Bengali Grapheme Classification - Capstone Report

The github repo for this project can be found here

Dataset Description:

The dataset, obtained from Kaggle, comes split into training and test datasets. The training dataset contains 200,840 images of the size 137 x 236 pixels. The test set contains 12 images of the same size. Along with the image data, there are csv files which describe the labels for each image.

Data Cleaning:

Since it was a Kaggle dataset, minimal data cleaning was required.

Findings from Exploratory Data Analysis:

Before analyzing the dataset, it is important to define what a grapheme and a diacritic are. A grapheme is, according to Wikipedia, “the smallest unit of a writing system of any given language. … a grapheme is a letter or a set of letters that represent a sound (more correctly, phoneme) in a word”. A diacritic is essentially an accent on a character.

Exploring the dataset, the 5 most frequent grapheme roots, vowel diacritics and consonant diacritics are shown below:

Figure 1: 5 Most Frequent Grapheme Roots

Figure 2: 5 Most Frequent Vowel Diacritics

Figure 3: 5 Most Frequent Consonants

The above characters however, are not hand-written characters. They are generated using the computer font *Kalpurush*. The hand-written characters themselves will not be as neat, and will be combinations of grapheme roots, vowel diacritics and consonants (although many characters might not possess all three components).

The figure below is a random image taken from the training data set (far left) compared against its constituent components.

Hand-written Character Grapheme Root Vowel Diacritic Consonant

Figure 3: 6 Hand-written Character Decomposition

From the figure we can make out the grapheme root and vowel diacritic from the hand-written character. But the consonant takes a different shape when integrated into the character. This is a characteristic of Indic languages and is something the classifier we build must be cognizant of.

Methodology of Experimentation:

The overall methodology of building the classifier adopted in this project contains the following steps:

1. Create an experimental matrix
2. Preprocess training data (resizing and threshold filtering)
3. Run experiments and track artifacts
4. Assess results from step 3 and select top 5 models
5. Ensemble the top 5 models

To maintain reproducibility, the random seed is set at the start of training and is kept constant. This enables like-to-like comparisons of the models. The following sections describe the key components of this methodology.

Creating an Experimental Matrix:

Since the project is image analysis based, it was decided that the classifier should be a convolutional neural network. The Keras Functional API was used to build the models. Instead of employing transfer learning and using a pre-trained network, a custom architecture was implemented.

The custom architecture consisted of alternating convolution and pooling layers. This will be followed by 2 hidden dense layers and finally, 3 dense output layers for the grapheme root, vowel diacritic and consonants. A comprehensive experiment matrix was constructed to check the performance of different model architectures by iterating on the following parameters:

- Convolutional kernel size
- Number of convolutional filters
- Number of convolutional layers
- Drop-out rate

The total number of experiments in the matrix was 81. The number of nodes in the two hidden dense layers were kept at 1024 and 512 respectively. Prior to these two dense layers, batch normalization was done with a momentum factor of 0.99.

Early stopping is implemented on the root prediction accuracy on the validation set with a patience of 15 epochs.

A sample model architecture is shown below:

