Block Diagram:

Input Image (Kannada Word)

Pre-processing:

Extract images of individual characters from the image of the word

CNNs (Trained with dataset)

Get application-defined UID of Kannada character

Output Unicode characters to text file

Dataset: Sample Images of all possible Kannada characters

Lookup Table:

From UID get Unicode encoding of Kannada character

Pre-processing Block Diagram:

Input Image (KannadaWord)

Threshold, Invert and Dilate

Rearrange the labels in proper order

Crop out extra whitespace

Label all the connected components

Get position of base line

Crop out individual characters from the image (Using the labels as reference)

Identify whether the character is a regular character or vattakshara (subscript) character [Using position of base line as reference]

CNNs for normal characters

CNNs for vattakshara characters

CNN block diagram:

Dataset: Sample Images of all characters

Split dataset into 4 size classes based on aspect ratio

Identify to which size class it belongs

Image of character to be identified

Class 0

Class 1

Class 2

Class 3

e.g. Let us say it belongs to class 3

Resize

Resize

Resize

Resize

Resize appropriately

Use to train CNN

Give as input to appropriate CNN

Use to train CNN

Use to train CNN

Use to train CNN

CNN 1

CNN 2

CNN 3

CNN 4

Get UID no. of character

**Neural Networks**

Artificial Neural Networks or ANNs are computing systems modelled after and the biological neural networks in animal brains and designed to mimic them. They consist of an input layer and an output layer with multiple hidden layers in between. Each layer consists of a number of processing units called as neurons.

A neuron, the basic unit of a neural network, takes multiple inputs and returns a single output. In a feedforward neural network, the outputs of the neurons in one layer are fed as inputs to the neurons in the next layer. The input layer is the first layer in this chain which takes its input signals from the user and the output layer is the last layer in the chain which returns its outputs to the user. A neural network may have multiple inputs and multiple outputs.

The attractive feature of these systems is that they do not require task-specific programming and instead they are designed to “learn” to perform the tasks themselves by considering examples. For example, in our application i.e. Optical Character Recognition, if we want to configure an ANN to recognize the character ‘A’ in an image, we just have to train it with a number of images labelled ‘A’ and ‘not A’ and then subsequently if an unknown image is presented the system can successfully recognize it as ‘A’ or ‘not A’. It is just like how we teach children to read the alphabet. In a non-learning system the programmer would have to define the rules to recognize the letter ‘A’ in the program itself, which would take hours of coding. In this way using Neural Networks for our application i.e. OCR would greatly simplify the design of the program.

In most neural networks a neuron is implemented as a unit which takes the weighted sum of its multiple inputs, applies an activation function to it and returns the result as output. Each input to a neuron has its own weight. When we “train” the neural network to “learn” something, what we are actually doing is adjusting these weights so as to get a desired output ‘y’ for a given input ‘x’. Algorithms such as Gradient Descent Algorithm are used for this.

**Neural Networks for Image Classification**

In a fully connected neural network, every neuron takes inputs from each and every neuron in the previous layer and feeds its output to each and every neuron in the next layer. In other words, every possible connection between two neurons in two different layers is made.

This architecture of neural network does not scale well for images. Firstly, the images would require a lot of manually engineered pre-processing if we want to obtain an accurate result. Secondly, it is because the size of the input is very large. Let us say we are giving a 200x200x3 RGB image as input to our network, by making each color value of each pixel as a separate input. Therefore we have 200x200x3 = 120,000 inputs. This means every neuron in the first layer takes 120,000 inputs and correspondingly has 120,000 weights, one for each input. For one neuron alone we need to store and process 120,000 variables. And one neuron is barely enough for a decent neural network. The complexity of the system increases greatly when we add more neurons and layers to the network.

The solution to this problem can be found by studying the visual cortex of a biological eye. In the visual cortex individual neurons respond to visual stimuli only in a restricted region of the visual field and not the entire visual field. The conclusion we can draw from this is that we have to create a network where each neuron processes data from only a restricted area of the image known as the neuron’s receptive field, and not the entire image.

**Convolutional Neural Networks (CNNs)**

In convolutional neural networks, we use three new types of layers, convolution layers, ReLU layers and pooling layers

*Convolution layers:* Convolution layers consist of a number of filters (say ‘n’). The convolution layer takes an input image as is and performs 2D convolution operation on it with each of its ‘n’ filters, and returns ‘n’ output images known as feature maps. The filters replace neurons and the filter coefficients are just like weights in the sense that they are trainable.

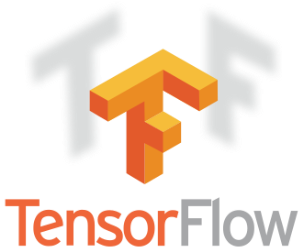
*ReLU layers:* ReLU layer is just an activation function layer; it performs ReLU activation function on each pixel in the input image to return an output image of same size. ReLU is an abbreviation for rectified linear unit, it is the activation function defined by f(x) = x for x >= 0 and f(x) = 0 for x < 0.

*Pooling layers:* Pooling is an operation where we downsample the input image by combining every m×n group of adjacent pixels together into a single pixel. In the most common type of pooling, max pooling, the max value in every group is retained while the other values are discarded.

These three layers are used along with the traditional fully connected layers to form a neural network. CNNs are classified as deep learning networks because they usually contain a large number of hidden layers.

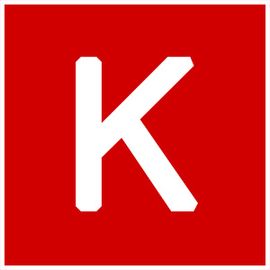
Using CNNs greatly reduces the amount of manual pre-processing needed for the image. In fact since we use trainable filters we can say that the CNN learns the pre-processing itself.

