Image Classification of Languages By Handwritten Letters

by Gray Simpson

This project classifies handwritten characters into a language. It reads letters and states them as English characters, Arabic, Chinese, Kannada, Baybayin, or Devanagari.

0, Chinese dataset: https://www.kaggle.com/datasets/gpreda/chinese-mnist

1, Arabic dataset: https://www.kaggle.com/datasets/mloey1/ahcd1

2, English dataset: https://www.kaggle.com/datasets/dhruvildave/english-handwritten-characters-dataset

3, Kannada dataset: https://www.kaggle.com/datasets/dhruvildave/kannada-characters

4, Baybayin dataset: https://www.kaggle.com/datasets/jamesnogra/baybayn-baybayin-handwritten-images

5, Devanagari dataset: https://www.kaggle.com/datasets/somnath796/devnagri-handwritten-character

In data cleaning, some colors have to be inverted, and many are sized down to 28x28 pixels as well. A lot of it was done before uploading to here with Python scripts I wrote. Since these are from entirely different datasets, I expect that while I can mitigate the damage done by variance in how the data was collected by inverting colors and the like, there are aspects of these datasets that may just be too different. It may not be able to generalize as well as it should given data that is not taken from these specific datasets. For example, the thinner lines in the Chinese dataset, or how the Devanagari characters regularly take up their whole square.

Here is my data, as used in this project: https://drive.google.com/drive/folders/1D7XCP6JMI0QvxvCDaENuswcAMCC7mpKI?usp=sharing

The format of each dataset is made to consist of the pathway to the image, and then the language of origin. No heed is paid to the specific character in the language. For accuracy in practical applications, it would make more sense to focus on determining the language of origin first, and then translating it to the actual content now that a smaller series of classfications can be had.

Importing and Bounds

With help from this Kaggle notebook on reading in the images: https://www.kaggle.com/code/akshayt19nayak/getting-started-image-processing-basics

```
import tensorflow as tf
from PIL import Image
import PIL.ImageOps
import numpy as np
import pandas as pd
import os
from os.path import exists
import pathlib
import seaborn as sb
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')

import cv2

num_classes = 6
batchsize = 128
imagesize = (28,28)
i = 0 #for naming the files as we move them
```

Next, we'll simply set up the dataframe we are going to create from the assorted datasets.

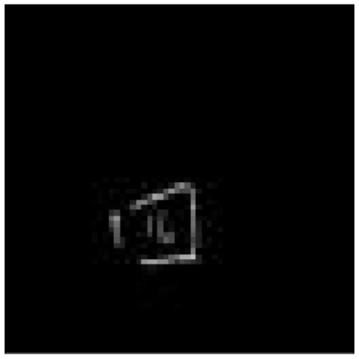
```
data = { "imagepath": [], "language": []}
df = pd.DataFrame(data)
```

Now let's take some test peeks into our data. We want to understand it better, and see what all we can do to make sure our input ends up uniform. Taken and modified from https://www.kaggle.com/code/akshayt19nayak/getting-started-image-processing-basics .

Chinese

```
chn_training_paths = pathlib.Path('drive/MyDrive/data/handwrittencharacters/chn').glob('input*.jpg')
chn_training_sorted = sorted([x for x in chn_training_paths])
chn_im_path = chn_training_sorted[25]
chn_bgrimg = cv2.imread(str(chn_im_path))
plt.imshow(chn_bgrimg)
plt.xticks([]) #To get rid of the x-ticks and y-ticks on the image axis
plt.yticks([])
```

```
([], <a list of 0 Text major ticklabel objects>)
```



Chinese needs to be shrunk a bit, so we'll do that.

```
i = 0
for x in chn_training_sorted:
  newpath = 'drive/MyDrive/data/handwrittencharacters/finaldata/' + str(i) + '.png'
  observation = [newpath,"chinese"]
  df.loc[-1] = observation
  df.index = df.index + 1
  if pathlib.Path(newpath).is_file() == False:
    old_img = Image.open(x)
    new_img = old_img.resize(imagesize)
    new_img.save(newpath)
  i+=1
```

Now let's peek at how our dataframe is looking, real quick.

```
df.head()
```

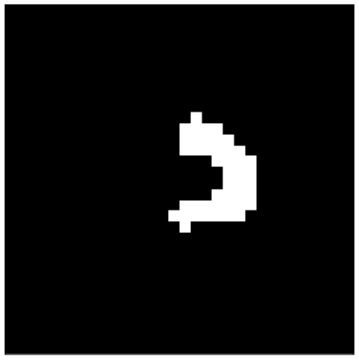
	imagepath	language
4255	drive/MyDrive/data/handwrittencharacters/final...	chinese
4254	drive/MyDrive/data/handwrittencharacters/final...	chinese
4253	drive/MyDrive/data/handwrittencharacters/final...	chinese

The data on our end is just folders full of images. We'll organize these into relevant dataframes on our own.

Arabic

```
abc_training_paths = pathlib.Path('drive/MyDrive/data/handwrittencharacters/abc').glob('id*label*.png')
abc_training_sorted = sorted([x for x in abc_training_paths])
abc_im_path = abc_training_sorted[25]
abc_bgrimg = cv2.imread(str(abc_im_path))
plt.imshow(abc_bgrimg)
plt.xticks([]) #To get rid of the x-ticks and y-ticks on the image axis
plt.yticks([])
```

```
([], <a list of 0 Text major ticklabel objects>)
```



