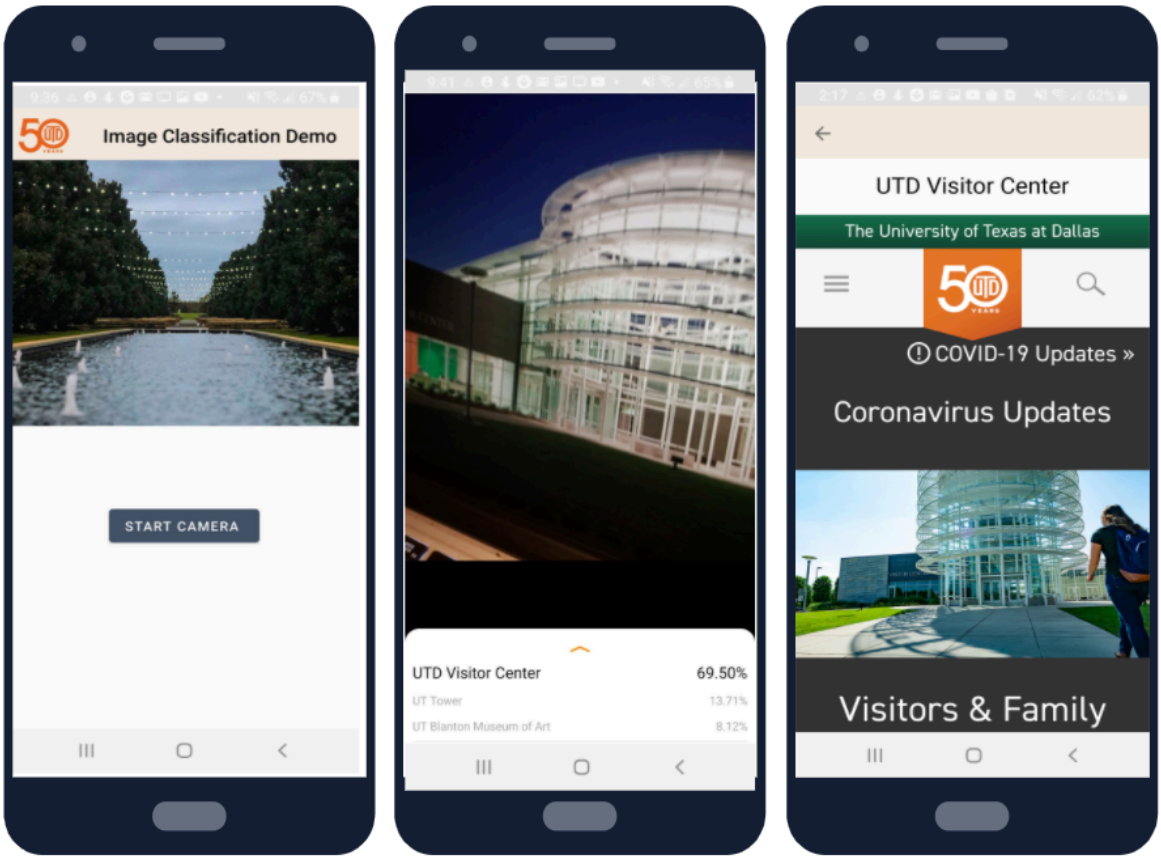
University of Texas at Dallas Artificial Intelligence Project

University Landmarks Image Classification on Android

Jairam Manikandan
7/13/20 - 7/29/20



Introduction

To quickly introduce myself before guiding my project, my name is Jairam Manikandan and I'm a rising high school senior from Austin, Texas. In brief, I created Android application that identifies University of Texas at Austin and University of Texas at Dallas landmarks using image classification. Specifically, my application identifies a total of 5 classes: UT's Tower, UT's Littlefield Fountain, UT's McCombs School of Business, UT's Blanton Museum of Art, and UTD's Visitor Center.

As seen by the introductory visual above, my project's key factors revolve around Android devices. A detailed explanation will be seen in the later pages of this paper, but I have included this graphic to make my project easier to understand and follow. In the example above, my application was able to classify UTD's Visitor Center with 69.5% accuracy. This was the result over one specific frame, so the number fluctuates between 70% and 90% when the user opens the camera for a couple of seconds long. Essentially, my app classifies each frame of the camera and the result of this are the percentages. Once the accuracy percentage reaches over 80%, the user will then be navigated to another page in my app which includes more information about the landmark.

Project Overview

In order to create a successful image classifier on Android, I followed a series of steps that guided me along my project to ultimately identify landmarks. These steps are for context and will be explained in detail in the following pages.

1. Identifying how my application would benefit students or people at various universities who may not be familiar with a campus.
2. Familiarizing myself with prior applications of image classification on mobile devices and understanding the topics of image classification, convolutional neural networks, and neural networks.
3. Familiarizing myself and understanding the machine learning topic of transfer learning, feature extraction, and fine tuning a pre-trained model. The pre-trained model I have used for my Android application is VGG16 by Oxford.
4. Utilizing the pre-trained VGG16 model that was trained on the ImageNet dataset by creating a new model. My new model use and aspect of transfer learning and fine tuning. This step also required me to learn about VGG16's architecture and work my new model around the structure of the pre-trained model.
5. Utilizing libraries and frameworks such as Keras in Tensorflow, Tensorflow Lite, and Android Studio to create my project, as well as other notable tools such as Google Colab.
6. Evaluating my new model that utilized transfer learning and fine-tuning. This process required me to showcase various graphs and my analysis on them.
7. by showing how easy it is to upscale the scope of my project by introducing new factors Representing the "bigger picture" of my project such as IOS, new classes to be identified, and new universities to add on to my current project.

Importance of University Landmark Classification on Mobile Devices

My friends and family members always recall finding an intergerning landmark on a university's campus only to find no information about it anywhere around the landmark. I faced this issue many times as well. For example, I still remember when I first saw the University of Texas at Austin's Blanton Museum of Art. The building was unlike anything I've seen before in a university and seeking to find more information about the museums only left me re-reading the brief sign in front of the landmark. It was times like these were my friends, family and I needed a mobile device that can provide valuable information, history, and significance about various landmarks on campus

in a matter of seconds. This facilitated the goal of my project to create an application that can identify and provide knowledge to a user about any landmark he or she may be interested about.