**TensorFlow:**

****

TensorFlow is an open source machine learning framework developed by the Google Brain team. It is a symbolic math library that is used for machine learning applications like neural networks. It is available for PCs and servers running Windows, MacOS or Linux and even for mobile Android or iOS devices. It provides APIs in Python, C++, Java and many more programming languages. TensorFlow is used in Google applications such as RankBrain and DeepDream.

**Keras:**

****

Keras is an even higher level library that runs on top of TensorFlow, which simplifies the process of implementing deep neural networks even more. It is an open source neural network library written in Python and can operate on TensorFlow, Theano, Microsoft Cognitive Toolkit (CNTK) or Apache MXNet. If it takes 11 lines of code to implement a CNN in Keras the same would take 42 lines of code in TensorFlow. [source: <http://adventuresinmachinelearning.com/keras-tutorial-cnn-11-lines/>]

**Unicode:**

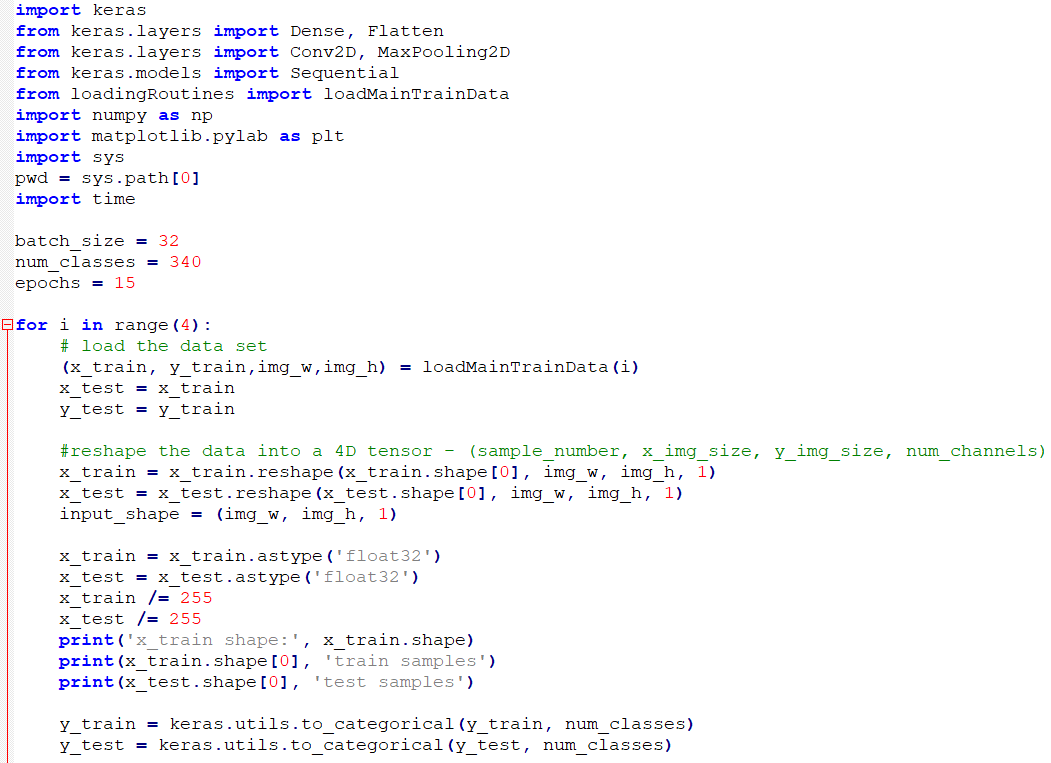
Unicode is defined by Wikipedia as “a computing industry standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems.” Unicode was developed when 8-bit encoding systems such as ASCII were still popular. Since ASCII could hold only 256 characters, only Roman characters were represented.

Many countries had developed their own versions of ASCII for their native languages. For example India developed ISCII. Alternatively, early Kannada writing software such as Baraha used customized ASCII fonts that merely rendered their own Kannada glyphs in place of the correct ASCII glyphs. While this solution is good for printing Kannada text on paper it is not suitable for applications such as transmitting Kannada text online or displaying Kannada text in web pages or on mobile devices. A universal encoding standard is needed. Unicode uses 16 bits (specifically UTF-16 uses 16 bits), which is way more than enough to represent characters in all of the world’s living languages, as well as historic scripts such as Brahmi.

UTF-16 assigns each of its characters with a unique 16-bit identification number known as a code point, and leaves the rendering of the character to the software. The code points for Kannada characters are in the range of 0x0C82 to 0x0CF2. This range of code points is reserved exclusively for Kannada characters, unlike in ISCII where the same character in different Indian languages is assigned the same code point.

**Codes and explanation:**

**cnn\_train.py** : This code is used to build the CNN model, train it with the dataset and save it to a .h5 file.



**Description:**

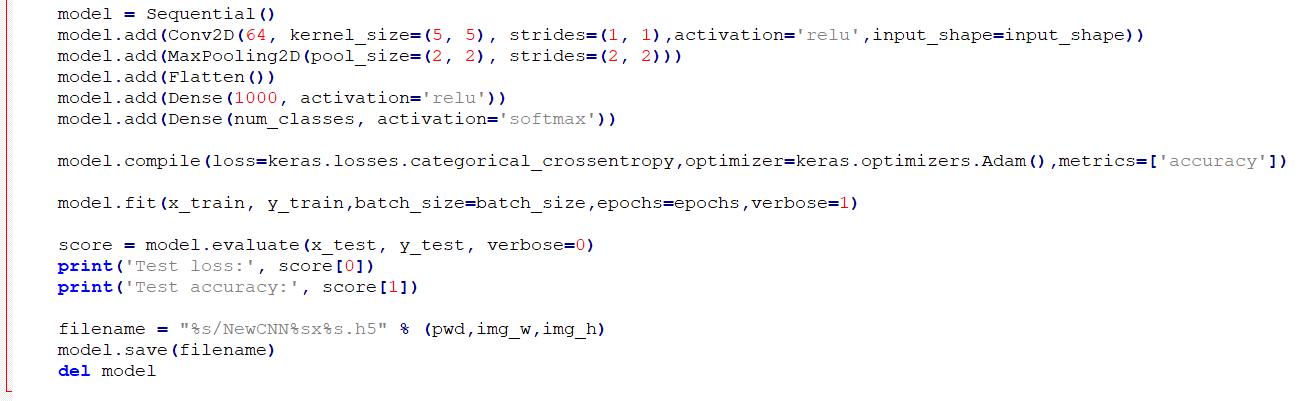
The batch\_size variable represents the number of samples per update. The num\_classes variable represents the number of classes, in our case the number of possible Kannada characters.

