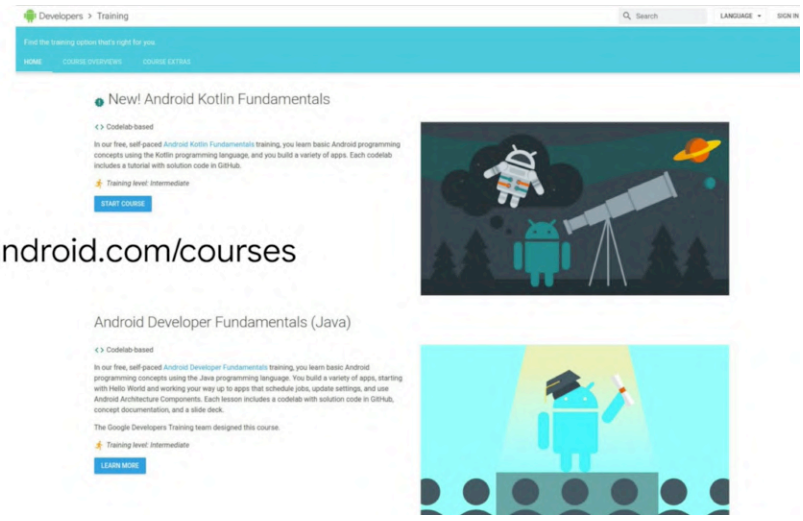
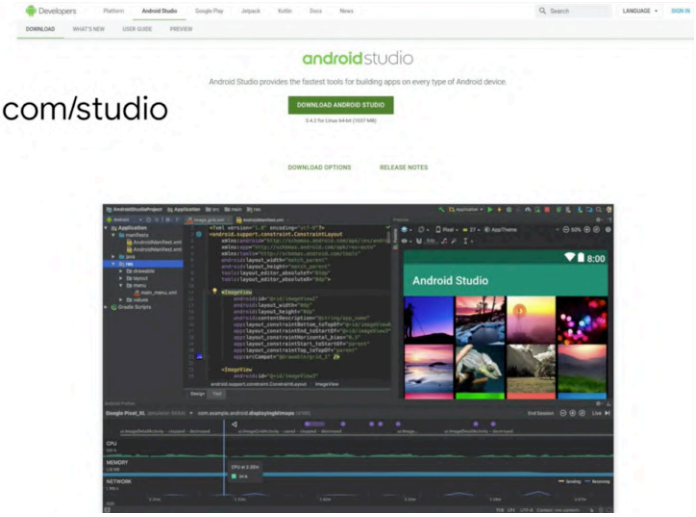
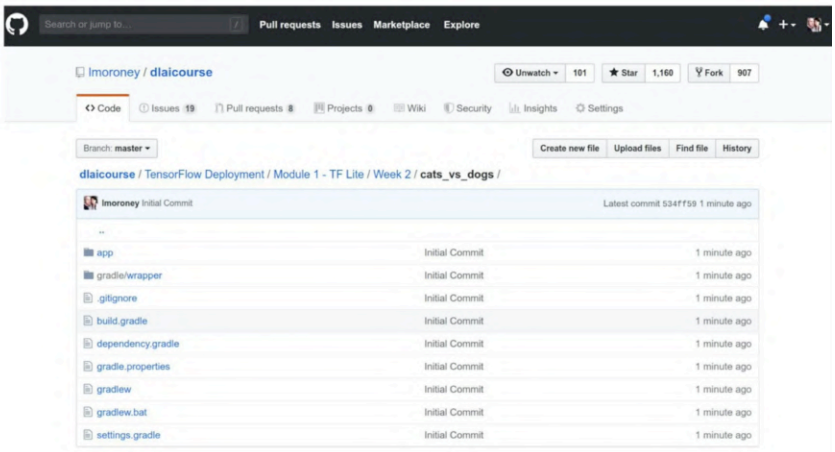
developer.android.com/courses



developer.android.com/studio



github.com/lmoroney/dlaicourse



build.gradle

Adding TensorFlow Lite to your Android project

```
dependencies {
    implementation 'org.tensorflow:tensorflow-lite:0.0.0-nightly'
}

android {
    // ...
    aaptOptions {
        noCompress "tflite"  // Your model's file extension: "tflite", "lite", etc.
    }
}
```

Check latest version at:
https://bintray.com/google/tensorflow/tensorflow-lite

To use tensorflow lights in an Android app built using Android studio. You need to set the tensorflow like dependencies and some options in your build.gradle file. Keep an eye on the latest release for the exact implementation details for now. This site gives you details on what to put in your build.gradle to get the latest.