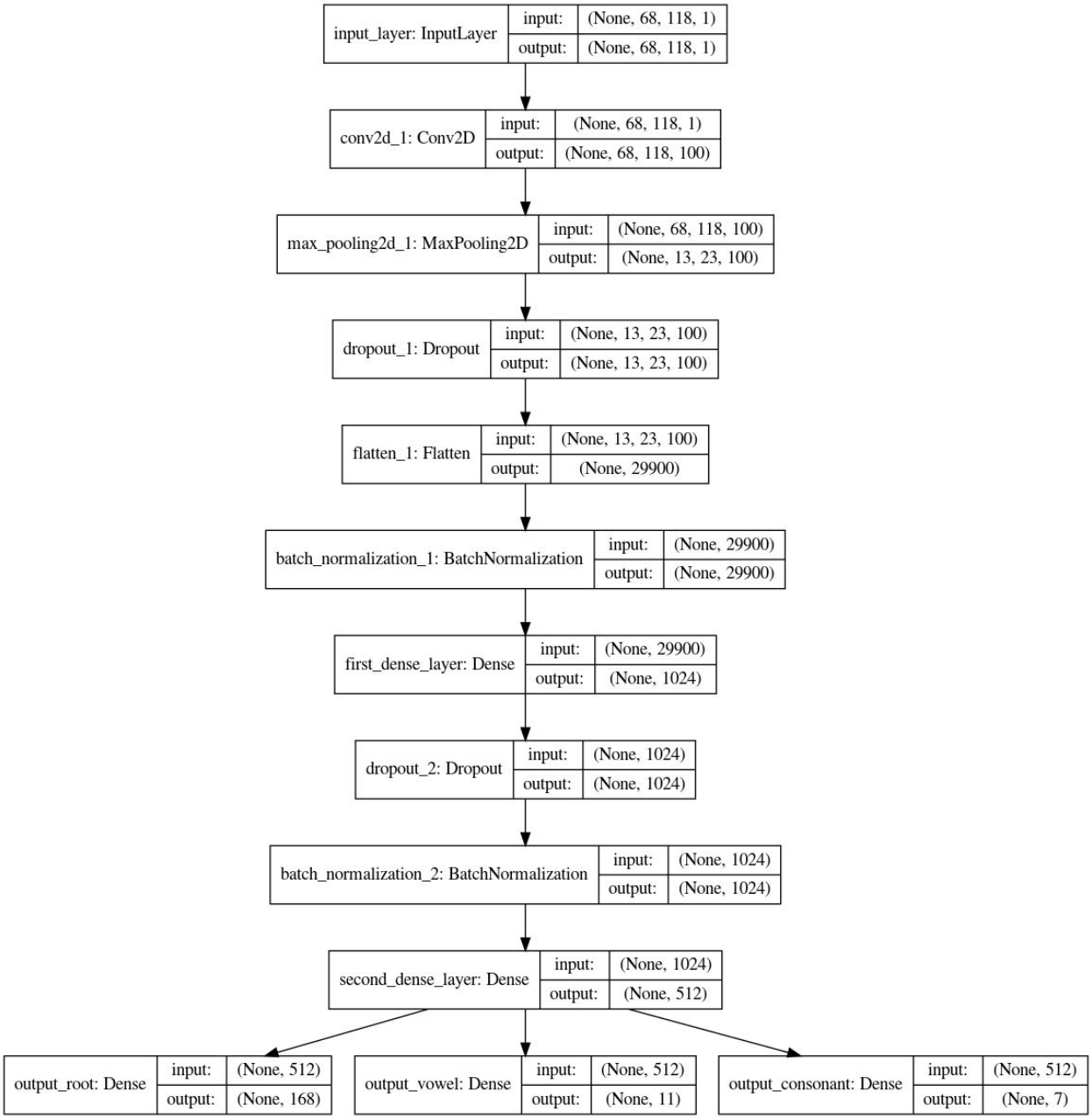

Figure 4: A Sample Model Architecture

Data Pre-processing and Augmentation:

The images in the dataset are 137 x 236 pixels. Training a classifier on such a "high resolution" will cause training to be slow (because of the number of calculations required). Also, at this resolution, augmenting the image data by adding rotations and pixel shifts caused my computer to run out of memory. Therefore, resizing the images was important. However, while resizing it is important to keep the aspect ratio of the image. Otherwise, the hand-written characters will

be skewed one way or another. Therefore, the OpenCV module was used for image data pre-processing.

Firstly, a threshold filter was applied to increase the contrast between the character itself and the background. Otsu filtering was used for this purpose, to eliminate the trial and error of manually specifying the threshold. Following the threshold filtering step, the image was resized to half the original resolution, i.e 68 x 118 pixels. The figure below compares the original image, the resized image, the threshold filter applied to the original and the threshold filter applied to the resized image.

