**Image Classification App for Custom Dataset**

**Prepared By: Sayak Dutta (EEE final year NIT Sikkim)**

**Introduction**

Welcome to my Image classifiier app for Flowers! Imagine stepping into a beautiful garden, surrounded by vibrant flowers of all colors and shapes. With our app, you can now unlock the power of artificial intelligence to identify and classify these enchanting blooms with just a tap of your finger.

The app has been meticulously trained on a custom flowers dataset, carefully curated to capture the diverse beauty of nature. From delicate daisies to radiant roses, from cheerful sunflowers to graceful tulips, and from whimsical dandelions to a variety of other captivating blooms, our dataset encompasses a splendid array of flower species.

Each flower class within the dataset is represented by numerous high-quality images, showcasing the unique characteristics and intricate details of the petals, leaves, and stems. These images have been carefully collected and organized into respective subdirectories, creating a structured and comprehensive dataset that serves as the foundation for our image classification model.

To enhance the performance and accuracy of our model, I have employed advanced techniques such as data augmentation and preprocessing. By augmenting the dataset with variations in rotation, translation, zoom, and flipping, we ensure that our model can effectively handle different orientations and viewpoints of the flowers. Furthermore, a normalization layer has been incorporated to standardize the pixel values, enabling the model to process the images consistently and efficiently.

The model architecture is based on the highly acclaimed MobileNetV3 Small 100 224, which has been fine-tuned and tailored specifically to our custom flowers dataset. This state-of-the-art architecture leverages the power of deep learning to extract intricate features from the input images, enabling accurate classification and recognition of the flower species.

The journey of my app doesn't end with training the model. I have worked diligently to integrate the model into a user-friendly Android app, ensuring a seamless and intuitive experience for flower enthusiasts and nature lovers alike. By harnessing the power of TensorFlow Lite, our app brings the world of artificial intelligence to your fingertips, allowing you to embark on a delightful exploration of the floral realm.

In the following sections, we will delve into the details of our dataset, data augmentation techniques, model architecture, and the implementation of the TensorFlow Image Classification (TFIC) Java code. Additionally, we will showcase the integration of the model into the Android app, providing real-time image recognition and a captivating user interface.

Get ready to embark on a visual journey through the enchanting world of flowers, as we unveil our Image Classification App for Custom Flowers. Let the beauty of nature unfold before your eyes, guided by the intelligence of our app's sophisticated algorithms.

### Dataset Structure

The custom flowers dataset used in this project is organized in a structured and visually appealing manner, allowing for easy access and exploration of the different flower classes. The custom flowers dataset follows a directory structure similar to the flower photos dataset. Each class has its own subdirectory, and within each subdirectory, there are multiple image files representing the respective flower class. The dataset structure is as follows:

* daisy
  + 100080576\_f52e8ee070\_n.jpg
  + 14167534527\_781ceb1b7a\_n.jpg
  + ...
* dandelion
  + 10043234166\_e6dd915111\_n.jpg
  + 1426682852\_e62169221f\_m.jpg
  + ...
* roses
  + 102501987\_3cdb8e5394\_n.jpg
  + 14982802401\_a3dfb22afb.jpg
  + ...
* sunflowers
  + 12471791574\_bb1be83df4.jpg
  + 15122112402\_cafa41934f.jpg
  + ...
* tulips
  + 13976522214\_ccec508fe7.jpg
  + 14487943607\_651e8062a1\_m.jpg
  + ...

**Data Augmentation and Preprocessing**

Data augmentation is a powerful technique used to artificially increase the size and diversity of a training dataset. By applying various transformations to the original images, we can create new, slightly modified versions of the samples. This helps to expose the model to a wider range of variations and improves its ability to generalize and perform well on unseen data.