Epochs is the number of iterations over the entire dataset. Higher number of epochs means we train the network for longer duration of time, which gives higher accuracy.

Here we are training not one but 4 CNNs, with input sizes 15x20, 25x20, 30x30 and 40x20.

The function loadMainTrainData(i) returns the training data (x\_train,y\_train) and the input size (img\_w,img\_h) for the CNN labelled i. This function is defined by us in another file.

Initially we have to perform a number of transformations on the training data so as to express it in correct form (i.e. reshaping matrices, re-mapping values and such. Then we move on to create and train the CNNs.

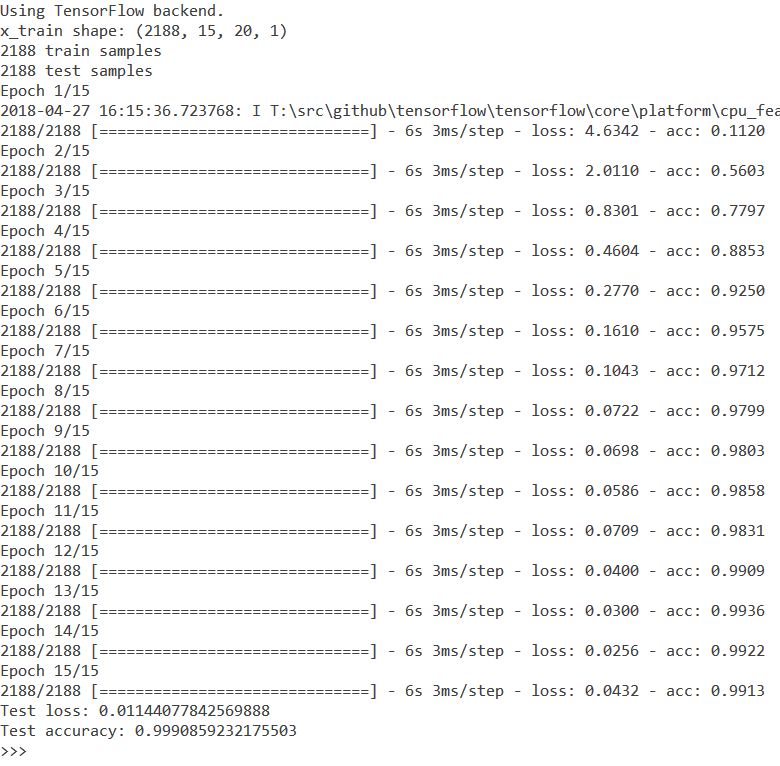


Now we use the Keras library to create a Sequential model. In the Sequential model the layers are stacked one after another in the order of input to output.

We are implementing a CNN that begins with one convolutional layer (Conv2D) with 64 5x5 filters. This layer takes a single input image and returns 64 feature maps, on which we apply ReLU activation function. This is followed by a 2x2 max-pooling layer (MaxPooling2D) which reduces the size of the maps. Next we convert everything into a single dimension (Flatten). This becomes an input for a fully connected (Dense) layer of 1000 hidden neurons with ReLU activation function and finally we have a fully connected output layer of 340 neurons with softmax activation function.

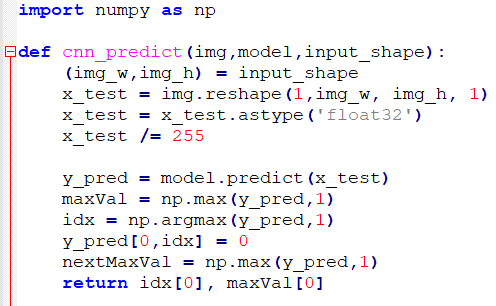
Now we compile the model using the compile() function. We use categorical cross-entropy as our loss function. Loss function is the parameter we need to minimize using our optimization algorithm when we train our network. The optimization algorithm used is the Adam optimizer, explained in detail in the paper *Adam: A Method for Stochastic Optimization*. Also we want to monitor the accuracy at each stage so we configure the network so.

Next we do the training using the fit() function. This is the part of the code that takes the longest time to excecute. A demonstration on the console is shown below:

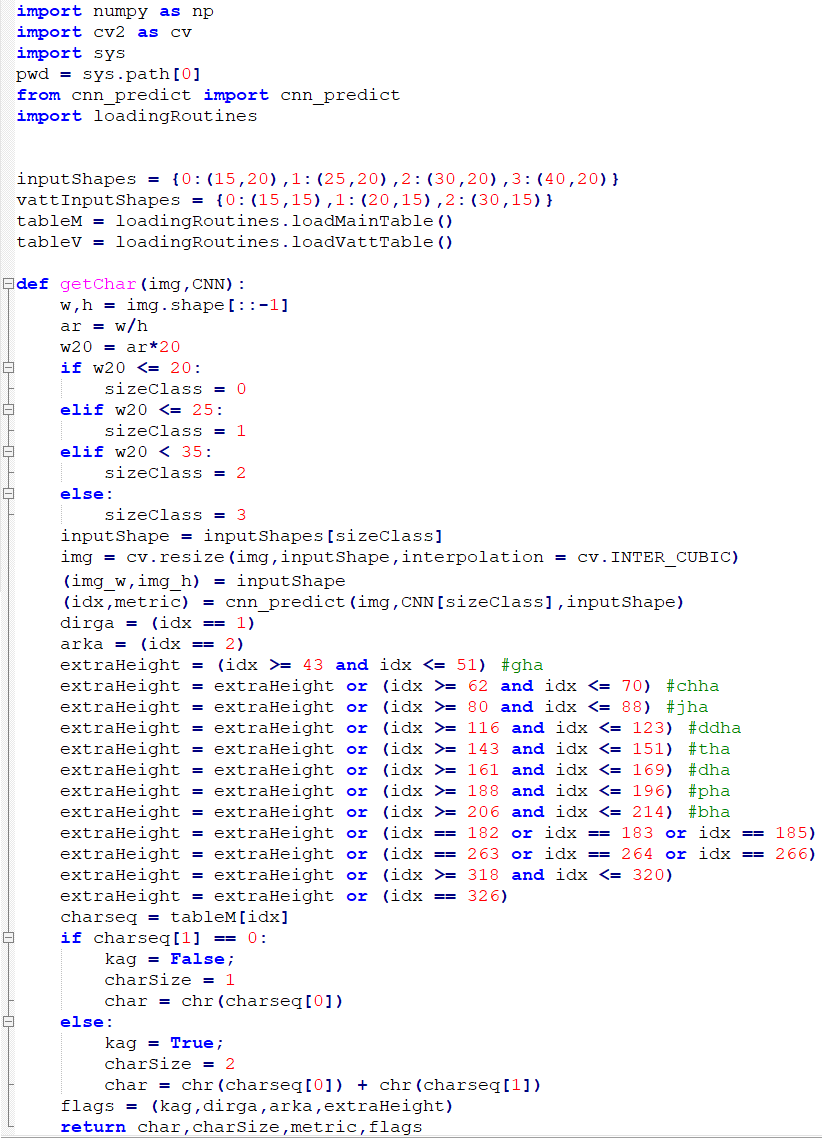


In the end we use the evaluate() function to evaluate the test loss and the test accuracy of the final trained model. Finally we save the trained model in a .h5 file on the secondary storage, then we delete it from the RAM.

**cnn\_predict function**: This function is called when we want to query one of our trained CNNs with an input. It takes an input image and a reference variable to a CNN which has already been loaded into the RAM by the external program that calls this function, and returns the UID of the character, obtained by taking the index of the output neuron that outputs the maximum value. Along with this we give the max. value itself as another output, to use as an accuracy metric.

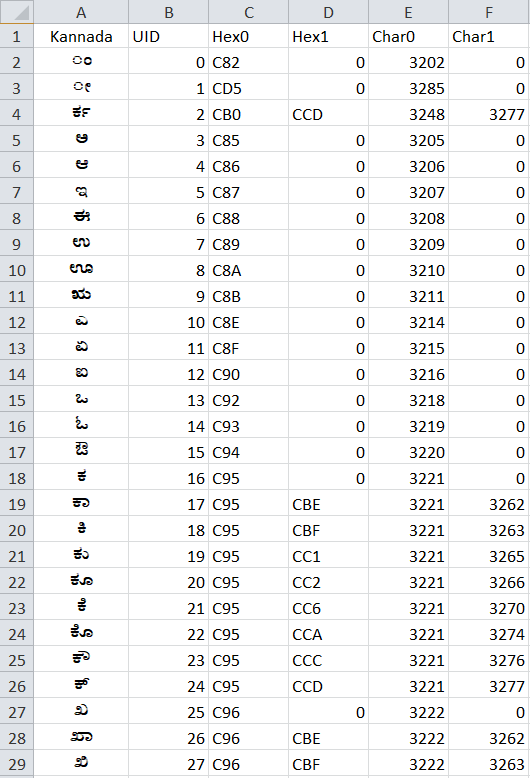


**getChar function:** This function is used for identifying characters

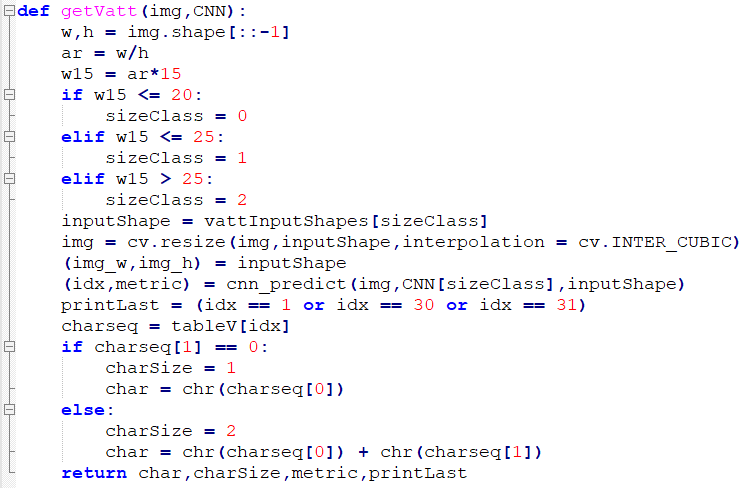


This function takes an input image and a reference variable to a list of CNNs which has already been loaded into the RAM by the external program that calls this function. It classifies the image into one of 4 size classes based on aspect ratio, then resizes and feeds this image to the respective CNN (using the cnn\_predict function).

