Machine Learning Engineer Nanodegree

Capstone Project

Rohan Khanna
December 23rd, 2017

You can have a look at my project's proposal review here:
https://review.udacity.com/#!/reviews/816473 (https://review.udacity.com/#!/reviews/816473)

I. Definition

(approx. 1-2 pages)

Project Overview

Machine powered Handwritten Character recognition is a problem statement that is being solved globally by various organizations for academic as well as business purposes. As Moore's law advances further, Computers will continue to get faster, cheaper and more capable of handling larger and larger magnitudes of data. Especially image and video stream related data. Leveraging this capability to understand and translate characters from languages other than the mainstream few becomes increasingly important, and now possible.
In this project, I have created a web app capable of accepting mouse-drawn Devanagari characters and recognizing this given character. This app uses a convolutional neural network trained on the Devanagari character dataset provided by Rishi Anand and hosted on Kaggle.com. The dataset consists of more than 92000 hand written monochrome bitmap images of each Devanagari character and numeral, except vowels.

Problem Statement

The goal is to build a web application that asks the user for mouse drawn input, and outputs the most likely Devanagari character the user has drawn.

- Download and pre-process the devenagari-character-set by Rishi Anand from kaggle.com.
- Train a convolutional neural network that can recognize these hand drawn characters.
- Expose neural network to a front end application via an api written in flask.
- Build a web application that allows a user to input this application a mouse drawn Devanagari character and get an output from the neural network about what user the drawn character is.

Metrics

Accuracy is the metric that is generally chosen to maximize for this neural network. The training considers validation accuracy maximization at every training iteration as its priority.
A metric such as F1 score or log loss is also preferrable, but it is important to note that F1 score and log loss are derivable directly from the accuracy score. In such a case, it is preferrable to pick accuracy as the defining metric. It should also be noted that the entire dataset contains an exact number of data points for each class, and as such does not contain any class imbalance. F1 score maximization based neural network training is usually reserved for those cases where the dataset

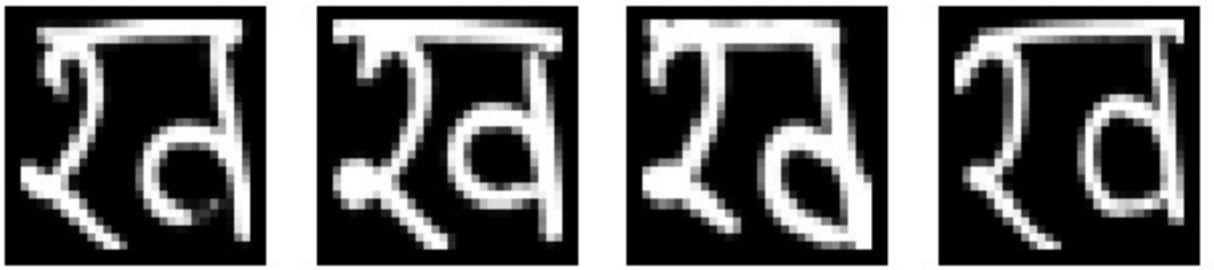
contains a highly imbalanced dataset, which is not the case here.
Accuracy at every step is determined by the equation:
accuracy = correct predictions / dataset size
There are 2 accuracy variables to consider here.

- Training accuracy :- This pertains to how accurate the model is when labelling data it has already been trained upon. This accuracy is low in the initial stages of the training period, but gradually converges to 100 percent over the course of the training period
- Validation accuracy :- This metric is the one that the model is actually trained to maximize. Since the validation set is kept isolated from the training set, only those models that show an improvement in validation accuracy are checkpointed, and further training only continues to build on top of the previous model checkpoint.

II. Analysis

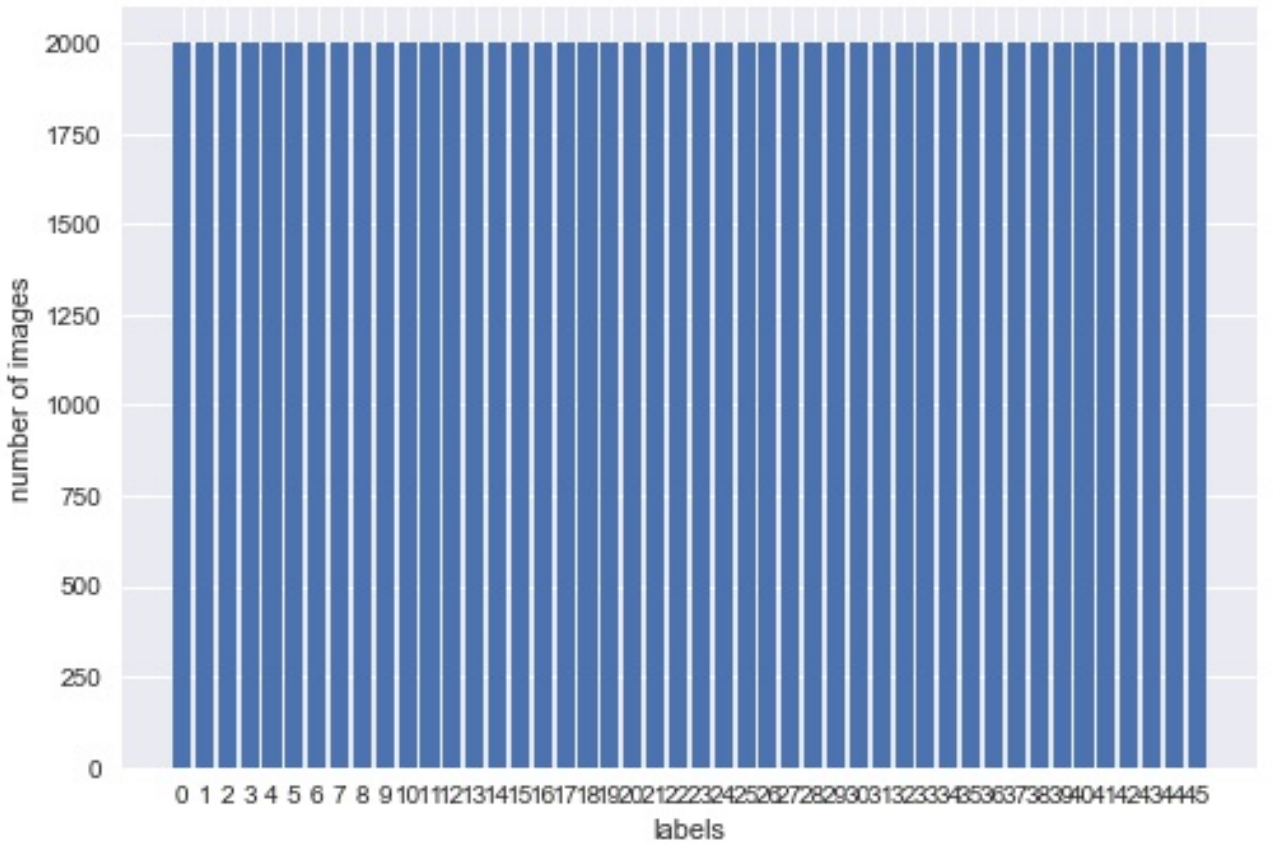
(approx. 2-4 pages)

