Image Classification

Authors:
Andrew Sen
Neo Zhao

In this notebook, we will use various types of neural networks to classify images of birds. The dataset, which can be found here, is a collection of over 75,000 images of birds of 450 different species. We will train each network to classify the images according to the species of bird depicted in each image.

Data Exploration

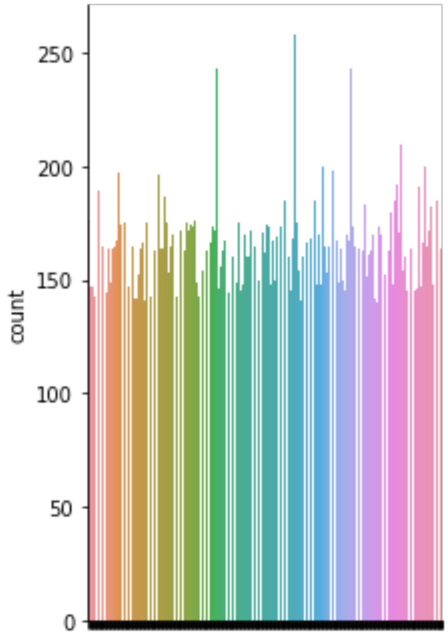
First, we'll have to read in the dataset.

In [ ]:
```
import pandas as pd
import os

csvpath = r'../input/100-bird-species/birds.csv'
source_dir = r'../input/100-bird-species'

df = pd.read_csv(csvpath)
df['filepaths'] = df['filepaths'].apply(lambda x: os.path.join(source_dir,x))
classes = sorted(df['labels'].unique())
num_classes = len(classes)
img_size = (224, 224)
img_shape = (img_size[0], img_size[1], 3)
```

Let's make a graph showing the distribution of classes.

In [ ]:
```
import seaborn as sb

graph = sb.catplot(x='labels', kind='count', data=df)
graph.set(xticklabels=[])
graph.set(xlabel='')
graph
```

Out[ ]:
```
<seaborn.axisgrid.FacetGrid at 0x7f640b8d8850>
```



As we can see, the classes are about evenly distributed, with all of them having at least 100 associated images. Some classes have a much larger amount of images, but it shouldn't have too much of an effect on the final results.

Model Training

First, we'll have to divide the data into train, test, and validate sets.

In [ ]:

```
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import models
from tensorflow.keras import layers
from sklearn.metrics import accuracy_score
import numpy as np

# setting random seeds
np.random.seed(1234)

# train/test/validate split
i = np.random.rand(len(df)) < 0.8
train = df[i]
test = df[~i]
i = np.random.rand(len(train)) < 0.8
valid = train[~i]
train = train[i]

batch_size = 30
gen = ImageDataGenerator()

train_gen = gen.flow_from_dataframe(
    train,
    x_col = 'filepaths',
    y_col = 'labels',
    target_size = img_size,
    class_mode = 'categorical',
    color_mode = 'rgb',
    shuffle = True,
    batch_size = batch_size
)

valid_gen = gen.flow_from_dataframe(
    valid,
    x_col = 'filepaths',
    y_col = 'labels',
    target_size = img_size,
    class_mode = 'categorical',
    color_mode = 'rgb',
    shuffle = True,
    batch_size = batch_size
)


test_gen = gen.flow_from_dataframe(
    test,
    x_col = 'filepaths',
    y_col = 'labels',
    target_size = img_size,
    class_mode = 'categorical',
    color_mode = 'rgb',
    shuffle = False,
    batch_size = batch_size
)
```

```
Found 48182 validated image filenames belonging to 450 classes.
Found 11907 validated image filenames belonging to 450 classes.
Found 15037 validated image filenames belonging to 450 classes.
```

