Handwritten Devanagari Script Recognition

Author : Sahil Bharodiya
sahilbharodiya2002@outlook.com

Abstract
This is a Neural Network based model that classifies the Devanagari handwritten characters. I used python libraries like Pandas, NumPy, Matplotlib, Sci-kit learn and TensorFlow. You will get data used in this code from Train, Test and new_images. I recommend you all to use google colab which is a Jupyter notebook environment that runs in the browser using Google Cloud.

About Data
Data used in this code is in microsoft excel comma separated value. First column of this data is a range of values from 01 to 36 and digit-0 to digit-9 Which refers to Devanagari letters 'क' to 'ज्ञ' and digits '०' to '९'. And the rest of columns are pixel values ranging from 0 to 256. All the letters are in the same amount.

About imported libraries

1) Pandas - for reading csv files
2) Numpy - for array related work
3) Matplotlib - for plotting images
4) Sklearn - for labelling data
5) Tensorflow - for neural network
6) Random - for generating a random number

Data preprocessing
I divided the array with 255.0 for setting the image pixel value from 0 to 1. Complete data is well arranged in training and testing.

Build the model
Building the neural network requires configuring the layers of the model, then compiling the model.

(1) Set up the layers : The basic building block of a neural network is the layer. Layers extract representations from the data fed into them. Hopefully, these representations are meaningful for the problem at hand. Most of deep learning consists of chaining together simple layers. Most layers, such as tf.keras.layers.Dense, have parameters that are learned during training.

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(1024,)),
    tf.keras.layers.Dense(1024, activation='relu'),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(46)])
```

The first layer in this network, tf.keras.layers.Flatten, transforms the format of the images from a two-dimensional array (of 1024 × 1 pixels) to a one-dimensional array (of 1024 pixels). Think of this layer as unstacking rows of pixels in the image and lining them up. This layer has no parameters to learn; it only reformats the data. **You can also use input size as (32 × 32 pixels)**.

After the pixels are flattened, the network consists of a sequence of three tf.keras.layers.Dense layers. These are densely connected, or fully connected, neural layers. The first Dense layer has 1024 nodes (or neurons). The second Dense layer has 512 nodes (or neurons). The third Dense layer has 256. The forth Dense layer has 128 nodes (or neurons). The sixth (and last) layer returns a logits array with length of 46. Each node contains a score that indicates the current image belongs to one of the 46 classes.

(2) Compile the model : Before the model is ready for training, it needs a few more settings. These are added during the model's *compile* step,

- *Loss function* —This measures how accurate the model is during training. You want to minimize this function to "steer" the model in the right direction.
- *Optimizer* —This is how the model is updated based on the data it sees and its loss function.
- *Metrics* —Used to monitor the training and testing steps. The following example uses *accuracy*, the fraction of the images that are correctly classified.

```
model.compile(optimizer='adam',
loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

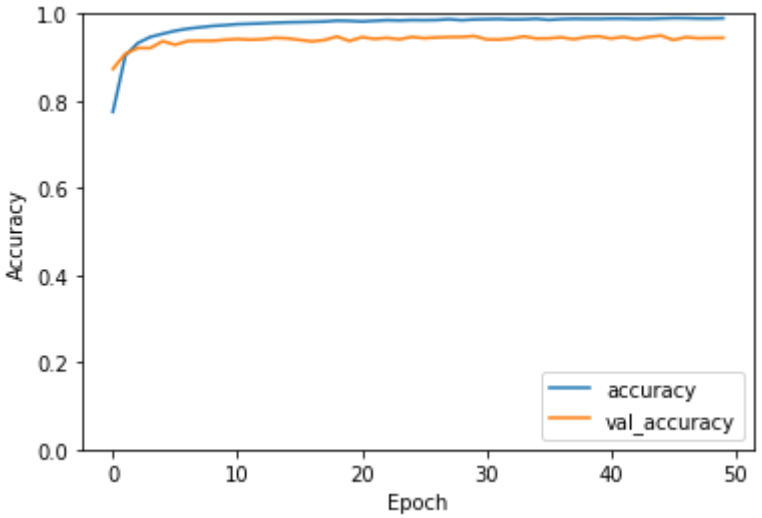
(3) Train the model : Training the neural network model requires the following steps:

- Feed the training data to the model. In this example, the training data is in the train_X and train_y arrays.
- The model learns to associate images and labels.
- You ask the model to make predictions about a test set—in this example, the test_X array.
- Verify that the predictions match the labels from the test_y array.

(4) Feed the model : To start training, call the model.fit method—so called because it "fits" the model to the training data.

As the model trains, the loss and accuracy metrics are displayed. This model reaches an accuracy of about 0.9911 (or 99.11%) on the training data.

(5) Accuracy vs Epoch grach :



(6) Evaluate accuracy : Next, compare how the model performs on the test dataset.

```
test_loss, test_acc = model.evaluate(test_X,  test_y, verbose=2)
print('Test accuracy:', test_acc)
```

Our model accuracy on test data is 94.47 %. It turns out that the accuracy on the test dataset is a little less than the accuracy on the training dataset. This gap between

training accuracy and test accuracy represents *overfitting*. Overfitting happens when a machine learning model performs worse on new, previously unseen inputs than it does on the training data. An overfitted model "memorizes" the noise and details in the training dataset to a point where it negatively impacts the performance of the model on the new data. For more information, see the following:

- Demonstrate overfitting
- Strategies to prevent overfitting

Predicting Results
Now, we have trained our model. I made a function which will preprocess the image data and predict the result.

Saving our model
Save the trained model in .h5 format.

Loading pretrained model
Open another notebook or existing notebook and load libraries. Load model using load_model (“path of your saved model file”). I also wrote some alphabets in paint and verified all images in model. Which pretty well works. You can also draw images of characters and verify on this model.
