CAPSTONE PROJECT REPORT

By: Rahul Passi

February 11th, 2018

Machine Learning Advanced Nanodegree

Devanagri character (HINDI) Recognition using Deep Learning techniques

1. Defintion-

Project overview

-Character Recognition

Character recognition is the ability of a computer to interpret intelligible characters from sources such as paper documents, photographs, touch-screens and other devices. The image of the text may be sensed "off line" from a piece of paper by optical scanning (optical character recognition) or intelligent word recognition.

Character Recognition plays an important role in storage and retrieval of typed and handwritten information. A wealth of historical papers and archives exist in a physical format. This includes genealogical information, old research papers ,letters to and from important people in history, old family records, written manuscripts, personal diaries and many other important historical data. Consistent review of this information damages the original paper and can lead to physical data corruption.

Character recognition allows people to translate this information into easily readable electronic formats .Modern technology has allowed us to recognize characters from images. The new prediction algorithms of machine learning and artificial intelligence has allowed us useful and trustable methods of character recognition .This has extended from predicting words to predicting entire sentences, all the way to predicting entire

letters. This also has allowed new breakthroughs in language translation. Now, predicting characters from different languages is getting easier day by day with advancement in machine learning field and predictive algorithms.

In this project, I will be working on the field of character recognition to predict 'Devanagri characters' (Hindi) from images. This would be done with the help of image classification techniques of Deep learning (Convolutional Neural Network) .I used the dataset available publicly on kaggle with link provided below-

https://www.kaggle.com/rishianand/devanagari-character-set/version/3/data

History-

The term *Deep Learning* was introduced to the machine learning community by Rina Detcher in 1986, and to Artificial neural Networks by Igor Aizenberg and colleagues in 2000, in the context of Boolean threshold neurons. through neural networks for *reinforcement learning*. In 2006, a publication by Geoff Hinto, Osindero and Teh showed how a many-layered feedforward neural network could be effectively pre-trained one layer at a time, treating each layer in turn as an unsupervised restricted Boltzmann machine, then fine-tuning it using supervised backpropagation. The paper referred to *learning* for *deep belief nets*.

The first general, working learning algorithm for supervised, deep, feedforward, multilayer perceptrons was published by Alexey Ivakhnenko and Lapa in 1965. A 1971 paper described a deep network with 8 layers trained by the group method of data handling algorithm.

Problem Statement

The goal is to predict the ' Devanagri characters' (Hindi) in an image. There are various models for classification of characters of English and other languages but there are comparatively less for classification of Hindi characters. This is basically an image classification problem which would be solved using deep learning.

Why deep learning is used here->

Deep Learning is a subset of Machine Learning Algorithms that is very good at recognizing patterns but typically requires a large number of data.

Deep learning excels in recognizing objects in images as it's implemented using 3 or more layers of artificial neural networks where each layer is responsible for extracting one or more feature of the image.

The model should be able to classify which Devanagri character is in the image, with high accuracy.

Evaluation Metrics

The trained CNN model will be tested for accuracy. This accuracy will be compared with the benchmark models.

Accuracy is measured as ratio between correctly predicted dataset and and tested dataset. Higher the accuracy, better the model.

There are around 2000 images in each of the 46 subdirectories. As the frequency of characters is same in each of the directories (classes are balanced), Accuracy would be a very good evaluation metric here. Also I avoided using loss as a metric because there is less difference (sometimes negligible) in loss between successive epochs .

2. Analysis-

Data Exploration

The dataset of characters is taken from Kaggle . It comprises of about 92000 grayscale images in a total of 46 sub-directories corresponding to 46 characters including digits and consonants but excluding vowels. Each image is of 32x32, making a total of 1024 pixels.

