Capstone 3 - Report

Introduction

The theme of this project was to explore the power of Convolutional Neural Networks, a neural network model that performs well on spatially-sensitive data like images. They're often used for image classification tasks.

The image classification task I chose to explore was classifying different types of Indian foods. This was the dataset I used to train the model - Indian Food Images Dataset

Characteristics of the dataset:

- 4000 images
- 80 different classes of indian foods
- 50 images in every class, so perfectly balanced

Just like with any other Data Science project, the project was divided into meaningful, distinct stages:

1. Data Wrangling
2. Exploratory Data Analysis
3. Modeling

Unlike the previous Capstone, the environment for this project was Google Colaboratory/Google Drive. The rationale behind this choice was:

1. The dataset would be somewhat large, so it's easier to manage on the cloud
2. Training CNNs is computationally expensive, so training it remotely on a Google Colaboratory instance works well, and moreover, there are options to work with GPUs to accelerate training times.
3. Google Colaboratory integrates seamlessly with Github for version control
4. It's far easier for other people to play with the notebook themselves

Also, all the CNN training would be done with Tensorflow/Keras, along with other standard Python data processing libraries like numpy, etc.

Data Wrangling

A majority of the Data Wrangling process was mostly geared towards just bringing the images into a workflow that made training the CNN as simple as possible.

The first step was loading in the dataset from Kaggle. I decided to learn how to use the Kaggle API to do this - it just involves downloading an API key (kaggle.json file) and running certain terminal commands on the notebook to load the dataset into a particular directory on Google Drive.

```
projectDir = "/content/drive/My Drive/Springboard-Capstone-3"
dataDir = projectDir + "/" + "data"
dataset1 = "iamsouravbanerjee/indian-food-images-dataset"
```

```
#Upload kaggle.json file to Drive
! mkdir ~/.kaggle
! cp kaggle.json ~/.kaggle/
! chmod 600 ~/.kaggle/kaggle.json
!kaggle datasets download $dataset1 -p "$dataDir" --unzip
```

This downloaded all the images into the `dataDir` folder, divided by subdirectories representing their class names. This format of image storage is actually already quite convenient since Keras' image generators process images by looking at the subdirectory they are under and assigning that subdirectory as the class name for that image.

For the CNN workflow, the dataset needed to be divided into train, validation, and test sets. To do that, I split each of the 80 subdirectories via the same train/validation/test split ratio and stored them in the same 80 subdirectories, except those subdirectories were under parent folders of train, validation, and test instead.

The code to do this wasn't all that trivial, so it's not worth including in the report. Refer to the Data Wrangling/EDA notebook for how this was setup.

Exploratory Data Analysis

Exploratory Data Analysis for the project wasn't as involved as it might have been with other more numerical data, since it was just a dataset of images. Nevertheless, there were still some interesting things to explore.

The first thing worth inspecting was seeing the class distribution of all the images. Imbalanced classes can severely impact the model's effectiveness after training as the model is implicitly prioritizing some classes over others. The code to build a bar chart of class counts is shown below:

```
categoryCounts = []
for category in categories:
 categoryDir = os.path.join(dataDir, category)
 numImages =  len([name for name in os.listdir(categoryDir) if
os.path.isfile(os.path.join(categoryDir, name))])
 categoryCounts.append(numImages)
```

```
plt.figure(figsize=(12, 16), dpi=80)
sns.barplot(y=categories, x=categoryCounts)
```

Which produced the following bar chart:

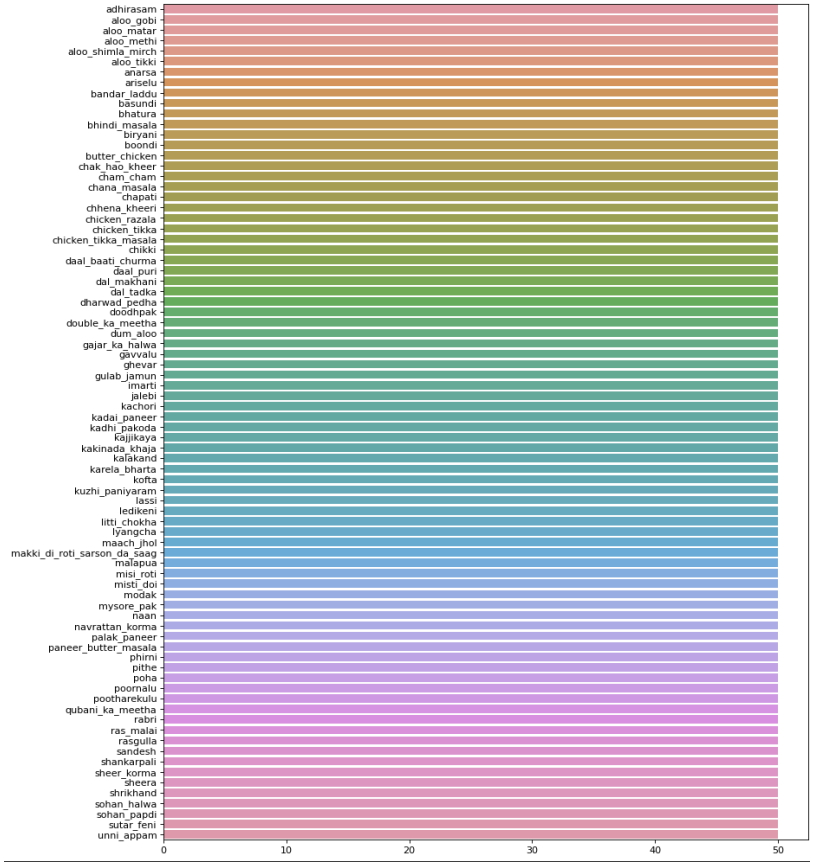

Figure 3.1 - Bar Chart of Class Distribution

Clearly, this shows all the classes are perfectly balanced at 50 images each, which is relieving. If they weren't, we would need to return to the Data Wrangling phase and figure out ways to balance the classes, either through some forms of image augmentation or downsampling.

Next was exploring the dataset, by actually visualizing the images! I wrote some code to choose 5 random categories out of the 80 in the dataset, and then choose 3 images each to render for inspection.

```
indices = np.random.choice(80, 5)
randomCategories = np.array(categories)[indices]
randomCategoriesImagePaths = np.array(image_paths)[indices]
print(randomCategories)
```

```
fig = plt.figure(figsize=(20, 20))
for i in range(len(randomCategories)):
 category = randomCategories[i]
 imagePaths = randomCategoriesImagePaths[i]
 indices = np.random.choice(50, 3)
 randomImagePaths = np.array(imagePaths)[indices]
 for j in range(len(randomImagePaths)):
   imagePath = randomImagePaths[j]
   image = imageio.imread(imagePath)
   ax = fig.add_subplot(5, 3, 3*i + (j+1))
   ax.imshow(image)
   ax.set_title(f'{category} image {j+1}')
```

Results:

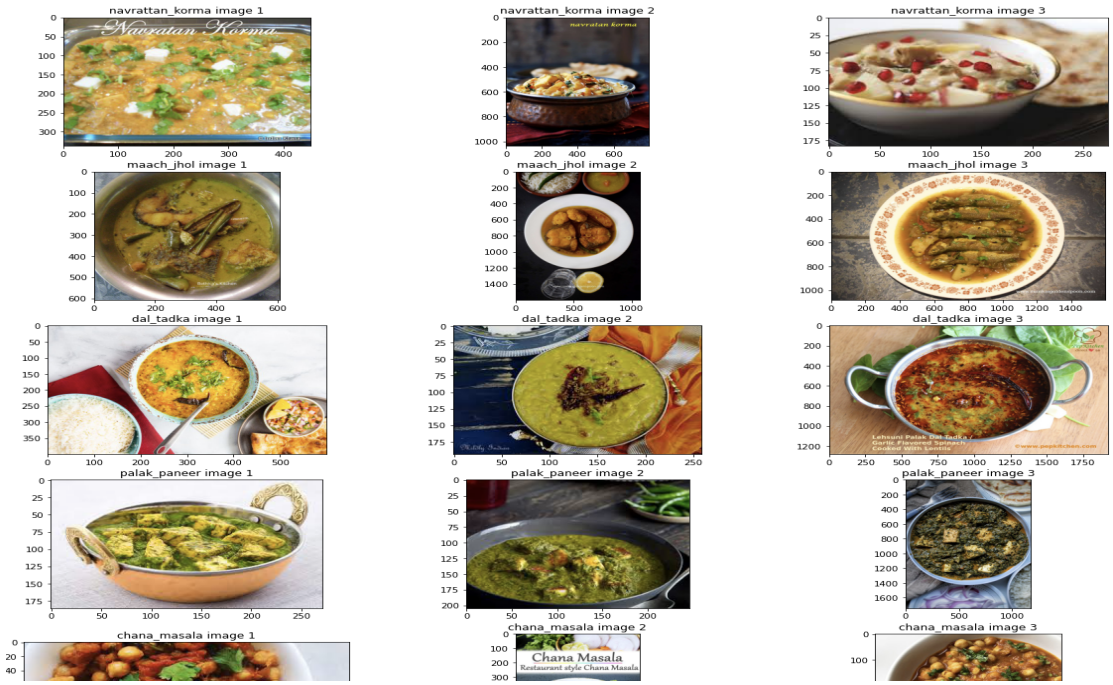

Figure 3.2 - Random set of images from the dataset

Running the random image selection code multiple times, I eventually noticed that there was an non-insignificant number of images that had text in them identifying the food. This could be a potential problem during training since textual features are distracting. A solution would be to manually remove such images in the dataset.

Another thing I thought was worth exploring was the resolutions of all the images. On average, what were the widths and heights of the images in each class? Were they high resolution images? This information I believed would be relevant as I anticipated building ImageDataGenerators with Keras, where this is a target_size parameter. Figuring out a good value for that parameter is essential, it basically controls the degree of resolution you believe is sufficient for the model to distinguish images. A large discrepancy between this parameter and the original image's resolution could hamper the model's ability to extract important features.

A simple but mundane task, I loaded all 4000 images and extracted these data points. Note, this takes around 20 minutes to run, but I justified it by reasoning that it would only be run once and thus the runtime cost would be amortized.

The results of this exploration are as shown below:

Figure 3.3 - Average Image Widths

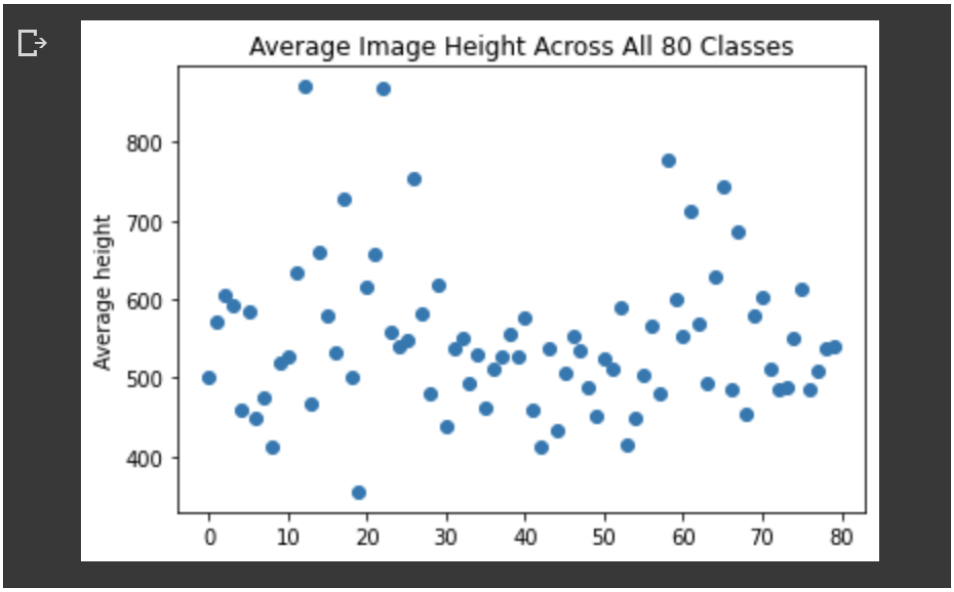

Figure 3.4 - Average Image Height

On average, the images had a resolution of around 400 x 500. That's not an extremely high resolution, but good enough for most practical purposes, and definitely good enough for a CNN to train on.