Data Exploration



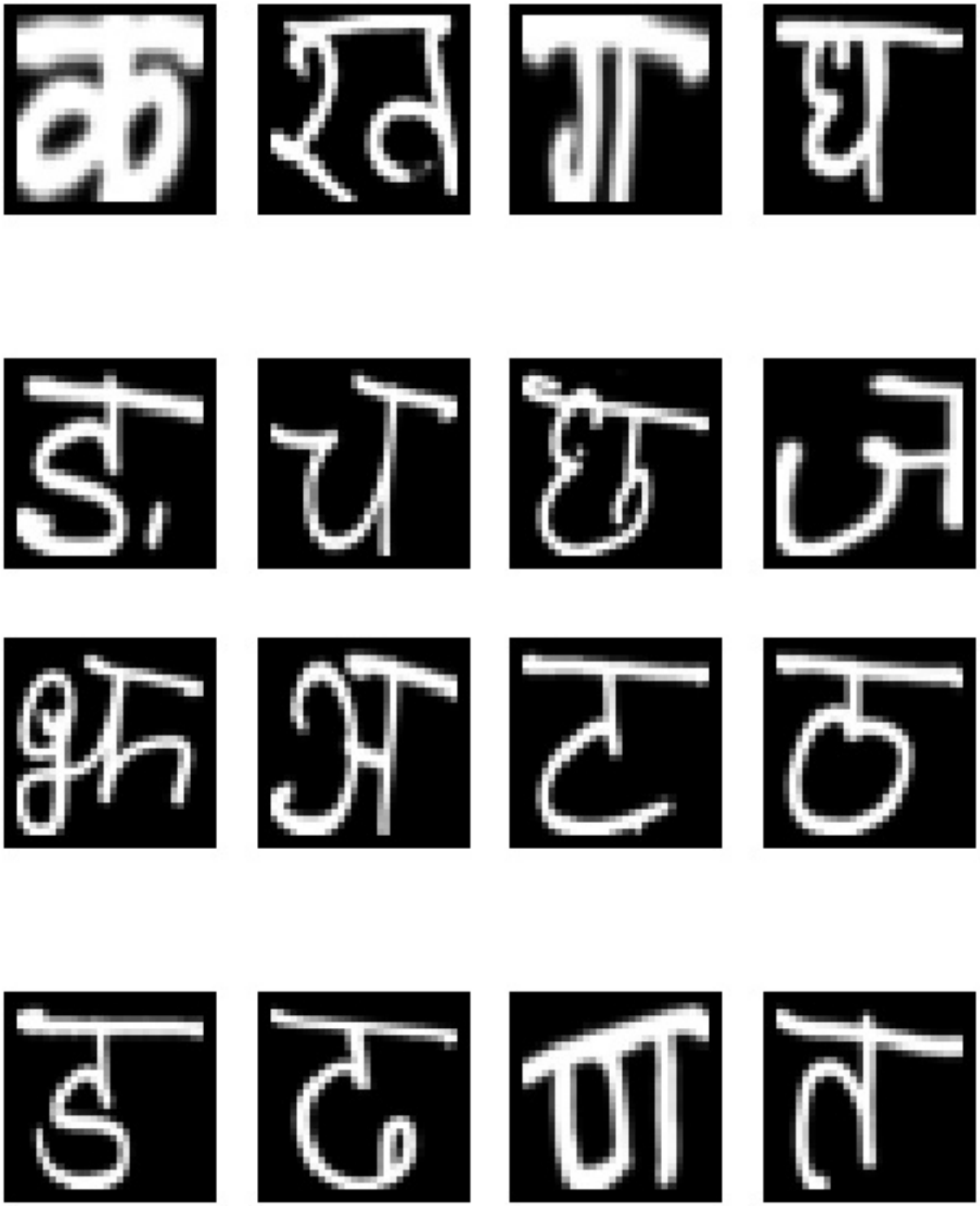
- Every input image provided to the model for training is of resolution 32x32 pixels. Each image is a monochrome bitmap image with 8bits of depth per pixel.
- The entire dataset consists of more than 92000 images
- There are a total of 46 classes that the 92000 images are split into. with 36 consonants, and 10 numerals. The dataset does not consist of any of the Devanagari vowels.
- We receive the data in 2 formats, one being an organized file tree of images and the other being a csv that represents each image in a single row containing 1026 columns. with 1024 columns representing each pixel of the image, and the remaining 2 columns representing the row_id and the label for the specific image.
- We have decided to use the csv data since it eliminates a huge portion of the preprocessing involved, and significantly speeds up the training process since all the data is no longer required to be loaded serially from the file system, thus eliminating delays induced by the repetitive action of reading each file individually from the filesystem at the OS level.