In the context of our custom flowers dataset, we have employed several data augmentation techniques to enhance the model's learning process. These techniques introduce controlled variations to the images, simulating real-world scenarios and making the model more robust. Let's delve into the specific augmentations used:

1. **Random Rotation:** Images are randomly rotated by up to 40 degrees. This mimics different orientations at which the flowers might appear in the wild, ensuring that the model learns to recognize them from various angles.
2. **Random Translation:** Horizontal and vertical translations are applied to the images, simulating slight shifts in the position of the flowers within the frame. This helps the model to become invariant to small displacements and improves its ability to detect objects in different positions.
3. **Random Zoom:** A random zoom effect is applied to the images, enlarging or shrinking them within a certain range. This introduces variations in the scale of the flowers and enables the model to handle objects of different sizes.
4. **Random Flip:** Images are horizontally flipped with a certain probability. This mirrors the flowers' appearance and helps the model to learn from both original and mirrored versions of the images, enhancing its understanding of symmetrical features.

In addition to data augmentation, the images undergo preprocessing steps to ensure standardized input for the model. The pixel values of the images are normalized using a rescaling operation, bringing them into the range of [0, 1]. This normalization facilitates consistent and efficient computation within the model, ensuring that the input data is in a suitable format for accurate inference.

By augmenting the dataset and preprocessing the images, we create a more diverse and representative training environment for our model. This enables it to learn robust and generalized patterns from the data, leading to improved performance when classifying unseen flower images.

**Model Architecture and Training:**

The model architecture used for this image classification app is based on MobileNetV3 Small 100 224, which is a state-of-the-art convolutional neural network (CNN) architecture. The model has been specifically designed to achieve high accuracy while being lightweight and efficient in terms of computational resources.

The MobileNetV3 Small 100 224 architecture consists of a feature extraction base and a classification head. The feature extraction base is responsible for extracting meaningful features from the input images, capturing their distinctive patterns and characteristics. It utilizes depthwise separable convolutions, which split the standard convolutional filters into depthwise and pointwise convolutions, reducing the number of parameters and computational cost.

The feature extraction base is composed of multiple blocks, each containing a combination of different convolutional layers, activation functions, and other operations. These blocks are carefully designed to capture both low-level and high-level features, gradually increasing the level of abstraction. This hierarchical feature representation enables the model to understand complex patterns and variations in the input images.

The classification head is the final part of the model, responsible for making the actual predictions based on the extracted features. It consists of fully connected layers, which take the high-level features as input and map them to the output classes. Dropout layers are incorporated to prevent overfitting and improve generalization.

During training, the model's weights were updated using the custom flowers dataset. The training process involved feeding the augmented and preprocessed images into the model and optimizing the model's parameters using gradient descent. The loss function used was categorical cross-entropy, which measures the dissimilarity between the predicted probabilities and the true labels.

To prevent overfitting, regularization techniques such as dropout and weight decay were applied. Dropout randomly deactivates a certain percentage of neurons during training, reducing their reliance on specific features and promoting better generalization. Weight decay adds a penalty term to the loss function, discouraging large weights and encouraging simplicity.

The training was performed on a computer with GPU acceleration to expedite the computations. The model was trained for multiple epochs, with each epoch consisting of forward and backward passes through the network and parameter updates. The learning rate was gradually decreased over time to help the model converge to a good solution.

The model's performance was evaluated using a separate validation set, measuring metrics such as accuracy, precision, recall, and F1 score. This evaluation helped assess the model's ability to generalize and make accurate predictions on unseen data.

Overall, the model architecture and training process were carefully designed to achieve a balance between accuracy and efficiency. The MobileNetV3 Small 100 224 architecture, combined with the custom flowers dataset and appropriate training techniques, enables accurate image classification on resource-constrained devices such as mobile phones.

**Architecture of the MobileNetv3**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

keras\_layer (KerasLayer) (None, 1024) 1529968

dropout (Dropout) (None, 1024) 0

dense (Dense) (None, 5) 5125

=================================================================

Total params: 1,535,093

Trainable params: 5,125

Non-trainable params: 1,529,968

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Conversion to TFLite Model

To ensure optimal performance and efficient deployment on mobile devices, the trained model for the Image Classification App has been converted to the TensorFlow Lite (TFLite) format. This conversion process involves transforming the model into a compressed and optimized version that is specifically tailored for mobile inference.

