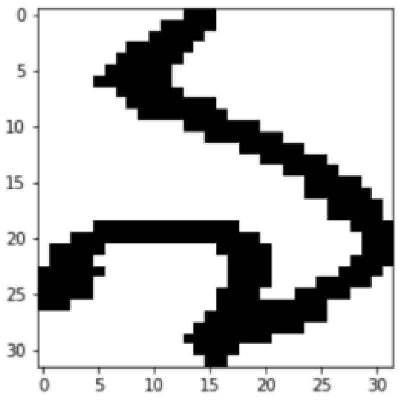
The adjoining figure shows that each image has size (32 x 32 x 3), which means height = 32 pixels, width = 32 pixels and 3 color channels.



```
# checking the size of a single image
training['images'][2].shape
```

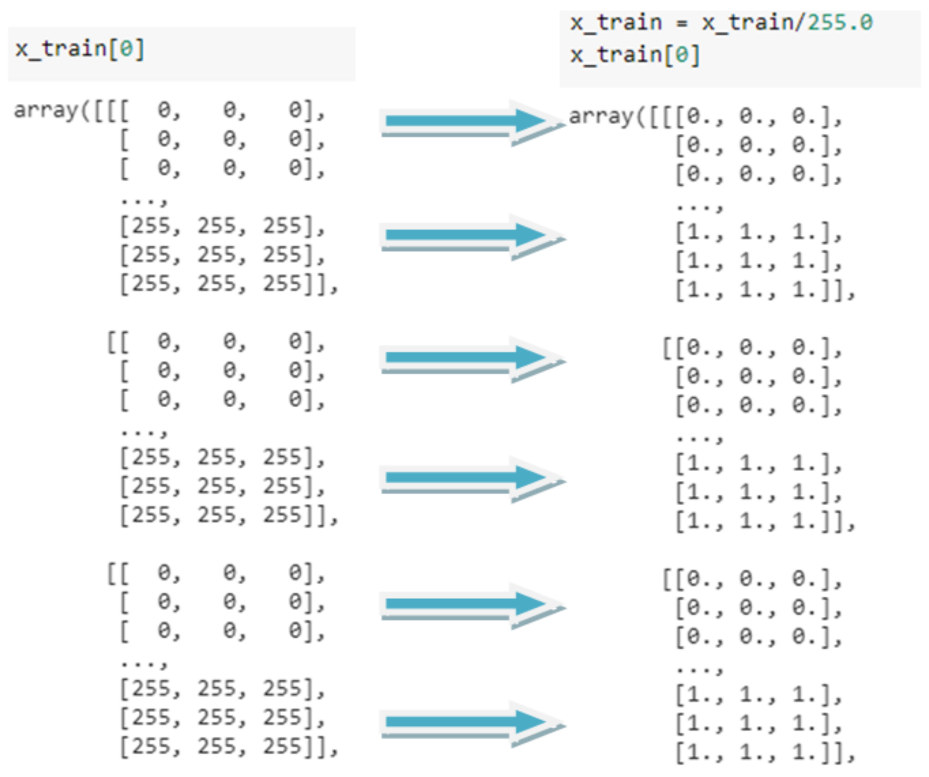
```
(32, 32, 3)
```

Sample image represented using matplotlib:



Images Normalization:

Normally an image is defined by 256 pixels numbered from 0 to 255. While this is okay for day to day use in our case in order to reduce the computation of the CNN model that we have normalized the image by dividing all the pixels by the maximum size which is 255. This doesn't affect the actual image as can be shown by plotting the image before and after normalization.



```
x_train[0]

array([[[  0,   0,   0],
        [  0,   0,   0],
        [  0,   0,   0],
        ...,
        [255, 255, 255],
        [255, 255, 255],
        [255, 255, 255]],

       [[  0,   0,   0],
        [  0,   0,   0],
        [  0,   0,   0],
        ...,
        [255, 255, 255],
        [255, 255, 255],
        [255, 255, 255]],

       [[  0,   0,   0],
        [  0,   0,   0],
        [  0,   0,   0],
        ...,
        [255, 255, 255],
        [255, 255, 255],
        [255, 255, 255]],
```

```
x_train = x_train/255.0
x_train[0]

array([[[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        ...,
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]],

       [[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        ...,
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]],

       [[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        ...,
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]],
```

So after normalization we have a 32x32 image with 3 channels and each pixel intensity value lies between 0 and 1.