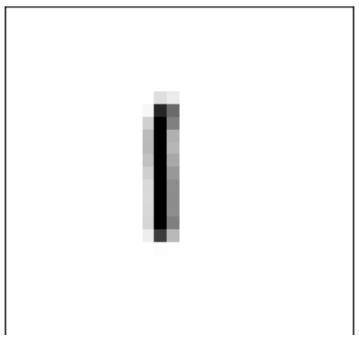
The Arabic characters still need to be shrunk slightly as we put them into the dataframe, so its not representative of their final product.

```
for x in abc_training_sorted:
  newpath = 'drive/MyDrive/data/handwrittencharacters/finaldata/' + str(i) + '.png'
  observation = [newpath,"arabic"]
  df.loc[-1] = observation
  df.index = df.index + 1
  if pathlib.Path(newpath).is_file() == False:
    old_img = Image.open(x)
    new_img = old_img.resize(imagesize)
    new_img.save(newpath)
  i+=1
```

English

```
eng_training_paths = pathlib.Path('drive/MyDrive/data/handwrittencharacters/eng').glob('img*.png')
eng_training_sorted = sorted([x for x in eng_training_paths])
eng_im_path = eng_training_sorted[25]
eng_bgrimg = cv2.imread(str(eng_im_path))
plt.imshow(eng_bgrimg)
plt.xticks([]) #To get rid of the x-ticks and y-ticks on the image axis
plt.yticks([])
```

```
([], <a list of 0 Text major ticklabel objects>)
```



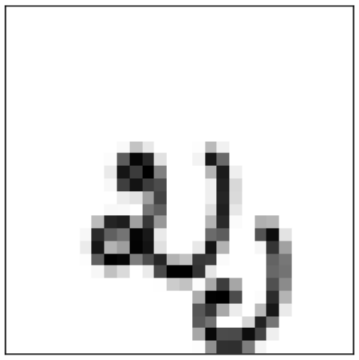
Due to how high res these character were before, they were shrunk with a Python script I wrote before being put into Google Drive for Colab to see. We'll have to invert them as we put them in.

```
for x in eng_training_sorted:
  newpath = 'drive/MyDrive/data/handwrittencharacters/finaldata/' + str(i) + '.png'
  observation = [newpath,"english"]
  df.loc[-1] = observation
  df.index = df.index + 1
  if pathlib.Path(newpath).is_file() == False:
    old_img = Image.open(x)
    new_img = PIL.ImageOps.invert(old_img)
    new_img.save(newpath)
  i+=1
```

Kannada

```
knd_training_paths = pathlib.Path('drive/MyDrive/data/handwrittencharacters/knd').glob('img*.png')
knd_training_sorted = sorted([x for x in knd_training_paths])
knd_im_path = knd_training_sorted[25]
knd_bgrimg = cv2.imread(str(knd_im_path))
plt.imshow(knd_bgrimg)
plt.xticks([]) #To get rid of the x-ticks and y-ticks on the image axis
plt.yticks([])
```

```
([], <a list of 0 Text major ticklabel objects>)
```



This was also too large to move before shrinking, so that part is done. Like English, we just have to invert the colors.

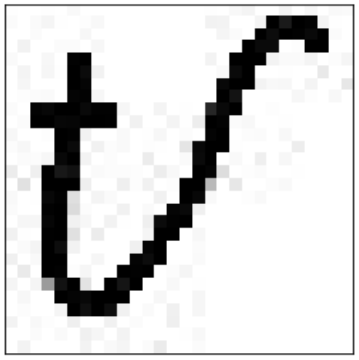
```
for x in knd_training_sorted:
  newpath = 'drive/MyDrive/data/handwrittencharacters/finaldata/' + str(i) + '.png'
  observation = [newpath,"kannada"]
  df.loc[-1] = observation
  df.index = df.index + 1
  if pathlib.Path(newpath).is_file() == False:
```

```
    old_img = Image.open(x)
    new_img = PIL.ImageOps.invert(old_img)
    new_img.save(newpath)
  i+=1
```

Baybayin

```
bbn_training_paths = pathlib.Path('drive/MyDrive/data/handwrittencharacters/bbn').glob('*.jpg')
bbn_training_sorted = sorted([x for x in bbn_training_paths])
bbn_im_path = bbn_training_sorted[25]
bbn_bgrimg = cv2.imread(str(bbn_im_path))
plt.imshow(bbn_bgrimg)
plt.xticks([]) #To get rid of the x-ticks and y-ticks on the image axis
plt.yticks([])
```

```
([], <a list of 0 Text major ticklabel objects>)
```



We can see that this data is not entirely clean, and it is already 28x28 pixels. We still need to invert it. It not being the same style as others and being so much more stark will create some error in our accuracy, but it is what we have and expected from using multiple different datasets.

This is also only a fraction of the amount of data they had, so that there are about 4600 entries of this now to comply with the others who have 3000-4000 samples while still keeping a decent amount of data for each individual character from the language.

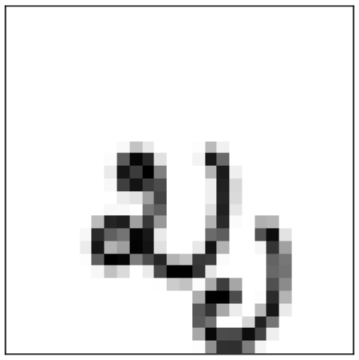
```
for x in bbn_training_sorted:
  newpath = 'drive/MyDrive/data/handwrittencharacters/finaldata/' + str(i) + '.png'
  observation = [newpath,"baybayin"]
  df.loc[-1] = observation
  df.index = df.index + 1
  if pathlib.Path(newpath).is_file() == False:
    old_img = Image.open(x)
    new_img = PIL.ImageOps.invert(old_img)
    new_img.save(newpath)
  i+=1
```

Devanagari

```
dvg_training_paths = pathlib.Path('drive/MyDrive/data/handwrittencharacters/dvg').glob('img*.png')
dvg_training_sorted = sorted([x for x in dvg_training_paths])
dvg_im_path = dvg_training_sorted[25]
dvg_bgrimg = cv2.imread(str(dvg_im_path))
plt.imshow(knd_bgrimg)
```

```
plt.xticks([]) #To get rid of the x-ticks and y-ticks on the image axis
plt.yticks([])
```

```
([], <a list of 0 Text major ticklabel objects>)
```