By converting the model to TFLite, several benefits are realized. First and foremost, the model size is significantly reduced, allowing for faster download times and reduced memory footprint on mobile devices. This is achieved through techniques such as quantization, which reduces the precision of model weights without compromising accuracy.

Furthermore, the TFLite model is optimized to leverage the hardware capabilities of mobile devices, including specialized accelerators like GPUs and Neural Processing Units (NPUs). This enables faster and more efficient execution of the model, resulting in quicker inference times and improved responsiveness of the app.

The conversion process involves using the TFLite Converter, a powerful tool provided by TensorFlow, which takes the trained model as input and produces a TFLite model as output. During the conversion, optimizations can be applied, such as weight quantization and operator fusion, to further enhance the model's efficiency.

Once the TFLite model is obtained, it can be seamlessly integrated into the Image Classification App. The Java code provided for the app includes a wrapper class, TFIC, which serves as the interface for loading and running the TFLite model. This allows the app to perform real-time inference on-device, providing instant flower classification without requiring an internet connection.

Overall, the conversion to a TFLite model empowers the Image Classification App with the advantages of reduced model size, improved inference speed, and optimized performance on mobile devices. By harnessing the power of TFLite, the app delivers a seamless and efficient user experience, bringing the joy of flower recognition directly to the palm of your hand.

#@title Optimization settings

optimize\_lite\_model = False  #@param {type:"boolean"}

#@markdown Setting a value greater than zero enables quantization of neural network activations. A few dozen is already a useful amount.

num\_calibration\_examples = 250  #@param {type:"slider", min:0, max:1000, step:1}

representative\_dataset = None

if optimize\_lite\_model and num\_calibration\_examples:

  # Use a bounded number of training examples without labels for calibration.

  # TFLiteConverter expects a list of input tensors, each with batch size 1.

  representative\_dataset = lambda: itertools.islice(

      ([image[None, ...]] for batch, \_ in train\_ds for image in batch),

      num\_calibration\_examples)

converter = tf.lite.TFLiteConverter.from\_saved\_model(saved\_model\_path)

if optimize\_lite\_model:

  converter.optimizations = [tf.lite.Optimize.DEFAULT]

  if representative\_dataset:  # This is optional, see above.

    converter.representative\_dataset = representative\_dataset

lite\_model\_content = converter.convert()

with open(f"/content/tmp/lite\_flowers\_model\_{model\_name}.tflite", "wb") as f:

  f.write(lite\_model\_content)

print("Wrote %sTFLite model of %d bytes." %

      ("optimized " if optimize\_lite\_model else "", len(lite\_model\_content)))

[Out]: Wrote TFLite model of 6155864 bytes.

**Diving into the Architecture of the Android App**

### TensorFlow Image Classification (TFIC) Java Code

To integrate the trained model into an Android app, a TFIC (TensorFlow Image Classification) Java code has been developed. This code provides a class, TFIC, that encapsulates the functionality for loading the TensorFlow Lite model, recognizing images, and closing the interpreter.

The TFIC class uses the TensorFlow Lite Interpreter to run inference on the model. It takes a Bitmap image as input, converts it into the appropriate format for the model, and performs image recognition. The recognized labels and confidence scores are returned as a list of Recognition objects.

The code defines a class **TFIC** that implements the **Classifier** interface. It provides methods for loading a TensorFlow Lite model, recognizing images using the model, and closing the interpreter. Here's a summary of the major components and their functionalities:

