**Progress report:**

The handwritten text recognition is a version of OCR (Optical Character Recognition). It enables to scan the handwritten text on paper into digital text that can be edited or changed accordingly.

It is very useful and important because everything is now digitalized and it is not possible to carry documents everywhere. So, it these documents can be digitalized without manually typing much time will be saved.

Here is the link to the tensorflow documentation for the better understanding: <https://www.tensorflow.org/api_docs/python/tf/dtypes/DType>

1. **Data Collection**

We take out data from the IAM dataset. IAM dataset is large database for handwritten text of various formats. From there we take the data and shape it to our use.

1. **Imports of the basic library**

Here we used different libraries such as Keras, Tensorflow, matplotlib, numpy

**Keras** is free and open source high level neural library and supports high leve backend computation engines. It is very much user friendly and uch faster to use.

**Tensorflow** is a framework that support high and low level api (application programming interface)

**Matplotlib** is a comprehensive library for creating static, animated, and interactive visualizations in Python.

**NumPy** is a Python library used for working with arrays. It also has functions for working in domain of linear algebra, fourier transform, and matrices.

**StringLookup** is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference

The **OS module** in Python is a part of the standard library of the programming language. When imported, it lets the user interact with the native OS Python is currently running on

1. **Dataset Splitting**

In the IAM words dataset there are lots of unnecessary data that are not needed. So, in our data splitting section we first remove those unnecessary data.

We look up with line that start with ‘#’ and after that we remove lines that have errors.

After that we split the the data for training testing and validation. So we split it into 90:5:5 [Trainning:Testing:Validation].

1. **Data input Pipeline**

Here we take the whole path of the data that we can pipeline for the data input segment. So, we classify all the paths by each folder and its subfolder and finally the image/png file path name.

Then from that we append and take only the word from the list. And we clean the test validation data set also.

1. **Building Character Vocabulary**

In this section we take our words as input for example ‘cat’ , ‘dog’. So, preprocessing the labels of the characters the vocabulary we should get is (a,c,d,g,o,t) and it doesn’t include the special tokens.

Then we convert the characters into numerical forms because in machine learning the data should be in numerical form for the ease of calculation and after getting the result we convert it into its original format. The char to num converts the strings into mathematical array.

1. **Resizing Image without Distortion**

In OCR models they work with rectangular images. SO, for getting an accurate result we resize our images and that too without any distortion. It helps image to go in a uniform size which is required for mini-batching.

At first here we will read the image, them decode the image and preprocess it. The padding and the transpose and flipping of the image is a proven method to increase the accuracy of tests. The details can be found here : <https://www.academia.edu/43006409/Application_of_Matrices_in_Flipping_the_Image_Using_Python_Program>

1. **Prepare tf.data.Dataset objects**

Here we call the train\_img\_paths and train\_labels\_cleaned. They give the input of the input path and its words respectively. Because it will return the numerical array value and batch the values which we will push into the model for training.

1. **Visualize a few samples**

Here the labels are now defined and put on the axis plot along with some test data from the wordset. This is done for the purpose of checking whether the image processing was done without any flaws, so that our trained data can infer correctly

1. **The model**

The above code defines a convolutional recurrent neural network (CNN-RNN) model for recognizing handwritten text from images. The model architecture consists of:

* Input layer for the image with shape (image\_width, image\_height, 1)
* Two convolutional layers with 32 and 64 filters respectively, each followed by max pooling layers
* Reshaping layer to transform the output from the convolutional layers into a shape of (image\_width // 4, (image\_height // 4) \* 64)
* A fully connected dense layer with 64 units and a dropout layer to prevent overfitting
* Two bidirectional LSTM layers with 128 and 64 units respectively, which process the sequence of feature vectors extracted from the image
* A dense layer with a softmax activation function, which outputs a probability distribution over the vocabulary of possible characters
* A CTC layer that calculates the CTC (Connectionist Temporal Classification) loss at each step. This layer takes two inputs: the true labels and the predicted probabilities from the softmax layer.
* Finally, the model is compiled using the Adam optimizer.

The build\_model() function returns the compiled model, which can be used for training and evaluation on handwriting recognition tasks.

The block consists of a convolutional layer with 64 filters of size (3,3), with the ReLU activation function, He normal weight initialization, and same padding. After the convolutional layer, a max pooling layer is applied with a pool size of (2,2) and the name "pool2".

The feature maps produced by the second convolutional block are downsampled by a factor of 4 compared to the input, so they need to be reshaped before being passed to the recurrent neural network (RNN) part of the model. This is done using a Reshape layer with a target shape of (image\_width // 4, (image\_height // 4) \* 64), which flattens the spatial dimensions and concatenates the feature maps along the channel dimension

.

After reshaping, a dense layer with 64 units and the ReLU activation function is applied, followed by a dropout layer with a rate of 0.2 to prevent overfitting. Finally, two bidirectional LSTM layers with 128 and 64 units, respectively, are applied, both with return sequences set to True to output the hidden state at each time step. Each LSTM layer also has a dropout layer with a rate of 0.25 to further prevent overfitting.

1. **Edit distance callback**

The calculate\_edit\_distance function takes in two arguments, labels and predictions, and calculates the edit distance between them. It converts the labels to sparse tensors, makes predictions and converts them to sparse tensors, then computes the individual edit distances and returns their average.

The EditDistanceCallback class takes in a prediction model as an argument in its constructor. It has an on\_epoch\_end method that computes the edit distance between the validation labels and predictions using the calculate\_edit\_distance function and prints out the mean edit distance for the epoch.

1. **Training**

the model is trained using the fit() function on the train\_ds dataset, with validation\_ds as the validation data, for epochs number of epochs. The edit\_distance\_callback is also passed as a callback during training to compute and print the mean edit distance after each epoch.

1. **Inference**

the code uses the decode\_batch\_predictions function to visualize the predictions made by the model on a sample batch of test images. The code takes a batch of 16 images from the test dataset, makes predictions using the prediction\_model (which is defined earlier as a keras.models.Model object), and then decodes the predictions using the

decode\_batch\_predictions function.

The resulting predicted text is displayed below each image. The images are flipped horizontally and rotated by 90 degrees clockwise for better visualization. Finally, the images and predicted text are plotted using matplotlib.