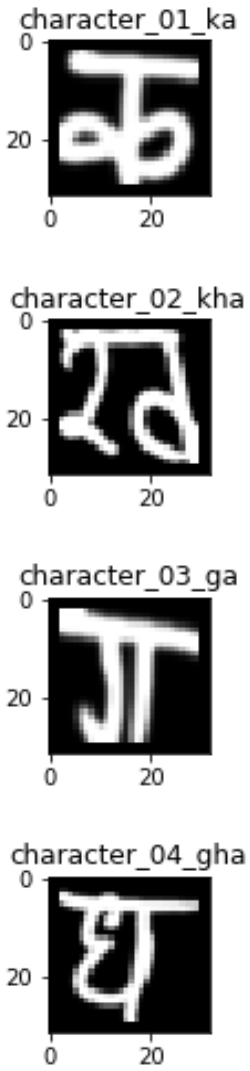


Figure 1 (sample dataset)

Also a csv file containing pixel information of the characters it is of dimension 92000*1025. Each of the 1024 pixels of the grayscale images are given pixel values between 0 to 255. The pixel value tell the brightness level of the pixel. The column "character" represents the Devanagri Character Name corresponding to each image.

Images would be divided on an **80/20 split** between training data and testing data respectively using **train_test_split imported from sklearn.model_selection.**

One unusual thing about the dataset is that generally characters are drawn black on white background in a grayscale image but here opposite is happening .

#Acknowledgement- This dataset was originally created by Computer Vision Research Group, Nepal.

Exploratory Visualization

I plotted the 32x32 images (using matplotlib library) from the dataset to present the data, some of which are seen below-

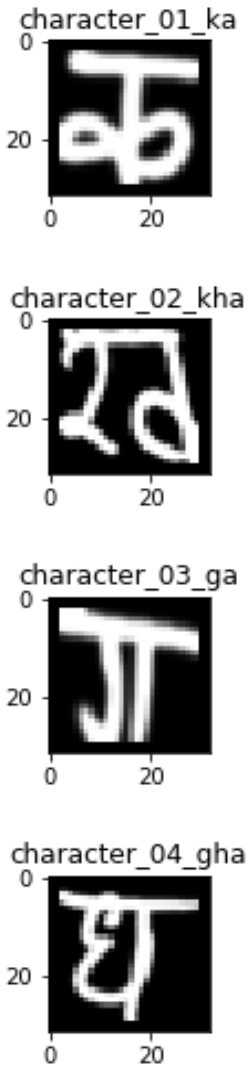


Figure 1 (sample dataset)

Also I calculated the frequency of each character in the dataset, which can be seen below (first 5 characters are taken as sample)-

```
In [1]: import os
list=os.listdir('Images/')
print len(list)
list1=os.listdir('Images/character_01_ka')
print len(list1)
list2=os.listdir('Images/character_02_kha')
print len(list2)
list3=os.listdir('Images/character_03_ga')
print len(list3)
list4=os.listdir('Images/character_04_gha')
print len(list4)
list5=os.listdir('Images/character_05_kna')
print len(list5)
```

```
46
2000
2000
2000
2000
2000
```

We can observe from above there are 46 types of characters and each character is having 2000 images.

Therefore, our dataset of 92000 images is evenly divided into 46 subdirectories (types of characters) having 2000 images each or we can say that classes are balanced.

Algorithm and techniques

The algorithm chosen for this project is Deep learning using Convolutional Neural Network (CNN).

Advantages

- Has best-in-class performance on problems that significantly outperforms other solutions in multiple domains. This includes speech, language, vision, playing games like Go etc. This isn't by a little bit, but by a significant amount.

- Reduces the need for feature engineering, one of the most time-consuming parts of machine learning practice.
- Is an architecture that can be adapted to new problems relatively easily. (e.g. Vision, time series, language etc.)