Sequential Model

In [ ]:
```python
num_epochs = 5

# define model topology
model_seq = models.Sequential()
model_seq.add(layers.Input(shape=img_shape))
model_seq.add(layers.Flatten())
model_seq.add(layers.Dense(256, activation='relu'))
model_seq.add(layers.Dense(256, activation='relu'))
model_seq.add(layers.Dense(256, activation='relu'))
model_seq.add(layers.Dense(num_classes, activation='softmax'))

# train
model_seq.compile(
    optimizer = 'adam',
    loss = 'categorical_crossentropy',
    metrics = ['accuracy']
)

# apply to test data
model_seq.fit(
    x = train_gen,
    epochs = num_epochs,
    validation_data = valid_gen,
    validation_steps = None,
    shuffle = False
)

pred_seq = model_seq.predict(test_gen) # get predictions as label probabilities
pred_seq = np.argmax(pred_seq, axis=1) # get most likely label from probabilities
print('\naccuracy: ', accuracy_score(test_gen.labels, pred_seq))
```

```
Epoch 1/5
1607/1607 [==============================] - 206s 128ms/step - loss: 110.1773 - acc
uracy: 0.0029 - val_loss: 6.1090 - val_accuracy: 0.0031
Epoch 2/5
1607/1607 [==============================] - 204s 127ms/step - loss: 6.1058 - accur
acy: 0.0032 - val_loss: 6.1102 - val_accuracy: 0.0031
Epoch 3/5
1607/1607 [==============================] - 206s 128ms/step - loss: 6.1048 - accur
acy: 0.0032 - val_loss: 6.1108 - val_accuracy: 0.0031
Epoch 4/5
1607/1607 [==============================] - 202s 126ms/step - loss: 6.1044 - accur
acy: 0.0031 - val_loss: 6.1111 - val_accuracy: 0.0031
Epoch 5/5
1607/1607 [==============================] - 198s 123ms/step - loss: 6.1043 - accur
acy: 0.0031 - val_loss: 6.1116 - val_accuracy: 0.0031

accuracy:  0.0029926182084192327
```

The accuracy of this model is extremely bad, being almost 0%. Although the model was only given 5 epochs for training, the accuracy had already plateaued by the second epoch. It's likely because the network is too simple to learn any valuable information about the dataset. Three dense layers with 256 nodes each do not make a complex enough network to process the large inputs.