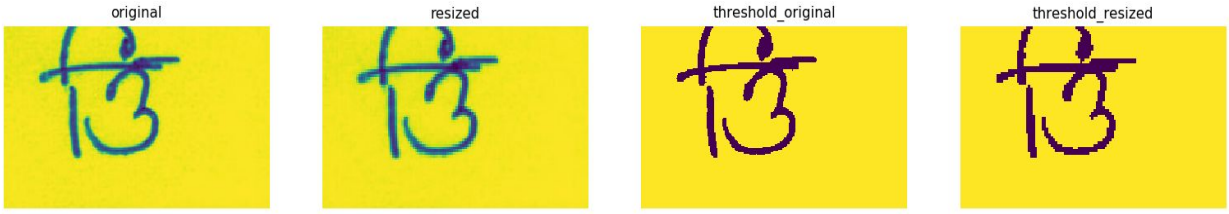

Figure 3: 7 Comparison of Input and Processed Images

For data augmentation, a multi-output generator was defined. The augmentations done are to adjust rotation range by 10^{o} and the width and height shifted by 20%.

Running Experiments and Tracking Artifacts:

The MLflow API was used to run experiments and track the model artifacts and metrics. The MLflow library comes with an autologger for Keras models, which was used to automatically log all the artifacts, metrics and parameters for each model.

Model Selection:

The metric used to evaluate model performance was accuracy. The accuracy metric was calculated on the predictions for grapheme root, vowel diacritic and consonants.

Results and Discussion:

The experimental matrix was implemented on a local machine with an 8GB NVIDIA GTX 1070 GPU and 16 GB RAM. Each experiment took roughly 1.4 hours, resulting in a total experimentation time of 4.7 days.

Table 1 below shows details of the best 5 models from experimentation along with the performance on the validation data (obtained from training data), sorted by performance on predicting validation root accuracy. The public and private scores from the Kaggle competition are also provided for reference.

Table 1: Best Performing Models - Architecture and Results

Model #	Kernel Size	Drop out Rate	# Convolutional Layers	# Convolutional Filters	Validation Set Root Accuracy	Validation Set Vowel Diacritic Accuracy	Validation Set Consonant Accuracy	Kaggle Competition Public Score	Kaggle Competition Private Score
57	6	0.25	5	100	0.922	0.985	0.975	0.928	0.865
33	5	0.25	5	75	0.921	0.976	0.977	0.923	0.861
56	6	0.25	4	100	0.911	0.980	0.977	0.923	0.863
60	5	0.25	5	100	0.909	0.979	0.979	0.920	0.858
63	4	0.25	5	100	0.914	0.974	0.977	0.920	0.858

The Kaggle competition scores are calculated by computing a 'macro-averaged recall' calculated for each component and then taking a weighted average of these three scores, with the grapheme root given double weight. Since the models developed in this study struggled with predicting grapheme roots accurately, when compared to vowels and consonants, the Kaggle scores are affected.

To significantly increase the performance on grapheme root prediction, more complicated convolutional neural network architectures are required. For reference, the starter kernel used 20 convolutional layers and attained scores of 0.95 in the public test set and 0.88 on the private test set.

The figures below show the training performance of the best 5 models on the training and validation data.