Literature Review

The topic of image classification has a lot of papers written bout it and there are many online resoucres that explain the essence of this field of classification. However, when it comes to implementing this on mobile devices, there aren't nearly as much credible papers available. Most mobile developers used ML kits with Android frameworks such as Firebase in the past, but now with a trend of mobile intelligence, developers are starting to integrate machine learning frameworks such as TensorFlow and OpenCV (Hindawi's *https://bit.ly/3fgpt2I* and Aditya Timmaraju's and Anirban Chatterjees' *https://tinyurl.com/y6boh9ol* respectively are a few examples). In terms of university landmark classification, I was unable to find credible sources that created successful android applications.

Image Classification

Classifying various objects and pictures seem innate for us humans, but for computers, it is quite the opposite and complex. In the vast field of computer vision, image classification plays a crucial role as there are a plethora of applications. For example, labeling x-rays for broken forearms, classifying handwritten digits, or even autonomous driving which identifies pedestrians, stop signs, and vehicles are all examples of image classification in our current generation. More specifically, this type of classification is the process of taking an input and resulting in a specific class, or probability of that class occurring. In my project's scenario, an image a university landmark would be the input, while the output would be the percentage of probability (refer to the above 65.9% probability on UTD's Visitor Center). While there are numerous techniques of utilizing image classification, I used neural networks and convolutional neural networks for my project.

Neural Networks and Convolutional Neural Networks

Neural networks are the intricates under the hood of computer vision related image classification applications. These networks are inspired by the structure of biological neurons in our brains, and hence the name neural network. Mimicking a human brain, a basic neural network would include an input, hidden, and output layer. As inputs are fed into the layers, the network learns information such as weights to ultimately come up with predicating outcomes. However, this process consists of information passed into layers in a single file fashion which may not be optimal for all

applications. Namely, image classification is frequently used with convolution neural networks (CNNs).

For example, an image of UT's Tower that is sent into the input layer of a CNN would gradually have its features extracts as the image travels further across the CNN's layers. Neurons earlier in the network would pick up on features such edges and corners while the later layers would pick on features such as windows, color, and structure of the Tower. The network convolves over a given image with filter for each image's recognizable features. Eliminating the need for manual feature extraction by using feature detectors and maps, convolutional neural networks were revolutionary when introduced to computer vision.

Transfer Learning and Fine Tuning

An important new idea in machine learning is transfer learning, which just like neural networks, are related to how the human brain works. For example, we humans are able to transfer knowledge across various tasks. What we learn about one task, we utilize this knowledge to solve related tasks. The more related the tasks are, the easier it is for us to transfer knowledge. Some example includes learning how to drive a car and then learning how to ride a motorcycle, or even learning about math in school and then learning another subject. Traditional machine learning has so far been taught to work in isolation and create models for specific tasks. Unless one has the resources for this such as a high-end GPU, this method is just not feasible. This is where transfer learning comes into play as it is the idea that machines can learn certain aspects of a previous model (such as weights and features) and utilize this information on a new model. There are different transfer learning techniques and strategies, but the two most popular methods are using a pretrained model as a feature extractor, or fine tuning a pretrained model.

Many pretrained image classification models have layered neural and convolutional networks that, as I explained before, pick up on features of an input image in a hierarchy representation (refer back to CNN UT Tower example). These layers are connected to a last layer (usually a fully connected layer) that can identify the input based on the classes it was trained on. In terms of feature extraction, the last layer can be removed of the pretrained model and since the layers work on a hierarchy representation of features, this method would result the pretrained model to extract all the relevant features of the input image. In terms of fine tuning, this method is more involved than just replacing the final classification layer like feature extraction, but also selectively retain certain layers. As discussed earlier, the initial layers in a network pick up on generic features of an input image, while the later layers pick up on more specific features related to the input image. With this

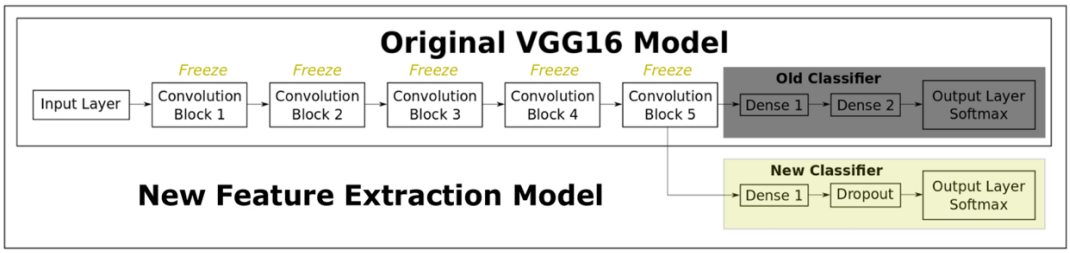
information, fine tuning is the process of freezing (or fixing weights) certain layers while retraining (or fine tuning) the rest of the layers in the model to better tailor to an image.

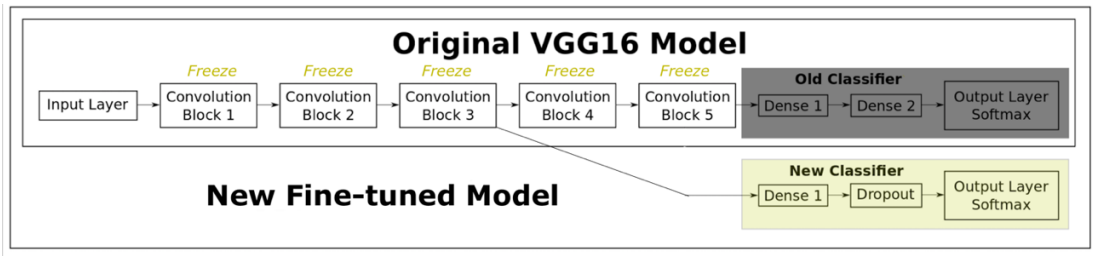
With understanding the basic concepts of my project, I now want to discuss the methods of transfer learning I implemented on VGG16 and provide an overview of the libraries and framework resources I utilized to create my project.