Disadvantages

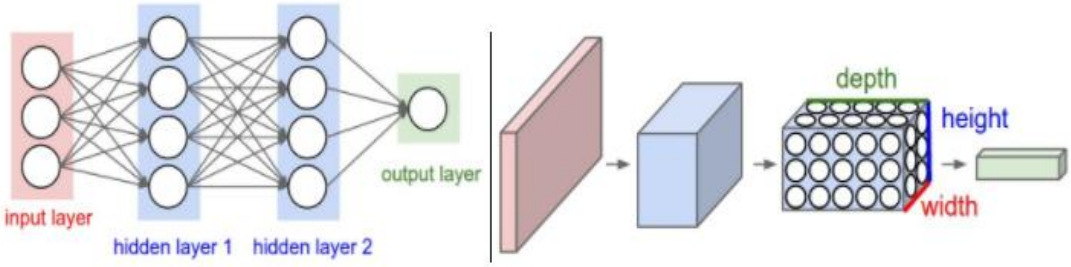
- Requires a large amount of data — if you only have thousands of example, deep learning is unlikely to outperform other approaches.
- Is extremely computationally expensive to train. The most complex models take weeks to train using hundreds of machines equipped with expensive GPUs.
- Determining the topology/flavor/training method/hyperparameters for deep learning is a black art with no theory to guide you.

This technique is chosen because of it's very high accuracy compared to others.

A typical neural network has anything from a few dozen to hundreds, thousands, or even millions of artificial neurons called **units** arranged in a series of layers, each of which connects to the layers on either side. Some of them, known as **input units**, are designed to receive various forms of information from the outside world that the network will attempt to learn about, recognize, or otherwise process. Other units sit on the opposite side of the network and signal how it responds to the information it's learned; those are known as **output units**. In between the input units and output units are one or more layers of **hidden units**, which, together, form the majority of the artificial brain. Most neural networks are **fully connected**, which means each hidden unit and each output unit is connected to every unit in the layers either side. The connections between one unit and another are represented by a number called a **weight**, which can be either positive (if one unit excites another) or negative (if one unit suppresses or inhibits another). The higher the weight, the more influence one unit has on another. (This corresponds to the way actual brain cells trigger one another across tiny gaps called synapses.)

Convolutional Neural Networks (ConvNet) take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way. In particular, unlike a regular Neural Network, the layers of a ConvNet have neurons arranged in 3 dimensions: **width, height, depth**. (Note that the word *depth* here refers to the third dimension of an activation volume, not to the depth of a full Neural Network, which can refer to the total number of layers in a network.) For example, the

input images in CIFAR-10 are an input volume of activations, and the volume has dimensions 32x32x3 (width, height, depth respectively). As we will soon see, the neurons in a layer will only be connected to a small region of the layer before it, instead of all of the neurons in a fully-connected manner. Moreover, the final output layer would for CIFAR-10 have dimensions 1x1x10, because by the end of the ConvNet architecture we will reduce the full image into a single vector of class scores, arranged along the depth dimension. Here is a visualization:



Left: A regular 3-layer Neural Network. Right: A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).

Below are the common types of layers to build ConvNet architectures:-

Conv2D: This layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs.

The CONV layer's parameters consist of a set of learnable filters. Every filter is small spatially (along width and height), but extends through the full depth of the input volume. For example, a typical filter on a first layer of a ConvNet might have size 5x5x3 (i.e. 5 pixels width and height, and 3 because images have depth 3, the color channels). During the forward pass, we slide (more precisely, convolve) each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. As we slide the filter over the width and height of the input volume we will produce a 2-dimensional activation map that gives the responses of that filter at every spatial position. Intuitively, the network will learn filters that activate when they see some type of visual feature such as an edge of some orientation or a blotch of some color on the first layer, or eventually entire honeycomb or wheel-like patterns on higher layers of the network. Now, we will have an entire set of filters in each CONV layer (e.g. 12 filters), and each of them will produce a separate 2-dimensional activation map. We will stack these activation maps along the depth dimension and produce the output volume.