Another one to invert. Due to the file structure, was also processed outside of Colab with a Python script to preshrink them, rename them, and move them.

```
for x in dvg_training_sorted:
  newpath = 'drive/MyDrive/data/handwrittencharacters/finaldata/' + str(i) + '.png'
  observation = [newpath,"devanagari"]
  df.loc[-1] = observation
  df.index = df.index + 1
  if pathlib.Path(newpath).is_file() == False:
    old_img = Image.open(x)
    new_img = PIL.ImageOps.invert(old_img)
    new_img.save(newpath)
  i+=1
```

And now, we can look at our dataframe one last time to understand it.

```
df.tail()
```

	imagepath	language
4	drive/MyDrive/data/handwrittencharacters/final...	devanagari
3	drive/MyDrive/data/handwrittencharacters/final...	devanagari
2	drive/MyDrive/data/handwrittencharacters/final...	devanagari
1	drive/MyDrive/data/handwrittencharacters/final...	devanagari
0	drive/MyDrive/data/handwrittencharacters/final...	devanagari

```
df['language'].describe()
```

```
count        21946
unique           6
top       baybayin
freq          4559
Name: language, dtype: object
```

```
df.dtypes
```

```
imagepath    object
language     object
dtype: object
```

All looks good. Onto the next part.

Prepare Data

Now, we can prepare the data from our dataframe, and split it up.

```
df['language'] = df['language'].astype('category')
```

One important part is reading in our data as pixels.

```
column = list()
for x in range(28*28):
  column.append("p" + str(x))
#column.append("language")

pixeldf = pd.DataFrame(columns = column)

index = 0

for pathindex in df.imagepath:
  pixels = cv2.imread(pathindex, cv2.IMREAD_GRAYSCALE)
  listpix = pixels.ravel()
  pixeldf.loc[index] = listpix/255.0
  index += 1

pixeldf['language'] = df['language']
index
```

```
21946
```

```
pixeldf['language'] = df['language'].astype('category').cat.codes
pixeldf.head()
```

	p0	p1	p2	p3	p4	p5	p6	p7	p8	p9	...	p775	p776	p777	p778	p779	p780	p781	p782	p783	langua
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	

5 rows × 785 columns

```
pixeldf.dtypes
```

```
p0          float64
p1          float64
p2          float64
p3          float64
p4          float64
             ...
p780        float64
p781        float64
p782        float64
p783        float64
language       int8
Length: 785, dtype: object
```

Now, we'll save it to a CSV for later.

```
pixeldf.to_csv("drive/MyDrive/data/handwrittencharacters/langhandwriting.csv", index=False)
```

Alternately, if we're instead here to load it:

```
pixeldf = pd.read_csv("drive/MyDrive/data/handwrittencharacters/langhandwriting.csv")
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-21-97a5179f2931> in <module>
----> 1 pixeldf = pd.read_csv("langhandwriting.csv")

7 frames
/usr/local/lib/python3.8/dist-packages/pandas/io/common.py in get_handle(path_or_buf, mode, encoding,
compression, memory_map, is_text, errors, storage_options)
    700         if ioargs.encoding and "b" not in ioargs.mode:
    701             # Encoding
--> 702             handle = open(
    703                 handle,
    704                 ioargs.mode,

FileNotFoundError: [Errno 2] No such file or directory: 'langhandwriting.csv'

SEARCH STACK OVERFLOW
```

```
from sklearn.model_selection import train_test_split

X = pixeldf.loc[:, pixeldf.columns!='language']
Y = pixeldf.language

xtrain, xtest, ytrain, ytest = train_test_split(X, Y, test_size=0.2, random_state=800)

print('train size:', xtrain.shape)
print('test size:', xtest.shape)
```

```
train size: (17556, 784)
test size: (4390, 784)
```

```
ytrain.dtypes
```

```
dtype('int8')
```

And now, let's graph our data. We know we have a very large amount, and the data was filtered through by hand, but let's put this visually for ease of access.

```
plt.hist(df['language'])
```

```
(array([4256.,    0., 3124.,    0., 3410.,    0., 3285.,    0., 4559.,
        3312.]),
 array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. ]),
 <a list of 10 Patch objects>)
```

We can see that while Baybayin has more than others, it is overall well distributed and there is plenty of data to learn on for each language. It should be able to predict well, but since it is learning categories that very different characters belong to, we'll have to see how it works. All the same, it will be a wonderful learning experience.

Sequential: Build The Model

Now we'll get into the machine learning. We'll go through a few basic sortings of layers and see what works best.

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28,28)),
    tf.keras.layers.Dense(512,activation='elu'),
    tf.keras.layers.Dropout(0.20),
    tf.keras.layers.Dense(512,activation='relu'),
    tf.keras.layers.Dropout(0.20),
    tf.keras.layers.Dense(num_classes,activation='softmax'),
])
model.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_4 (Flatten)         (None, 784)               0

 dense_8 (Dense)             (None, 512)               401920

 dropout_6 (Dropout)         (None, 512)               0

 dense_9 (Dense)             (None, 512)               262656

 dropout_7 (Dropout)         (None, 512)               0

 dense_10 (Dense)            (None, 6)                 3078

=================================================================
Total params: 667,654
Trainable params: 667,654
Non-trainable params: 0
_________________________________________________________________
```

While different activation functions, layer organizations, and dropouts were interesting to explore, they all raised loss and lowered accuracy. This was found to be the best layer setup for the data.

Sequential: Train and Evaluate

To make sure our data is looking alright, we'll take a quick peek into our training data.

```
xtrain.head()
```

	p0	p1	p2	p3	p4	p5	p6	p7	p8	p9	...	p774	p775	p776	p777	p
1298	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	...	0.000000	0.000000	0.0	0.000000	0.000
20928	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000	0.000000	0.031373	...	0.564706	0.164706	0.0	0.015686	0.039
5909	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	...	0.000000	0.000000	0.0	0.000000	0.000
8710	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	...	0.000000	0.000000	0.0	0.000000	0.000

A number of optimizers and different formats were tried to up the accuracy and improve things, but further improvement was quite small and did not significantly change results. When a good setup was found, the number of epochs were raised to polish it further.