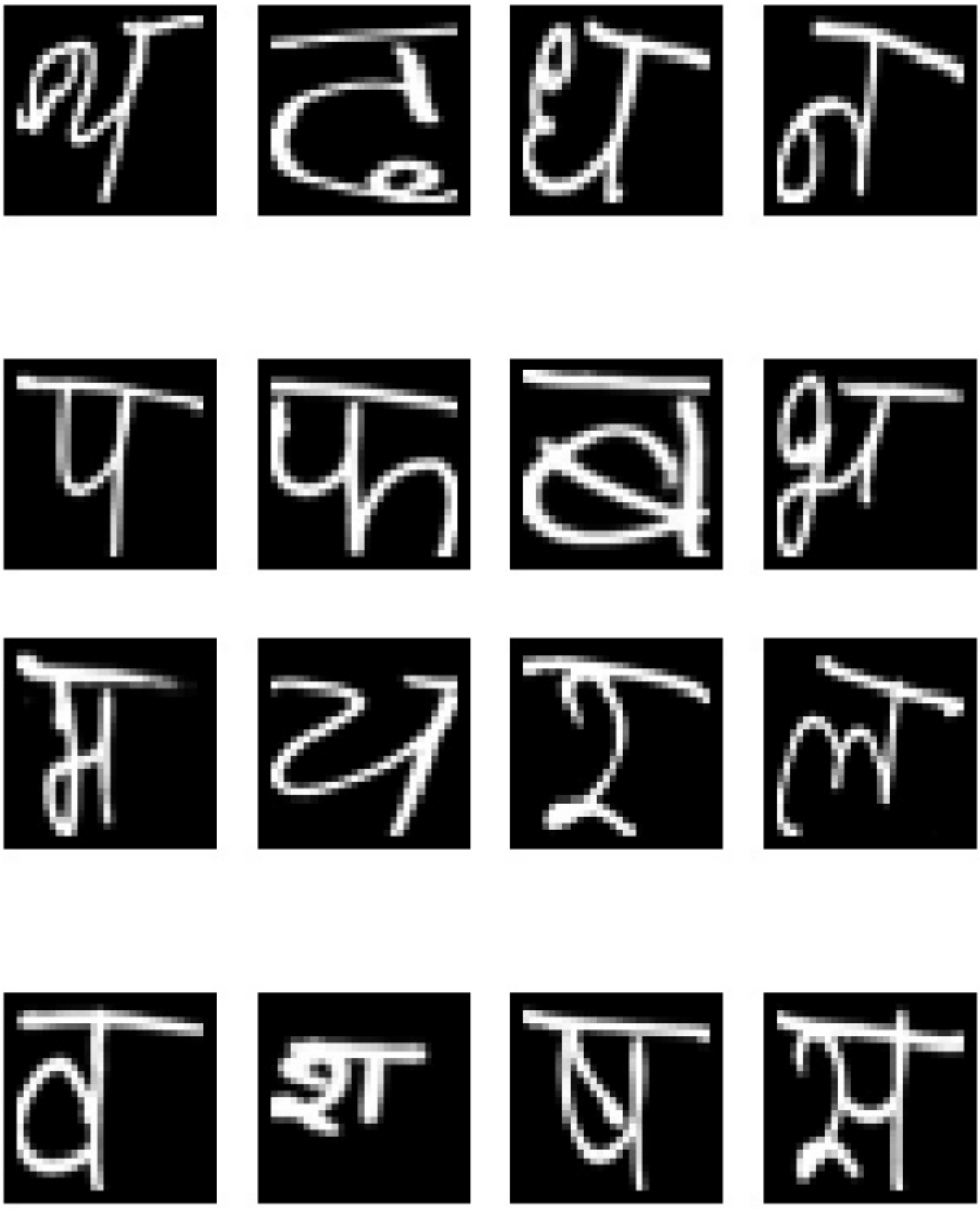
Exploratory Visualization

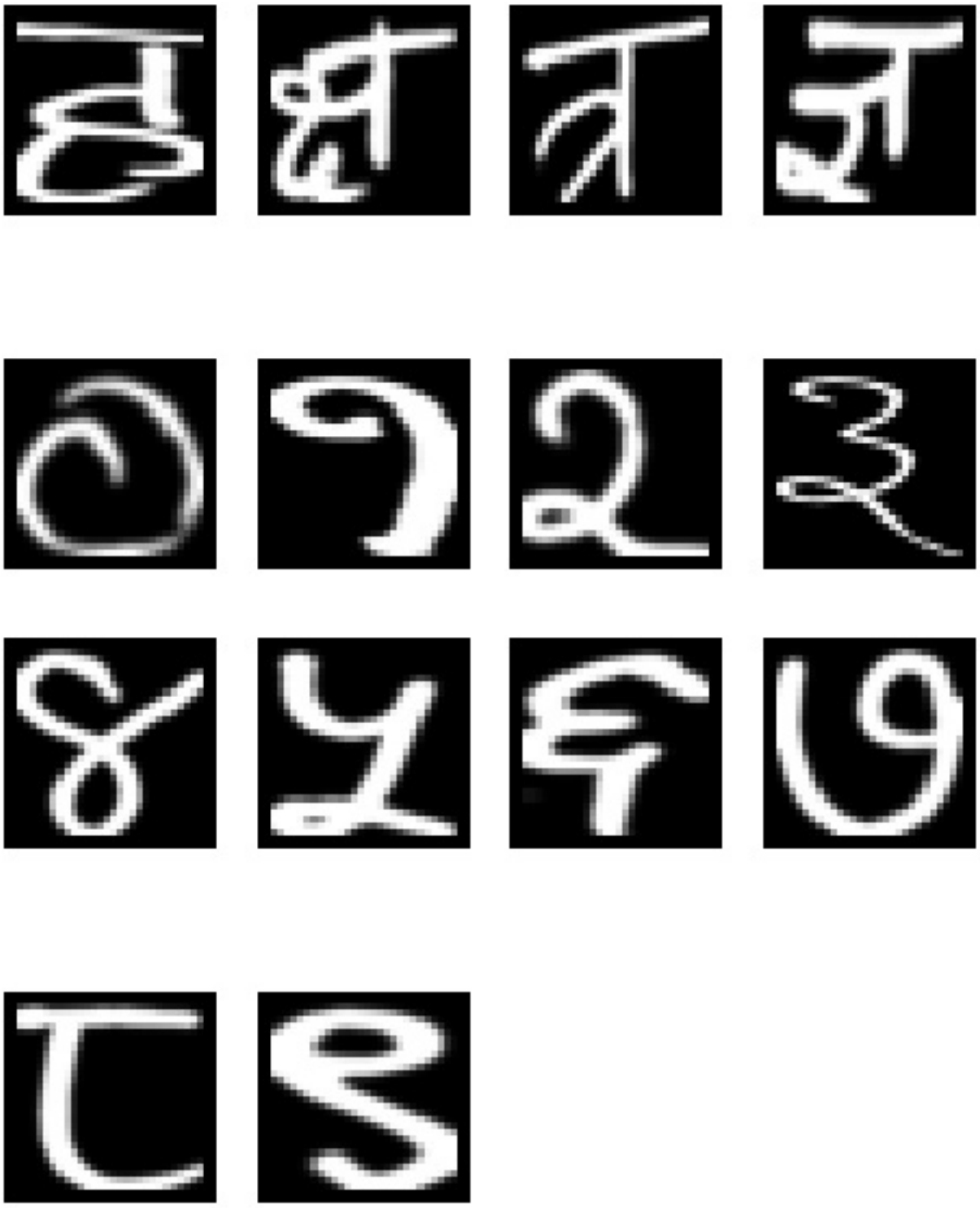


This plot describes the number of images per unique label in our dataset. As we can see from this plot, there are an exact 2000 images per character label.

Here's a visualization of one sample each from every unique label:







The last 10 samples are those of how the numerals are written in the Devanagari script. Corresponding to 0 1 2 3 4 5 6 7 8 and 9 respectively.

Algorithms and Techniques

This project is a classification problem. To solve it, we have chosen to use a convolutional neural network or a CNN. A convolutional Neural network is a state-of-the-art system used to solve data classification problems, along with having other image processing applications. This approach requires a large amount of data, as compared to other approaches, which seeing the size of the

dataset should not be a problem. The training algorithm outputs an assigned probability for each label or class. This is then used to reduce the number of false positives by using a threshold. (The tradeoff here is that this method leads to an increase in the number of false negatives.)

The following parameters can be tuned to optimize the classifier:

- Classification threshold (see above)
- Training parameters
    - Training length (number of epochs)
    - Batch size (how many images to look at, simultaneously during each training step)
    - Type of Solver (what algorithm is to be used for learning)
    - Learning rate (how fast to learn. This can be dynamic)
    - Weight decay (prevents the model being dominated by a select few neurons)
    - Momentum (takes the previous learning step into account when calculating the next one)
- Neural network architecture
    - Number of layers in the Neural network.
    - Layer types (convolutional, fully-connected, or pooling)
    - Layer parameters
- Preprocessing parameters

During training, both the training and the validation sets are loaded into the RAM. Then,, random batches are selected to be loaded into GPU memory for processing. The training is done using the Adam optimization algorithm.

Benchmark

For benchmarking purposes, I used Yann LeCun's Lenet-5 which is an industry standard when it comes to character recognition. Lenet-5 has been shown to achieve an astounding 99.7% accuracy when it comes to training hand written alphanumeric character datasets. Since I will be unable to train my own Lenet-5 equivalent model with the Devanagari dataset, due to the fact that LeNet-5 only achieves that accuracy after nearly 50,000 iterative epochs, I will be assuming comparability between the accuracy obtained on the MNIST dataset, which is available publicly, and that of my model and dataset. I will not be benchmarking any other variables like classification delay or processing delay as they are each dependent on a variety of factors such has server and client hardware and operating system, thus leading to unreliability of measurement and comparison.

III. Methodology

(approx. 3-5 pages)

Data Preprocessing

There are two core components to this project.