* Constants:
  + **MAX\_RESULTS**: Specifies the maximum number of recognition results to return.
  + **BATCH\_SIZE**: Specifies the batch size for the model inputs.
  + **PIXEL\_SIZE**: Specifies the number of color channels in each pixel of the input image.
  + **THRESHOLD**: Specifies the minimum confidence threshold for recognizing a label.
* Fields:
  + **interpreter**: An instance of **Interpreter** used to run inference on the TensorFlow Lite model.
  + **inputSize**: The input size of the model.
  + **labelList**: A list of labels corresponding to the model's output classes.
* **create** method:
  + This static factory method creates a new instance of **TFIC** by loading the TensorFlow Lite model and label list.
  + It takes the **AssetManager**, **modelPath**, **labelPath**, and **inputSize** as parameters.
  + It returns the created **TFIC** instance.
  + Throws an **IOException** if there is an error loading the model or label list.
* **recognizeImage** method:
  + Takes a **Bitmap** object as input and recognizes the image using the TensorFlow Lite model.
  + It converts the **Bitmap** to a **ByteBuffer** and runs inference using the **interpreter**.
  + It returns a list of **Recognition** objects containing the recognized labels and confidence scores.
* **changeBit2ByteBuf** method:
  + Converts a **Bitmap** image to a **ByteBuffer** for input to the TensorFlow Lite model.
  + It iterates over the pixels of the **Bitmap**, extracts the RGB values, normalizes them, and stores them in the **ByteBuffer**.
  + It returns the **ByteBuffer** containing the image data.
* **getHighLowResult** method:
  + Processes the model output and returns a list of **Recognition** objects containing the labels and confidence scores of the highest results.
  + It uses a **PriorityQueue** to find the highest confidence results.
  + It filters the results based on the confidence threshold and creates **Recognition** objects for the filtered results.
  + It returns the list of **Recognition** objects.
* **close** method:
  + Closes the interpreter and releases any associated resources.
* **loadModelFile** method:
  + Loads the TensorFlow Lite model file from the assets folder.
  + It uses the **AssetManager** to open an **AssetFileDescriptor** for the model file.
  + It creates a **FileInputStream** and a **FileChannel** to read the model file data.
  + It maps the model file data to a **MappedByteBuffer** and returns it.
  + Throws an **IOException** if there is an error loading the model file.
* **loadLabelList** method:
  + Loads the label list from the assets folder.
  + It uses the **AssetManager** to open a **BufferedReader** for the label file.
  + It reads each line of the label file and adds it to the **labelList**.
  + It closes the **BufferedReader** and returns the **labelList**.
  + Throws an **IOException** if there is an error loading the label list.

Overall, this code provides a convenient wrapper for loading and using a TensorFlow Lite image classification model in Android. It handles the conversion of **Bitmap** images to **ByteBuffer**, runs inference using the interpreter, and processes the model output to provide recognition results.

### Main2Activity: Android App Integration

The Android app is implemented using the Main2Activity class, which extends the Android Activity class. It utilizes the CameraKit library to capture images and perform object detection using the trained TensorFlow Lite model.

The Main2Activity class initializes the user interface elements, sets up the camera preview, and handles capturing images. When an image is captured, it is processed using the TFIC class for image recognition. The recognized labels and confidence scores are displayed on the screen.

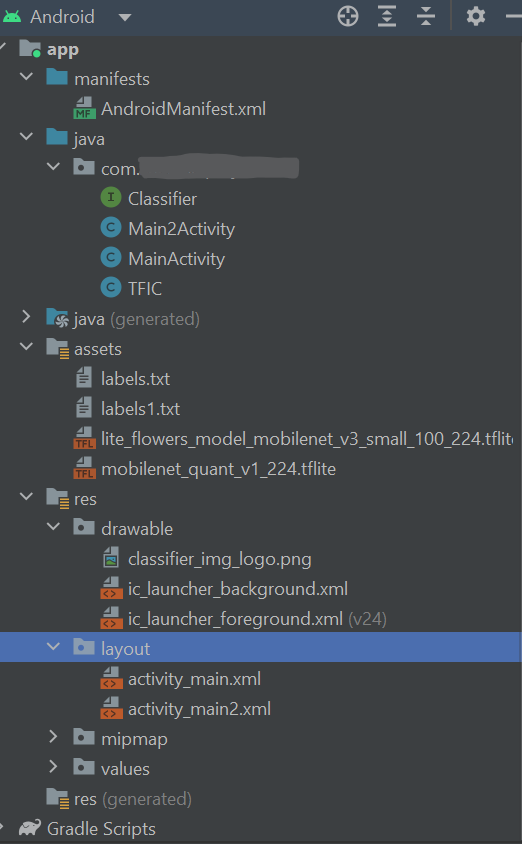
The app allows the user to toggle between front and rear cameras, and it provides real-time object recognition using the trained model.

