Contest 4 First Place Solution

Sauman Das, Edward Zhang (Super Sophomores)

August 26, 2020

1 Data

Before beginning to write the code for this contest, we looked at the data and analyzed the images. According to the description, there were 16,854 images given for training and another 5,641 unlabeled images used for the submission. The size of all images were constant at (100, 100) with 3 channels. We also analyzed the data frequency to see if there was any evidence of data imbalance.

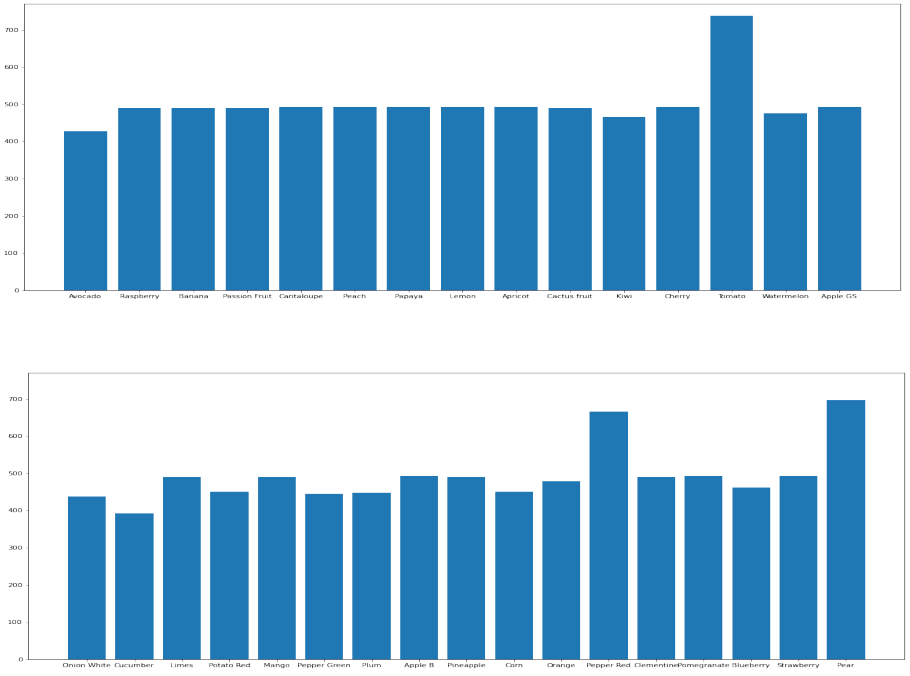

Figure 1: Number of Images in each Category

Although some fruits such as Tomato have more images than others, most categories have around the same number of training images. Therefore, we do not need to worry too much about the imbalance.

2 Data Augmentation

Although we are provided with a large amount of data, we can increase the number of images for training by using data augmentation. Using TensorFlow, applying these augmentations is quite simple. First, we import the necessary libraries.

```
#deep learning libraries
import tensorflow as tf
from tensorflow import keras
```

```
#data augmentation
from tensorflow.keras.preprocessing.image import ImageDataGenerator

#building the model
from tensorflow.keras.models import Sequential
from tensorflow.keras import layers as L

#import pretrained model
from tensorflow.keras.applications.inception_resnet_v2 import InceptionResNetv2
```

Now, we can start doing the image augmentation by using the ImageDataGenerator.

```
#we include several types of augmentations as the parameters
img_datagen = ImageDataGenerator(rescale=1./255,
                                 vertical_flip=True,
                                 horizontal_flip=True,
                                 width_shift_range=0.2,
                                 height_shift_range=0.2,
                                 zoom_range=0.1,
                                 validation_split=0.2)
```

When applying image data augmentation, it is important to think about what each augmentation does. For example, if we included the parameter **brightness_range** (makes an image darker or lighter), then a really dark papaya could be mistaken as an avocado.

Figure 2: (a) Papaya (b) Avocado

The augmentations we included will flip, translate, zoom in and out, and rescale the image so that all pixel values are between 0.0 and 1.0. This makes it easier for the model to fit the data.