- There is the model training component which consists of the dataset, the model training codebase, written in python in a jupyter notebook environment.
- Then there is the user facing application, which takes the weights file output by the jupyter notebook and loads them in a python flask application, which further consists of a server and client application.

- The client application uses javascript libraries like canvas.js and allows a user to hand-draw a Devanagari character with their mouse or trackpad inside a 200x200 pixel window. The client application then converts this image into a data stream and sends it to the server for further processing
- The server application receives this data stream, and converts it back into a 200x200 px RGB .png image. The server then downsamples this image from 200x200 px to 32x32 px using and converts it into a greyscale image. each pixel is then rangemapped from a [0...1] to a [0...255] value per image by multiplying each pixel value with 255. This thereby converts the image into a format that is acceptable to the neural network.
  In terms of the actual training dataset used, we have obtained a csv compiled version of the data from the data provider at kaggle.com. The compiled csv file converts the 32x32 pixel values of each image into a single row of 1024 pixels, with each pixel being represented by a value between 0 and 255. Also, the training process for Lenet-5 requires that the dataset be passed through a rectification process by which all pixels be divided by 255, thereby preserving only those pixels that are 255 valued, and losing those pixels that are anywhere below that number. On testing, the majority of the images in the dataset ended up losing all relavant image features and some images got transformed into those with just a handful of dots. Clearly this would not do, which is why conversion from integer formatted pixels to float formatted pixels becomes necessary.

Implementation

The implementation process can be split into two main stages:

1. The classifier training stage
2. The application development stage

During the first stage, the classifier was trained on the preprocessed training data. This was done in a Jupyter notebook , and can be further divided into the following steps:

- Load both the training and validation images into memory, preprocessing them as described in the previous section
- Define the network architecture and training parameters
- Define the loss function and accuracy
- Train the network, logging the validation/training loss and the validation accuracy
- Plot the logged values
- If the accuracy is not high enough, return to step 3
- Save and pickle the trained network

The application development stage can be split into the following steps:

- Create a simple Flask application that deploys a single page application at the root URL route
- Create a secondary route to /ocr that exposes the neural network trained in the first stage to the frontend application via a simple image predictive API

Refinement

Ive made a few refinements of my own, both when it comes to training the model on the current dataset as well as the user facing application that accepts a user's hand drawn character input.