Utilizing VGG16 and ImageNet

The pre-trained model I chose from my project is VGG16 (also called OxfordNet). This model has a convolutional neural network architecture and was named after the Visual Geometry Group you developed this model from Oxford University. VGG16 was used to win the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) competition in 2014, due to its state-of-the-art performance. With over 14 million images and 1000 image classes, ImageNet was used as the dataset to train VGG16 in 2014.

In terms of transfer learning, I experimented with the two most popular methods: feature extraction and fine tuning on VGG16 to see with method would yield the highest accuracy and lowest loss for my project of identifying university landmarks. Below is a visual representation of the original model and the new model I created. For feature extraction, I have frozen and fixed the weights of the five previous convolutional blocks and created a new classifier tailored to my own dataset of university landmark images. For fine tuning the pre-trained model, I have frozen and fixed convolution block one through three, while utilizing the same new classifier I created during feature extraction. The method would fine tune and retrain the weights for VGG16 to better suit my own dataset.





Libraries, Frameworks, and Tools

The main resources I used for my project were Tensorflow, Keras, Tensorflow Lite, Android Studio, Google Drive and Google Colab. Although the RAM and GPU limits on colab were limited, and I had my runtime disconnected many times, my project was relatively intuitive when using Keras and Tensorflow.

Model Creation Steps

Data Collection

Since I wasn't able to find a dataset online regarding university landmarks, I went to the University of Austin to gather images for my very own dataset and took a total of 18051 pictures of UT's Tower, Littlefield Fountain, Blanton Museum of Art, and McCombs School of Business. I also downloaded close to 700 pictures of UTD's Visitor Center for the fifth class my project could classify. This totaled to around 70GB of storage needed, so I utilized google drive to store all my files and later mount drive into colab to access my files.

Data Pre-processing

Since I had an imbalance of images in specific classes, I performed augmentation. This process was done using the ImageDataGenerator() function in Keras to randomly augment images and save them to a certain directory of my choice.

```
[ ]  # helper function
     def plots(ims, figsize=(12,6), rows=1, interp=False, titles=None):
         if type(ims[0]) is np.ndarray:
             ims = np.array(ims).astype(np.uint8)
             if (ims.shape[-1] != 3):
                 ims = ims.transpose((0,2,3,1))
         f = plt.figure(figsize=figsize)
         cols = len(ims)//rows if len(ims) % 2 == 0 else len(ims)//rows +1
         for i in range(len(ims)):
             sp = f.add_subplot(rows,cols,i+1)
             sp.axis('Off')
             if titles is not None:
                 sp.set_title(titles[i], fontsize = 16)
             plt.imshow(ims[i], interpolation=None if interp else 'none')

[ ]  gen = ImageDataGenerator(rotation_range=10,width_shift_range=0.1,
         height_shift_range=0.1, shear_range=0.15, zoom_range=0.1,brightness_range=[0.5, 1.5],
         channel_shift_range=10., horizontal_flip=True)

[ ]  image_path = '/content/drive/My Drive/AI/AI_Dataset/UTD Visitor Center/augment_pic.jpeg'

[ ]  image = np.expand_dims(ndimage.imread(image_path),0)
     plt.imshow(image[0])

     /usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:1: DeprecationWarning: `imread` is deprecated!
     `imread` is deprecated in SciPy 1.0.0.
     Use ``matplotlib.pyplot.imread`` instead.
       """Entry point for launching an IPython kernel.
     <matplotlib.image.AxesImage at 0x7f7c22c4ddd8>
```

```
[ ]  #DONE

     #dir = '/content/drive/My Drive/AI/AI_Dataset'
     #to_dir = '/content/drive/My Drive/AI/AI_Dataset/UT Blanton Museum of Art/result'
     #aug_iter = gen.flow_from_directory(dir, save_to_dir=to_dir, classes=['UT Blanton Museum of Art'])
```

I then needed to create subdirectories in my main directories and split the images based on a 80/20 percent ratio of training and testing images respectfully. This was done using NumPy arrays and essentially splitting the array based on the ratios I discussed earlier.

Splitting data into test and train directories

```
##DONE##

split data into train and test directories
import shutil
import random

root_dir = '/content/drive/My Drive/AI'
classes_dir = ['/UT Blanton Museum of Art',
               '/UT McCombs',
               '/UT Tower',
               '/UT Tower Fountain',
               '/UTD Visitor Center']

test_ratio = 0.20

for cls in classes_dir:
    os.makedirs(root_dir +'/train' + cls, exist_ok=True)
    os.makedirs(root_dir +'/test' + cls, exist_ok=True)

    # Creating partitions of the data after shuffeling
    src = root_dir + '/AI_Dataset' + cls # Folder to copy images from

    allFileNames = os.listdir(src)
    np.random.shuffle(allFileNames)
    train_FileNames, test_FileNames, val_FileNames = np.split(np.array(allFileNames),
                                                              [int(len(allFileNames)* (1 - test_ratio)),
                                                               int(len(allFileNames))])

    train_FileNames = [src+'/'+ name for name in train_FileNames.tolist()]
    test_FileNames = [src+'/' + name for name in test_FileNames.tolist()]

    print('Total images: ', len(allFileNames))
    print('Training: ', len(train_FileNames))
    print('Testing: ', len(test_FileNames))

    # Copy-pasting images
    for name in train_FileNames:
        shutil.copy(name, root_dir +'/train' + cls)

    for name in test_FileNames:
        shutil.copy(name, root_dir +'/test' + cls)
```

```
[ ]
# get train and test directories
train_dir = '/content/drive/My Drive/AI/train'
test_dir = '/content/drive/My Drive/AI/test'
```

Input Pipeline

Pre-trained models accept various shapes of tensors as input, and my project's case VGG16 accepts images of shape 224 x 224 x 3. I then used ImageDataGenerator() again to create my data compatible enough to input into VGG16's neural architecture, as well as rescaling the pixel values of my training and images. Further, this creates batches of images that can be used by Tensorflow. Regarding batches, my batch size was relatively low due to the large model size of VGG16 and a higher batch size would cause both RAM and GPU shortage. I then directed the data generators to read my images from their respective directories passing in parameters into this function such as Boolean data shuffling and setting the size of which the generator needs to resize my images.