```
ytrain = tf.keras.utils.to_categorical(ytrain,num_classes)
ytest = tf.keras.utils.to_categorical(ytest,num_classes)
xtrain = xtrain.values.reshape(-1,28,28,1)
xtest = xtest.values.reshape(-1,28,28,1)
model.compile(loss='categorical_crossentropy', optimizer = 'adam', metrics = ['accuracy'])
seqresult = model.fit(xtrain,ytrain,
                      batch_size=batchsize,
                      epochs=25,
                      verbose=1,
                      validation_data=(xtest,ytest))
score = model.evaluate(xtest, ytest, verbose=0)
print('Test Loss: ', score[0])
print('Test Accuracy: ', score[1])
```

```
Epoch 1/25
138/138 [==============================] - 5s 35ms/step - loss: 0.7662 - accuracy: 0.7097 - val_loss: 0.5604 -
Epoch 2/25
138/138 [==============================] - 3s 23ms/step - loss: 0.4979 - accuracy: 0.8078 - val_loss: 0.4811 -
Epoch 3/25
138/138 [==============================] - 3s 23ms/step - loss: 0.4244 - accuracy: 0.8360 - val_loss: 0.4271 -
Epoch 4/25
138/138 [==============================] - 3s 22ms/step - loss: 0.3640 - accuracy: 0.8558 - val_loss: 0.4275 -
Epoch 5/25
138/138 [==============================] - 3s 23ms/step - loss: 0.3205 - accuracy: 0.8718 - val_loss: 0.4058 -
Epoch 6/25
138/138 [==============================] - 3s 23ms/step - loss: 0.2998 - accuracy: 0.8787 - val_loss: 0.3958 -
Epoch 7/25
138/138 [==============================] - 3s 23ms/step - loss: 0.2774 - accuracy: 0.8885 - val_loss: 0.4095 -
Epoch 8/25
138/138 [==============================] - 3s 23ms/step - loss: 0.2542 - accuracy: 0.8971 - val_loss: 0.4106 -
Epoch 9/25
138/138 [==============================] - 3s 23ms/step - loss: 0.2478 - accuracy: 0.8976 - val_loss: 0.3988 -
Epoch 10/25
138/138 [==============================] - 3s 23ms/step - loss: 0.2276 - accuracy: 0.9078 - val_loss: 0.4317 -
Epoch 11/25
138/138 [==============================] - 3s 23ms/step - loss: 0.2198 - accuracy: 0.9102 - val_loss: 0.4070 -
Epoch 12/25
138/138 [==============================] - 3s 22ms/step - loss: 0.2062 - accuracy: 0.9177 - val_loss: 0.4263 -
Epoch 13/25
138/138 [==============================] - 3s 23ms/step - loss: 0.2022 - accuracy: 0.9164 - val_loss: 0.4305 -
Epoch 14/25
138/138 [==============================] - 3s 23ms/step - loss: 0.1865 - accuracy: 0.9231 - val_loss: 0.4218 -
Epoch 15/25
138/138 [==============================] - 3s 22ms/step - loss: 0.1872 - accuracy: 0.9246 - val_loss: 0.4468 -
Epoch 16/25
138/138 [==============================] - 3s 23ms/step - loss: 0.1828 - accuracy: 0.9253 - val_loss: 0.4168 -
Epoch 17/25
138/138 [==============================] - 3s 23ms/step - loss: 0.1746 - accuracy: 0.9283 - val_loss: 0.4091 -
Epoch 18/25
138/138 [==============================] - 3s 22ms/step - loss: 0.1663 - accuracy: 0.9315 - val_loss: 0.4635 -
Epoch 19/25
138/138 [==============================] - 3s 23ms/step - loss: 0.1658 - accuracy: 0.9334 - val_loss: 0.4752 -
```

```
Epoch 20/25
138/138 [==============================] - 3s 23ms/step - loss: 0.1639 - accuracy: 0.9326 - val_loss: 0.4664 -
Epoch 21/25
138/138 [==============================] - 3s 23ms/step - loss: 0.1515 - accuracy: 0.9369 - val_loss: 0.4666 -
Epoch 22/25
138/138 [==============================] - 3s 23ms/step - loss: 0.1510 - accuracy: 0.9379 - val_loss: 0.4692 -
Epoch 23/25
138/138 [==============================] - 3s 23ms/step - loss: 0.1443 - accuracy: 0.9397 - val_loss: 0.4786 -
Epoch 24/25
138/138 [==============================] - 3s 23ms/step - loss: 0.1444 - accuracy: 0.9415 - val_loss: 0.5531 -
Epoch 25/25
138/138 [==============================] - 3s 23ms/step - loss: 0.1405 - accuracy: 0.9413 - val_loss: 0.5176 -
Test Loss:  0.5176120400428772
Test Accuracy:  0.8546696901321411
```

While loss is quite high, so is accuracy! It did very well for such a straightforward model. It will be interesting to see how other networks compare.

RNN: Build

Now onto RNNs. A number of different things were tried, but too many layers caused the accuracy to go down. Two layers with flattening provided the best results, and a dropout rate of .2 has again proved to be the most stable option.