- For the Model training phase, I was facing unbelievably low accuracy towards freshly hand

drawn character inputs when passing them through the model. This was corrected when I observed that during training, the process of dividing each integer valued pixel by 255 to eliminate all pixel values less than 255 was in fact creating unrecognizable images from the sample inputs. Some did manage to preserve their characteristic features, but most of them simply deteriorated to a handful of dots. This was not the objective of the model training. So regardless of the reason why most other programmers choose to train models this way, I decided to forego this approach and trained the model with scaled pixel data instead, by scaling the pixels from 0:255 to 0:1

- Same was the case for the user facing application. I had to write the application so as to use the exact same keras model I used in my model training phase. I also made sure to take user drawn input at a higher resolution and downsampled the input instead of taking the input directly in the form of a 32x32 pixel greyscale matrix. User drawn accuracy is found to be significantly better after making these refinements, and accuracy on the previously defined validation and testing sets is also found to be closer to the 97% mark than it was found to be before these refinements were put in place.

IV. Results

(approx. 2-3 pages)

Model Evaluation and Validation

During development, a validation set was used to evaluate the model.

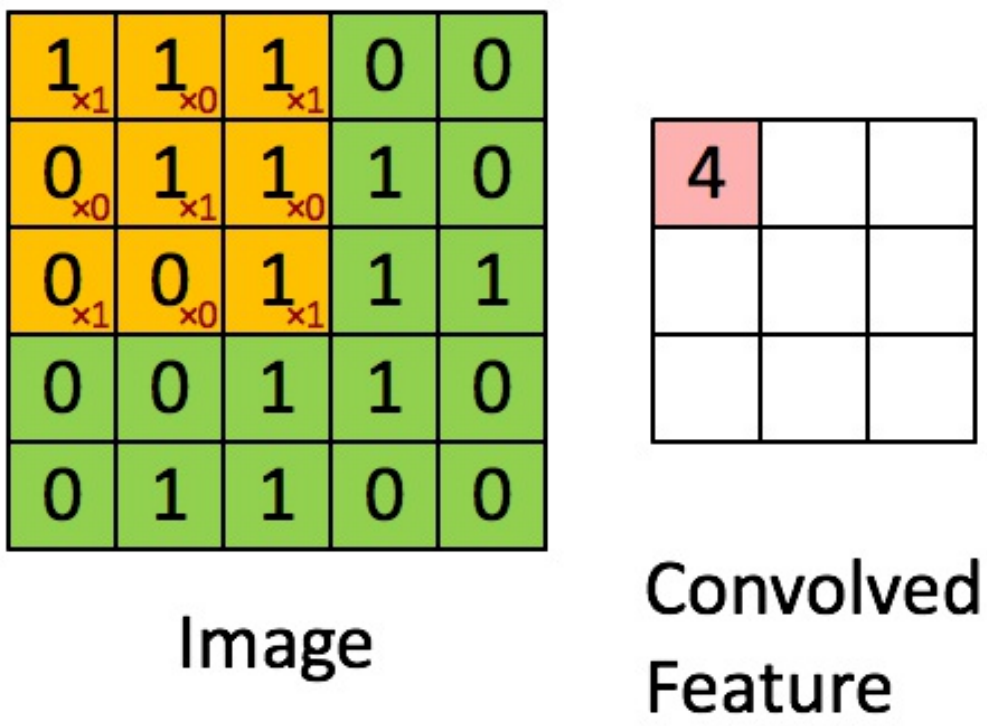
The final architecture and hyperparameters were chosen because they performed the best among the tried combinations.