Input Pipeline

```
[ ]  # find shape of the tensors expected as input by the pre-trained VGG16 model
     input_shape = model.layers[0].output_shape[0][1:3]
     input_shape
```

```
(224, 224)
```

```
[ ]  # further augmentation
     datagen_train = ImageDataGenerator(
           rescale=1./255,
           rotation_range=180,
           shear_range=0.1,
           zoom_range=[0.9, 1.5],
           horizontal_flip=True,
           vertical_flip=True,
           fill_mode='nearest')

     datagen_test = ImageDataGenerator(rescale=1./255)
```

```
[ ]  batch_size = 32
```

```
[ ]  if True:
         save_to_dir = None
     else:
         save_to_dir='/content/drive/My Drive/temp/'
```

```
[ ]  # reading data from drive
     generator_train = datagen_train.flow_from_directory(directory=train_dir,
                                                         target_size=input_shape,
                                                         batch_size=batch_size,
                                                         shuffle=True,
                                                         save_to_dir=save_to_dir)
```

```
Found 15159 images belonging to 5 classes.
```

```
[ ]  # reading data from drive
     generator_test = datagen_test.flow_from_directory(directory=test_dir,
                                                       target_size=input_shape,
                                                       batch_size=batch_size,
                                                       shuffle=False)
```

```
Found 3792 images belonging to 5 classes.
```

```
[ ]  # testing steps size
     steps_test = generator_test.n / batch_size
     steps_test
```

```
118.5
```

```
[ ]  # file paths for later
     image_paths_train = path_join(train_dir, generator_train.filenames)
     image_paths_test = path_join(test_dir, generator_test.filenames)
```

Adjusting My Class Weights

The images I took of the various landmark were not the same number for each class or landmark. In order for my new model to generalize my data to its maximum potential I adjusted each class's weight using skikit-learn.

Adjusting class weights due to my imbalanced dataset

```
[ ]  # balancing dataset
     from sklearn.utils.class_weight import compute_class_weight

     class_weights = compute_class_weight(class_weight='balanced',
                                          classes=np.unique(cls_train),
                                          y=cls_train)
     class_weight_dict = dict(enumerate(class_weights))
```

```
[ ]  class_weight_dict
```

```
{0: 3.661594202898551,
 1: 0.5696730552423901,
 2: 0.6479589655909382,
 3: 0.7791827293754818,
 4: 6.906150341685649}
```

Feature Extraction

I first crated an instance of VGG16 using Keras. The model came with its weights learned on ImageNet and contained a convolutional section and a fully connected dense part used for classification of classes.

Pre-trained Model: VGG16

```
[ ]  # download full VGG16 model
     model = VGG16(include_top=True, weights='imagenet')
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels.h5
553467904/553467096 [==============================] - 11s 0us/step
```

The Convolutional Neural Network layers prior to VGG16's last few dense layers are great for generalizing and picking up important features. The last dense layers take these features and outputs a prediction of classes. Hence, I have created new layers such as a dropout layer to help prevent overfitting my data and a dense, fully connected layer tailoring to number of classes I had. Furthermore, activation functions such as ReLU and Softmax were utilized.

```
[ ]  # get last layer
     transfer_layer = model.get_layer('block5_pool')
     transfer_layer.output
```

```
<tf.Tensor 'block5_pool/Identity:0' shape=(None, 7, 7, 512) dtype=float32>
```

```
[ ]  # New model that has everything from input to conv block 5
     conv_model = Model(inputs=model.input,
                        outputs=transfer_layer.output)
```

```
[ ]  # adding more layers
     new_model = Sequential()

     # Add the convolutional part of the VGG16 model from above.
     new_model.add(conv_model)

     # Flatten the output of the VGG16 model because it is from a
     # convolutional layer.
     new_model.add(Flatten())

     # Add a dense (aka. fully-connected) layer.
     # This is for combining features that the VGG16 model has
     # recognized in the image.
     new_model.add(Dense(1024, activation='relu'))

     # Add a dropout-layer which may prevent overfitting and
     # improve generalization ability to unseen data e.g. the test-set.
     new_model.add(Dropout(0.5))

     # Add the final layer for the actual classification.
     new_model.add(Dense(num_classes, activation='softmax'))
```

To avoid a great risk of overfitting, I fixed the weights of the previous layers only updated the weights of the new classification layers I added. This was a necessary step to consider as the new classification layers I added can propagate back into the previous layers and distort the weights of these layers. I then compiled my new model to take into account the changes I have made learn which parameters I have set such as the specific metrics and learning rate.

```
[ ]  # disable base
     conv_model.trainable = False
     for layer in conv_model.layers:
         layer.trainable = False
     print_layer_trainable()
```

```
False:  input_1
False:  block1_conv1
False:  block1_conv2
False:  block1_pool
False:  block2_conv1
False:  block2_conv2
False:  block2_pool
False:  block3_conv1
False:  block3_conv2
False:  block3_conv3
False:  block3_pool
False:  block4_conv1
False:  block4_conv2
False:  block4_conv3
False:  block4_pool
False:  block5_conv1
False:  block5_conv2
False:  block5_conv3
False:  block5_pool
```

```
[ ]  new_model.compile(optimizer=optimizer, loss=loss, metrics=metrics)
```