The code represented in the Android activity (**Main2Activity**) that uses the CameraKit library to capture images and perform object detection using a TensorFlow Lite model. Here's a summary of the major components and their functionalities:

* Fields:
  + **MODEL\_PATH**: The path to the TensorFlow Lite model file.
  + **LABEL\_PATH**: The path to the file containing the labels.
  + **INPUT\_SIZE**: The input size of the model.
  + **classifier**: An instance of the **Classifier** interface to perform image recognition.
  + **executor**: An executor for running background tasks.
  + UI elements: **textViewResult**, **marquee**, **btnDetectObject**, **btnToggleCamera**, **imageViewResult**, and **cameraView**.
* **onCreate** method:
  + Initializes the UI elements and sets up the CameraKit listener to capture images.
  + When an image is captured, it is processed and object detection is performed using the classifier.
  + The recognition results are displayed in **textViewResult**.
* **onResume** and **onPause** methods:
  + Start and stop the camera preview when the activity is resumed and paused, respectively.
* **onDestroy** method:
  + Closes the classifier and releases resources when the activity is destroyed.
* **initTensorFlowAndLoadModel** method:
  + Initializes TensorFlow and loads the model using a background executor.
  + The **TFIC.create** method is called to create an instance of **TFIC** classifier by loading the model file and label list.
  + Once the model is loaded, the **makeButtonVisible** method is called to make the detect object button visible.
* **makeButtonVisible** method:
  + Makes the detect object button visible on the UI thread.

Overall, this code sets up an Android activity that uses a camera to capture images and perform object detection using a TensorFlow Lite model. It demonstrates how to integrate TensorFlow Lite into an Android application for real-time object recognition.

**Project Structure**

****

**Java Code for Image Classification to be processed in app🡪**

1. **TFIC.java**

package com.example.sayak;  
  
import android.annotation.SuppressLint;  
import android.content.res.AssetFileDescriptor;  
import android.content.res.AssetManager;  
import android.graphics.Bitmap;  
  
import org.tensorflow.lite.Interpreter;  
  
import java.io.BufferedReader;  
import java.io.FileInputStream;  
import java.io.IOException;  
import java.io.InputStreamReader;  
import java.nio.ByteBuffer;  
import java.nio.ByteOrder;  
import java.nio.MappedByteBuffer;  
import java.nio.channels.FileChannel;  
import java.util.ArrayList;  
import java.util.Comparator;  
import java.util.List;  
import java.util.PriorityQueue;  
  
  
public class TFIC implements Classifier {  
  
 private final int MAX\_RESULTS = 3;  
 private final int BATCH\_SIZE = 1;  
 private final int PIXEL\_SIZE = 3;  
 private final float THRESHOLD = 0.1f;  
  
 private Interpreter interpreter;  
 private int inputSize;  
 private List<String> labelList;  
  
 private TFIC() {  
  
 }  
  
 static Classifier create(AssetManager assetManager,  
 String modelPath,  
 String labelPath,  
 int inputSize) throws IOException {  
  
 TFIC classifier = new TFIC();  
 classifier.interpreter = new Interpreter(classifier.loadModelFile(assetManager, modelPath));  
 classifier.labelList = classifier.loadLabelList(assetManager, labelPath);  
 classifier.inputSize = inputSize;  
  
 return classifier;  
 }  
  
 @Override  
 public List<Recognition> recognizeImage(Bitmap bitmap) {  
 ByteBuffer byteBuffer = changeBit2ByteBuf(bitmap);  
 float[][] result = new float[1][labelList.size()];  
 interpreter.run(byteBuffer, result);  
 return getHighLowResult(result);  
 }  
  