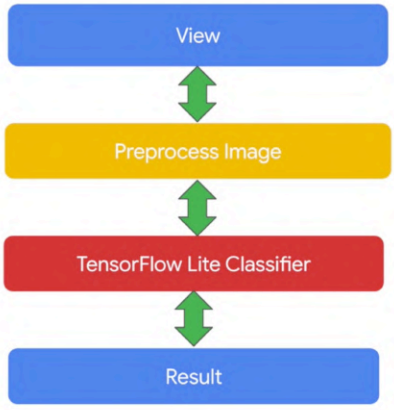
build.gradle

Adding TensorFlow Lite to your Android project

```
dependencies {
    implementation 'org.tensorflow:tensorflow-lite:0.0.0-nightly'
}

android {
    // ...
    aaptOptions {
        noCompress "tflite"  // Your model's file extension: "tflite", "lite", etc.
    }
}
```

Importantly, you'll also need these options set up this informs the Android compiler not to compile or compress the TF light file. You need the raw bytes to be copied to the device uncompressed or The Interpreter cannot load them. It's the number one issue. I've seen with new developers and they run into this when using tensorflow light on Android.

Architecture



ImageClassifierActivity.kt

The first line is that the current view is found, and its image is extracted as a bitmap, and then loaded into the bitmap variable.

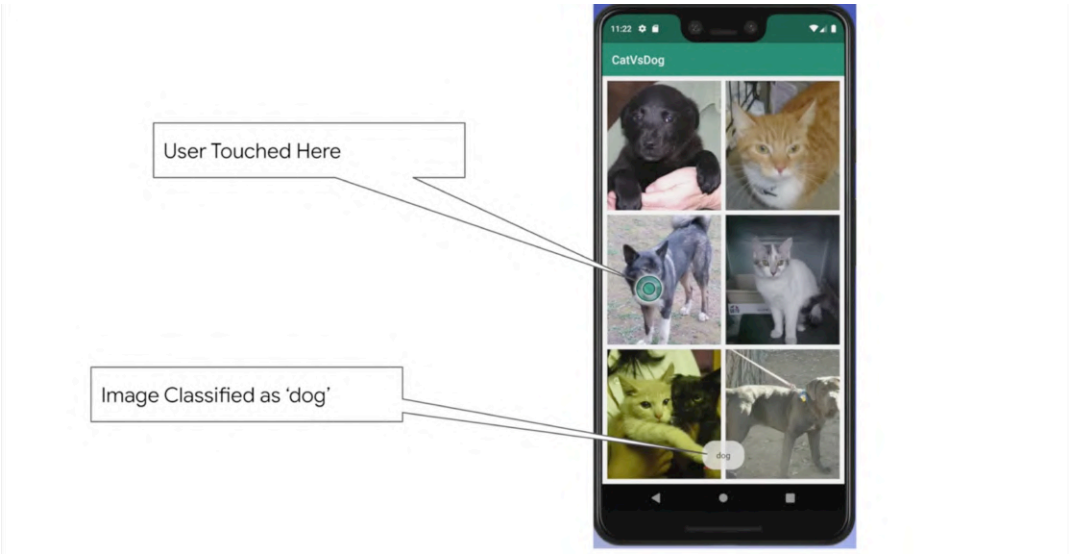
```
override fun onClick(view: View?) {
    val bitmap = ((view as ImageView).drawable as BitmapDrawable).bitmap

    val result = classifier.recognizeImage(bitmap)

    runOnUiThread
       { Toast.makeText(this, result.get(0).title, Toast.LENGTH_SHORT).show() }
}
```

ImageClassifierActivity.kt

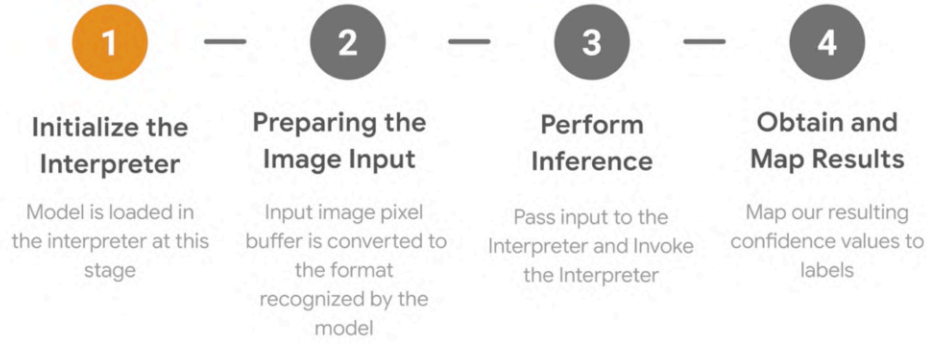
This is then passed to the classifier module, which is Kotlin code, that wraps the interpreter object, as you'll see later, and a result is returned.

```
override fun onClick(view: View?) {
    val bitmap = ((view as ImageView).drawable as BitmapDrawable).bitmap

    val result = classifier.recognizeImage(bitmap)

    runOnUiThread
       { Toast.makeText(this, result.get(0).title, Toast.LENGTH_SHORT).show() }
}
```