Modeling

Usually, the Modeling phase follows another training preprocessing phase where the dataset might be normalized and split into the appropriate sets. But some of this was already done in the Data Wrangling phase, and most meaningful image processing operations are taken care of by Keras' ImageDataGenerator, such as:

1. **Morphological transformations** on the images, like shearing, rotations, flips, resolution-shifting, zooming, etc
2. **Shuffling** the dataset
3. **Normalizing** the pixel ranges to [0,1]

So that meant we could dive straight into modeling.

The ideology I decided to follow for the modeling was leveraging Transfer Learning. Transfer Learning involves adapting pretrained models that have already learned useful/relevant information to a different learning task. Transfer Learning is quite common for Neural Networks because:

1. It uses pretrained models saves time in terms of training the network (since the trained weights are just pre-loaded)
2. CNNs might have already learned useful features and therefore just topping off the network with additional layers to learn more specific features might be sufficient.

In the context of CNN Transfer Learning, it's standard to use models that performed well on the ImageNet dataset as part of the ImageNet Large Scale Visual Recognition Challenge that happens annually. There have been certain models that have performed particularly well on these challenges, so some of the models I chose to create the base CNN models were:

1. VGG-16
2. ResNet-50
3. AlexNet (No transfer learning, just the architecture was borrowed)
4. VGG-19

I did not choose DenseNet201 for example, because the dataset isn't all that large (50 per class isn't objectively a lot) so there was some fear of overfitting. Deeper, denser networks have that tendency because there are more weights to train and therefore more opportunities for the network to learn needlessly specific features.

Here is an example of setting up the Transfer Learning with Keras and Tensorflow:

```
input_shape = (240, 240, 3)
learning_rate = 0.001
fine_tune = 2
dropout = 0.2
layer1 = 500
layer2 = 1500

#Build data generators
from tensorflow.keras.applications import vgg16

train_datagen = ImageDataGenerator(
       preprocessing_function = vgg16.preprocess_input,
       rescale=1./255,
       shear_range=0.6,
       zoom_range=0.7,
       horizontal_flip=True,
       vertical_flip=True
       )
test_datagen = ImageDataGenerator(
   preprocessing_function = vgg16.preprocess_input,
```

```
   rescale=1./255)
train_generator = train_datagen.flow_from_directory(
       trainDir,
       target_size=input_shape[:2],
       batch_size=64,
       class_mode='categorical')
validation_generator = test_datagen.flow_from_directory(
       validationDir,
       target_size=input_shape[:2],
       batch_size=64,
       class_mode='categorical')
test_generator = test_datagen.flow_from_directory(
       testDir,
       target_size=input_shape[:2],
       batch_size=64,
       class_mode='categorical')
```

```
vggModel =
tf.keras.applications.VGG16(input_shape=input_shape,include_top=False,weights='
imagenet')

#Make some of the final layers of the base model trainable
if fine_tune > 0:
 for layer in vggModel.layers[:-fine_tune]: layer.trainable = False
else:
 for layer in vggModel.layers: layer.trainable = False

#Build augmented model from base model
inputs = tf.keras.Input(shape=input_shape)
x = vggModel(inputs, training=False)
x = tf.keras.layers.GlobalAveragePooling2D()(x)
x = tf.keras.layers.Flatten(name="flatten")(x)
x = tf.keras.layers.Dense(layer1, activation='relu')(x)
x = tf.keras.layers.Dense(layer2, activation='relu')(x)
x = tf.keras.layers.Dropout(dropout)(x)
outputs = tf.keras.layers.Dense(80, activation='softmax')(x)

model1 = tf.keras.Model(inputs, outputs)

model1.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate),
               loss='categorical_crossentropy',
              metrics=['accuracy',
                       tf.keras.metrics.TopKCategoricalAccuracy(k=3),
                       tf.keras.metrics.Precision(),
```

```
                        tf.keras.metrics.Recall()
                        ])

history = model1.fit(
   train_generator,
   epochs=35,
   validation_data=validation_generator,

steps_per_epoch=int(math.ceil(train_generator.n/train_generator.batch_size)),

validation_steps=int(math.ceil(validation_generator.n/validation_generator.batc
h_size)),
   callbacks=[PlotLossesKeras()])

eval_loss, eval_accuracy, acc3, precision, recall =
model1.evaluate(validation_generator, verbose=1)
print("Accuracy: {:.2f}%".format(eval_accuracy * 100))
print("Loss: {}".format(eval_loss))
print("k = 3 Accuracy: {:.2f}%".format(acc3 * 100))
print("Precision: {}".format(precision))
print("Recall: {}".format(recall))
```

