Alison Sin (ms2697) 12/13/20

A Neuroscientist's Approach towards the Font Recognition

As a neuroscience major trying to finesse my way through the engineering and CS world, I am proud of my 0.88317-accurate model. I took a lot away from this competition: my first machine learning project.

I truly hammered down the concept of 'less being more', or the K-I-S-S principle. Starting ambitious, I would build my Convolution-Neural-Network from scratch. The model incorporated all of the data features given, and the network's architectures were tuned according to various existing ones that have been proven successful in image classifications, e.g. models for MNIST. I adopted transfer learning to incorporate winning neural networks such as Resnet50 and Googlenet etc. However, in the end, my customized CNN only ended up achieving an accuracy of 7X%. And my end model was made just by using the first 7 features in the dataset given, with an ensemble learning algorithm incorporating Adaboost and XGCBoost classifiers. This is a mind-boggling realization, and such a non-linear process is done as follows.

Exploratory Data Analysis / Pre-processing Step

I began with EDA, as building a good pipeline is foundational to the model's success. After sitting with the data, I categorized the given features into 3 data types: categorical, numerical, and image pixel. 'Categorical data' (`["strength"]`) were changed into boolean 0,1s, numerical data were transformed into the range of 0-1s using sklearn.preprocessing's MinMaxScaler function. As for image pixel preprocessing, I only intensified the pixel's intensity using the function `skimage.feature` from the sklearn library. This is because plotting the images would allow me to see that they were already cropped and centered.

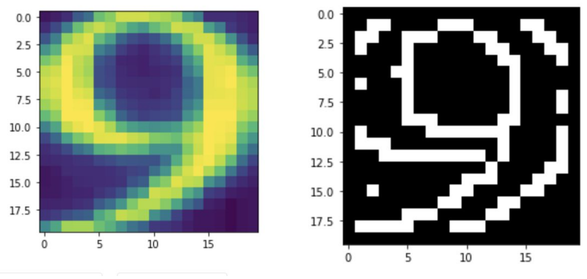


Fig 1. Image pixel before (left) and after (right) preprocessing.

Process of Building a CNN Model from Scratch

To build my CNN, I used the Keras and Tensorflow library. The model consists of 3 parts: 2 parallel networks that would converge into 1 final output network. One of the parallel CNN is a Multi-Layer Perceptron Layer, categorizing the categorical and numerical data, whereas the other parallel CNN is for classifying the image data. For the latter, I have attempted to use a classic 5,5 kernel and a 3-5-5, 3-5-5 kernel to increase non-linearity. I have also attempted to use transfer learning from existing winning neural networks e.g. Resnet50, and freeze the first chunk of layers and only train the final outputs. (Figure 2).

This attempt of customizing my own CNN made me realize that ML is truly as much as an art as it is of science. Changing the batch-size, learning rate, numbers of layers, etc. can drastically alter how the model performs. And finding the optimal complexity is hard, as using a complex model e.g. Resnet50 when unnecessary leads to a serious overfitting problem.

Process of Using Ensemble Methods / Voting Classifiers

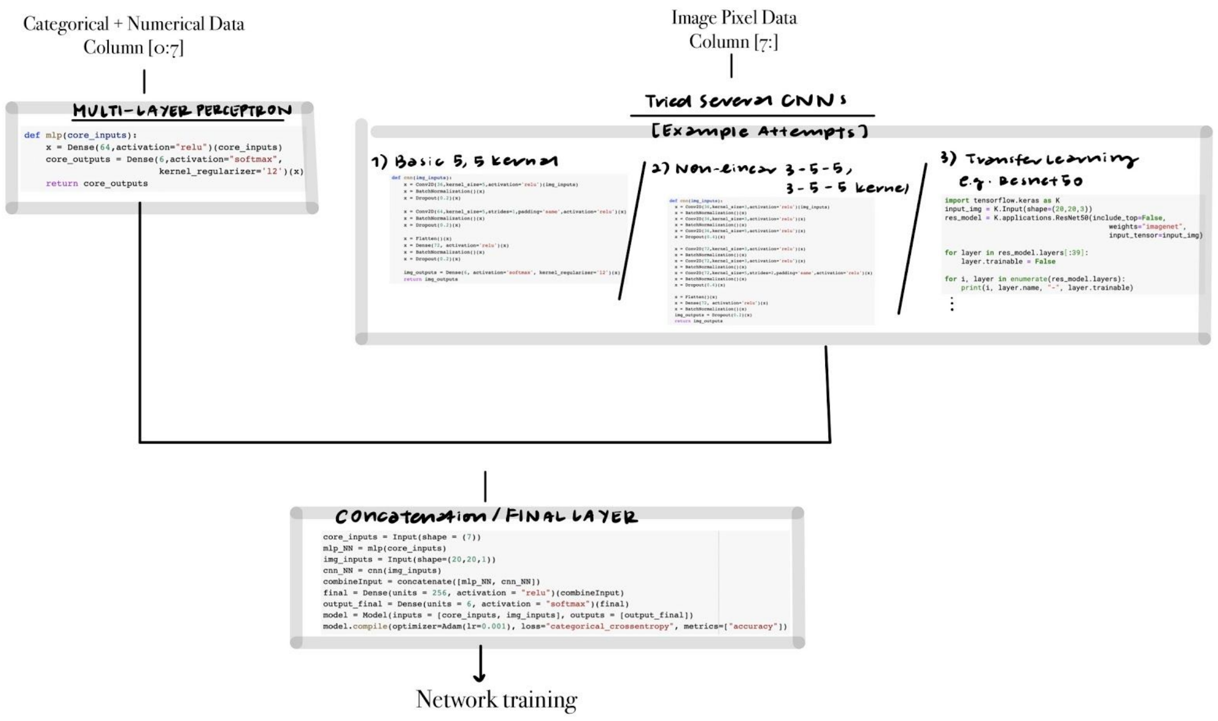
Since even simple organisms e.g. Zebrafish, uses a 'wisdom of crowd' method in action selection, I would then attempt to use scikit-learn's built-in ensemble algorithms. Through careful parameter tunings, I learned that feeding only the first 7 numerical + categorical features allows the model to achieve high accuracy, whereas adding in pixel data decreases it. (Table 1). My end model consisted of using the Voting-classifier to softly incorporate the 'AdaBoostClassifier' with the 'XGBClassifier', feeding in only the first 7 feature columns.