<u>Model building</u>

Choice of KERAS sequential API

We used KERAS sequential model API to build our model. The sequential model was chosen instead of functional because we are dealing with just one form of data, and intend the layers to be sequentially arranged so that every next layer has inputs from just the previous layer.

The Dropout layer randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting

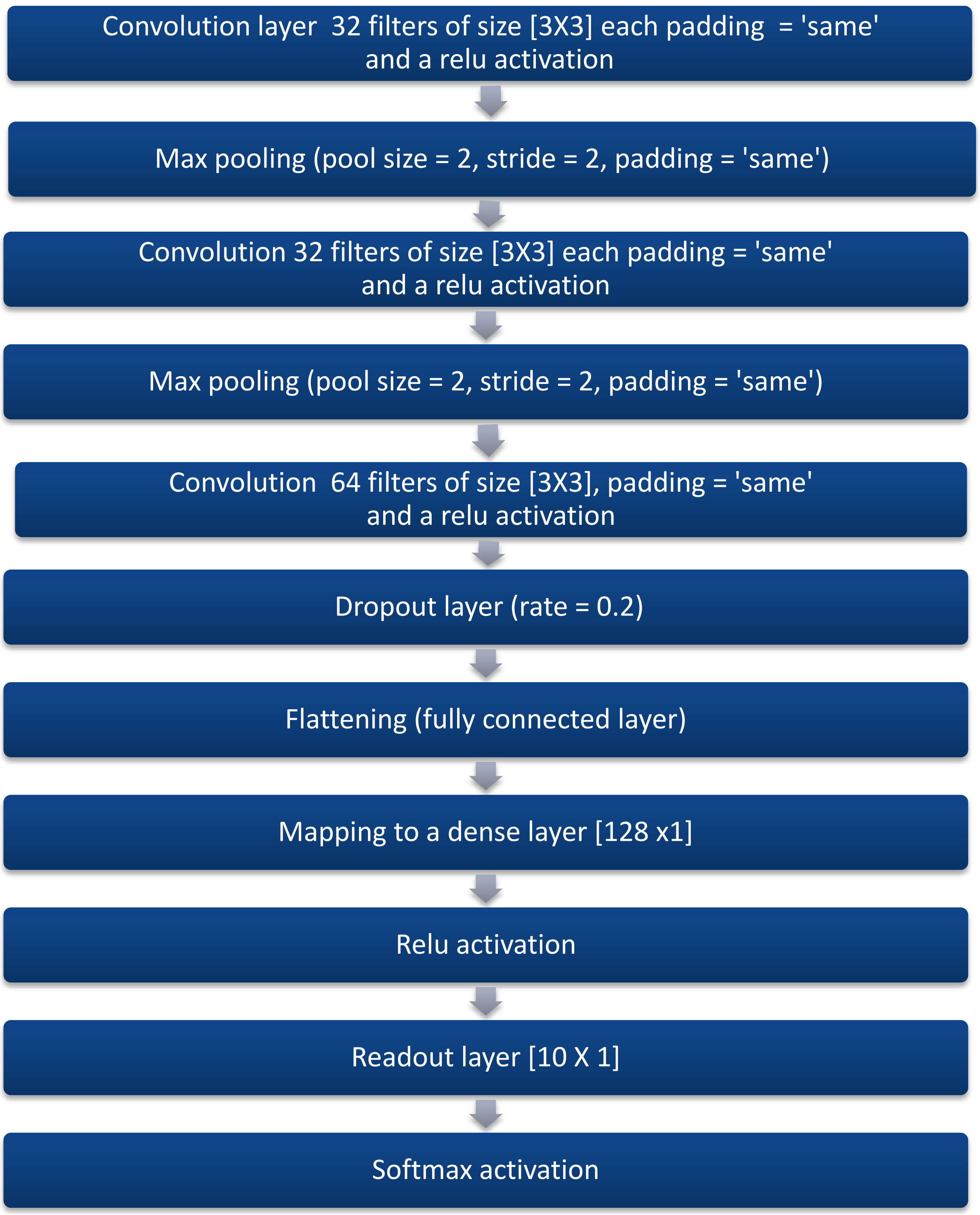
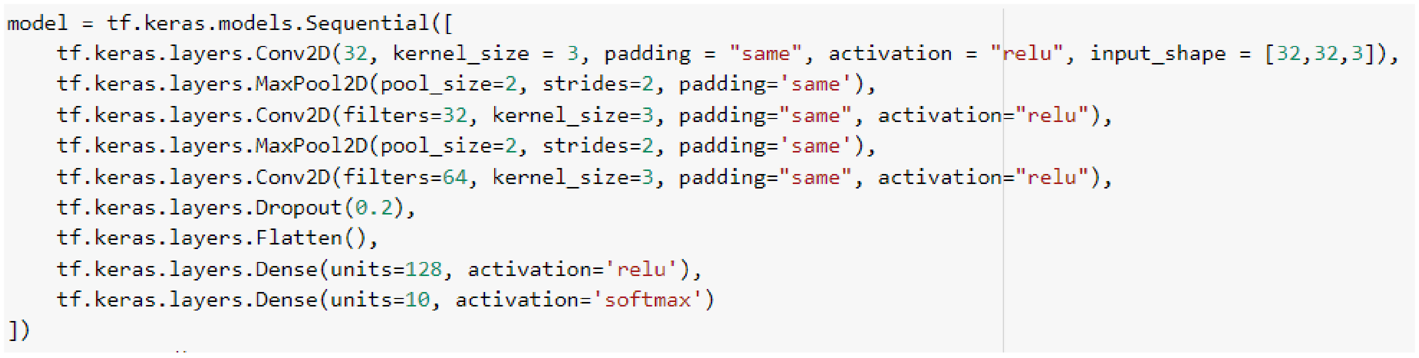