The title of the result is then passed to a Toast to be rendered.



Steps Involved in Performing Inference



Set the Interpreter's Options

Options: A class for controlling runtime interpreter behaviour

- `setNumThreads(int numThreads)`
- `setUseNNAPI(boolean useNNAPI)`
- `setAllowFp16PrecisionForFp32(boolean allow)`
- `addDelegate(Delegate delegate)`

Classifier.kt

Set the Interpreter's Options

```
val tfliteOptions = Interpreter.Options()
tfliteOptions.setNumThreads(5)
tfliteOptions.setUseNNAPI(true)
```

Classifier.kt

Loading the model and labels

- Get the file descriptor of the model file

```
assetManager.openFd("converted_model.tflite")
```

- Open the input stream

```
val inputStream = FileInputStream(fileDescript
```

- Read the file channels along with its offset and length as follows

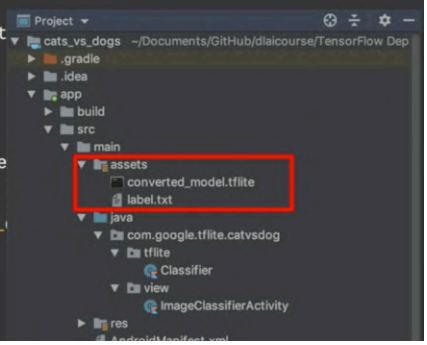
```
val fileChannel = inputStream.channel
val startOffset = fileDescriptor.startOffset
val declaredLength = fileDescriptor.declaredLe
```

- Finally we load the TFLite model as:

```
tfliteModel = fileChannel.map(FileChannel.MapMode.READ_
```

- Then set the labels as follows

```
labelList = Arrays.asList("cat","dog")
```



Classifier.kt

Loading the model and labels

- Get the file descriptor of the model file

```
assetManager.openFd("converted_model.tflite")
```

- Open the input stream

```
val inputStream = FileInputStream(fileDescriptor.fileDescriptor)
```

- Read the file channels along with its offset and length as follows

```
val fileChannel = inputStream.channel
val startOffset = fileDescriptor.startOffset
val declaredLength = fileDescriptor.declaredLength
```

- Finally we load the TFLite model as:

```
tfliteModel = fileChannel.map(FileChannel.MapMode.READ_ONLY, startOffset, declaredLength)
```

- Then set the labels as follows

```
labelList = Arrays.asList("cat","dog")
```

Classifier.kt

Loading the model and labels

- Get the file descriptor of the model file

```
assetManager.openFd("converted_model.tflite")
```

- Open the input stream

```
val inputStream = FileInputStream(fileDescriptor.fileDescriptor)
```

- Read the file channels along with its offset and length as follows

```
val fileChannel = inputStream.channel
val startOffset = fileDescriptor.startOffset
val declaredLength = fileDescriptor.declaredLength
```

- Finally we load the TFLite model as:

```
tfliteModel = fileChannel.map(FileChannel.MapMode.READ_ONLY, startOffset, declaredLength)
```

- Then set the labels as follows

```
labelList = Arrays.asList("cat","dog")
```

Classifier.kt

Loading the model and labels

- Get the file descriptor of the model file

```
assetManager.openFd("converted_model.tflite")
```

- Open the input stream

```
val inputStream = FileInputStream(fileDescriptor.fileDescriptor)
```

- Read the file channels along with its offset and length as follows

```
val fileChannel = inputStream.channel
val startOffset = fileDescriptor.startOffset
val declaredLength = fileDescriptor.declaredLength
```

- Finally we load the TFLite model as:

```
tfliteModel = fileChannel.map(FileChannel.MapMode.READ_ONLY, startOffset, declaredLength)
```

- Then set the labels as follows

```
labelList = Arrays.asList("cat","dog")
```

Classifier.kt

Initializing the Interpreter

```
tflite = Interpreter(tfliteModel, tfliteOptions)
```