Convolutional Neural Network

In [ ]:

```
num_epochs = 30

model_cnn = models.Sequential()
model_cnn.add(layers.Input(shape=img_shape))
model_cnn.add(layers.Conv2D(128, 3, strides=2, padding="same", activation='relu'))
model_cnn.add(layers.BatchNormalization())
model_cnn.add(layers.MaxPooling2D(3, strides=2, padding="same"))
model_cnn.add(layers.Conv2D(128, 3, strides=2, padding="same", activation='relu'))
model_cnn.add(layers.BatchNormalization())
model_cnn.add(layers.GlobalAveragePooling2D())
model_cnn.add(layers.Dense(num_classes, activation='softmax'))

model_cnn.compile(
    optimizer = 'adam',
    loss = 'categorical_crossentropy',
    metrics = ['accuracy']
)

model_cnn.fit(
    x = train_gen,
    epochs = num_epochs,
    validation_data = valid_gen,
    validation_steps = None,
    shuffle = False
)

pred_cnn = model_cnn.predict(test_gen)
pred_cnn = np.argmax(pred_cnn, axis=1)
print('\naccuracy: ', accuracy_score(test_gen.labels, pred_cnn))
```

```
Epoch 1/30
1607/1607 [==============================] - 144s 89ms/step - loss: 5.2833 - accura
cy: 0.0464 - val_loss: 5.2095 - val_accuracy: 0.0563
Epoch 2/30
1607/1607 [==============================] - 144s 89ms/step - loss: 4.5272 - accura
cy: 0.1254 - val_loss: 4.4753 - val_accuracy: 0.1419
Epoch 3/30
1607/1607 [==============================] - 156s 97ms/step - loss: 4.0400 - accura
cy: 0.1986 - val_loss: 4.1044 - val_accuracy: 0.1969
Epoch 4/30
1607/1607 [==============================] - 175s 109ms/step - loss: 3.6605 - accur
acy: 0.2607 - val_loss: 3.8514 - val_accuracy: 0.2352
Epoch 5/30
1607/1607 [==============================] - 143s 89ms/step - loss: 3.3562 - accura
cy: 0.3132 - val_loss: 3.4461 - val_accuracy: 0.2939
Epoch 6/30
1607/1607 [==============================] - 144s 90ms/step - loss: 3.0954 - accura
cy: 0.3585 - val_loss: 3.9787 - val_accuracy: 0.2560
Epoch 7/30
1607/1607 [==============================] - 145s 90ms/step - loss: 2.8772 - accura
cy: 0.3948 - val_loss: 5.1052 - val_accuracy: 0.1709
Epoch 8/30
1607/1607 [==============================] - 141s 88ms/step - loss: 2.6904 - accura
cy: 0.4249 - val_loss: 3.2533 - val_accuracy: 0.3327
Epoch 9/30
1607/1607 [==============================] - 140s 87ms/step - loss: 2.5328 - accura
cy: 0.4540 - val_loss: 2.8963 - val_accuracy: 0.4025
Epoch 10/30
1607/1607 [==============================] - 142s 88ms/step - loss: 2.4093 - accura
cy: 0.4785 - val_loss: 3.7872 - val_accuracy: 0.2824
Epoch 11/30
1607/1607 [==============================] - 154s 96ms/step - loss: 2.2991 - accura
cy: 0.4972 - val_loss: 2.9159 - val_accuracy: 0.3934
Epoch 12/30
1607/1607 [==============================] - 153s 95ms/step - loss: 2.1833 - accura
cy: 0.5167 - val_loss: 3.6589 - val_accuracy: 0.3275
Epoch 13/30
1607/1607 [==============================] - 142s 88ms/step - loss: 2.1064 - accura
cy: 0.5324 - val_loss: 3.6984 - val_accuracy: 0.3135
Epoch 14/30
1607/1607 [==============================] - 144s 89ms/step - loss: 2.0339 - accura
cy: 0.5433 - val_loss: 4.8984 - val_accuracy: 0.2809
Epoch 15/30
1607/1607 [==============================] - 167s 104ms/step - loss: 1.9627 - accur
acy: 0.5589 - val_loss: 2.8317 - val_accuracy: 0.4027
Epoch 16/30
1607/1607 [==============================] - 145s 90ms/step - loss: 1.8967 - accura
cy: 0.5682 - val_loss: 2.7574 - val_accuracy: 0.4318
Epoch 17/30
1607/1607 [==============================] - 147s 91ms/step - loss: 1.8446 - accura
cy: 0.5805 - val_loss: 2.7416 - val_accuracy: 0.4418
Epoch 18/30
1607/1607 [==============================] - 141s 88ms/step - loss: 1.7983 - accura
cy: 0.5888 - val_loss: 2.7273 - val_accuracy: 0.4397
Epoch 19/30
1607/1607 [==============================] - 140s 87ms/step - loss: 1.7569 - accura
```

```
cy: 0.5950 - val_loss: 2.3679 - val_accuracy: 0.4937
Epoch 20/30
1607/1607 [==============================] - 146s 91ms/step - loss: 1.7154 - accura
cy: 0.6030 - val_loss: 2.6426 - val_accuracy: 0.4497
Epoch 21/30
1607/1607 [==============================] - 143s 89ms/step - loss: 1.6690 - accura
cy: 0.6116 - val_loss: 3.5512 - val_accuracy: 0.3847
Epoch 22/30
1607/1607 [==============================] - 147s 91ms/step - loss: 1.6289 - accura
cy: 0.6194 - val_loss: 2.9445 - val_accuracy: 0.4124
Epoch 23/30
1607/1607 [==============================] - 208s 129ms/step - loss: 1.5949 - accur
acy: 0.6268 - val_loss: 2.4848 - val_accuracy: 0.4763
Epoch 24/30
1607/1607 [==============================] - 176s 109ms/step - loss: 1.5692 - accur
acy: 0.6334 - val_loss: 3.0800 - val_accuracy: 0.4128
Epoch 25/30
1607/1607 [==============================] - 145s 90ms/step - loss: 1.5355 - accura
cy: 0.6399 - val_loss: 2.8071 - val_accuracy: 0.4258
Epoch 26/30
1607/1607 [==============================] - 145s 90ms/step - loss: 1.5065 - accura
cy: 0.6440 - val_loss: 2.7487 - val_accuracy: 0.4324
Epoch 27/30
1607/1607 [==============================] - 148s 92ms/step - loss: 1.4838 - accura
cy: 0.6479 - val_loss: 3.8793 - val_accuracy: 0.3839
Epoch 28/30
1607/1607 [==============================] - 143s 89ms/step - loss: 1.4472 - accura
cy: 0.6580 - val_loss: 2.7906 - val_accuracy: 0.4462
Epoch 29/30
1607/1607 [==============================] - 148s 92ms/step - loss: 1.4266 - accura
cy: 0.6565 - val_loss: 2.5159 - val_accuracy: 0.4923
Epoch 30/30
1607/1607 [==============================] - 141s 88ms/step - loss: 1.4037 - accura
cy: 0.6634 - val_loss: 2.6715 - val_accuracy: 0.4785
```

```
accuracy:  0.4719691427811398
```

This model's accuracy is considerably improved over the first sequential network. The accuracy is still less than half, but given there are 450 different target classes, this performance is perhaps to be expected of a network that simply consists of two convolutional layers in sequence. The model likely could have improved further with even more epochs, but 30 epochs already took a long time with a GPU. Either way, it seems a more complex network will be required to achieve a test accuracy over 80 or 90%.