Convolution layer 32 filters of size [3X3] each padding = 'same' and a relu activation
Max pooling (pool size = 2, stride = 2, padding = 'same')
Convolution 32 filters of size [3X3] each padding = 'same' and a relu activation
Max pooling (pool size = 2, stride = 2, padding = 'same')
Convolution 64 filters of size [3X3], padding = 'same' and a relu activation
Dropout layer (rate = 0.2)
Flattening (fully connected layer)
Mapping to a dense layer [128 x1]
Relu activation
Readout layer [10 X 1]
Softmax activation

Fig: Layers of sequential model used

Model building code:



```
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(32, kernel_size = 3, padding = "same", activation = "relu", input_shape = [32,32,3]),
    tf.keras.layers.MaxPool2D(pool_size=2, strides=2, padding='same'),
    tf.keras.layers.Conv2D(filters=32, kernel_size=3, padding="same", activation="relu"),
    tf.keras.layers.MaxPool2D(pool_size=2, strides=2, padding='same'),
    tf.keras.layers.Conv2D(filters=64, kernel_size=3, padding="same", activation="relu"),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(units=128, activation='relu'),
    tf.keras.layers.Dense(units=10, activation='softmax')
])
```

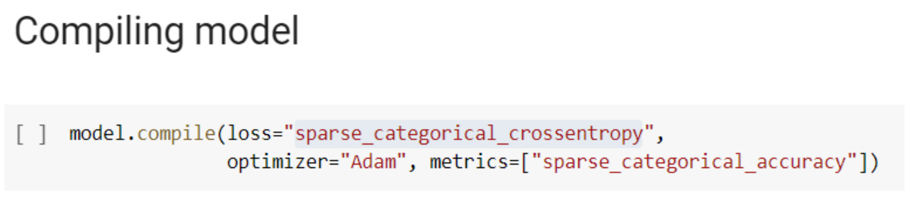
Model compilation:

Choice of Optimizer: For the best optimizer to be used we referred to the following.