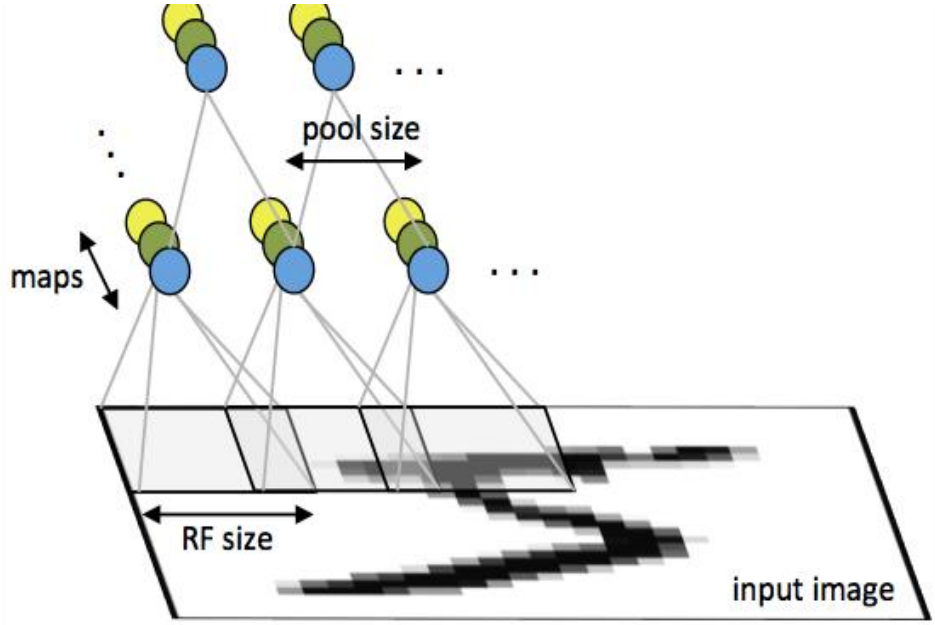


Fig 1: First layer of a convolutional neural network with pooling. Units of the same color have tied weights and units of different color represent different filter maps.

We added relu as non- linear activation function in each of the Convolutional layer.

MaxPooling2D: The objective of max pooling layer is to down-sample an input representation (image, hidden-layer output matrix, etc.), reducing its dimensionality and allowing for assumptions to be made about features contained in the sub-regions binned.

Flatten layer: The Flatten layer is a utility layer that flattens an input of shape (a , b , c) to a simple vector output of shape a*b*c .

Dense: In a dense layer, every node in the layer is connected to every node in the next layer.

Dropout layer was introduced as a regularization technique for reducing overfitting . Certain neurons in the layer will be deactivated .

Benchmark Model

The benchmark model is chosen from one of the kernels of the Kaggle Devanagri character recognition .

The benchmark models have an accuracy around and above 90% .Aim would be to match it and get a score of over 90% accuracy, which would be considered as a success .

3. Methodology-

Data preprocessing

CSV file containing the pixel information of all the images is loaded into memory using pandas.

In order to reduce the computational load, images are normalized which will reduce values from 0 - 255 to 0 - 1 by dividing each pixel values by 255.

The character labels were then converted to a one-hot encoding format using sklearn's label binarizer for classification purposes. For example ,we converted the first category of characters to the vector [1 0 0 0 ……0] and so on . The length of vectors will be 46 as there are a total of 46 types of characters .

Implementation

1. **Import Datasets**
   We used pandas library to import the csv file containing the pixel information of images . We used matplotlib library to plot images to observe if our datasets imported correctly.

2. **Data Pre-processing**

We discussed it in the data preprocessing section.

3. **Build the model**

I have used the accuracy metric as explained in the Metric section, for optimizer I have used adam optimizer. The model runs with 10 epochs and 400 batch size .

The CNN model is developed using the layers discussed in algorithms and techniques section-

```
In [80]: model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
c_1 (Conv2D)                 (None, 29, 29, 32)        544
_________________________________________________________________
p_1 (MaxPooling2D)           (None, 14, 14, 32)        0
_________________________________________________________________
c_2 (Conv2D)                 (None, 12, 12, 64)        18496
_________________________________________________________________
p_2 (MaxPooling2D)           (None, 6, 6, 64)          0
_________________________________________________________________
flatten_11 (Flatten)         (None, 2304)              0
_________________________________________________________________
d_1 (Dense)                  (None, 128)               295040
_________________________________________________________________
modeloutput (Dense)          (None, 46)                5934
=================================================================
Total params: 320,014
Trainable params: 320,014
Non-trainable params: 0
_________________________________________________________________
```