- The model's summary can be seen here with the explanation further down this page:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 29, 29, 50)        850
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 50)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 11, 11, 100)       80100
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 5, 5, 100)         0
_________________________________________________________________
dropout_1 (Dropout)          (None, 5, 5, 100)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 2500)              0
_________________________________________________________________
dense_1 (Dense)              (None, 200)               500200
_________________________________________________________________
dense_2 (Dense)              (None, 46)                9246
=================================================================
Total params: 590,396
Trainable params: 590,396
Non-trainable params: 0
_________________________________________________________________
```

- The shape of the filters of the convolutional layers is 4*4.

- The first convolutional layer learns 50 filters, the second learns 100 filters.
- The 2 pooling layers sandwiched between the 2 convolutional layers and the final densely connected layer halve the resolution.
- The weights of the convolutional layers are initialized by using random weights.
- The last layer is a fully connected layer and has 200 outputs
- The training runs for 10 epochs
- The original data set is divided into a testing and validation set, where the testing set never influences the training process, and the validation set is used at every iteration as an accuracy metric to converge the model.
- The original dataset reserves 20% of its data for testing purposes, and 15% of the remaining dataset for validation purposes.
- The process by which CNN's are trained is called backpropagation. A model is essentially a mathematical representation of data, with each node referring to a simple function that takes n inputs, places multiplicative weights on each of those inputs, sums the resultant and passes it through the function to produce an output.
- The training phase involves passing the inputs one by one through the network and comparing the received output with that of the expected output. Then the neural network's weights are tweaked by a small factor each, so as to make the model come closer to 100% accuracy. The CNN's weights are tweaked in a reverse fashion, starting with the furthest connected neuron pairs from the input layer, and then the second furthest and so on. This process is executed for each input. Once all inputs have been passed through this algorithm, we can say that the model has been trained for one epoch. Neural networks are usually trained for multiple epochs. If we use a checkpointer, we can guarantee that per epoch, only the models with the best validation accuracy will remain preserved. Thereby minimizing any chances of overfitting to occur.
- The model essentially consists of 3 hidden layers, 1 input layer and 1 output layer. After the first input layer, we have 2 convolution layers with 50 and 100 neurons respectively. The convolution layer is specially designed to process spatial data, and is sensitive to data pixels that are closer to each other spatially.
- Convolutional Neural Nets are especially successful at image classification because Convolution layers can take advantage of individual pixel sensitivity relative to it's position in 2 dimensional space. This has benefits over traditional neural networks that will not be able to make the distinction, and will treat each pixel independently, as if they were a different input altogether. Dark spaces on the corners especially, might not even get considered by the model, since statistically, hand drawn characters rarely stretch all the way to either of the 4 corners, leading to near-complete insensitivity to corner pixel inputs to the neural network, if a homogenous neural network were to be used.
- Convolution in 2 dimensional space may be visualized with the following diagram: available at
  https://imgur.com/jWC74tc.gif

- 

- The outputs of these layers are passed through a Maxpooling function that esentially downsamples the output from the convolution layer by returning the brightest (highest bit valued) pixel from each 2x2 pixel window tweened across the input pixel map. I say tweened because the stride has been set to 1 here. The previous diagram can also be used to explain max pooling, except the output of each window is the value of the brightest pixel.
- The final layer before the output layer is a densely connected layer that consists of 200 neurons. It is provided a flattened single dimensional input from the second layer.
- The final densely connected layer contains exactly as many neurons as there are classes to be detected.
- Using the adam optimizer, the model achieves a testing set accuracy of close to 97%. This model is reasonable for our purposes, considering that this model was trained on a 2012 model 13" intel core i3 dual core MacBook pro with an intel 4000 integrated GPU and 16 GiB of RAM. The training time was close to 17 minutes. Following Pierto's principle, any incremental increases in accuracy will require significantly more training time and GPU intensive capabilities.
- The model training and frontend app development was fairly straightforward, seeing that the data was clean and free of any significant outliers. I guess some input images did seem as if they were drawn with an excessively thick paintbrush, but overall there was nothing that could harm the performance of the model on the final output. As far as reproducibility is concerned, I had faced somewhat of a challenge trying to convert an image to greyscale and then into the required format that the neural network needed for the frontend part of this project, and also faced a similar problem regarding how dimly lit