Reference: https://medium.com/@onlytojay/mnist-cnn-optimizer-comparison-with-tensorflow-keras-163735862ecd

As per the above reference a comparison was made between various optimizers for training a CNN model for digit classification of MNIST dataset images. It concluded that Adam, Nadam and RMSProp performed the best on the metric of 'accuracy' on the validation data. We selected the Adam optimizer for our case.

Choice of loss function: The type of problem we are dealing with is a multiclass classification of images dataset, so we choose sparse_categorical_crossentropy. Also the metric 'sparse categorical accuracy' was chosen as it is a commonly used metric for multiclass classification.



Compiling model

```
[ ] model.compile(loss="sparse_categorical_crossentropy",
                  optimizer="Adam", metrics=["sparse_categorical_accuracy"])
```

Loss function: sparse category cross-entropy

Optimizer used: Adam optimizer

Defining a callback: We want our model to stop at a threshold of 0.99 accuracy, therefore we used a callback called myCallback to terminate the training process as soon as we reach accuracy of 99%.

Training the model:

```
model.fit(x_train, y_train, epochs = 10, callbacks = [callbacks])
```

The epochs were set to 10 but as the accuracy threshold was set to 0.99 the model gets trained within 5-6 epochs with the threshold consideration. The training results are as follows:

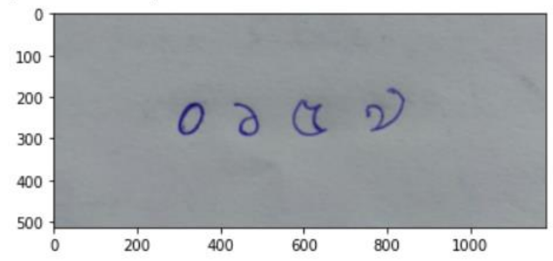
Final loss value: 0.0238 at end of 5th epoch.

Sparse categorical accuracy: 0.9930

The test data:

The basic idea is to consider a series of letters or an image containing multiple letters, so we considered an image for testing as shown in the adjoining figure.

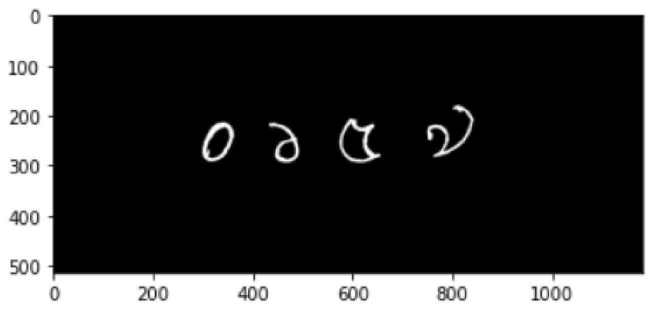
```
(514, 1181, 3)
```



Preprocessing on the test data

Image size is [514 X 1181] with 3 channels (RBG). There are 2 operations to be performed on this image now to extract out the images to be fed to the model.

[1] Image binarisation:



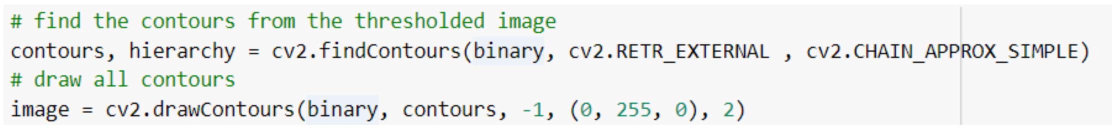
This step basically involves in reducing the depth of the image and converting it to single grayscale image. This image is called as a binary image.