Accuracy reported by the initial model is 90.15% which is actually pretty good but can be certainly improved .

```
In [68]: scores = model.evaluate(X_test, y_test, verbose=2)
         print("Accuracy: %.2f%%" % (scores[1]*100))
```

```
Accuracy: 90.15%
```

Environment setting-

Hyper-parameters used –>

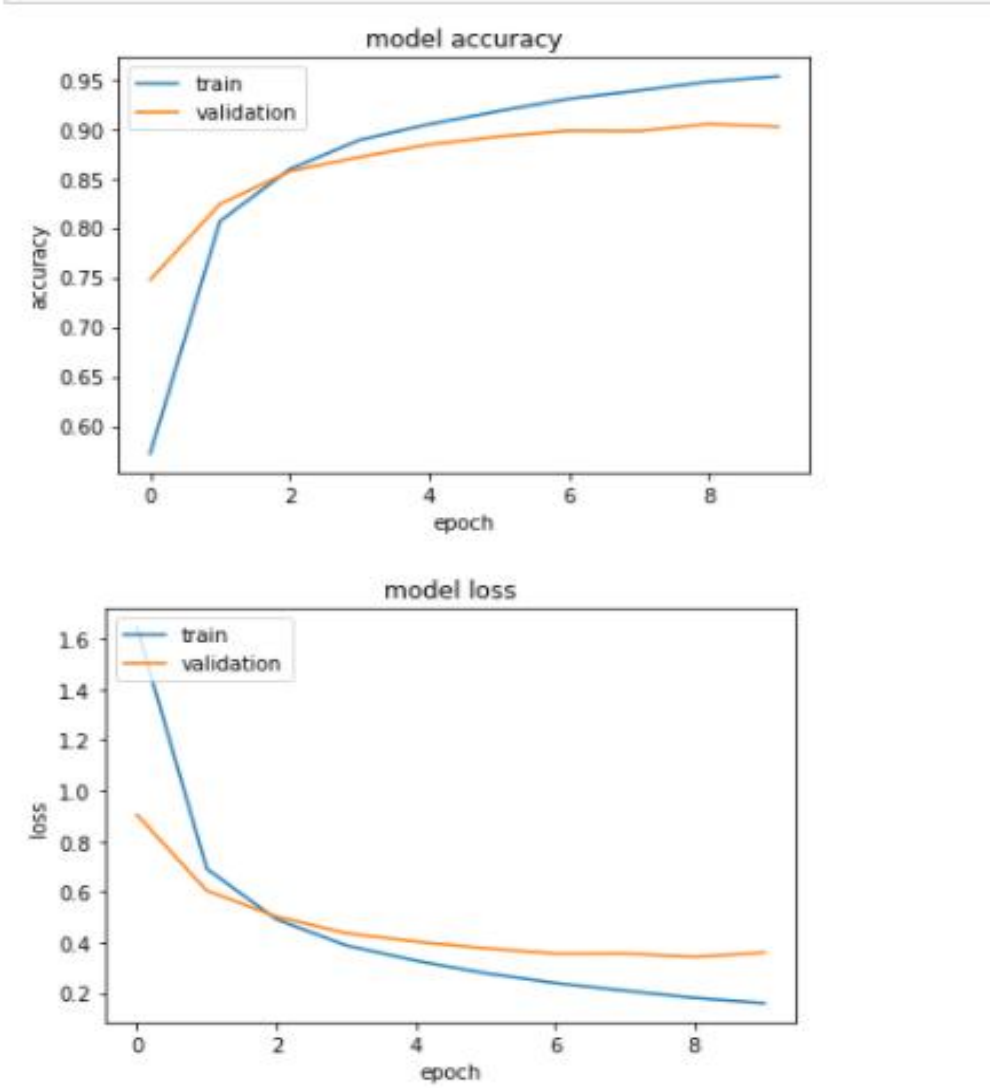
Epochs -10

Batch Size-400

Optimizer used was adam's optimizer which I found to be the most effective as using it resulted in higher accuracy.

As model testing takes a lot of time (atleast 10 mins) for every run, tampering with environment setting is difficult as increase in epochs and batch size would make model testing even more consuming and sometimes inefficient.