```
RNNmodel = tf.keras.models.Sequential([
    tf.keras.layers.Input(shape=(28,28)),
    tf.keras.layers.SimpleRNN(28, activation='relu', return_sequences = True),
    tf.keras.layers.Dropout(0.20),
    tf.keras.layers.SimpleRNN(28, activation='relu'),
    tf.keras.layers.Dropout(0.20),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(num_classes,activation='softmax'),
])
RNNmodel.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 simple_rnn_4 (SimpleRNN)    (None, 28, 28)            1596

 dropout_10 (Dropout)        (None, 28, 28)            0

 simple_rnn_5 (SimpleRNN)    (None, 28)                1596

 dropout_11 (Dropout)        (None, 28)                0

 flatten_6 (Flatten)         (None, 28)                0

 dense_12 (Dense)            (None, 6)                 174

=================================================================
Total params: 3,366
Trainable params: 3,366
Non-trainable params: 0
_________________________________________________________________
```

RNN: Train and Evaluate

Due to modifications to the data that the purely sequential model needed, we will split it into train and test again so that the data is the right format needed for the RNN. From the successes of prior trials, the optimizer is kept as Adam.

```
xtrain, xtest, ytrain, ytest = train_test_split(X, Y, test_size=0.2, random_state=800)
ytrain = tf.keras.utils.to_categorical(ytrain,num_classes)
ytest = tf.keras.utils.to_categorical(ytest,num_classes)
xtrain = xtrain.values.reshape(-1,28,28,1)
xtest = xtest.values.reshape(-1,28,28,1)

RNNmodel.compile(loss='categorical_crossentropy', optimizer = 'adam', metrics = ['accuracy'])
RNNresult = RNNmodel.fit(xtrain,ytrain,
                     batch_size=batchsize,
                     epochs=90,
                     verbose=1,
                     validation_data=(xtest,ytest))
score = RNNmodel.evaluate(xtest, ytest, verbose=0)
print('Test Loss: ', score[0])
print('Test Accuracy: ', score[1])
```

```
Epoch 62/90
138/138 [==============================] - 3s 24ms/step - loss: 0.3790 - accuracy: 0.8565 - val_loss: 0.3948
Epoch 63/90
138/138 [==============================] - 3s 25ms/step - loss: 0.3771 - accuracy: 0.8584 - val_loss: 0.3945
Epoch 64/90
138/138 [==============================] - 3s 25ms/step - loss: 0.3642 - accuracy: 0.8626 - val_loss: 0.3867
Epoch 65/90
138/138 [==============================] - 3s 24ms/step - loss: 0.3646 - accuracy: 0.8639 - val_loss: 0.4843
Epoch 66/90
138/138 [==============================] - 3s 25ms/step - loss: 0.3804 - accuracy: 0.8566 - val_loss: 0.4377
Epoch 67/90
138/138 [==============================] - 3s 24ms/step - loss: 0.3693 - accuracy: 0.8610 - val_loss: 0.4027
Epoch 68/90
138/138 [==============================] - 3s 25ms/step - loss: 0.3580 - accuracy: 0.8632 - val_loss: 0.3840
Epoch 69/90
138/138 [==============================] - 3s 24ms/step - loss: 0.3627 - accuracy: 0.8657 - val_loss: 0.4217
Epoch 70/90
138/138 [==============================] - 3s 25ms/step - loss: 0.3716 - accuracy: 0.8607 - val_loss: 0.4441
Epoch 71/90
138/138 [==============================] - 3s 24ms/step - loss: 0.3566 - accuracy: 0.8648 - val_loss: 0.4157
Epoch 72/90
138/138 [==============================] - 4s 25ms/step - loss: 0.3513 - accuracy: 0.8663 - val_loss: 0.3959
Epoch 73/90
138/138 [==============================] - 3s 25ms/step - loss: 0.3626 - accuracy: 0.8645 - val_loss: 0.3869
Epoch 74/90
138/138 [==============================] - 3s 25ms/step - loss: 0.3512 - accuracy: 0.8664 - val_loss: 0.3811
Epoch 75/90
138/138 [==============================] - 3s 25ms/step - loss: 0.3458 - accuracy: 0.8680 - val_loss: 0.4076
Epoch 76/90
138/138 [==============================] - 3s 25ms/step - loss: 0.3660 - accuracy: 0.8625 - val_loss: 0.3969
Epoch 77/90
138/138 [==============================] - 3s 25ms/step - loss: 0.3491 - accuracy: 0.8686 - val_loss: 0.3776
Epoch 78/90
138/138 [==============================] - 3s 25ms/step - loss: 0.3453 - accuracy: 0.8693 - val_loss: 0.4014
Epoch 79/90
138/138 [==============================] - 3s 25ms/step - loss: 0.3509 - accuracy: 0.8677 - val_loss: 0.3821
Epoch 80/90
138/138 [==============================] - 3s 25ms/step - loss: 0.3513 - accuracy: 0.8678 - val_loss: 0.3936
Epoch 81/90
138/138 [==============================] - 3s 25ms/step - loss: 0.3423 - accuracy: 0.8721 - val_loss: 0.3835
Epoch 82/90
138/138 [==============================] - 3s 24ms/step - loss: 0.3377 - accuracy: 0.8733 - val_loss: 0.3734
Epoch 83/90
138/138 [==============================] - 3s 24ms/step - loss: 0.3370 - accuracy: 0.8748 - val_loss: 0.3957
Epoch 84/90
138/138 [==============================] - 3s 24ms/step - loss: 0.3421 - accuracy: 0.8713 - val_loss: 0.4516
Epoch 85/90
138/138 [==============================] - 3s 24ms/step - loss: 0.3503 - accuracy: 0.8683 - val_loss: 0.4173
Epoch 86/90
138/138 [==============================] - 3s 25ms/step - loss: 0.3605 - accuracy: 0.8641 - val_loss: 0.3796
Epoch 87/90
138/138 [==============================] - 3s 24ms/step - loss: 0.3400 - accuracy: 0.8722 - val_loss: 0.3740
Epoch 88/90
```