[2] Digit segmentation and resize

The major part of the coding done for image preprocessing was referred from the online openCV Documentations on contour features.

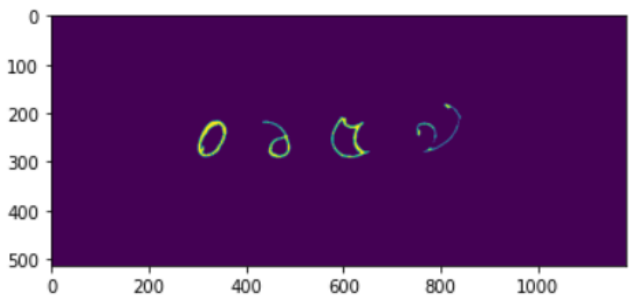
[https://docs.opencv.org/3.4/dd/d49/tutorial_py_contour_features.html**]**

At this point we need to extract the digits from the image. As per the above example, 4 images can be extracted. We approach the problem by first finding out the



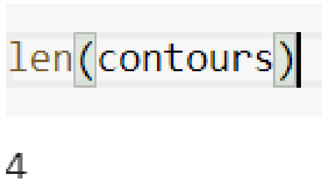
```
# find the contours from the thresholded image
contours, hierarchy = cv2.findContours(binary, cv2.RETR_EXTERNAL , cv2.CHAIN_APPROX_SIMPLE)
# draw all contours
image = cv2.drawContours(binary, contours, -1, (0, 255, 0), 2)
```

contours in the image. We used 'cv2' library to recognize these contours. 'cv2.findcontours' was used to assign contours. As there was inner and outer contours, we choose RETR_EXTERNAL and not RETR_TREE which was giving us 8 contours, as it was also taking the inner parts for the hierarchy.



The cv2.drawcontours can be used to visually represent the contours. We obtain 4 contours. The contours were found out and saved in vectors

'Contours' and 'hierarchy'.



```
len(contours)
4
```

Now each digit is extracted by creating a rectangular box around it by using 'cv2.boundingRect'. Each of these rectangles were segmented out from the picture.

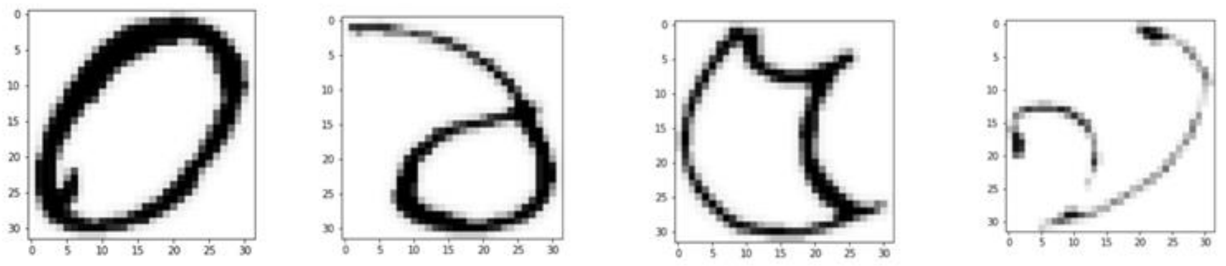
For the bounding rectangle:

```
x,y,w,h = cv2.boundingRect(cnt)
segmented_image=binr[y:y+h,x:x+w]
```

(x,y) → top left corner location
W → width of image
H → height of image
These extracted parameters are used to extract digit boxes from the binarised image.