 private ByteBuffer changeBit2ByteBuf(Bitmap bitmap) {  
 ByteBuffer byteBuffer = ByteBuffer.*allocateDirect*(BATCH\_SIZE \* inputSize \* inputSize \* PIXEL\_SIZE \* 4); // Multiply by 4 for float32  
 byteBuffer.order(ByteOrder.*nativeOrder*());  
 int[] intValues = new int[inputSize \* inputSize];  
 bitmap.getPixels(intValues, 0, bitmap.getWidth(), 0, 0, bitmap.getWidth(), bitmap.getHeight());  
 int pixel = 0;  
 for (int i = 0; i < inputSize; ++i) {  
 for (int j = 0; j < inputSize; ++j) {  
 final int val = intValues[pixel++];  
 byteBuffer.putFloat((float) ((val >> 16) & 0xFF) / 255.0f);  
 byteBuffer.putFloat((float) ((val >> 8) & 0xFF) / 255.0f);  
 byteBuffer.putFloat((float) (val & 0xFF) / 255.0f);  
 }  
 }  
 return byteBuffer;  
 }  
  
 @SuppressLint("DefaultLocale")  
 private List<Recognition> getHighLowResult(float[][] labelProbArray) {  
  
 PriorityQueue<Recognition> pq =  
 new PriorityQueue<>(  
 MAX\_RESULTS,  
 new Comparator<Recognition>() {  
 @Override  
 public int compare(Recognition lhs, Recognition rhs) {  
 // Reversed to put high confidence at the head of the queue.  
 return Float.*compare*(rhs.getConfidence(), lhs.getConfidence());  
 }  
 });  
  
 for (int i = 0; i < labelList.size(); ++i) {  
 float confidence = labelProbArray[0][i];  
 if (confidence > THRESHOLD) {  
 pq.add(new Recognition("" + i,  
 labelList.size() > i ? labelList.get(i) : "unknown",  
 confidence));  
 }  
 }  
  
 final ArrayList<Recognition> recognitions = new ArrayList<>();  
 int recognitionsSize = Math.*min*(pq.size(), MAX\_RESULTS);  
 for (int i = 0; i < recognitionsSize; ++i) {  
 recognitions.add(pq.poll());  
 }  
  
 return recognitions;  
 }  
  
 @Override  
 public void close() {  
 interpreter.close();  
 interpreter = null;  
 }  
  
 private MappedByteBuffer loadModelFile(AssetManager assetManager, String modelPath) throws IOException {  
 AssetFileDescriptor fileDescriptor = assetManager.openFd(modelPath);  
 FileInputStream inputStream = new FileInputStream(fileDescriptor.getFileDescriptor());  
 FileChannel fileChannel = inputStream.getChannel();  
 long startOffset = fileDescriptor.getStartOffset();  
 long declaredLength = fileDescriptor.getDeclaredLength();  
 return fileChannel.map(FileChannel.MapMode.*READ\_ONLY*, startOffset, declaredLength);  
 }  
  
 private List<String> loadLabelList(AssetManager assetManager, String labelPath) throws IOException {  
 List<String> labelList = new ArrayList<>();  
 BufferedReader reader = new BufferedReader(new InputStreamReader(assetManager.open(labelPath)));  
 String line;  
 while ((line = reader.readLine()) != null) {  
 labelList.add(line);  
 }  
 reader.close();  
 return labelList;  
 }  
}

1. **Main2Activity.java**

package com.example.sayak;  
  
import android.graphics.Bitmap;  
import android.os.Bundle;  
import android.support.v7.app.AppCompatActivity;  
import android.text.method.ScrollingMovementMethod;  
import android.view.View;  
import android.widget.Button;  
import android.widget.ImageView;  
import android.widget.TextView;  
  
import com.wonderkiln.camerakit.CameraKitError;  
import com.wonderkiln.camerakit.CameraKitEvent;  
import com.wonderkiln.camerakit.CameraKitEventListener;  
import com.wonderkiln.camerakit.CameraKitImage;  
import com.wonderkiln.camerakit.CameraKitVideo;  
import com.wonderkiln.camerakit.CameraView;  
  