Model	**Accuracy**
`AdaBoostClassifier(DecisionTreeClassifier(random_state=300),n_estimators=400, learning_rate=1)`	`0.8687692307692307`
`RandomForestClassifier(n_estimators=1000)`	`0.8601025641025641`
`ExtraTreesClassifier(n_estimators=1000)`	`0.8603076923076923`
`XGBClassifier(n_estimators=6000,learning_rate = 1)`	`0.8824615384615384`

Table 1.

Summary

This is truly a mind-boggling revelation: I should always try to find the easiest path ('Keep-It-Simple-Stupid'). We could be given the most powerful tools, but they would be rendered useless if we do not use them properly (e.g. resnet50). Although time and effort do not correlate with the end product, I am still very proud and happy to have learned the skills of building my own custom CNN using TensorFlow. After all, it is the process/journey that counts.



```
def mlp(core_inputs):
    x = Dense(64,activation="relu")(core_inputs)
    core_outputs = Dense(6,activation="softmax",
                         kernel_regularizer='l2')(x)
    return core_outputs
```

```
def cnn(img_inputs):
    x = Conv2D(36,kernel_size=5,activation='relu')(img_inputs)
    x = BatchNormalization()(x)
    x = Dropout(0.2)(x)

    x = Conv2D(64,kernel_size=5,strides=1,padding='same',activation='relu')(x)
    x = BatchNormalization()(x)
    x = Dropout(0.2)(x)

    x = Flatten()(x)
    x = Dense(72, activation='relu')(x)
    x = BatchNormalization()(x)
    x = Dropout(0.2)(x)

    img_outputs = Dense(6, activation='softmax', kernel_regularizer='l2')(x)
    return img_outputs
```

```
def cnn(img_inputs):
  x = Conv2D(36,kernel_size=3,activation='relu')(img_inputs)
  x = BatchNormalization()(x)
  x = Conv2D(36,kernel_size=3,activation='relu')(x)
  x = BatchNormalization()(x)
  x = Conv2D(36,kernel_size=5,activation='relu')(x)
  x = Dropout(0.4)(x)

  x = Conv2D(72,kernel_size=3,activation='relu')(x)
  x = BatchNormalization()(x)
  x = Conv2D(72,kernel_size=3,activation='relu')(x)
  x = BatchNormalization()(x)
  x = Conv2D(72,kernel_size=5,strides=2,padding='same',activation='relu')(x)
  x = BatchNormalization()(x)
  x = Dropout(0.4)(x)

  x = Flatten()(x)
  x = Dense(72, activation='relu')(x)
  x = BatchNormalization()(x)
  img_outputs = Dropout(0.2)(x)
  return img_outputs
```

```
import tensorflow.keras as K
input_img = K.Input(shape=(20,20,3))
res_model = K.applications.ResNet50(include_top=False,
                                    weights="imagenet",
                                    input_tensor=input_img)

for layer in res_model.layers[:39]:
    layer.trainable = False

for i, layer in enumerate(res_model.layers):
    print(i, layer.name, "-", layer.trainable)
  ⋮
```

```
core_inputs = Input(shape = (7))
mlp_NN = mlp(core_inputs)
img_inputs = Input(shape=(20,20,1))
cnn_NN = cnn(img_inputs)
combineInput = concatenate([mlp_NN, cnn_NN])
final = Dense(units = 256, activation = "relu")(combineInput)
output_final = Dense(units = 6, activation = "softmax")(final)
model = Model(inputs = [core_inputs, img_inputs], outputs = [output_final])
model.compile(optimizer=Adam(lr=0.001), loss="categorical_crossentropy", metrics=["accuracy"])
```

Fig 2. Custom CNN Architecture