each pixel in the image was, when converting a canvas object to a color png image and then to a pixel array, and scaling the brightest pixel to 1, with the other pixels accordingly.

- With a 97% validation accuracy score, I would say that we can come to trust this model, but a lot more iterations of training will be required before we can say that this model is significantly robust enough. But I would also include that a more nuanced technique of splitting the training, testing and validation sets could be used to significantly improve the model's ability to generalize. If the data contained within the training, testing, and validation datasets each contained data pointing to an equal distribution of labels, we would be able to better train the model, and given enough iterations, come up with a better validation accuracy.

- Cloud training is an option here, but simply observing how well this model trains on a consumer grade laptop from 5 years ago should also be considered when it comes to the robustness of this model. However, a 99% accuracy is possibly implementable with more training iterations.

- With a 97% validation accuracy score, I would say that we can come to trust this model, but a lot more iterations of training will be required before we can say that this model is significantly robust enough. But I would also include that a more nuanced technique of splitting the training, testing and validation sets could be used to significantly improve the model's ability to generalize. If the data contained within the training, testing, and validation datasets each contained data pointing to an equal distribution of labels, we would be able to better train the model, and given enough iterations, come up with a better validation accuracy.

- One can get a deeper understanding of the model's accuracy towards unseen data by testing it against freshly hand drawn models by running the user facing app contained in the second directory. It is for this very purpose that the frontend application was implemented in the first place. For a majority of the tests that I had run by hand, I had found the model to perform significantly well. The training data used didnt seem to contain any outliers as such, like unlabelled data or deliberately misslabelled data, as the quality of the dataset has been verified by the data provider and other independent entities providing quality metrics for the same. The matter of sensitivity is also seemingly not a concern, as the model would not be able to train to anywhere near such a validation accuracy that we have observed, due to the learning rate being just right. However, as is the case here, a larger training period, combined with a more even data splitting may result in the model being able to generalize significantly better.

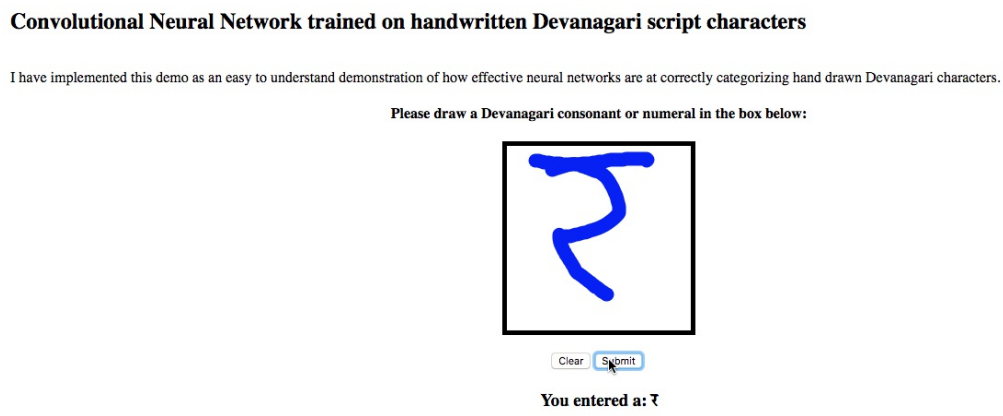
- It should also be noted that we are using a checkpointer to train the model, and preserving only the model weights with the best validation accuracy score. Since we are certain that the we are only picking the models that are performing the best on unseen data, a strong case for the model not overfitting the training data can be created.

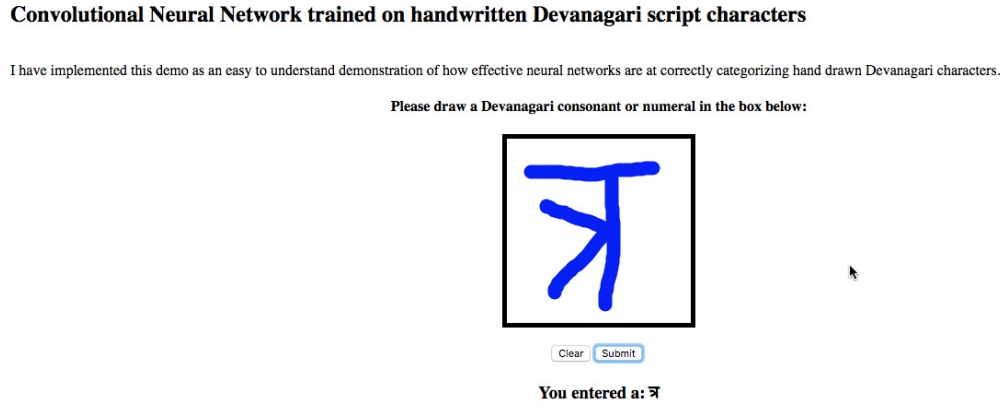
Justification

- The benchmark that I have chosen for this project, is arguably the best trained model that exists publically, and has been tested on a similar dataset of hand written english numeric characters by Yann LeCun. The model is reportedly 99.7% accurate and expectedly so. This current project has only been inspired by Yann LeCun's Lenet-5 and does attempt to get an accuracy as close as possible to the benchmark model. But seeing that I do not have the industrial grade GPUs and TPUs at my disposal, my laptop grade integrated GPU

resulted in a testing and validation accuracy of a little over 97%. Which I believe is satisfactory.

- Neural network training in keras allows for a very convenient methodology which enables reloading of weights into a neural network architecture after pickling them in such a case where the number of training iterations needs to be dynamic, or when a model needs to be trained to a certain percentage accuracy, regardless of the number of training iterations. This way, one can design a neural network where training can take place dynamically, thereby resuming, and therefore enabling building upon model training from a certain point instead of having to restart the training from scratch.
- For the purposes of recognizing freshly hand drawn characters from scratch, I believe that the solution offers significant enough accuracy, as shown here:

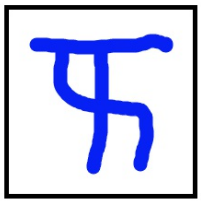




Convolutional Neural Network trained on handwritten Devanagari script characters

I have implemented this demo as an easy to understand demonstration of how effective neural networks are at correctly categorizing hand drawn Devanagari characters.

Please draw a Devanagari consonant or numeral in the box below:



Clear Submit

You entered a: फ

Convolutional Neural Network trained on handwritten Devanagari script characters

I have implemented this demo as an easy to understand demonstration of how effective neural networks are at correctly categorizing hand drawn Devanagari characters.

Please draw a Devanagari consonant or numeral in the box below:

Clear Submit

You entered a: क

Convolutional Neural Network trained on handwritten Devanagari script characters

I have implemented this demo as an easy to understand demonstration of how effective neural networks are at correctly categorizing hand drawn Devanagari characters.