The function returns the application-defined UID of the character identified which we query into a table to get the Unicode encoding (in decimal) of the aforementioned character. A Kannada character may be a single Unicode character or may actually be a combination of two Unicode characters. The output of this function is the Unicode string of the identified character, along with the size of the character, a value from 0 to 1 indicating the accuracy of the identification, and a number of flags.



**getVatt function:** This is used for identifying vattaksharas. It is similar to the getChar function but with only 3 size classes, and different flags.

****

**getWord function:** This function is used to identify an entire Kannada word from an image.

It first draws a bounding box around the word and crops out the extra whitespace around it using the selfCrop function (defined in imgProc\_util.py). Then it gets the position of the base line using getBase function (defined in imgProc\_util.py). Next it performs thresholding and inverting followed by dilation and then labels the connected components i.e. characters. Any component whose height is less than one-fifth of the image height is discarded. Then it has to rearrange the labels in the correct order.

For every label in the reordered list of labels, we isolate the corresponding character from the image, then we identify it as a regular character or vattakshara character based on whether it is above or below the base line. We then use the getChar function or the getVatt function respectively.

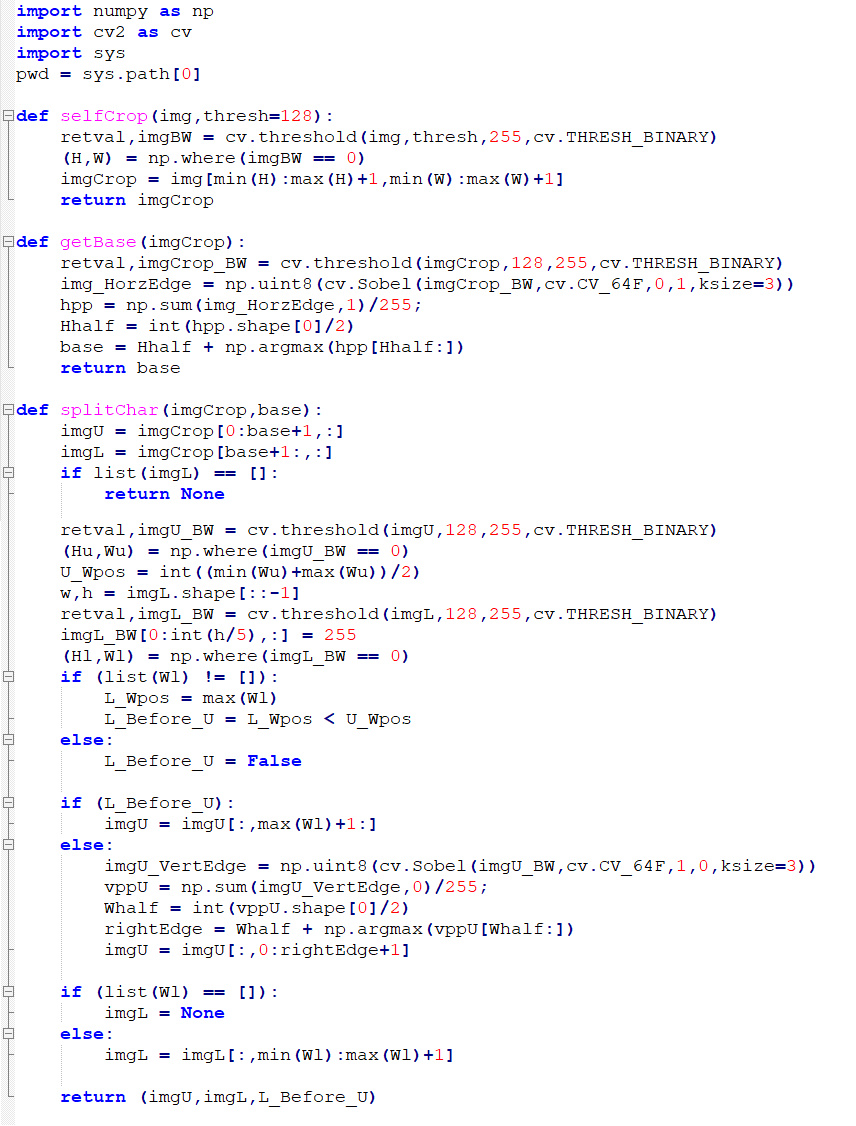
After the character is identified it is usually appended to the end of the output string ‘word’. But for vattaksharas and some other special characters we have to insert the new character in between the word.Occasionally we get a regular character joined with a vattakshara. In this case what we do is split the image into two parts along the base line, then apply getChar function to the upper part and getVatt function to the lower part, and then prints them both, usually character first and vattakshara next. Sometimes the vattakshara is joined with the next character and we may have to print the vattakshara before the character. We implement all of these special cases in our program, with separate printing procedures for each.

Finally after all the characters are identified we return the string ‘word’ as the output of this function.