I trained simple, non-tuned versions of all the 4 models and realized that VGG-16 gave the most promising initial performance on the test set of around 20%. So, I decided to channel most of my efforts into tuning parameters for VGG-16.

The parameters I tuned for the models were:

1. Target size of the image fed to the model
2. Shear range
3. Zoom range
4. Dropout rate of one of the layers
5. Number of nodes in the first augmented layer
6. Number of nodes in the second augmented layer
7. Number of layers in the base model to fine-tune, i.e re-training some of the final layers of the base model
8. Epochs

Some changes that helped were increasing the target size resolution (started at (50, 50)), and reducing the number of nodes in the augmented layers because the model seemed to be overfitting, as evidenced by large discrepancies in the training and validation accuracies during training.

After many, many rounds of tweaking combinations of parameters, I was able to bring the accuracy of the VGG-16 to past 50%! This was my target, and it was difficult to increase the accuracy much beyond this.

For perspective, a model that randomly guessed would be accurate 1/80th of the time or 1.25% of the time. This model is 40 times better than a random-chance model, which is definitely significant.

Here is a summary of stats for the model:

Accuracy: 50.38%
Accuracy with the top 3 predictions: 71.88%
Precision: 0.57
Recall: 0.46

I saved this model for future use as a .h5 file:

```
model1.save(projectDir + '/' + 'Models/VGG16ModelV9.h5')
```

Which could be loaded again as:

```
model1 = tf.keras.models.load_model(projectDir + '/' +
'Models/VGG16ModelV9.h5')
```

After shortlisting the best model, I decided to play around with a little more. The first thing I did was actually inspect the predictions it was making on the test set. And I noticed some very interesting patterns:

1. The model had a lot of trouble recognizing "aloo shimla mirch", getting only 1 prediction correct. But if you look at the other predictions, they are actually quite reasonable - it suggests other "aloo"/potato related dishes like "aloo matar," "dum aloo," or "aloo gobi". So clearly, the model has at least been able to understand that this is some kind of a potato dish
2. The model seems to confuse "anarsa" and "ariselu" sometimes too, which is also reasonable because they are very similar looking foods
3. Butter chicken was very often confused with chicken tikka masala, also fair. Chicken tikka also confused with chicken tikka masala a couple of times
4. In the Indian sweet arena, the model sometimes mistakes imarti with jalebi and vice versa - they are also extremely similar looking foods!
5. It mistook naan for chapati once