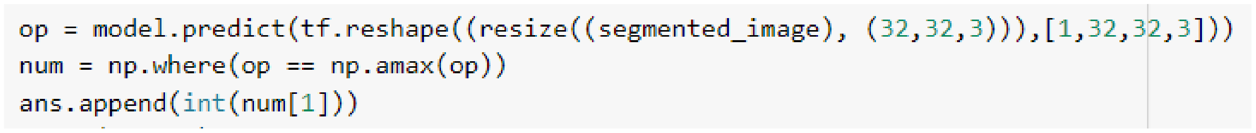
```
if abs(x1 - x) > 0.5 and abs(y-y1) > 0.5:
    if h >= 30 and w >= 30:
        segmented_image = resize(abs(segmented_image), (32,32))
        plt.imshow(segmented_image)
        plt.show()
        cv2.imwrite(str(image_name) + '.jpg',segmented_image)

x1,y1 = x,y
```

The code is formulated in such a way that it will make rectangular boxes around the contours without repeating the same contour. Now we crop out these images and resize them to

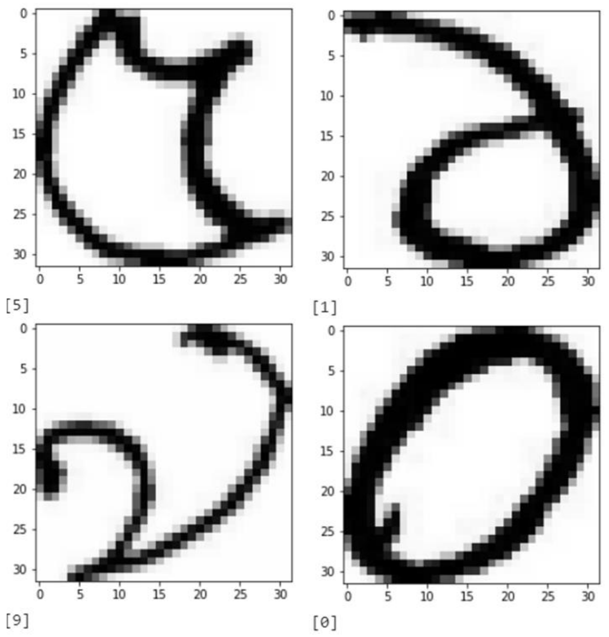


[32 X 32] and save them.

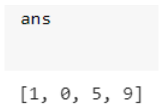
Predictions: for every digit image we get an output from the model as an array of 10 probabilities. The np.amax function was used to find the index of the highest probability and that should be the predicted number.



```
op = model.predict(tf.reshape((resize((segmented_image), (32,32,3))),[1,32,32,3]))
num = np.where(op == np.amax(op))
ans.append(int(num[1]))
```



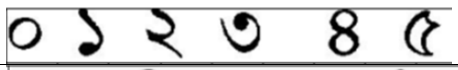
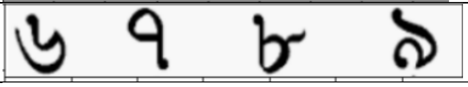
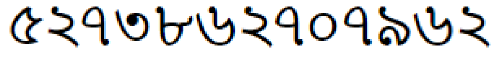
```
[5]  [1]
[9]  [0]
```



```
ans
```

```
[1, 0, 5, 9]
```

Some other predictions:

Image for test	desired	result
	**0,1,2,3,4,5**	[0, 3, 5, 4, 2, 1]
	**6,7,8,9**	[8, 6, 9, 7]
	5,2,7,3, 8,6,2,7,0, 7,9,6,2	[0, 3, 2, 6, 9, 7, 7, 2, 6, 8, 7, 2, 5] **Correctly predicted all 13 digits**

Result: The digits in the image are being recognized by the model very efficiently, but the order in which these digits should be arranged is not proper.

References:

[1] A. Choudhury and J. Mukherjee, “An Approach towards Recognition of Size and Shape Independent Bangla Handwritten Numerals,” *Int. J. Sci. Eng. Res.*, vol. 1, no. 1–3, pp. 223–226, 2013.