Steps Involved in Performing Inference

1 — 2 — 3 — 4

Initialize the Interpreter

Model is loaded in the interpreter at this stage

Preparing the Image Input

Input image pixel buffer is converted to the format recognized by the model

Perform Inference

Pass input to the Interpreter and Invoke the Interpreter

Obtain and Map Results

Map our resulting confidence values to labels

Original Size: 1280 x 720

Desired Size: 224 x 224

Color: Green(ish)

ARGB Value: #0010FF10

RGB Values: 16, 255, 16

Normalized Values: .06275, 1, .06275

ImageClassifierActivity.kt

Rescaling and allocating a buffer

- Resize the bitmap to 224 x 224

```
Bitmap.createScaledBitmap(bitmap, INPUT_SIZE, INPUT_SIZE, false)
```

- Convert bitmap to bytebuffer

```
// Batch size is 4 (Floating point model)
val byteBuffer = ByteBuffer.allocateDirect(BATCH_SIZE *
                                           INPUT_SIZE * INPUT_SIZE *
                                           PIXEL_SIZE)
byteBuffer.order(ByteOrder.nativeOrder())
```

Classifier.kt

Get R-G-B channels of the image

```
int red = (input.shr(16)  and 0xFF)
int green = (input.shr(8)  and 0xFF)
int blue = (input and 0xFF)
```

Classifier.kt

Preparing the input

```
for (i in 0 until INPUT_SIZE) {
   for (j in 0 until INPUT_SIZE) {
       val input = intValues[pixel++]
       byteBuffer.putFloat((((input.shr(16)  and 0xFF) - IMAGE_MEAN) / IMAGE_STD))
       byteBuffer.putFloat((((input.shr(8) and 0xFF) - IMAGE_MEAN) / IMAGE_STD))
       byteBuffer.putFloat((((input and 0xFF) - IMAGE_MEAN) / IMAGE_STD))
    }
}
```

Steps Involved in Performing Inference

1 — 2 — 3 — 4

Initialize the Interpreter

Model is loaded in the interpreter at this stage

Preparing the Image Input

Input image pixel buffer is converted to the format recognized by the model

Perform Inference

Pass input to the Interpreter and Invoke the Interpreter

Obtain and Map Results

Map our resulting confidence values to labels

Classifier.kt

Running inference and accumulating the results

```
val result = Array(1) { FloatArray(2) }
interpreter.run(byteBuffer, result)
```

Classifier.kt

Running inference and accumulating the results

```
val result = Array(1) { FloatArray(2) }
interpreter.run(byteBuffer, result)
```

Getting And Processing the Result

1 — 2 — 3 — 4

Initialize the Interpreter

Model is loaded in the interpreter at this stage

Preparing the Image Input

Input image pixel buffer is converted to the format recognized by the model

Perform Inference

Pass input to the Interpreter and Invoke the Interpreter

Obtain and Map Results

Map our resulting confidence values to labels

Classifier.kt

Sorting the results

- Here we instantiate a queue to accumulate the results with its size indicating the number of results to be shown

```
val pq = PriorityQueue(
            MAX_RESULTS,
            Comparator<Classifier.Recognition> {...})
```

- The list of labels are stored in:

```
labelList = Arrays.asList("cat","dog")
```

```
for (i in labelList.indices) {
    val confidence = labelProbArray[0][i]
    if (confidence >= 0.5) {
        pq.add(Classifier.Recognition("" + i,
                if (labelList.size > i) labelList[i] else "Unknown", confidence))
    }
}
```

What is Image Classification?

- A common use of machine learning is to identify what an image represents.
- Quantized MobileNet trained on ImageNet dataset comprising of around 1000 different classes of objects including people, animals, etc.,

Steps Involved in Performing Inference

1. **Initialize the Interpreter**
   Model is loaded in the interpreter at this stage
2. **Preparing the Image Input**
   Prepare the bitmap so that it can be used by the interpreter
3. **Perform Inference**
   Pass input to the Interpreter and Invoke the Interpreter
4. **Obtain and Map Results**
   Map our resulting confidence values to labels

Classifier.kt

Set the Interpreter's Options

```
val tfliteOptions = Interpreter.Options()
tfliteOptions.setNumThreads(5)
tfliteOptions.setUseNNAPI(true)
```

Classifier.kt

Loading model and label file into the interpreter

- Get the file descriptor of the model

```
assetManager.openFd("mobilenet_v1_1.0_224_quant.tflite")
```