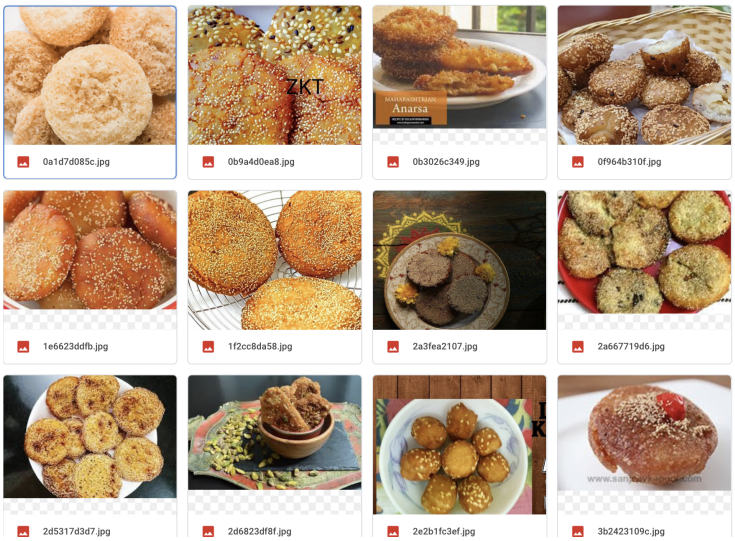

Figure 4.1 - Anarsa Images

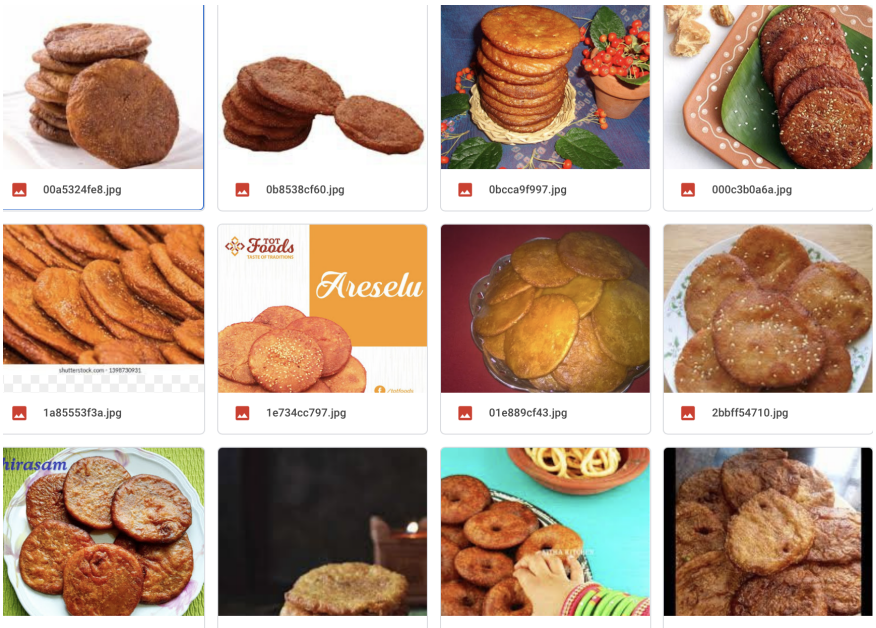

Figure 4.2 - Ariselu Images

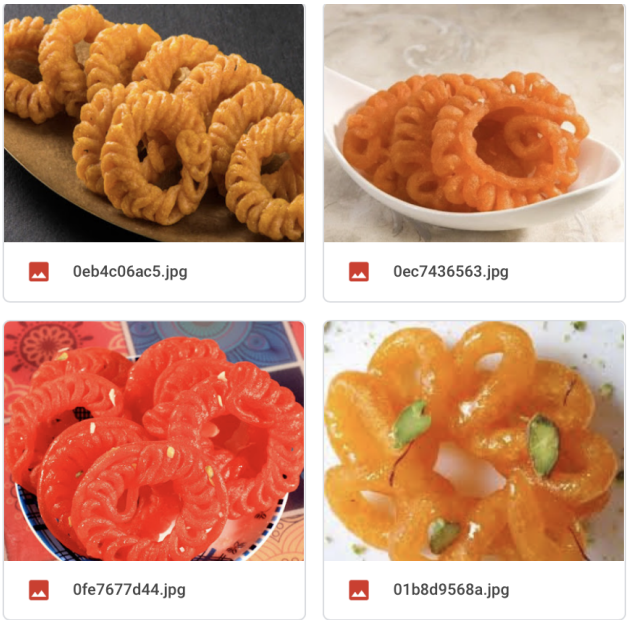

Figure 4.3 - Imarti Images

Figure 4.4 - Jalebi Images

A final thing I did was try to set up a pipeline for the model to make predictions on future images. This was somewhat annoyingly and unnecessarily challenging in my opinion, because Keras models are very rigid in terms of input shape. Since the model was trained in batches, calling .predict on the model requires tensors of rank 4 only, which means you'd have to extend the dimensions of the image before calling predict on it.

There were also other obstacles like manually doing all the needed preprocessing tasks, so I decided to just create a directory with the singular image in a subdirectory of its class and then have an ImageDataGenerator flow the image from that directory with a batch size of 1, similar to how the models were trained.