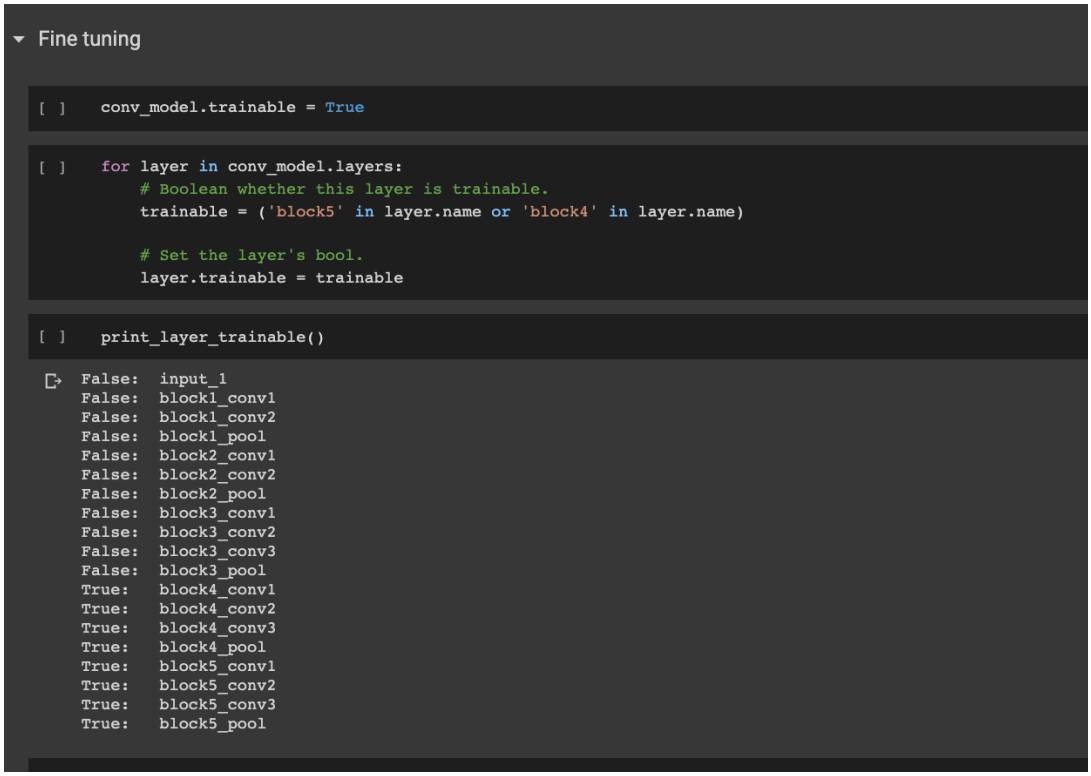
Finally, I set the number of epochs and steps per epoch I wanted my model to train for. I used the Keras function model.fit() to train my model which included my training generator, number of epochs, steps per epoch, class weight, and how I wanted to save the weights for each epoch in a sperate directory. The total run time took 12 hours to train completely.

```
[ ]  new_model.compile(optimizer=optimizer, loss=loss, metrics=metrics)

[ ]  epochs = 5
     steps_per_epoch = 237

[ ]  checkpoint_path = "/content/drive/My Drive/AI/checkpoints_tutorial_two/"
     # Create a callback that saves the model's weights
     cp_callback = tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_path,
                                                      save_weights_only=True,
                                                      verbose=1)

[ ]  history = new_model.fit(x=generator_train,
                             epochs=epochs,
                             steps_per_epoch=steps_per_epoch,
                             class_weight=class_weight_dict,
                             validation_data=generator_test,
                             validation_steps=steps_test,
                             callbacks=[cp_callback])
```

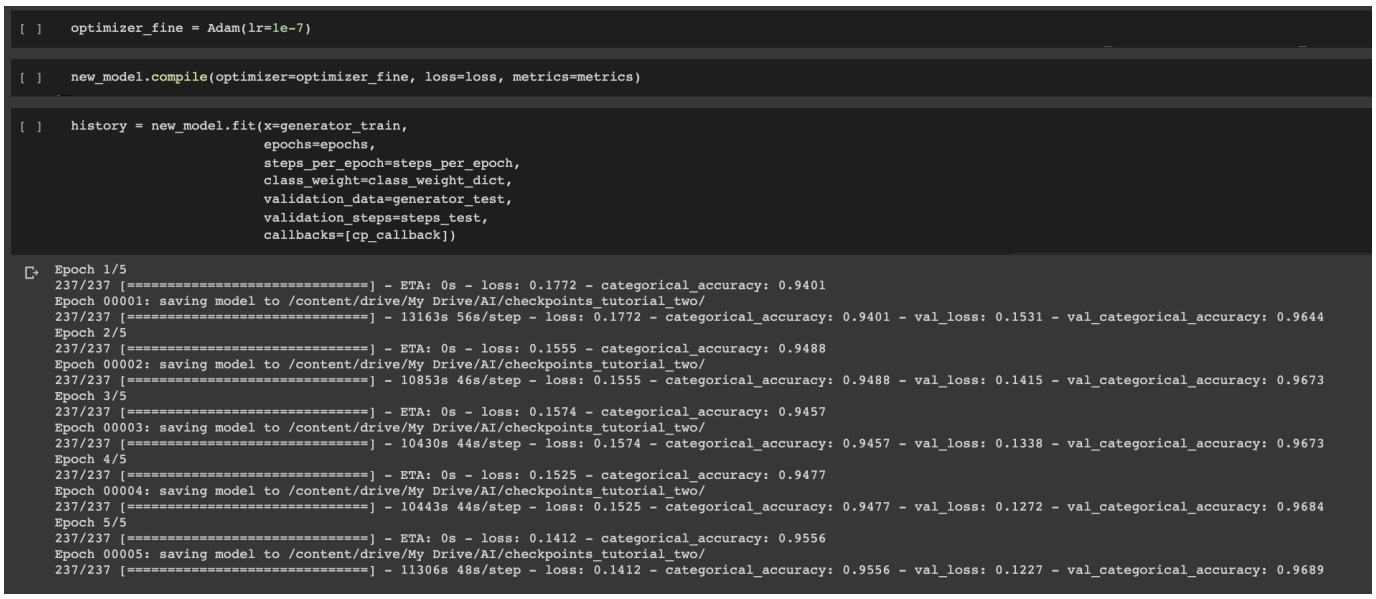
```
Epoch 1/5
237/237 [==============================] - ETA: 0s - loss: 1.0512 - categorical_accuracy: 0.6026
Epoch 00001: saving model to /content/drive/My Drive/AI/checkpoints_tutorial_two/
237/237 [==============================] - 12723s 54s/step - loss: 1.0512 - categorical_accuracy: 0.6026 - val_loss: 0.6091 - val_categorical_accuracy: 0.8499
Epoch 2/5
237/237 [==============================] - ETA: 0s - loss: 0.4586 - categorical_accuracy: 0.8514
Epoch 00002: saving model to /content/drive/My Drive/AI/checkpoints_tutorial_two/
237/237 [==============================] - 10033s 42s/step - loss: 0.4586 - categorical_accuracy: 0.8514 - val_loss: 0.4176 - val_categorical_accuracy: 0.8803
Epoch 3/5
237/237 [==============================] - ETA: 0s - loss: 0.3293 - categorical_accuracy: 0.8911
Epoch 00003: saving model to /content/drive/My Drive/AI/checkpoints_tutorial_two/
237/237 [==============================] - 10034s 42s/step - loss: 0.3293 - categorical_accuracy: 0.8911 - val_loss: 0.2675 - val_categorical_accuracy: 0.9343
Epoch 4/5
237/237 [==============================] - ETA: 0s - loss: 0.2584 - categorical_accuracy: 0.9163
Epoch 00004: saving model to /content/drive/My Drive/AI/checkpoints_tutorial_two/
237/237 [==============================] - 9821s 41s/step - loss: 0.2584 - categorical_accuracy: 0.9163 - val_loss: 0.2404 - val_categorical_accuracy: 0.9438
Epoch 5/5
237/237 [==============================] - ETA: 0s - loss: 0.2269 - categorical_accuracy: 0.9275
Epoch 00005: saving model to /content/drive/My Drive/AI/checkpoints_tutorial_two/
237/237 [==============================] - 10986s 46s/step - loss: 0.2269 - categorical_accuracy: 0.9275 - val_loss: 0.2209 - val_categorical_accuracy: 0.9494
```