Recurrent Neural Network with ConvLSTM1D

`ConvLSTM1D` is a type of recurrent network layer available in Keras that combines both LSTM and convolution. The training time is extremely long, so the number of epochs was reduced to just 2.

In [ ]:

```
num_epochs = 2

model_rnn = models.Sequential()
model_rnn.add(layers.Input(shape=img_shape))
model_rnn.add(layers.Rescaling(1. / 255))
model_rnn.add(layers.ConvLSTM1D(128, 3, activation='relu'))
model_rnn.add(layers.BatchNormalization())
model_rnn.add(layers.Flatten())
model_rnn.add(layers.Dense(num_classes, activation = 'softmax'))

model_rnn.compile(
    optimizer = 'adam',
    loss = 'categorical_crossentropy',
    metrics = ['accuracy']
)

model_rnn.fit(
    x = train_gen,
    epochs = num_epochs,
    validation_data = valid_gen,
    validation_steps = None,
    shuffle = False
)

pred_rnn = model_rnn.predict(test_gen)
pred_rnn = np.argmax(pred_rnn, axis = 1)
print('\naccuracy: ', accuracy_score(test_gen.labels, pred_rnn))
```

```
2022-12-05 02:09:26.750621: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:
937] successful NUMA node read from SysFS had negative value (-1), but there must b
e at least one NUMA node, so returning NUMA node zero
2022-12-05 02:09:26.871308: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:
937] successful NUMA node read from SysFS had negative value (-1), but there must b
e at least one NUMA node, so returning NUMA node zero
2022-12-05 02:09:26.872086: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:
937] successful NUMA node read from SysFS had negative value (-1), but there must b
e at least one NUMA node, so returning NUMA node zero
2022-12-05 02:09:26.873229: I tensorflow/core/platform/cpu_feature_guard.cc:142] Th
is TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN)
to use the following CPU instructions in performance-critical operations:  AVX2 AVX
512F FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compile
r flags.
2022-12-05 02:09:26.873538: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:
937] successful NUMA node read from SysFS had negative value (-1), but there must b
e at least one NUMA node, so returning NUMA node zero
2022-12-05 02:09:26.874251: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:
937] successful NUMA node read from SysFS had negative value (-1), but there must b
e at least one NUMA node, so returning NUMA node zero
2022-12-05 02:09:26.874901: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:
937] successful NUMA node read from SysFS had negative value (-1), but there must b
e at least one NUMA node, so returning NUMA node zero
2022-12-05 02:09:29.021112: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:
937] successful NUMA node read from SysFS had negative value (-1), but there must b
e at least one NUMA node, so returning NUMA node zero
2022-12-05 02:09:29.021976: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:
937] successful NUMA node read from SysFS had negative value (-1), but there must b
e at least one NUMA node, so returning NUMA node zero
2022-12-05 02:09:29.022643: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:
937] successful NUMA node read from SysFS had negative value (-1), but there must b
e at least one NUMA node, so returning NUMA node zero
2022-12-05 02:09:29.023243: I tensorflow/core/common_runtime/gpu/gpu_device.cc:151
0] Created device /job:localhost/replica:0/task:0/device:GPU:0 with 15401 MB memor
y:  -> device: 0, name: Tesla P100-PCIE-16GB, pci bus id: 0000:00:04.0, compute cap
ability: 6.0
2022-12-05 02:09:29.878089: I tensorflow/compiler/mlir/mlir_graph_optimization_pas
s.cc:185] None of the MLIR Optimization Passes are enabled (registered 2)
Epoch 1/2
```

```
2022-12-05 02:09:33.755933: I tensorflow/stream_executor/cuda/cuda_dnn.cc:369] Load
ed cuDNN version 8005
1607/1607 [==============================] - 2912s 2s/step - loss: 5.9362 - accurac
y: 0.0126 - val_loss: 6.0158 - val_accuracy: 0.0115
Epoch 2/2
1607/1607 [==============================] - 3000s 2s/step - loss: 5.4719 - accurac
y: 0.0423 - val_loss: 6.0517 - val_accuracy: 0.0167
```

```
accuracy:  0.01556161468378001
```

The accuracy shown is pretty bad, but we think this is only due to having so few epochs to train the model. We think that if the model were to be given 30 epochs like the other models in this notebook, it will have done at least as well as the convolutional network, if not better.