The training images are given to us in a convenient form. Under the directory ../input/nmlo-contest-4/train/train, all the folder names represent the different fruits, with all the training images inside each designated folder. Since this is the way the file is organized we used the ***flow_from_directory*** method. Here is the code:

```
TRAIN_DIR = "../input/nmlo-contest-4/train/train"

train_generator = img_datagen.flow_from_directory(TRAIN_DIR,
                                                  shuffle=True,
                                                  batch_size=32,
                                                  subset="training",
                                                  target_size=(100, 100))

valid_generator = img_datagen.flow_from_directory(TRAIN_DIR,
                                                  shuffle=True,
                                                  batch_size=16,
                                                  subset="validation",
                                                  target_size=(100, 100))
```

We specified the target_size as (100, 100) which also implies that we will be using all 3 channels. However, we did not need to specify this parameter as all the images are guaranteed to be of that size. This is the end of the data augmentation part. When we fit the data, the generator will automatically pass augmented images along with the original versions.

3 Model

3.1 Structure

There were several models that we could build to fit this dataset. At first, we tried building models manually, however, these did not receive the best results. Fortunately, keras offers pretrained models that we can use. One such model is called the **InceptionResNetv2**. The model has 54,276,192 trainable parameters. Building this model ourselves would waste a lot of unnecessary time. On top of the InceptionResNetv2, we added one dense hidden layer, followed by a dropout to prevent overfitting to some extent. Finally, we have the output layer which predicts the 33 classes using the softmax activation function. Here is the code to build the model:

```
model.add(InceptionResNetV2(weights="imagenet",
                            include_top=False,
                            input_shape=(100, 100, 3)))
model.add(L.Flatten())
model.add(L.Dense(256, activation="relu"))
model.add(L.Dropout(0.5))
model.add(L.Dense(33, activation="softmax"))
```

We loaded weights from imagenet so that we would not start training from randomly initialized weights. Since we already have weights that recognize basic features in the earlier layers, many times it isn't necessary to train these layers again, so their weights are frozen. This is especially beneficial if there are few images in the training set because training less layers decreases the chance of overfitting. Furthermore, the training time increases significantly to train the entire model. Fortunately, we can still manage to train the complete model since Kaggle offers free GPU.

We decided to train the entire model without freezing any of the layers since we had enough images to prevent overfitting (even more generated using data augmentation).

3.2 Compiling the Model

We used the Adam optimizer with a learning rate of 0.0001. Higher learning rates were making the model loss fluctuate too much. The loss function we used was categorical cross entropy and we monitored the accuracy metric.

```
model.compile(optimizer=keras.optimizers.Adam(lr=0.0001),
          loss="categorical_crossentropy",
          metrics=["accuracy"])
```

3.3 Training

Before fitting the model to the data, we created an EarlyStopping Callback to automatically stop training if the validation accuracy decreased 3 times in a row. Here is the code to create the callback:

```
early = keras.callbacks.EarlyStopping(monitor="val_accuracy",
                                      patience=3,
                                      mode="max",
                                      restore_best_weights=True)
```

Now we are ready to train! We trained for a maximum of 10 epochs, but our model stopped improving after the 7th epoch, so the callback automatically stopped training. Each epoch took a little over a minute to train with the Kaggle GPU on.

```
history = model.fit(train_generator,
                    validation_data=valid_generator,
                    steps_per_epoch=train_generator.n//train_generator.batch_size,
                    validation_steps=valid_generator.n//valid_generator.batch_size,
```

```
callbacks=[early],
epochs=10)
```

We ended with a validation accuracy of 99.58%. We will not show the code for making the submission since it is just a call of model.predict. The only preprocessing done to the test images is to rescale their values in between 0.0 and 1.0.

4 Conclusion

That was the entire code for preparing the data and training the model. The key factor that helped us achieve this score was using the InceptionResNetv2. Data augmentation definitely helped the model, but building a simple model with just a couple convolutional layers would not have reached a really high accuracy. We might have been able to improve the results by a little bit if we addressed the slight data imbalance. Some fruit categories have almost double the average number of images. Perhaps undersampling those categories could have slightly improved the results.