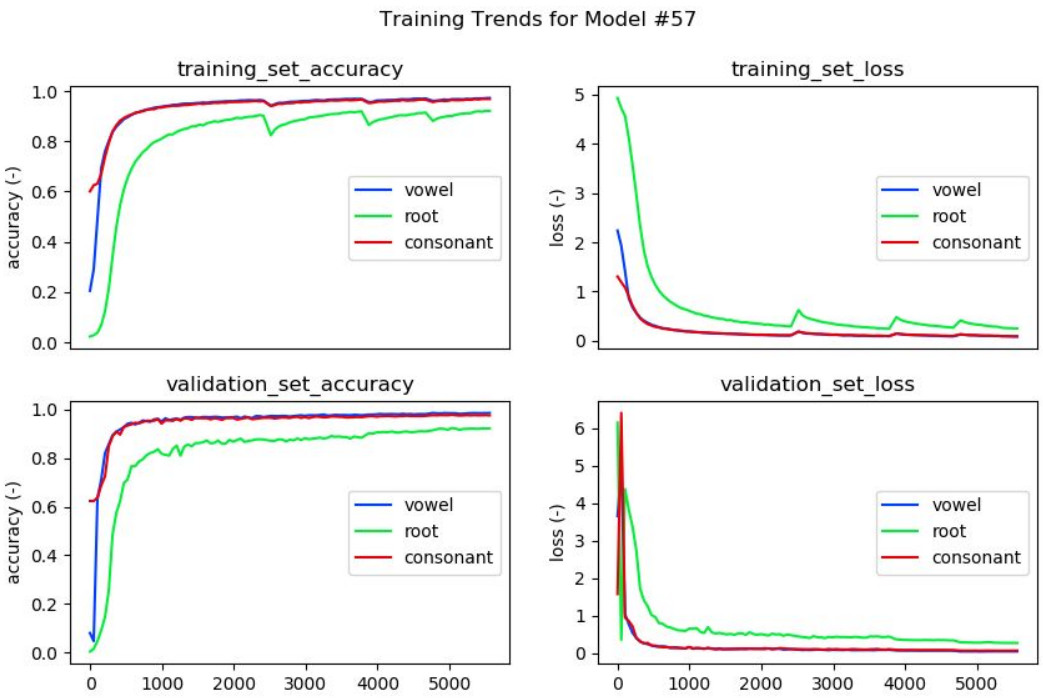

Figure 4: Training Curves for Model #57

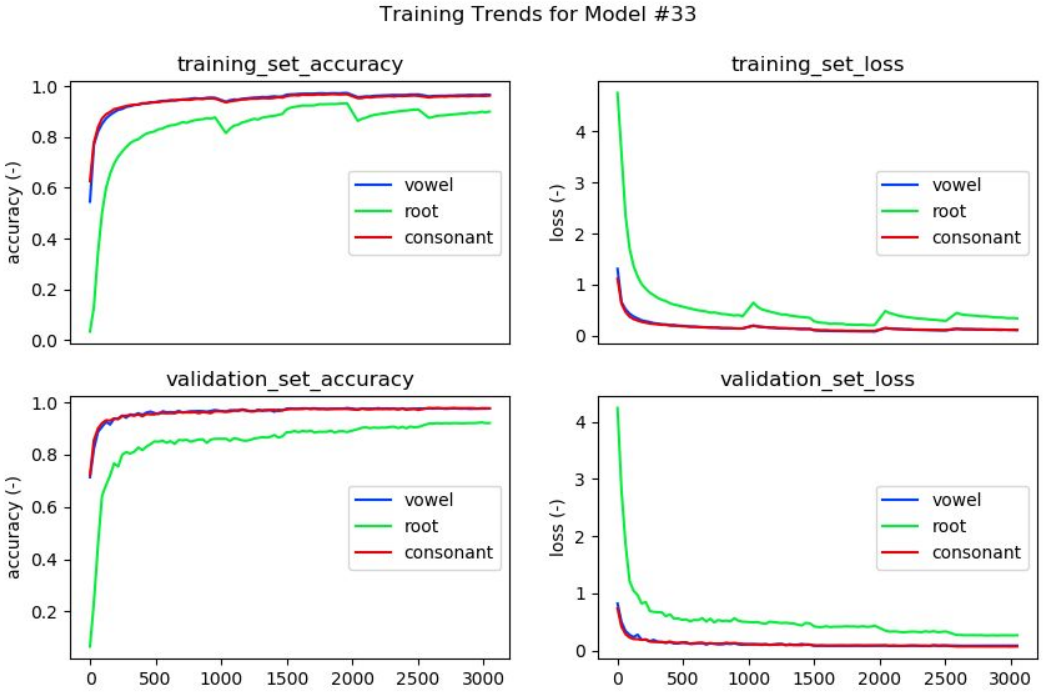

Figure 5: Training Curves for Model #33

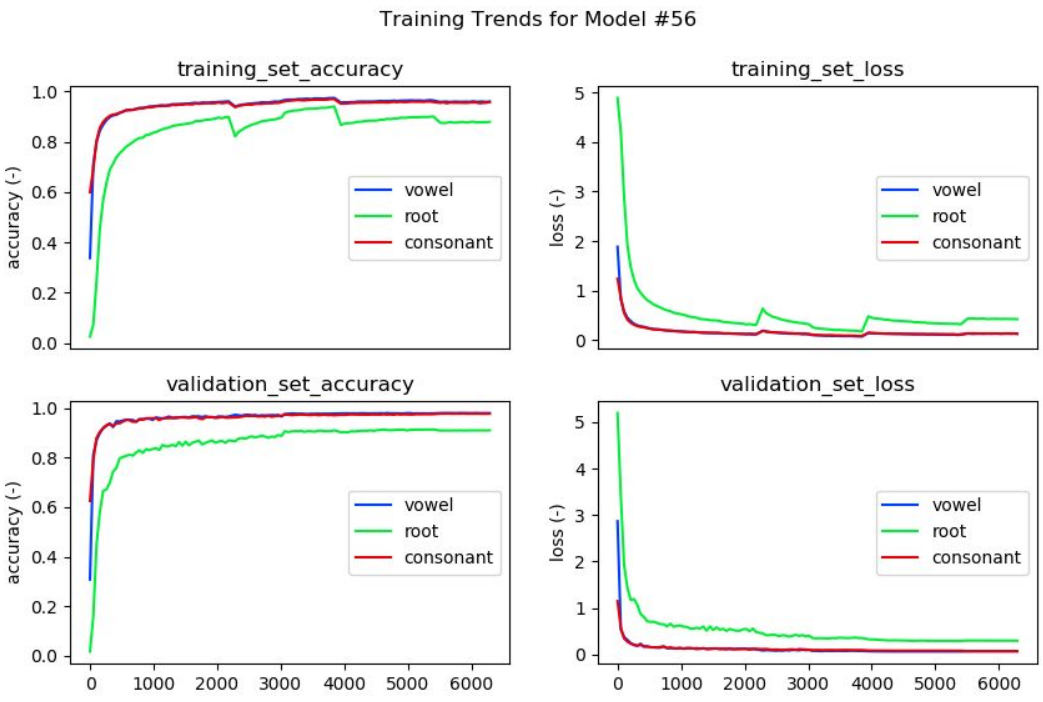

Figure 6: Training Curves for Model #56

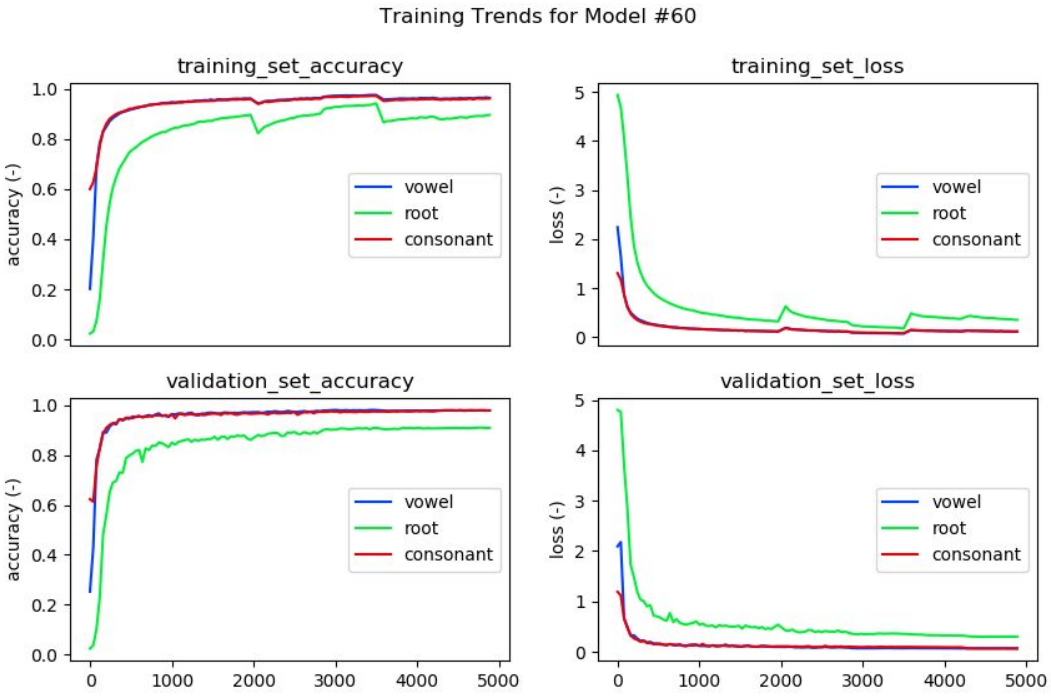

Figure 7: Training Curves for Model #60

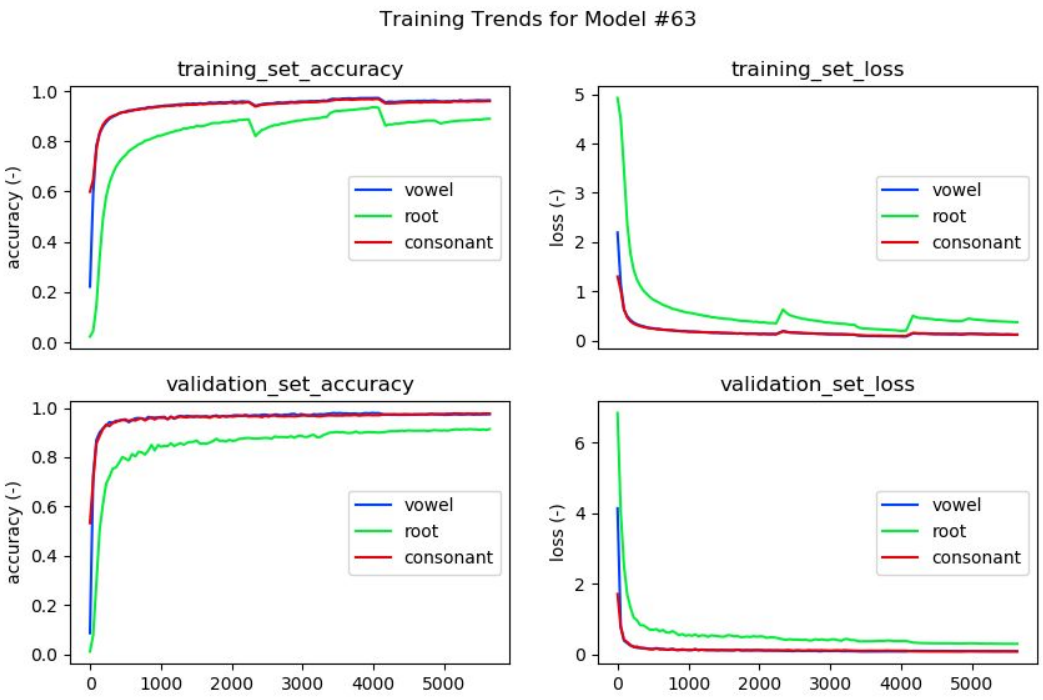

Figure 8: Training Curves for Model #63

Ensembling:

After evaluating model performance, ensembling methods were tested. The first ensembling method was a simple unweighted average. Essentially, the top 5 models as listed in Table 1 are averaged to get the final predictions for grapheme root, vowel diacritic and consonant. This ensemble method obtained a Kaggle competition public score of 0.9395 and a private score of 0.8784, which is an improvement on the single best performing model.

The second ensembling method attempted was stacking. Since the grapheme root prediction proved to be the hardest, it was decided to use a stacking algorithm to ensemble the grapheme root predictions. For the consonant and root predictions, it was decided to retain average ensembling. The stacking algorithm chosen was a simple logistic regression. For training this stacking algorithm, a different train-test split and random seed was used for better generalization. The best logistic regression model had an inverse regularization factor (C) of 0.1. This ensemble obtained a public score of 0.936 and a private score of 0.8748. Again, the ensemble performs slightly better than the single best performing model, but slightly worse than the average ensemble.

Conclusions:

- CNN based deep learning models were trained to predict handwritten Bengali characters
- 81 models were tested with varying custom architectures with parameters such as number of convolutional layers, convolutional kernel size, number of convolutional filters and dropout rates. Training performance was tracked using the MLFlow API
- The best performing models consistently obtained over 97% percent accuracy on vowel diacritic and consonant accuracy, and 91% accuracy on the grapheme root
- The highest obtained Kaggle public and private scores for single models where 0.928 and 0.865 respectively
- Two ensembling techniques were studied - average ensembling and a combination of a logistic regression stacker in combination with average ensembling. The simple average ensemble performed the best with Kaggle public and private scores of 0.9395 and 0.8784 respectively