Plotting accuracy and loss graphs-



Refinement

The initial model is improved by using the following technique-
-Two dropout layers (dropout rate of 0.20) are added to ensure that overfitting does not happen
(certain neurons here will be deactivated , not only it improves accuracy but it also

lowers the training time . Hence, total efficiency is improved as both accuracy and time taken is improved.

Following is the summary of improved model –

```
: model2.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
c_1 (Conv2D)                 (None, 29, 29, 32)        544
_________________________________________________________________
p_1 (MaxPooling2D)           (None, 14, 14, 32)        0
_________________________________________________________________
c_2 (Conv2D)                 (None, 12, 12, 64)        18496
_________________________________________________________________
p_2 (MaxPooling2D)           (None, 6, 6, 64)          0
_________________________________________________________________
dropout_11 (Dropout)         (None, 6, 6, 64)          0
_________________________________________________________________
flatten_13 (Flatten)         (None, 2304)              0
_________________________________________________________________
d_1 (Dense)                  (None, 128)               295040
_________________________________________________________________
dropout_12 (Dropout)         (None, 128)               0
_________________________________________________________________
modeloutput (Dense)          (None, 46)                5934
=================================================================
Total params: 320,014
Trainable params: 320,014
Non-trainable params: 0
```

Accuracy reported here is 91.80% which is clearly better than the initial model which had accuracy of 90.15% .

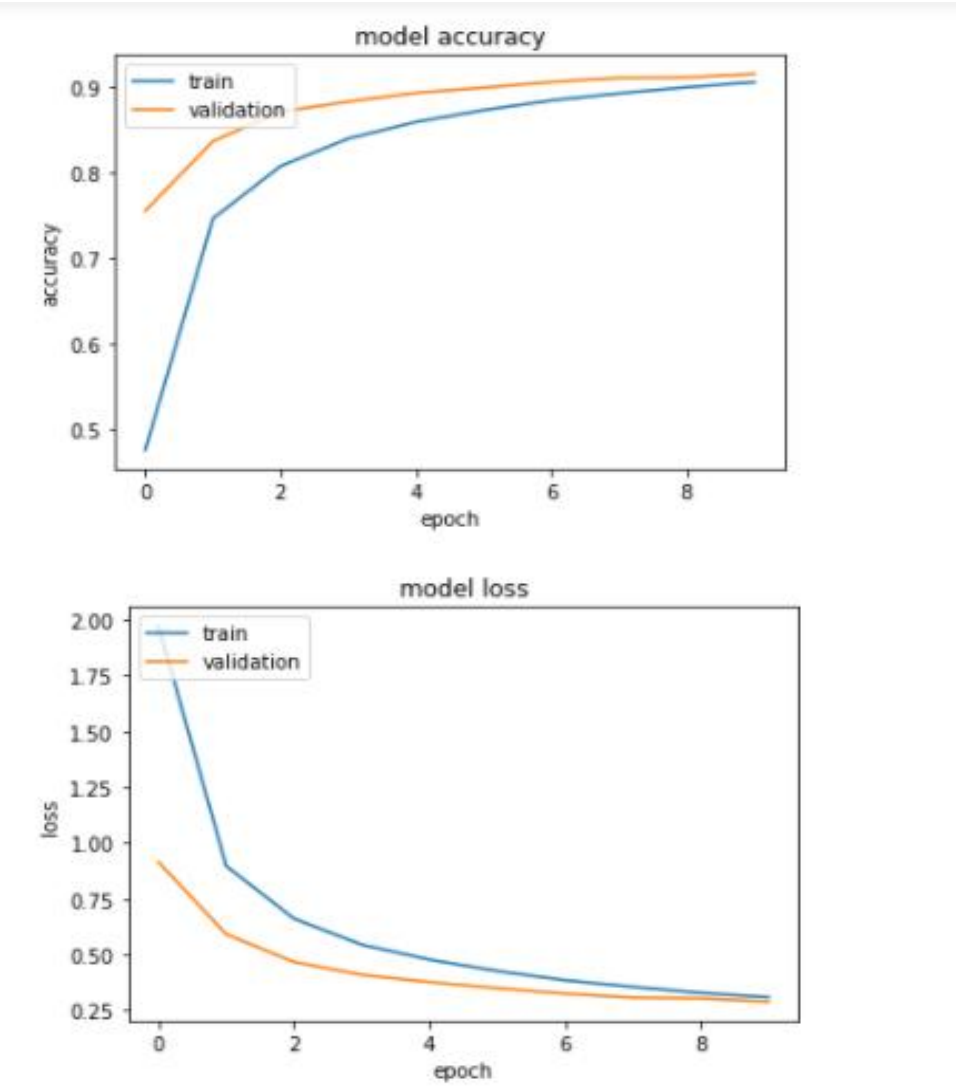
```
In [76]: scores = model2.evaluate(X_test, y_test, verbose=2)
         print("Accuracy: %.2f%%" % (scores[1]*100))

         Accuracy: 91.80%
```