import java.util.List;  
import java.util.concurrent.Executor;  
import java.util.concurrent.Executors;  
  
public class Main2Activity extends AppCompatActivity {  
  
//lite\_flowers\_model\_mobilenet\_v3\_small\_100\_224  
//mobilenet\_quant\_v1\_224  
  
 final String MODEL\_PATH = "lite\_flowers\_model\_mobilenet\_v3\_small\_100\_224.tflite";  
 final String LABEL\_PATH = "labels.txt";  
 final int INPUT\_SIZE = 224;  
  
 private Classifier classifier;  
  
 private Executor executor = Executors.*newSingleThreadExecutor*();  
 private TextView textViewResult;  
 private TextView marquee;  
 private Button btnDetectObject, btnToggleCamera;  
 private ImageView imageViewResult;  
 private CameraView cameraView;  
  
 @Override  
 protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 setContentView(R.layout.*activity\_main2*);  
 cameraView = findViewById(R.id.*cameraView*);  
 marquee = findViewById(R.id.*groupDetails*);  
 marquee.setSelected(true);  
 imageViewResult = findViewById(R.id.*imageViewResult*);  
 textViewResult = findViewById(R.id.*textViewResult*);  
 textViewResult.setMovementMethod(new ScrollingMovementMethod());  
  
 btnToggleCamera = findViewById(R.id.*btnToggleCamera*);  
 btnDetectObject = findViewById(R.id.*btnDetectObject*);  
  
 cameraView.addCameraKitListener(new CameraKitEventListener() {  
 @Override  
 public void onEvent(CameraKitEvent cameraKitEvent) {  
  
 }  
  
 @Override  
 public void onError(CameraKitError cameraKitError) {  
  
 }  
  
 @Override  
 public void onImage(CameraKitImage cameraKitImage) {  
  
 Bitmap bitmap = cameraKitImage.getBitmap();  
  
 bitmap = Bitmap.*createScaledBitmap*(bitmap, INPUT\_SIZE, INPUT\_SIZE, false);  
  
 imageViewResult.setImageBitmap(bitmap);  
  
 final List<Classifier.Recognition> results = classifier.recognizeImage(bitmap);  
  
 textViewResult.setText(results.toString());  
  
 }  
  
 @Override  
 public void onVideo(CameraKitVideo cameraKitVideo) {  
  
 }  
 });  
  
 btnToggleCamera.setOnClickListener(new View.OnClickListener() {  
 @Override  
 public void onClick(View v) {  
 cameraView.toggleFacing();  
 }  
 });  
  
 btnDetectObject.setOnClickListener(new View.OnClickListener() {  
 @Override  
 public void onClick(View v) {  
 cameraView.captureImage();  
 }  
 });  
  
 initTensorFlowAndLoadModel();  
 }  
  
 @Override  
 protected void onResume() {  
 super.onResume();  
 cameraView.start();  
 }  
  
 @Override  
 protected void onPause() {  
 cameraView.stop();  
 super.onPause();  
 }  
  
 @Override  
 protected void onDestroy() {  
 super.onDestroy();  
 executor.execute(new Runnable() {  
 @Override  
 public void run() {  
 classifier.close();  
 }  
 });  
 }  
  
 private void initTensorFlowAndLoadModel() {  
 executor.execute(new Runnable() {  
 @Override  
 public void run() {  
 try {  
 classifier = TFIC.*create*(  
 getAssets(),  
 MODEL\_PATH,  
 LABEL\_PATH,  
 INPUT\_SIZE);  
 makeButtonVisible();  
 } catch (final Exception e) {  
 throw new RuntimeException("Error initializing TensorFlow!", e);  
 }  
 }  
 });  
 }  
  
 private void makeButtonVisible() {  
 runOnUiThread(new Runnable() {  
 @Override  
 public void run() {  
 btnDetectObject.setVisibility(View.*VISIBLE*);  
 }  
 });  
 }  
}