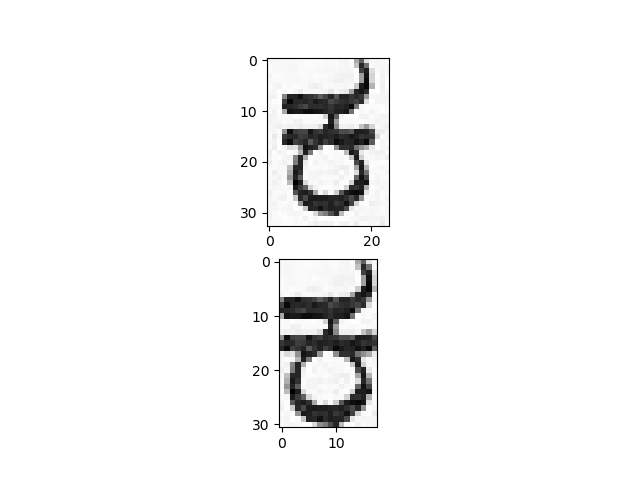


**(ADD FROM NEXT PAGE)**

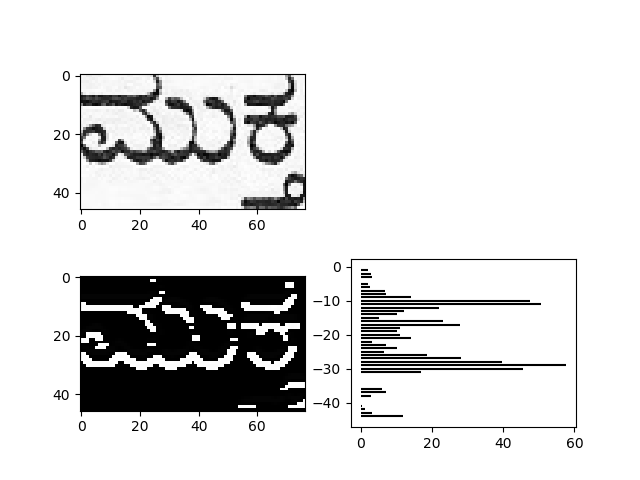
**imgProc\_util.py**: This code contains some pre-processing functions we use in our code:



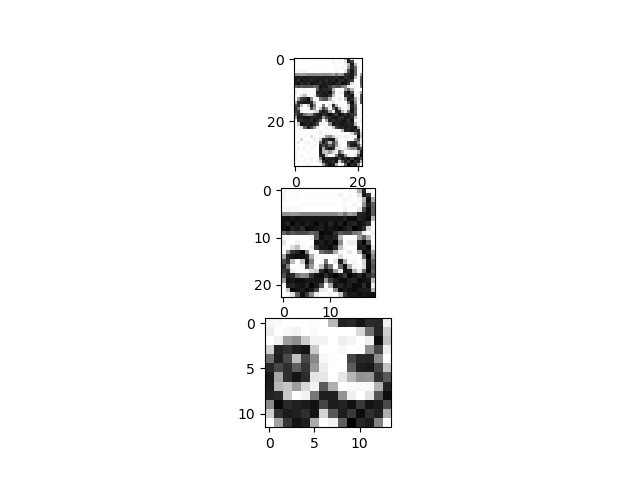
The function **selfCrop** takes an image of a word or character and trims out any extra white pixels surrounding the object in focus. A demonstration is shown below:



The function **getBase** is used for identifying the base line. It works by taking an input image of a word, then thresholding and using Sobel edge detection to get all the **lower edges** in the image. It takes the horizontal sum of edge detection result and finds the maximum value in the lower half. It is demonstrated for the image below. The base line has been identified to be at position 29 on the H (height) axis.

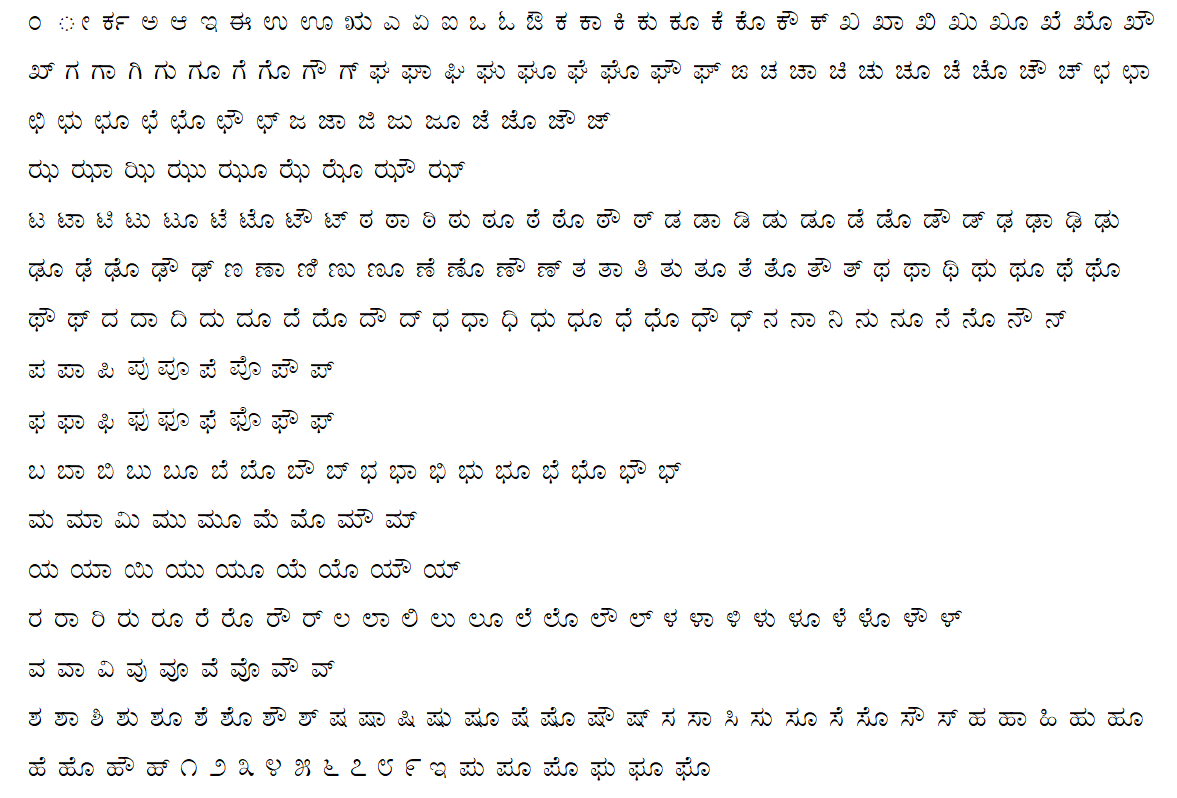


The function **splitChar** is used when a character and a vattakshara are joined and we want to separate them. The function takes an input image and splits it into two images along the base line.