So, to predict future images, you'd just need to put the image in a folder with the class label (obviously this label is just for verification and is extracted from the generator) and then call predict on a generator that flows that 1 image. The code to set that up was as follows (images were just extracted from the test set for now)

```
generatorDir = os.path.join(projectDir, 'image')
os.mkdir(generatorDir)
imagePath = ""
```

```
def randomImage():
 global imagePath
 global generatorDir
 #Remove old image if it exists
 if pathlib.Path(imagePath).is_file():
   os.remove(imagePath)
   os.rmdir(os.path.dirname(imagePath))

 randomCategory = np.random.choice(categories, 1)[0]
 imageDir = os.path.join(generatorDir, randomCategory)
 os.mkdir(imageDir)
 randomInt = np.random.randint(10)
 imagePath = f"{generatorDir}/{randomCategory}/image.jpg"
 imageio.imwrite(imagePath,
imageio.imread(f"{testDir}/{randomCategory}/image{randomInt}.jpg"))
 image_datagen = ImageDataGenerator(
   preprocessing_function = vgg16.preprocess_input,
   rescale=1./255)
 image_generator = image_datagen.flow_from_directory(
   generatorDir,
   target_size=(240, 240),
   batch_size=1,
   class_mode='categorical',
   shuffle=False
 )
 return imagePath, randomCategory, image_generator
```

```
imagePath, category, image_generator = randomImage()
image = imageio.imread(imagePath)
```

```
prediction = best_model.predict(image_generator)
prediction = categories[prediction[0].argmax(axis=-1)]
print(f"Truth: {category}, Prediction: {prediction}")
plt.imshow(image)
```

Sample result:

```
Found 1 images belonging to 1 classes.
Truth: chicken_tikka, Prediction: chicken_tikka
<matplotlib.image.AxesImage at 0x7fd4aeaa3e90>
```

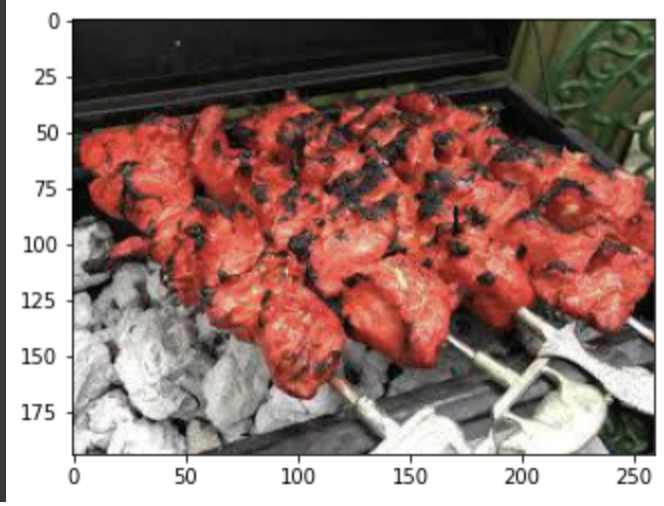

Figure 4.5 - Sample result of the random image predictor

Conclusion and scope for improvement

Overall, I'm generally pleased with how the project panned out and am satisfied with the results of the best performing model. 51% is a good accuracy in the context of 80 possible classes, and it means the model is (just barely) more often right than wrong. Beyond this tangible achievement, the project also taught me a lot of important things:

1. Setting up the right workflow for a Deep Learning project
2. Understanding how to use the Tensorflow/Keras API to train a CNN
3. The nuances of training neural networks - how to tackle overfitting, what parameters to tweak to measurably improve performance, etc
4. Appreciating the preprocessing involved with image classification, such as all the morphological transformations or base model preprocessing required before a network is ready to train on it

All this being said, however, there is certainly scope to improve the model or find better models, even. There are definitely more systematic ways to perform hyperparameter tuning with Keras (Keras Tuner), so one could definitely explore using the API for that to perhaps discover better combinations of hyperparameters that would lead to higher accuracies. Other base models could be explored as well - there are plenty of CNN architectures out there and I've only sampled a small subset of them - perhaps some of them are more conducive to a learning task like this. One could even venture outside the realm of CNNs and explore other model types. I've wondered if gradient boosting might work well for this project. Perhaps vision transformers? The point is, this is by no means the pinnacle of progress with this learning task, but I would hope that this excursion would be a good starting point for others to develop even more powerful models using insights from this capstone!