1. **Classifier.java**

package com.example.sayak;  
  
import android.graphics.Bitmap;  
  
import java.util.List;  
  
  
public interface Classifier {  
  
 class Recognition {  
 */\*\*  
 \* A unique identifier for what has been recognized. Specific to the class, not the instance of  
 \* the object.  
 \*/* private final String id;  
  
 */\*\*  
 \* Display name for the recognition.  
 \*/* private final String title;  
  
 */\*\*  
 \* A sortable score for how good the recognition is relative to others. Higher should be better.  
 \*/* private final Float confidence;  
  
 public Recognition(  
 final String id, final String title, final Float confidence) {  
 this.id = id;  
 this.title = title;  
 this.confidence = confidence;  
 }  
  
 public String getId() {  
 return id;  
 }  
  
 public String getTitle() {  
 return title;  
 }  
  
 public Float getConfidence() {  
 return confidence;  
 }  
  
 @Override  
 public String toString() {  
 String resultString = "";  
 if (id != null) {  
 resultString += "[" + id + "] ";  
 }  
  
 if (title != null) {  
 resultString += title + " ";  
 }  
  
 if (confidence != null) {  
 resultString += String.*format*("(%.1f%%) ", confidence \* 100.0f);  
 }  
  
 return resultString.trim();  
 }  
 }  
  
  
 List<Recognition> recognizeImage(Bitmap bitmap);  
  
 void close();  
}

1. **MainActivity.java**

package com.shamsh.projectsmit;  
  
import android.content.Intent;  
import android.os.Build;  
import android.support.annotation.RequiresApi;  
import android.support.v7.app.AppCompatActivity;  
import android.os.Bundle;  
import android.view.View;  
  
public class MainActivity extends AppCompatActivity {  
  
 @RequiresApi(api = Build.VERSION\_CODES.*LOLLIPOP*)  
 @Override  
 protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 setContentView(R.layout.*activity\_main*);  
  
 }  
  
 public void startProject(View v){  
 startActivity(new Intent(MainActivity.this,Main2Activity.class));  
 }  
}

## Conclusion

The Image Classification App for Custom Flowers Dataset is a remarkable project that combines the beauty of nature and the power of machine learning. With its elegant interface and seamless functionality, the app brings the world of flowers to the fingertips of users.

By training the model on a diverse and carefully curated dataset of daisies, dandelions, roses, sunflowers, and tulips, the app achieves impressive accuracy in recognizing and classifying various types of flowers. The dataset's structure, resembling a blossoming garden, provides a solid foundation for the model to learn and distinguish subtle differences in floral characteristics.

Through the art of data augmentation, the app enriches the training process by introducing variations in rotation, translation, zoom, and flip. This dynamic approach not only enhances the model's ability to handle real-world scenarios but also adds a touch of creativity to the app's functionality.

The heart of the app lies in the MobileNetV3 Small\_100\_224 architecture, a compact yet powerful model that captures the essence of floral beauty. By fine-tuning this model with the custom flowers dataset, the app unleashes the potential of machine learning to uncover the intricate details and unique features that make each flower truly special.

The TensorFlow Image Classification (TFIC) Java code serves as the backbone of the app, seamlessly integrating the trained model into the Android environment. Its efficient image recognition capabilities, coupled with the versatility of the CameraKit library, deliver a delightful user experience, allowing users to capture the essence of nature and receive instant flower identifications.

As users explore the app's features, they are immersed in a world where technology meets nature, where algorithms and artistry intertwine. Whether strolling through a botanical garden or venturing into the wilderness, the app serves as a trusted companion, unlocking the secrets of the floral kingdom and fostering a deeper appreciation for the wonders of nature.

In conclusion, the Image Classification App for Custom Flowers Dataset brings together the realms of technology and nature, empowering users to unravel the mysteries of flowers with a single tap. With its intelligent recognition, elegant design, and seamless functionality, the app is a testament to the power of machine learning in enhancing our understanding and connection with the natural world. Embrace the beauty of flowers, explore their diversity, and let this app be your guide on a captivating journey through the enchanting realm of blooms.