Environment settings (Hyper-parameters and optimizer) are same as that of initial model.

The only difference in this architecture was that two dropout layers (dropout rate 0.20) were added which deactivated 20% of the neurons each to prevent overfitting .

Also, our goal of exceeding 90% accuracy according to benchmarks has been succeeded.

Plotting accuracy and loss graphs-



4. Results-

Model evaluation and Validation

The final model has been clearly described above with model summary, environment settings and with loss v epochs and accuracy v epochs plotted .

The final architecture was chosen because it performed better than the other(initial) model that I tried, by improving its accuracy and decreasing its time ,which it does by adding 2 dropout layers of 0.20 drop rate.

An important advantage the final architecture has that it can work very well for new unseen datasets because of the dropout layers which are improving its learning. This makes the model very robust for testing different kinds of new data.

The robustness of the final model is verified by conducting a test against the final model using different images of characters other than our main dataset (80/20 split mentioned in data exploration section) .

The final model takes pixel information of images as input from dataset ,exact inputs taken are (Sample(could be None), Height, Width, Channels) and after data preprocessing ,splitting of testing and training data is done, then training and testing happens.

Here the final model is built which includes all the layers described in algorithms and techniques section and then after setting the Hyper-parameters is first trained on 80% of data than tested on 20% of the rest unseen data. This architecture and steps could be followed to develop a similar model to train and test on other problems related to character recognition and with minor adjustments can solve those problems.

The following observations are based on the results of the test:
The observations of test confirms that our model is reliable and robust enough to perform well on real unseen data. Use of the 2 Dropout layers in the final model ensures that small perturbations (changes) in the training data or the input space do not affect the results greatly .

Justification

The accuracy of the final model is 91.80% which matches our goal of getting above 90% accuracy and matching the various kernels (our benchmarks) of kaggle.
Our model has perfomed very well matching the benchmark model.
Thus, our model solved the problem with a good accuracy score and reliability in real time applications and I think our solution is good enough and could be used as a general setting to solve these kind of problems .