Pretrained Model

We will take a pretrained model and use transfer learning to apply the model to this particular task. This will jumpstart our model so that it can identify useful features in the data from the outset. Here, we will use the EfficientNetB3 model to make training slightly faster.

In [ ]:

```
from tensorflow.keras.applications.efficientnet import EfficientNetB3

num_epochs = 30

# init base model
base_model = EfficientNetB3(include_top=False, input_shape=img_shape, weights="imag

# set layers past 100 to be untrainable
fine_tune_at = 100
for layer in base_model.layers[:fine_tune_at]:
  layer.trainable = False

model_pre = models.Sequential()
model_pre.add(layers.Input(shape=img_shape))
model_pre.add(base_model)
model_pre.add(layers.BatchNormalization())
model_pre.add(layers.Dense(256, activation='relu'))
model_pre.add(layers.Dropout(rate=.2, seed=1234))
model_pre.add(layers.Dense(256, activation='relu'))
model_pre.add(layers.Dropout(rate=.2, seed=1234))
model_pre.add(layers.Dense(num_classes, activation='softmax'))

model_pre.compile(
    optimizer = 'adam',
    loss = 'categorical_crossentropy',
    metrics = ['accuracy']
)

model_pre.fit(
    x = train_gen,
    epochs = num_epochs,
    validation_data = valid_gen,
    validation_steps = None,
    shuffle = False
)

pred_pre = model_pre.predict(test_gen)
pred_pre = np.argmax(pred_pre, axis=1)
print('\naccuracy: ', accuracy_score(test_gen.labels, pred_pre))
```

```
Epoch 1/30
1607/1607 [==============================] - 323s 194ms/step - loss: 3.0409 - accur
acy: 0.3449 - val_loss: 1.3059 - val_accuracy: 0.6522
Epoch 2/30
1607/1607 [==============================] - 329s 205ms/step - loss: 1.3034 - accur
acy: 0.6608 - val_loss: 0.8500 - val_accuracy: 0.7748
Epoch 3/30
1607/1607 [==============================] - 312s 194ms/step - loss: 0.9696 - accur
acy: 0.7404 - val_loss: 0.7793 - val_accuracy: 0.7989
Epoch 4/30
1607/1607 [==============================] - 299s 186ms/step - loss: 0.8485 - accur
acy: 0.7740 - val_loss: 0.6370 - val_accuracy: 0.8378
Epoch 5/30
1607/1607 [==============================] - 297s 185ms/step - loss: 0.7615 - accur
acy: 0.7949 - val_loss: 0.5843 - val_accuracy: 0.8519
Epoch 6/30
1607/1607 [==============================] - 298s 186ms/step - loss: 0.6767 - accur
acy: 0.8156 - val_loss: 0.6004 - val_accuracy: 0.8560
Epoch 7/30
1607/1607 [==============================] - 298s 185ms/step - loss: 0.6238 - accur
acy: 0.8317 - val_loss: 0.5855 - val_accuracy: 0.8535
Epoch 8/30
1607/1607 [==============================] - 299s 186ms/step - loss: 0.5902 - accur
acy: 0.8406 - val_loss: 0.5823 - val_accuracy: 0.8622
Epoch 9/30
1607/1607 [==============================] - 404s 251ms/step - loss: 0.5609 - accur
acy: 0.8471 - val_loss: 0.5612 - val_accuracy: 0.8693
Epoch 10/30
1607/1607 [==============================] - 328s 204ms/step - loss: 0.5172 - accur
acy: 0.8584 - val_loss: 0.5247 - val_accuracy: 0.8723
Epoch 11/30
1607/1607 [==============================] - 302s 188ms/step - loss: 0.4763 - accur
acy: 0.8690 - val_loss: 0.5280 - val_accuracy: 0.8797
Epoch 12/30
1607/1607 [==============================] - 298s 186ms/step - loss: 0.4572 - accur
acy: 0.8755 - val_loss: 0.5538 - val_accuracy: 0.8798
Epoch 13/30
1607/1607 [==============================] - 306s 190ms/step - loss: 0.4352 - accur
acy: 0.8804 - val_loss: 0.5135 - val_accuracy: 0.8809
Epoch 14/30
1607/1607 [==============================] - 304s 189ms/step - loss: 0.4054 - accur
acy: 0.8882 - val_loss: 0.4564 - val_accuracy: 0.8929
Epoch 15/30
1607/1607 [==============================] - 320s 199ms/step - loss: 0.3874 - accur
acy: 0.8913 - val_loss: 0.4872 - val_accuracy: 0.8922
Epoch 16/30
1607/1607 [==============================] - 304s 189ms/step - loss: 0.3527 - accur
acy: 0.9006 - val_loss: 0.4747 - val_accuracy: 0.8910
Epoch 17/30
1607/1607 [==============================] - 299s 186ms/step - loss: 0.3577 - accur
acy: 0.8996 - val_loss: 0.4651 - val_accuracy: 0.8947
Epoch 18/30
1607/1607 [==============================] - 300s 187ms/step - loss: 0.3231 - accur
acy: 0.9094 - val_loss: 0.4638 - val_accuracy: 0.8939
Epoch 19/30
1607/1607 [==============================] - 298s 185ms/step - loss: 0.3162 - accur
```

```
acy: 0.9107 - val_loss: 0.4714 - val_accuracy: 0.9006
Epoch 20/30
1607/1607 [==============================] - 303s 189ms/step - loss: 0.2898 - accur
acy: 0.9178 - val_loss: 0.4751 - val_accuracy: 0.8977
Epoch 21/30
1607/1607 [==============================] - 319s 198ms/step - loss: 0.2862 - accur
acy: 0.9198 - val_loss: 0.5509 - val_accuracy: 0.8844
Epoch 22/30
1607/1607 [==============================] - 298s 185ms/step - loss: 0.2869 - accur
acy: 0.9178 - val_loss: 0.4679 - val_accuracy: 0.9026
Epoch 23/30
1607/1607 [==============================] - 298s 185ms/step - loss: 0.2683 - accur
acy: 0.9243 - val_loss: 0.4551 - val_accuracy: 0.9003
Epoch 24/30
1607/1607 [==============================] - 299s 186ms/step - loss: 0.2520 - accur
acy: 0.9280 - val_loss: 0.4781 - val_accuracy: 0.9043
Epoch 25/30
1607/1607 [==============================] - 301s 187ms/step - loss: 0.2570 - accur
acy: 0.9270 - val_loss: 0.4592 - val_accuracy: 0.9093
Epoch 26/30
1607/1607 [==============================] - 299s 186ms/step - loss: 0.2175 - accur
acy: 0.9372 - val_loss: 0.4741 - val_accuracy: 0.9048
Epoch 27/30
1607/1607 [==============================] - 311s 194ms/step - loss: 0.2276 - accur
acy: 0.9355 - val_loss: 0.4806 - val_accuracy: 0.9038
Epoch 28/30
1607/1607 [==============================] - 298s 186ms/step - loss: 0.2286 - accur
acy: 0.9350 - val_loss: 0.4917 - val_accuracy: 0.9010
Epoch 29/30
1607/1607 [==============================] - 297s 185ms/step - loss: 0.2225 - accur
acy: 0.9369 - val_loss: 0.4721 - val_accuracy: 0.9095
Epoch 30/30
1607/1607 [==============================] - 297s 185ms/step - loss: 0.2110 - accur
acy: 0.9394 - val_loss: 0.4773 - val_accuracy: 0.9055
```

```
accuracy:  0.9084258828223715
```