```
138/138 [==============================] - 3s 24ms/step - loss: 0.3482 - accuracy: 0.8702 - val_loss: 0.3922
Epoch 89/90
138/138 [==============================] - 3s 25ms/step - loss: 0.3420 - accuracy: 0.8728 - val_loss: 0.3788
Epoch 90/90
138/138 [==============================] - 3s 25ms/step - loss: 0.3444 - accuracy: 0.8716 - val_loss: 0.3835
```

Interestingly enough, it did not get better results than Sequential after Sequential had been through more epochs. The loss function also evened out quite thoroughly. The different does not seem to be too significant here, but interesting, and with much lower loss.

CNN: Build

```
CNNmodel = tf.keras.models.Sequential([
    tf.keras.layers.Input(shape=(28,28,1)),
    tf.keras.layers.Conv2D(32,padding='same', activation='elu', kernel_size=(3,3)),
    tf.keras.layers.MaxPooling2D(pool_size=(2,2)),
    tf.keras.layers.Conv2D(64,padding='same', activation='relu', kernel_size=(3,3)),
    tf.keras.layers.MaxPooling2D(pool_size=(2,2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(num_classes,activation='softmax'),
])
CNNmodel.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 32)        320

 max_pooling2d (MaxPooling2D  (None, 14, 14, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 14, 14, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 7, 7, 64)         0
 2D)

 flatten_2 (Flatten)         (None, 3136)              0

 dense_4 (Dense)             (None, 6)                 18822

=================================================================
Total params: 37,638
Trainable params: 37,638
Non-trainable params: 0
_________________________________________________________________
```

CNN: Train and Evaluate

```
xtrain, xtest, ytrain, ytest = train_test_split(X, Y, test_size=0.2, random_state=800)
ytrain = tf.keras.utils.to_categorical(ytrain,num_classes)
ytest = tf.keras.utils.to_categorical(ytest,num_classes)
xtrain = xtrain.values.reshape(-1,28,28,1)
xtest = xtest.values.reshape(-1,28,28,1)

CNNmodel.compile(loss='categorical_crossentropy', optimizer = 'adam', metrics = ['accuracy'])
cnnresult = CNNmodel.fit(xtrain,ytrain,
                      batch_size=batchsize,
                      epochs=85,
                      verbose=1,
```

```
                    validation_data=(xtest,ytest))
score = CNNmodel.evaluate(xtest, ytest, verbose=0)
print('Test Loss: ', score[0])
print('Test Accuracy: ', score[1])
```

```
Epoch 1/85
138/138 [==============================] - 27s 191ms/step - loss: 0.7859 - accuracy: 0.7001 - val_loss: 0.483
Epoch 2/85
138/138 [==============================] - 29s 211ms/step - loss: 0.4535 - accuracy: 0.8352 - val_loss: 0.437
Epoch 3/85
138/138 [==============================] - 31s 227ms/step - loss: 0.3922 - accuracy: 0.8534 - val_loss: 0.400
Epoch 4/85
138/138 [==============================] - 25s 178ms/step - loss: 0.3477 - accuracy: 0.8672 - val_loss: 0.353
Epoch 5/85
138/138 [==============================] - 25s 179ms/step - loss: 0.3260 - accuracy: 0.8739 - val_loss: 0.332
Epoch 6/85
138/138 [==============================] - 26s 191ms/step - loss: 0.3085 - accuracy: 0.8796 - val_loss: 0.348
Epoch 7/85
138/138 [==============================] - 25s 180ms/step - loss: 0.2906 - accuracy: 0.8872 - val_loss: 0.310
Epoch 8/85
138/138 [==============================] - 25s 179ms/step - loss: 0.2735 - accuracy: 0.8935 - val_loss: 0.311
Epoch 9/85
138/138 [==============================] - 25s 179ms/step - loss: 0.2611 - accuracy: 0.8992 - val_loss: 0.318
Epoch 10/85
138/138 [==============================] - 26s 190ms/step - loss: 0.2549 - accuracy: 0.9012 - val_loss: 0.294
Epoch 11/85
138/138 [==============================] - 26s 190ms/step - loss: 0.2400 - accuracy: 0.9066 - val_loss: 0.329
Epoch 12/85
138/138 [==============================] - 25s 180ms/step - loss: 0.2427 - accuracy: 0.9041 - val_loss: 0.285
Epoch 13/85
138/138 [==============================] - 25s 179ms/step - loss: 0.2276 - accuracy: 0.9118 - val_loss: 0.288
Epoch 14/85
138/138 [==============================] - 25s 178ms/step - loss: 0.2149 - accuracy: 0.9150 - val_loss: 0.286
Epoch 15/85
138/138 [==============================] - 25s 178ms/step - loss: 0.2067 - accuracy: 0.9173 - val_loss: 0.292
Epoch 16/85
138/138 [==============================] - 24s 176ms/step - loss: 0.2056 - accuracy: 0.9183 - val_loss: 0.280
Epoch 17/85
138/138 [==============================] - 24s 177ms/step - loss: 0.1910 - accuracy: 0.9261 - val_loss: 0.279
Epoch 18/85
138/138 [==============================] - 25s 178ms/step - loss: 0.1866 - accuracy: 0.9260 - val_loss: 0.280
Epoch 19/85
138/138 [==============================] - 25s 178ms/step - loss: 0.1789 - accuracy: 0.9306 - val_loss: 0.299
Epoch 20/85
138/138 [==============================] - 25s 178ms/step - loss: 0.1765 - accuracy: 0.9302 - val_loss: 0.293
Epoch 21/85
138/138 [==============================] - 25s 178ms/step - loss: 0.1688 - accuracy: 0.9329 - val_loss: 0.287
Epoch 22/85
138/138 [==============================] - 25s 178ms/step - loss: 0.1652 - accuracy: 0.9345 - val_loss: 0.293
Epoch 23/85
138/138 [==============================] - 25s 178ms/step - loss: 0.1608 - accuracy: 0.9347 - val_loss: 0.288
Epoch 24/85
138/138 [==============================] - 25s 178ms/step - loss: 0.1505 - accuracy: 0.9410 - val_loss: 0.293
Epoch 25/85
138/138 [==============================] - 25s 181ms/step - loss: 0.1494 - accuracy: 0.9400 - val_loss: 0.306
Epoch 26/85
138/138 [==============================] - 25s 180ms/step - loss: 0.1518 - accuracy: 0.9374 - val_loss: 0.287
Epoch 27/85
138/138 [==============================] - 25s 181ms/step - loss: 0.1405 - accuracy: 0.9441 - val_loss: 0.308
Epoch 28/85
138/138 [==============================] - 25s 178ms/step - loss: 0.1399 - accuracy: 0.9443 - val_loss: 0.304
Epoch 29/85
```

And the convolutional network wins with the highest accuracy, though the loss is quite high here. It actually performed slightly worse with more epochs run on it, and this was determined to be the best result.

Pretrained Model

The search for a pretrained model to handle any sort of handwriting recognition was tough, but even outside of that it was difficult to figure out one that could handle images as small as 28x28 like I have, plus work effectively in greyscale. As it stands, I'm not certain this would be helpful for my exact problem I am trying to solve. Even the pretrained models I found (which I could not figure out how to implement) seemed to also be trained with full words. Or, perhaps, I misunderstood the angle. All the same, here is the link for what I found potentially useful for my problem, but could not manage:

https://github.com/google-research/bert/blob/master/multilingual.md

And below is a bit of code to show that I understood, if nothing else, one aspect of the code. I tried many other things as well here, but could not manage to get it to execute without errors due to the issues with me having greyscale, 28x28 data.