Fine-tuning

It is now clear that in feature extraction, the original pre-trained model is frozen and ensures that the weights to not change. This eliminates the chance for overfitting as my new classification head will not propagate large gradients back through the VGG16 model to distort its weights. For the process of fine-tuning I have removed two convolutional blocks in VGG16 (four and five) and retrained my classification head. This allows for deeper convolution neural network layers in VGG16 to better tailor weights towards my dataset. The total run time took 10 hours to train completely.



```
Fine tuning

[ ]  conv_model.trainable = True

[ ]  for layer in conv_model.layers:
         # Boolean whether this layer is trainable.
         trainable = ('block5' in layer.name or 'block4' in layer.name)

         # Set the layer's bool.
         layer.trainable = trainable

[ ]  print_layer_trainable()

     False:  input_1
     False:  block1_conv1
     False:  block1_conv2
     False:  block1_pool
     False:  block2_conv1
     False:  block2_conv2
     False:  block2_pool
     False:  block3_conv1
     False:  block3_conv2
     False:  block3_conv3
     False:  block3_pool
     True:   block4_conv1
     True:   block4_conv2
     True:   block4_conv3
     True:   block4_pool
     True:   block5_conv1
     True:   block5_conv2
     True:   block5_conv3
     True:   block5_pool
```

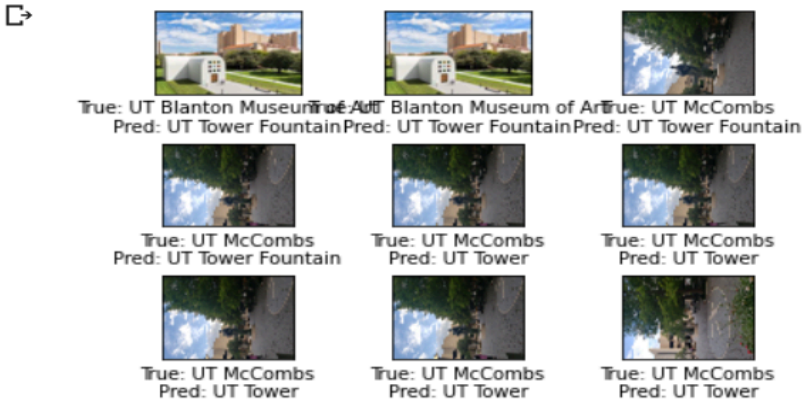


```
[ ]  optimizer_fine = Adam(lr=1e-7)

[ ]  new_model.compile(optimizer=optimizer_fine, loss=loss, metrics=metrics)

[ ]  history = new_model.fit(x=generator_train,
                             epochs=epochs,
                             steps_per_epoch=steps_per_epoch,
                             class_weight=class_weight_dict,
                             validation_data=generator_test,
                             validation_steps=steps_test,
                             callbacks=[cp_callback])

     Epoch 1/5
     237/237 [==============================] - ETA: 0s - loss: 0.1772 - categorical_accuracy: 0.9401
     Epoch 00001: saving model to /content/drive/My Drive/AI/checkpoints_tutorial_two/
     237/237 [==============================] - 13163s 56s/step - loss: 0.1772 - categorical_accuracy: 0.9401 - val_loss: 0.1531 - val_categorical_accuracy: 0.9644
     Epoch 2/5
     237/237 [==============================] - ETA: 0s - loss: 0.1555 - categorical_accuracy: 0.9488
     Epoch 00002: saving model to /content/drive/My Drive/AI/checkpoints_tutorial_two/
     237/237 [==============================] - 10853s 46s/step - loss: 0.1555 - categorical_accuracy: 0.9488 - val_loss: 0.1415 - val_categorical_accuracy: 0.9673
     Epoch 3/5
     237/237 [==============================] - ETA: 0s - loss: 0.1574 - categorical_accuracy: 0.9457
     Epoch 00003: saving model to /content/drive/My Drive/AI/checkpoints_tutorial_two/
     237/237 [==============================] - 10430s 44s/step - loss: 0.1574 - categorical_accuracy: 0.9457 - val_loss: 0.1338 - val_categorical_accuracy: 0.9673
     Epoch 4/5
     237/237 [==============================] - ETA: 0s - loss: 0.1525 - categorical_accuracy: 0.9477
     Epoch 00004: saving model to /content/drive/My Drive/AI/checkpoints_tutorial_two/
     237/237 [==============================] - 10443s 44s/step - loss: 0.1525 - categorical_accuracy: 0.9477 - val_loss: 0.1272 - val_categorical_accuracy: 0.9684
     Epoch 5/5
     237/237 [==============================] - ETA: 0s - loss: 0.1412 - categorical_accuracy: 0.9556
     Epoch 00005: saving model to /content/drive/My Drive/AI/checkpoints_tutorial_two/
     237/237 [==============================] - 11306s 48s/step - loss: 0.1412 - categorical_accuracy: 0.9556 - val_loss: 0.1227 - val_categorical_accuracy: 0.9689
```

Model Evaluation

After feature extraction, I plotted the performance metrices for this method. Over the course of my model being trained, its training and testing loss rapidly decreased significantly, while its training and testing accuracy increased gradually, eventually almost converging. This suggests that my model was a good fit for the data, as there is little generalization gap between the two loss values. The final accuracy for my model was 94.94%.