As we can see, adding the pretrained model drastically improved the accuracy, as was to be expected.

Analysis

As noted before, the sequential model's performance was extremely bad. This is likely because the network is far too simple to gleam any useful information from a given input. Given that the accuracy is very close to 0% and that the validation accuracy quickly plateaued, the model likely wasn't able to learn any useful information at all.

The CNN was more promising than a simple dense sequential model. This is likely because CNNs are generally useful for learning information from image data. The CNN's poor performance in this notebook speaks more to the complexity of the data than to the power of CNNs. More epochs, more convolutional layers, and more complexity in the network would have all likely improved the performance of the model.

It is hard to properly judge the performance of the RNN because we provided so little time for the model to properly train. The jump in accuracy between the two epochs was greater than what was demonstrated by the prior two models, indicating that at the very least, the RNN was learning more useful information more quickly. As stated before, we believe that with more epochs, this model would have performed at least as well as our CNN. Although RNNs are typically used for processing time-series data, we think that by combing LSTM with convolution, the network would be able to converge on a solution in a smaller number of iterations.

And as expected, the model based on the pretrained model was by far the best. Pretrained models are already primed to extract useful information that can help to differentiate images. This gives the network a headstart towards learning information that improves performance on this specific task. Furthermore, by using the EfficientNet model, the overall training time was not much worse than that of the CNN. This demonstrates that for any image classification task, using a pretrained model can immediately save a lot of time and effort.