5. Conclusion-

Free form visualization

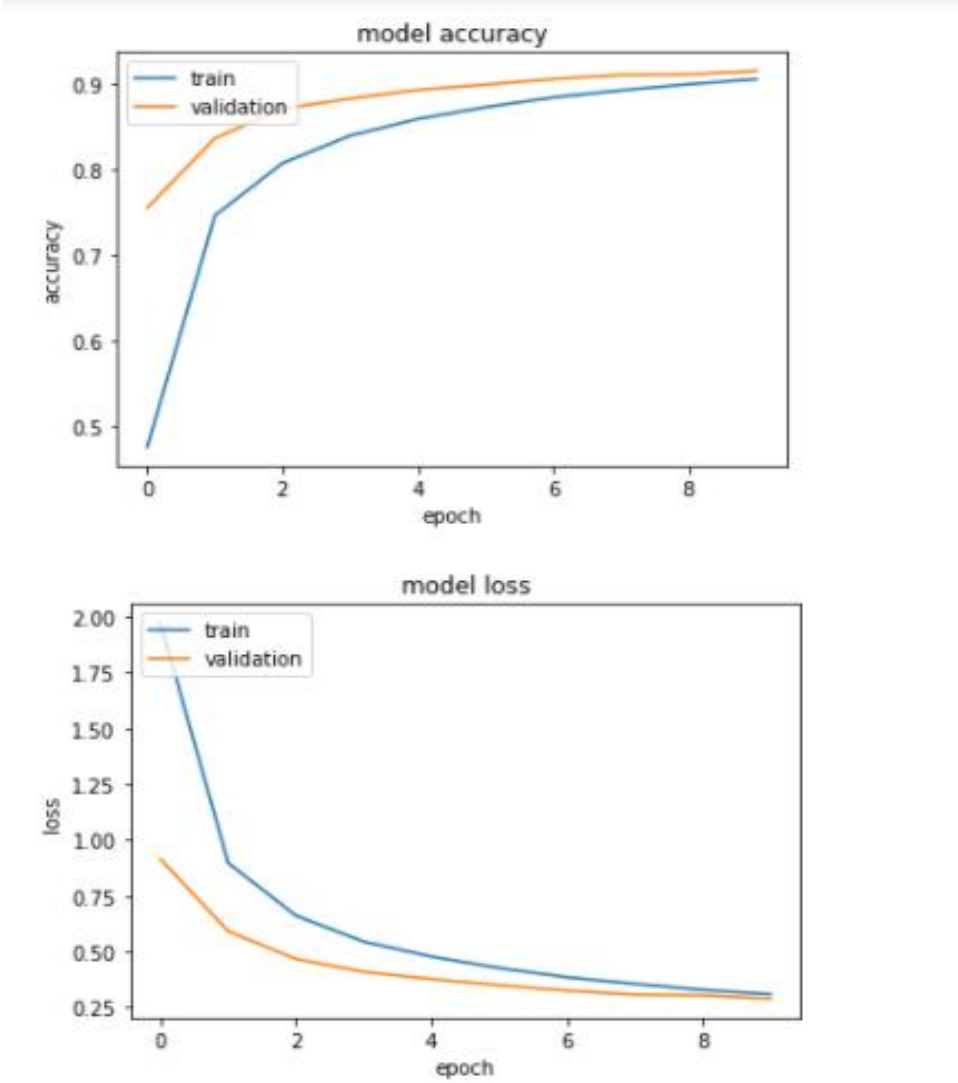
The final model was tested on testing dataset which was split from main dataset in order to test random images to see how best our model predicts on new images.

```
In [76]: scores = model2.evaluate(X_test, y_test, verbose=2)
         print("Accuracy: %.2f%%" % (scores[1]*100))

         Accuracy: 91.80%
```

testing accuracy on testing dataset

Our model had a good testing accuracy score of above 90% and good reliability as seen in the test.

Validation (testing) curves can be seen below in yellow and blue line in first and second graph respectively-



As it can be seen from above , increasing the no. of epochs would result in better performance (increased accuracy and decreased loss).

Here the model improves with increasing epochs, but the amount of improvement becomes lesser and lesser after each passing epoch, but after some time increase in epochs will result in overfitting before which we have to stop training to avoid decrease in performance (accuracy).

Reflection

Project can be summarized as following-

1. An initial problem and a relevant public dataset were found.
2. A benchmark model (kaggle) is chosen for the problem.
3. The dataset was downloaded and loaded in the code.

4. The data is preprocessed as required.
5. A deep neural network model using convolutional neural network(CNN) is built.
6. The model is trained on training dataset and tested on the unseen dataset which is split (80/20 split as mentioned in data exploration section) from the main dataset multiple times in order to find an optimum model architecture with good hyper-parameters.

The final model is plotted with accuracy v epoch and loss v epoch curves.

I found the last step (Step- 6) to be particularly tough although interesting as it took me a lot of time for model testing (10 minutes per model) .Yes, I was satisfied by the final model and solution and it could be used to solve these kind of problems.

Improvement

Accuracy could be improved by changing hyper-parameters through increasing epochs and batch size but even for improving accuracy by a small margin a lot of computational power and time would be required.

Also some adjustment with layers and filters could also improve accuracy.