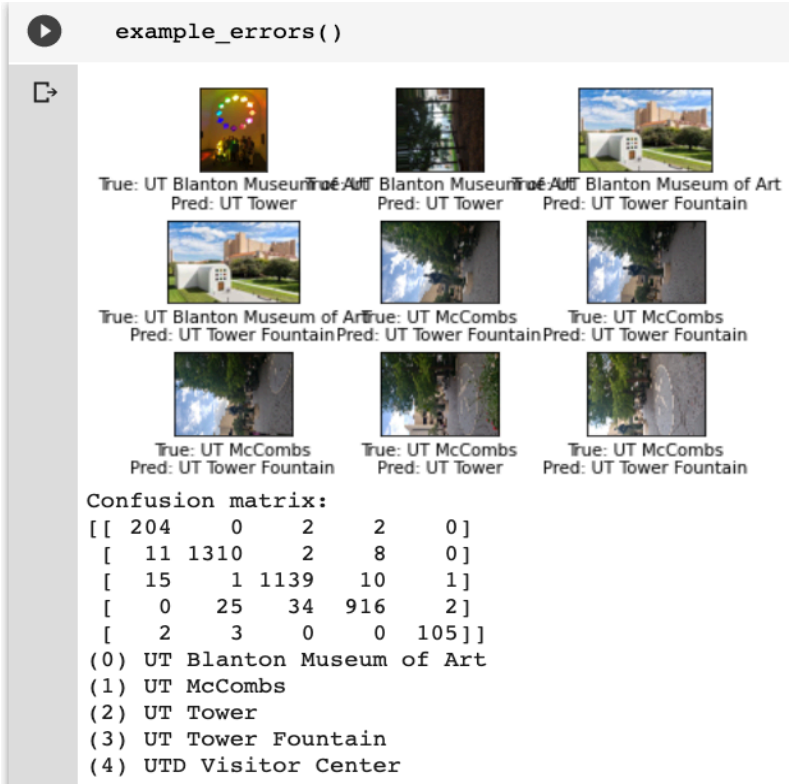
During training of feature extraction, my model mis-classified certain images as an incorrect class. An example of these images is plotted below, as well as a confusion matrix that explains what were the most commonly missed understood classes of my project. As seen, my model had confusion in identifying between Blanton Museum and the UT Tower.




```
Confusion matrix:
[[ 206    0    0    2    0]
 [  25 1293    8    5    0]
 [  43    3 1108    9    3]
 [  11   39   36  889    2]
 [   3    3    0    0  104]]
(0) UT Blanton Museum of Art
(1) UT McCombs
(2) UT Tower
(3) UT Tower Fountain
(4) UTD Visitor Center
```

After feature extraction, I plotted the performance metrices for this method and got better results than feature extraction. Over the course of my model being trained, its training and testing loss decreased slightly, but since this model was building off of feature extraction, this is expected. Moreover, the model's loss and accuracy is almost converging with tanning and testing suggesting this model was a great fit for my data. The little generation gap between these two has ultimately yielded an accuracy of 96.89%

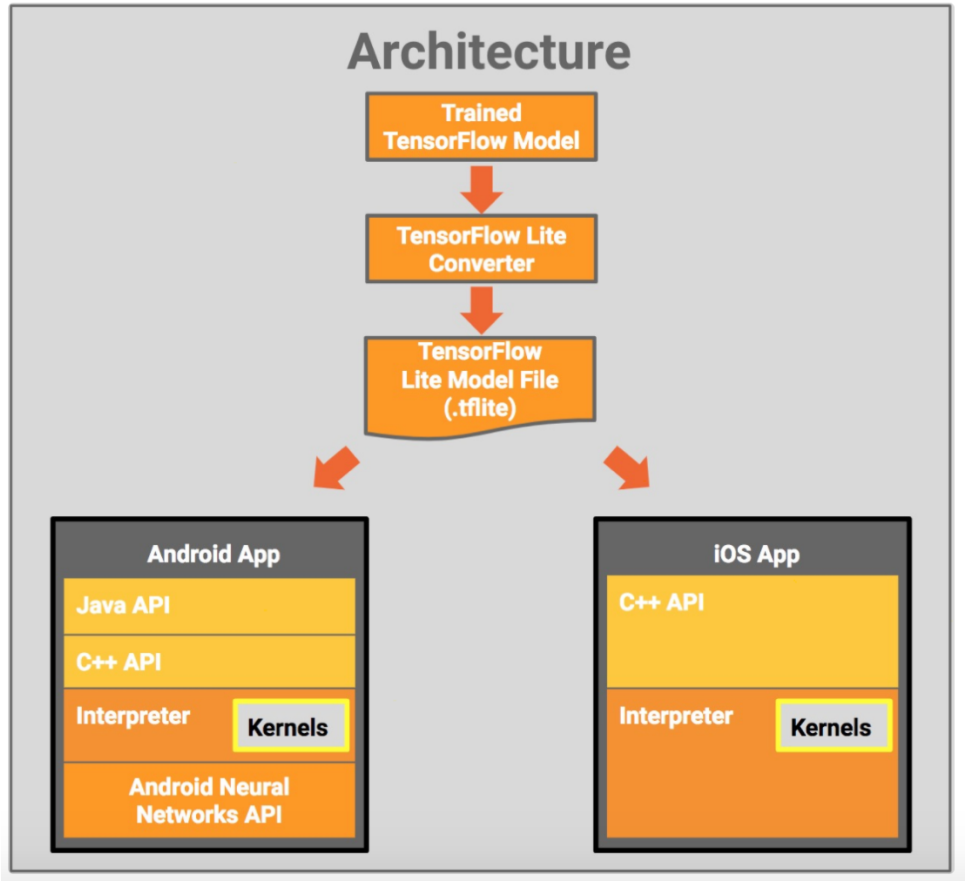
During training of fine-tuning, my model mis-classified certain images as an incorrect class. An example of these images is plotted below, as well as a confusion matrix that explains what were the most commonly missed understood classes of my project. As seen, my model had confusion in identifying between the UT Fountain and the UT Tower.