Please draw a Devanagari consonant or numeral in the box below:

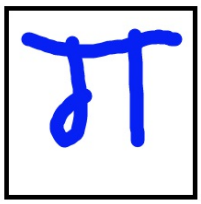
Clear Submit

You entered a: ल

Convolutional Neural Network trained on handwritten Devanagari script characters

I have implemented this demo as an easy to understand demonstration of how effective neural networks are at correctly categorizing hand drawn Devanagari characters.

Please draw a Devanagari consonant or numeral in the box below:



Clear Submit

You entered a: ग

Convolutional Neural Network trained on handwritten Devanagari script characters

I have implemented this demo as an easy to understand demonstration of how effective neural networks are at correctly categorizing hand drawn Devanagari characters.

Please draw a Devanagari consonant or numeral in the box below:

Clear Submit

You entered a: घ

Convolutional Neural Network trained on handwritten Devanagari script characters

I have implemented this demo as an easy to understand demonstration of how effective neural networks are at correctly categorizing hand drawn Devanagari characters.

Please draw a Devanagari consonant or numeral in the box below:

Clear Submit

You entered a: म

Convolutional Neural Network trained on handwritten Devanagari script characters

I have implemented this demo as an easy to understand demonstration of how effective neural networks are at correctly categorizing hand drawn Devanagari characters.

Please draw a Devanagari consonant or numeral in the box below:

Clear Submit

You entered a: स

Convolutional Neural Network trained on handwritten Devanagari script characters

I have implemented this demo as an easy to understand demonstration of how effective neural networks are at correctly categorizing hand drawn Devanagari characters.

Please draw a Devanagari consonant or numeral in the box below:

Clear Submit

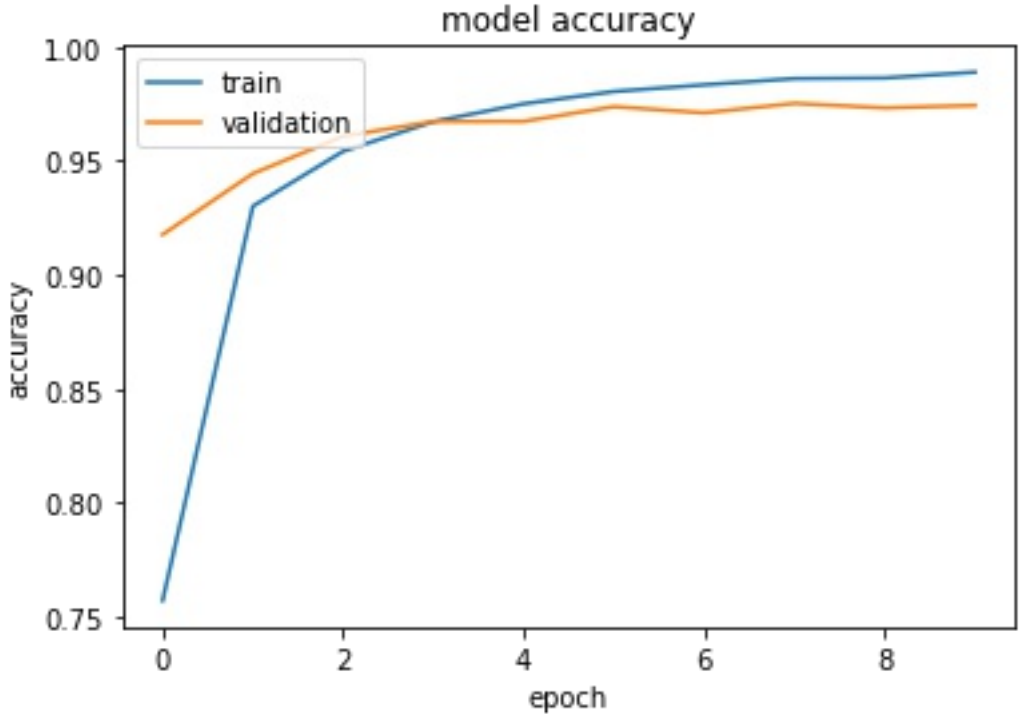
You entered a: ख

V. Conclusion

(approx. 1-2 pages)

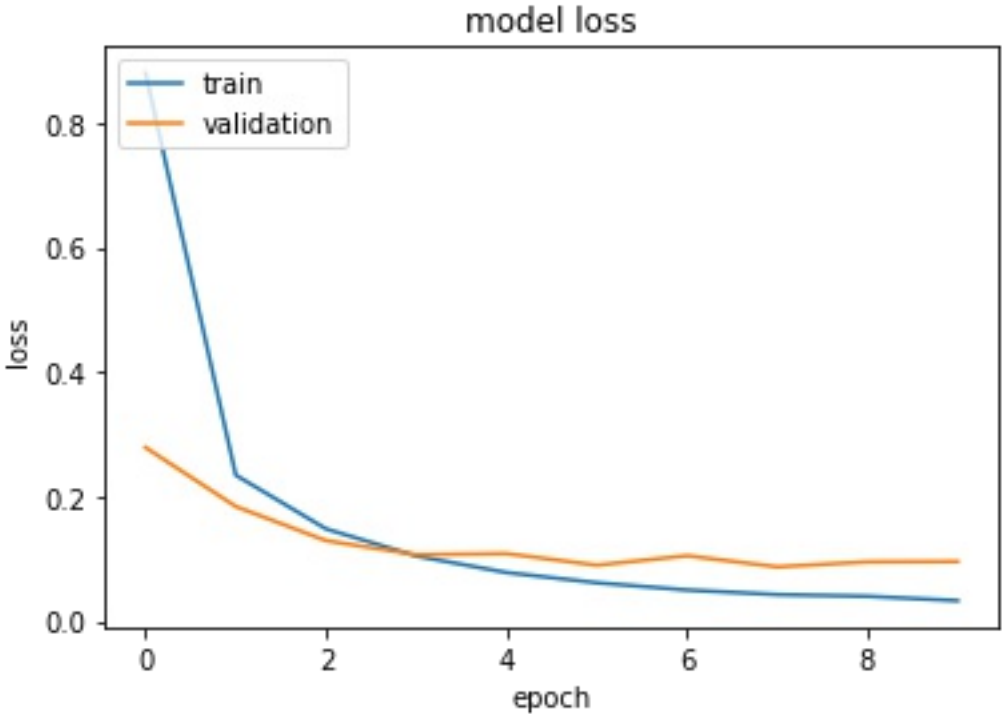
Free-Form Visualization

Here in this following graph, the iterative process of plotting the training and testing accuracy has been represented:



After 10 iterations, the model seems to converge at close to 97% accuracy.

In comparison, the training and validation loss graphs have been represented as follows:



On observation, we see that the model converges at somewhere near 97% validation accuracy, and at this point, it starts requiring greater and greater numbers of iterations to get closer to the 100% accuracy mark, although overfitting does become a concern at this point, since further training does, infact drive training accuracy to very close to 100%, but keeps validation accuracy stable at the aforementioned level.

Reflection

In summation, We have successfully completed a two part project.

- The first part being that of architecting, training, validating and testing a Convolutional Neural network, on a training set that consists of 92000 images of hand drawn consonants and numerals from the Devanagari script.
- The second part being that of creating a basic GUI web browser based application, also written in python using the Flask web framework, so as to make it compatible with the training portion of project that has been done, such that it effectively demonstrates the capabilities of the model to an outsider with no advance understanding of the technical aspects of this project.

The most challenging part of the project arguably, was determining the root cause of why freshly hand drawn symbols were facing low accuracy predictions in comparison with the hand drawn symbols of the testing set, even though they were relatively similar in handwriting and style. The root cause resided in the conversion technique utilized when converting a canvas drawing into an image and then again into a pixel bitmap array. The pixels were not being encoded brightly enough, and had to be scaled up, with the brightest pixel scaled up to 1, and the other pixels scaled accordingly.

Improvement

I didn't include any non standard capabilities to the model, although I do believe there could have been a variety of ways by which I could have augmented it. These methods are listed as follows:

- Translation. As with each image, a translation of pixels up, down, left or right could have drastic effects on the accuracy of the classification. Improvements could be made during preprocessing to translate each image up, down, left or right by means of a certain heuristic, having specific boundary conditions, and thereby generating multiple new images out of a single image so as to train the model better, by training on a larger spectrum of images.
- Transformation. Each image occupies a certain size on the pixel array a transformation of the pixels' size in any direction could have drastic effects on the accuracy of the classification. Improvements could be made during preprocessing to transform each image, making it bigger or smaller by means of a certain heuristic, having specific boundary conditions, and thereby generating multiple new images out of a single image so as to train the model better, by training on a larger spectrum of images.
- Dynamic training. By rewriting the training code block to keep training the model until it reaches a certain training accuracy, instead of training a certain number of iterations, one can create a model that is dynamically trainable. Some implementation of earlystopping would be required here, while training with an arbitrarily high number of epochs.

I am confident that a better solution exists in comparison with this model, especially one that can train a model on a laptop grade computer in a reasonable amount of time. As we discover these, greater possibilities will begin to emerge.