**Creating our Dataset:**

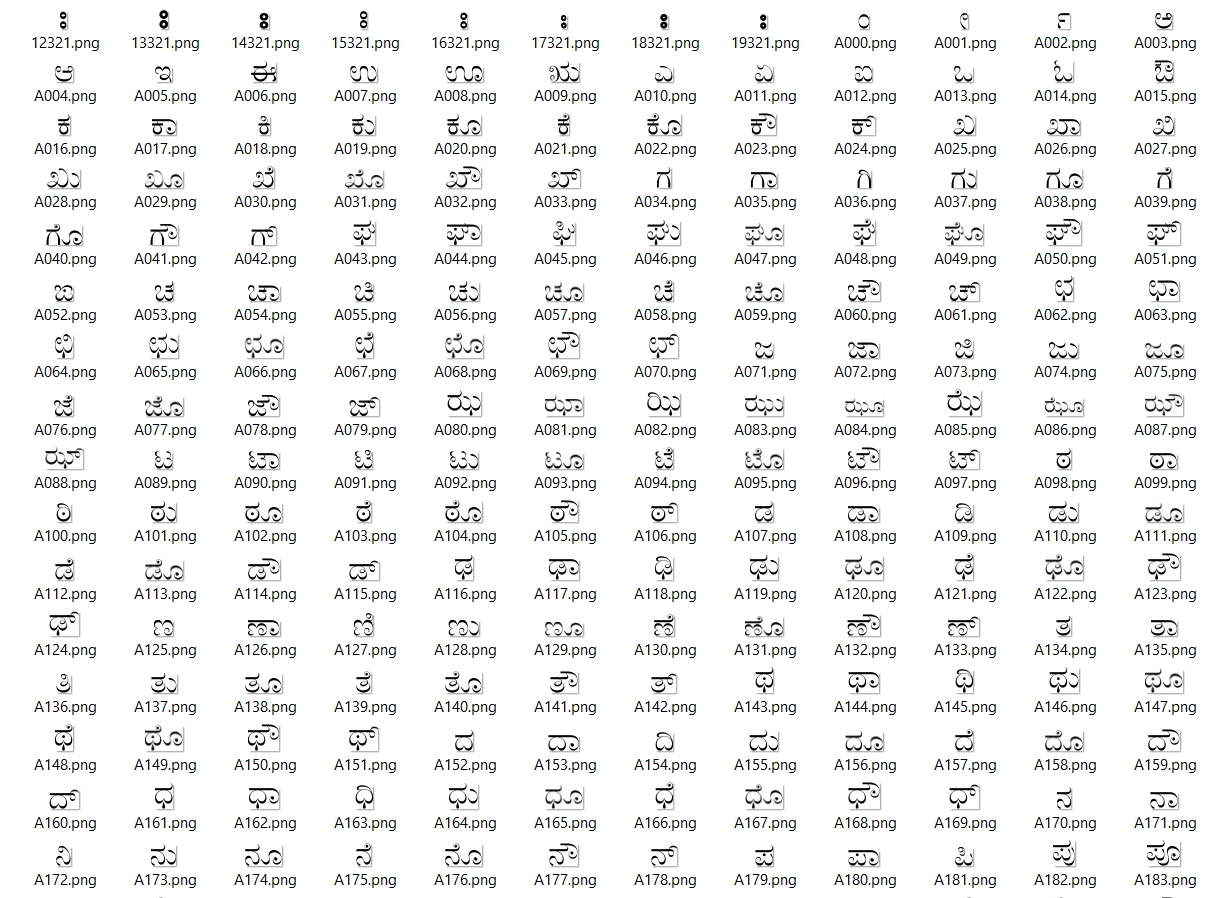
To create our dataset, we first list out all possible Kannada glyphs in a Microsoft Word document, like so:



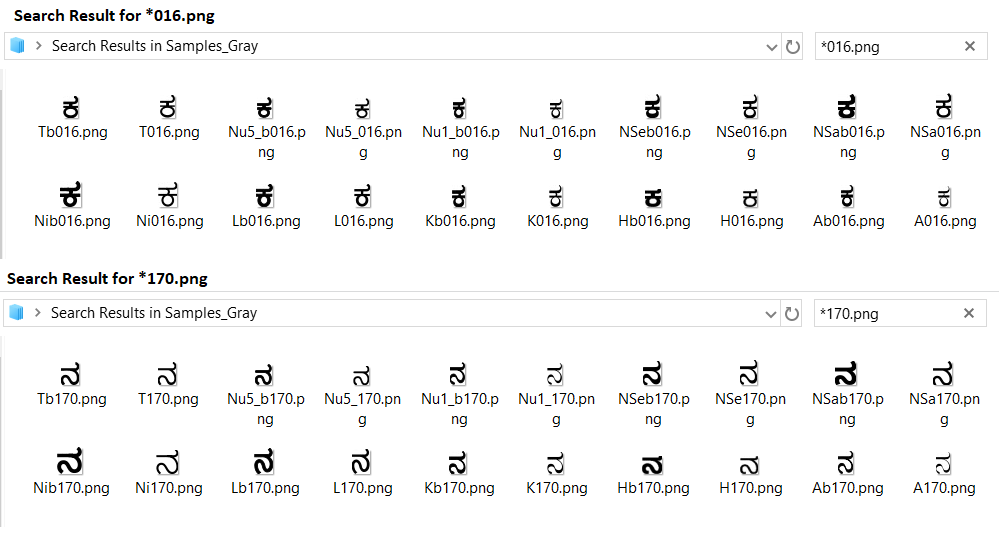
We do this for 10 Kannada fonts, in normal text and in bold text for each font. The fonts used are given below:



Next, we convert each page in the Word document into a PNG image, and then we run the image through a segmentation code written by us in GNU Octave. The code separates the individual characters and saves all of them into separate images. The results are shown below:



The code not only separates the characters but also labels each character in order with a 3-digit number from 0 to 339, which will become the UID of the character. Any images with the same 3-digit number as a suffix now contain the same character, we can demonstrate this using file search. Any inconsistencies are corrected manually:



Now we split the entire dataset into 4 groups as follows:

We resize each image to a height of 20 pixels keeping the aspect ratio constant, then we study the width of each image.