Android Implementation

Converting Model Into tf.lite Format

After my model was finished training and evaluated, the next step would be to convert my TensorFlow model into a TensorFlow Lite format that can be latter accepted by Android Studio. Presently supported on Android and IOS, TensorFlow Lite lets models run with low latency on mobile devices without needing a round trip to a server. This format is in essence a FlatBuffer-based file containing a compressed binary representation of my model.

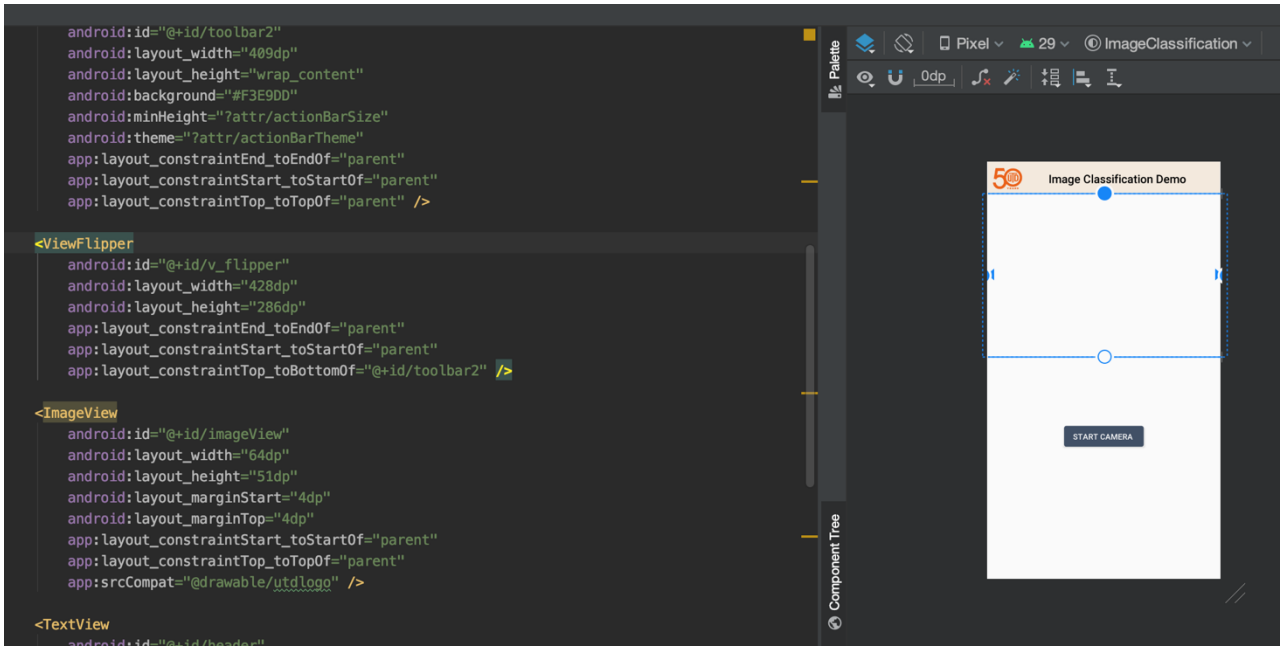



```
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT] #Dynamic range quantization 4x smaller
tflite_quant_model = converter.convert()
# Save the TF Lite model.
with tf.io.gfile.GFile('tutorial_two_model.tflite', 'wb') as f:
  f.write(tflite_quant_model)
```

Application Layouts and Activities

All Android applications consist of activities (action files) and layout (visual files) which work in harmony to produce an app. I was able to integrate the tf.lite file into Android studio and create layouts and activities.

```
// TODO: Specify model.tflite as the model file and backup_model_labels.txt as the label file
@Override
protected String getModelPath() { return "ut_model.tflite"; }

@Override
protected String getLabelPath() { return "ut_model_labels.txt"; }

@Override
protected TensorOperator getPreprocessNormalizeOp() {
  return new NormalizeOp(IMAGE_MEAN, IMAGE_STD);
}

@Override
protected TensorOperator getPostprocessNormalizeOp() {
  return new NormalizeOp(PROBABILITY_MEAN, PROBABILITY_STD);
}
```



Future Directions and Improvements

I believe my project can be highly applicable to day-to day scenarios. My goal for this project was to create a tool students at universities can potentially use to identify landmarks on the go and learn more about their history with ease. For example, students during freshman orientation can utilize my app and learn about interesting facts along with any campus tours at universities. Moreover, rising college freshman or the general public would benefit from having a classifier at their fingertips to satisfy their intriguing questions about landmarks. The scope for my project is highly expandable. I plan on making my application work with many more universities and expand the number of landmarks the app can identify. Further, an important aspect to bring would be to make my application IOS compatible as well. The technology in my project can ultimately be used in any computer vision task and is a pioneering step in the rapidly growing demand for mobile artificial intelligence.

Conclusion and Links

When I was in my freshman year of high school I first heard about AI. I always used to think this technology was only available to companies such as Facebook, Google, Amazon, or Apple. Creating a hands-on project implementing AI topics has been one of my long interests and I would like to thank Dr. Jey Veerasamay for providing me with this amazing opportunity. This course was my very first exposure to artificial intelligence and machine learning related topics, but through the teaching of Dr. Anurag Nagar, Mr. Omeed Ashtiani, and Mr. Ashutosh Senepati, these eight weeks sparked an interest in me to actively extend my skills in the vast field of data science.

Colab:
https://colab.research.google.com/drive/1dLbMKrj2IvKQK7Tsaj3yJBYD4HwhuzoM?usp=sharing

Git Hub:
https://github.com/mkdn007/Android-University-Landmark-Classifier

Android App APK:
https://drive.google.com/drive/folders/1hQ49j9axmHTqW9jU2pIEbp2E7D08gx6e

Presentation:
https://docs.google.com/presentation/d/1BvlCh3Blak2GbB5EC02DfkF4rJKbyP_3VNtnPvkkkZc/edit?usp=sharing

Dataset:
https://drive.google.com/drive/folders/1jNHG5D-1z0wcVHaOz3D50FHChapleMhy