```
from keras.applications.xception import Xception

#premodel = tf.keras.applications.MobileNetV2(input_shape=(32,32,3), include_top=False, weights='imagenet')
premodel = tf.keras.applications.Xception(input_shape=(71,71,3), include_top=False, weights='imagenet')
#premodel.add(Xception(weights='imagenet', input_shape=(28,28,1), include_top=False))
premodel.trainable = False
premodel.summary()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/xception/xception_weights_
83683744/83683744 [==============================] - 1s 0us/step
Model: "xception"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_3 (InputLayer)           [(None, 71, 71, 3)]  0           []

 block1_conv1 (Conv2D)          (None, 35, 35, 32)   864         ['input_3[0][0]']

 block1_conv1_bn (BatchNormaliz  (None, 35, 35, 32)  128         ['block1_conv1[0][0]']
 ation)

 block1_conv1_act (Activation)  (None, 35, 35, 32)   0           ['block1_conv1_bn[0][0]']

 block1_conv2 (Conv2D)          (None, 33, 33, 64)   18432       ['block1_conv1_act[0][0]']

 block1_conv2_bn (BatchNormaliz  (None, 33, 33, 64)  256         ['block1_conv2[0][0]']
 ation)

 block1_conv2_act (Activation)  (None, 33, 33, 64)   0           ['block1_conv2_bn[0][0]']

 block2_sepconv1 (SeparableConv  (None, 33, 33, 128)  8768       ['block1_conv2_act[0][0]']
 2D)

 block2_sepconv1_bn (BatchNorma  (None, 33, 33, 128)  512        ['block2_sepconv1[0][0]']
 lization)

 block2_sepconv2_act (Activatio  (None, 33, 33, 128)  0          ['block2_sepconv1_bn[0][0]']
 n)

 block2_sepconv2 (SeparableConv  (None, 33, 33, 128)  17536      ['block2_sepconv2_act[0][0]']
 2D)

 block2_sepconv2_bn (BatchNorma  (None, 33, 33, 128)  512        ['block2_sepconv2[0][0]']
 lization)

 conv2d_2 (Conv2D)              (None, 17, 17, 128)  8192        ['block1_conv2_act[0][0]']

 block2_pool (MaxPooling2D)     (None, 17, 17, 128)  0           ['block2_sepconv2_bn[0][0]']
```

```
batch_normalization (BatchNorm  (None, 17, 17, 128)  512        ['conv2d_2[0][0]']
alization)

add (Add)                       (None, 17, 17, 128)  0          ['block2_pool[0][0]',
                                                                 'batch_normalization[0][0]']

block3_sepconv1_act (Activatio  (None, 17, 17, 128)  0          ['add[0][0]']
n)

block3_sepconv1 (SeparableConv  (None, 17, 17, 256)  33920      ['block3_sepconv1_act[0][0]']
2D)

block3_sepconv1_bn (BatchNorma  (None, 17, 17, 256)  1024       ['block3_sepconv1[0][0]']
lization)
```

Analysis of Models

As far as analyzing the different models go, the loss would vary widely when running too many or too few epochs among other factors, but accuracy largely stayed the same. Pretrained models were not tested in this project, however given the type of classification it may not have worked out entirely well. The other three, in regards to accuracy, are ranked here:

1. CNN (loss .53, accuracy 90%)
2. RNN (loss .38, accuracy 86%)
3. Sequential (loss .52, accuracy 85%)

Convolutional neural networks provided the most accuracy and most often classified the characters as the correct language. After that, RNNs and sequentials were about the same-- depending on random chance, with each execution, there was a chance that one would outperform the other. They are effectively at the same accuracy, it just depends.

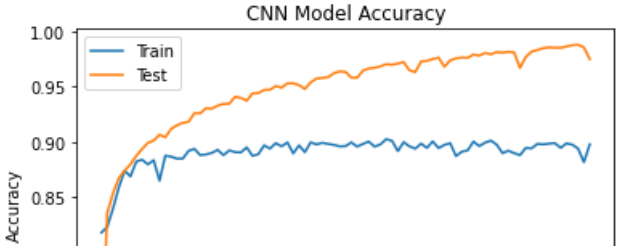
It's been particularly interesting to be able to experiment with the different types of layers. Even when I tried new types of layers, or orders, I could get dramatically different results. Often I would end up returning to something more close to what I started with based on readings on the topics, but not always. Sometimes I'd get to find a way that worked out better.

While far more time had to be spent cleaning and processing data than initially expected, it was all very interesting to learn and handle. Not only do I feel like I have learned much of image classification models, deep learning, and Keras, I feel like I'm finally starting to understand more of general Python use as well. Data preparation required a number of scripts and bits of writing to fully process. I also got to learn more about various languages, too.

However, onto the discussion on each different method of learning practiced here, in order of best to worst.

CNN Model