* Images of width 22 or less are resized to 15x20 and saved into a folder “Samples15”
* Images of width 18 to 27 are resized to 25x20 and saved into a folder “Samples25”
* Images of width 23 to 37 are resized to 30x20 and saved into a folder “Samples30”
* Images of width 33 or greater are resized to 40x20 and saved into a folder “Samples40”

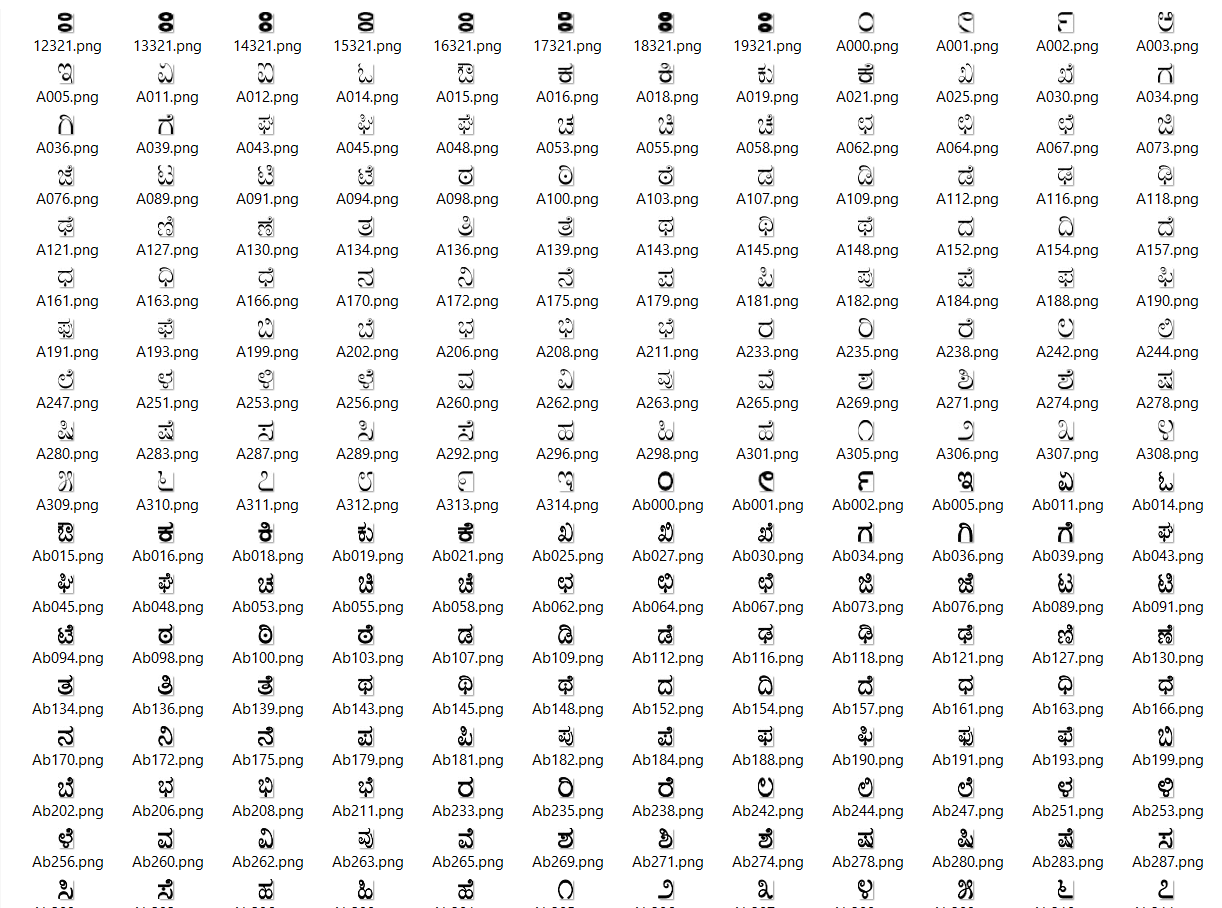
Note that the same sample may appear in two folders with different sizes.

***Why is this done?***

A CNN takes a constant input size, however the sizes and aspect ratios of our characters are variable. All characters must be resized to the constant input size before being input to the CNN. We could use a single CNN for all the characters ignoring the difference in aspect ratios, but this would give very inaccurate results. The solution is to divide the dataset into 4 classes based on the aspect ratios, then use these datasets to train 4 CNNs of different input sizes [i.e. 15x20,25x20,30x20 and 40x20.] In this way we try to roughly preserve the original aspect ratio of the characters so they can be identified properly.

The results of dividing and resizing samples are shown below:

Samples resized to 15x20 and stored in “Samples15”:



Samples resized to 40x20 and stored in “Samples40”:



Now we are ready to use this dataset to train our CNNs.

**Reference:**

[**http://cs231n.github.io/convolutional-networks/#overview**](http://cs231n.github.io/convolutional-networks/#overview)

[**https://en.wikipedia.org/wiki/Convolutional\_neural\_network**](https://en.wikipedia.org/wiki/Convolutional_neural_network)

[**http://adventuresinmachinelearning.com/keras-tutorial-cnn-11-lines/**](http://adventuresinmachinelearning.com/keras-tutorial-cnn-11-lines/)

[**https://keras.io/**](https://keras.io/)