```
plt.plot(cnnresult.history['val_accuracy'])
plt.plot(cnnresult.history['accuracy'])
plt.title("CNN Model Accuracy")
plt.xlabel('Epoch Number')
plt.ylabel('Accuracy')
plt.legend(['Train', 'Test'])
plt.show()
```

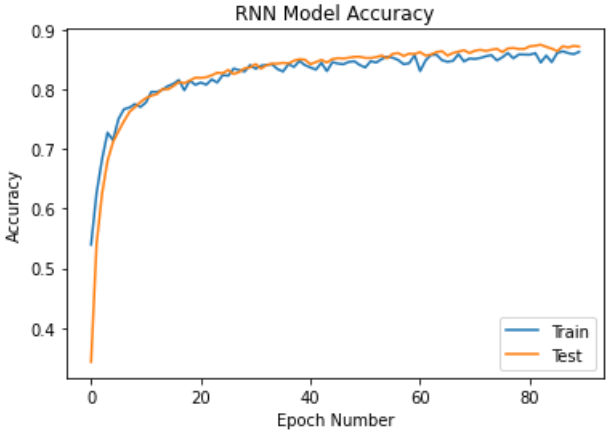


The CNN was the most accurate, at enough to round up to 90%, but even outside of this graph, I can denote that the loss actually spikes after the end of the epoch mark shown, giving a bit more insight into how it works. It also took the longest amount of time to train, which is to be expected from advanced neural networks. We can see an interesting trend to match it, as accuracy on training data remained largely the same even while the test data's accuracy and its ability to generalize continued to rise confidently.

It makes sense that the CNN was the most accurate, as it investigates small chunks of pixels to find traits that then can be replicated elsewhere, which is extremely helpful for handwriting. Each language well may tend to similar sets of traits-- such as how languages like Chinese may have many less circles and curves than English or Arabic. It focuses more on sharp edges, and a CNN is perfect to pick up these nuances.

RNN

```
plt.plot(RNNresult.history['val_accuracy'])
plt.plot(RNNresult.history['accuracy'])
plt.title("RNN Model Accuracy")
plt.xlabel('Epoch Number')
plt.ylabel('Accuracy')
plt.legend(['Train', 'Test'])
plt.show()
```



```
The recurrent neural network was about the same as sequent
finagling, and didn't look too promising, but was a model
more epochs well and went from 82% accuracy at 20 epochs t
with 90 epochs, loss of .38, and accuracy of 86%. All thin
aren't always considered the best for image data-- they be
sequential data, like text, predicting what would come nex
LSTM could be investigated further with more time, but kno
not its focus, there are other places to spend time.

From the graph, we can see that its ability to learn off o
test data are entirely intertwined. Since RNNs learn off o
```

The recurrent neural network was about the same as sequential. It took a bit of finagling, and didn't look too promising, but was a model that could appreciate more epochs well and went from 82% accuracy at 20 epochs to where it is now, with 90 epochs, loss of .38, and accuracy of 86%. All things considered, RNNs aren't always considered the best for image data-- they benefit more from sequential data, like text, predicting what would come next. Things such

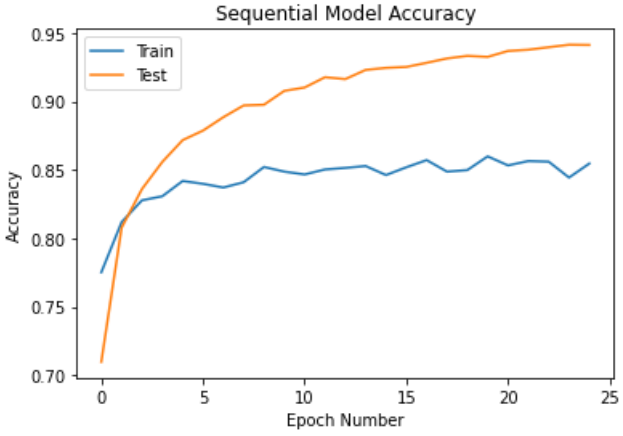
```
sequential data, this also makes sense as to its behavior.
conducive to image classfication and does not give us a hu
remains as good as a sequential model.
```

as LSTM could be investigated further with more time, but knowing that images are not its focus, there are other places to spend time.

From the graph, we can see that its ability to learn off of the training and test data are entirely intertwined. Since RNNs learn off of ordered and sequential data, this also makes sense as to its behavior. But it is not conducive to image classfication and does not give us a huge benefit, and it remains as good as a sequential model.

Sequential Model

```
plt.plot(seqresult.history['val_accuracy'])
plt.plot(seqresult.history['accuracy'])
plt.title("Sequential Model Accuracy")
plt.xlabel('Epoch Number')
plt.ylabel('Accuracy')
plt.legend(['Train', 'Test'])
plt.show()
```



The sequential model was the most straightforward and steady, but it did not mean that it was not effective. With a shocking 85% accuracy, it quickly and succinctly did its job. While we know that the data is in no way perfect, coming from different datasets entirely with different forms of collection, and that this probably made it easier than it should have been to identify characters, we can appreciate it all the same.

We can see from the chart that it continues to get better at generalizing to test data without consistently becoming too much better with the training data, a trend also obsevered from the CNN model. Sequential models did not benefit from increasing the number of epochs, so the lower number may explain why the graph seems "smoother" in comparison. It does teach us well about how models of this data learn.

Overall, CNNs are the best for image classification, and it has been a joy to explore them.

Colab paid products - Cancel contracts here

✓ 0s completed at 2:02 AM