- Read the model file's channels

```
val inputStream = FileInputStream(fil
val fileChannel = inputStream.channel
val startOffset = fileDescriptor.star
val declaredLength = fileDescriptor.d
```

Models Repository at:
https://www.tensorflow.org/lite/guide/hosted_models

- Load the TFLite model as:

```
tfliteModel = fileChannel.map(FileChannel.MapMode.READ_ONLY, startOffset, declaredLength)
```

- Load the labels

```
labelList = assetManager.open("labels_mobilenet_quant_v1_224.txt")
          .bufferedReader()
          .useLines { it.toList() }
```

Classifier.kt

Loading model and label file into the interpreter

- Get the file descriptor of the model

```
assetManager.openFd("mobilenet_v1_1.0_224_quant.tflite")
```

- Read the model file's channels

```
val inputStream = FileInputStream(fileDescriptor.fileDescriptor)
val fileChannel = inputStream.channel
val startOffset = fileDescriptor.startOffset
val declaredLength = fileDescriptor.declaredLength
```

- Load the TFLite model as:

```
tfliteModel = fileChannel.map(FileChannel.MapMode.READ_ONLY, startOffset, declaredLength)
```

- Load the labels

```
labelList = assetManager.open("labels_mobilenet_quant_v1_224.txt")
          .bufferedReader()
          .useLines { it.toList() }
```

Classifier.kt

Loading model and label file into the interpreter

- Get the file descriptor of the model

```
assetManager.openFd("mobilenet_v1_1.0_224_quant.tflite")
```

- Read the model file's channels

```
val inputStream = FileInputStream(fil
val fileChannel = inputStream.channel
val startOffset = fileDescriptor.star
val declaredLength = fileDescriptor.d
```

- Load the TFLite model as:

```
tfliteModel = fileChannel.map(FileChannel.M      READ_ONLY, startOffset, declaredLength)
```

- Load the labels

```
labelList = assetManager.open("labels_mobilenet_quant_v1_224.txt")
          .bufferedReader()
          .useLines { it.toList() }
```

Models Repository at:
https://www.tensorflow.org/lite/guide/hosted_models

Classifier.kt

Initializing the Interpreter

```
tflite = Interpreter(tfliteModel, tfliteOptions)
```

Steps Involved in Performing Inference

1 — 2 — 3 — 4

Initialize the Interpreter

Model is loaded in the interpreter at this stage

Preparing the Image Input

Prepare the bitmap so that it can be used by the interpreter

Perform Inference

Pass input to the Interpreter and Invoke the Interpreter

Obtain and Map Results

Map our resulting confidence values to labels

CameraActivity.kt

Preparing the input

- Get the image when available in the camera feed
- Convert it to YUV image format
- Then convert this to ARGB8888 so that we can extract the RGB channels

Camera2BasicFragment.kt

```
// Get The last image
val image: Image = it.acquireLatestImage()
val planes = image.getPlanes()

// Strip Y U V channels
yRowStride      = planes[0].getRowStride()
uvRowStride     = planes[1].getRowStride()
uvPixelStride   = planes[1].getPixelStride()

// Convert  to ARGB format
ImageUtils.convertYUV420ToARGB8888(
yuvBytes[0],yuvBytes[1],yuvBytes[2],previewSize.width,previewSize.height,
yRowStride,uvRowStride,uvPixelStride,rgbBytes)
```

Camera2BasicFragment.kt

```
// Get The last image
val image: Image = it.acquireLatestImage()
val planes = image.getPlanes()

// Strip Y U V channels
yRowStride      = planes[0].getRowStride()
uvRowStride     = planes[1].getRowStride()
uvPixelStride   = planes[1].getPixelStride()

// Convert  to ARGB format
ImageUtils.convertYUV420ToARGB8888(
yuvBytes[0],yuvBytes[1],yuvBytes[2],previewSize.width,previewSize.height,
yRowStride,uvRowStride,uvPixelStride,rgbBytes)
```

Camera2BasicFragment.kt

```
// Get The last image
val image: Image = it.acquireLatestImage()
val planes = image.getPlanes()

// Strip Y U V channels
yRowStride      = planes[0].getRowStride()
uvRowStride     = planes[1].getRowStride()
uvPixelStride   = planes[1].getPixelStride()

// Convert  to ARGB format
ImageUtils.convertYUV420ToARGB8888(
yuvBytes[0],yuvBytes[1],yuvBytes[2],previewSize.width,previewSize.height,
yRowStride,uvRowStride,uvPixelStride,rgbBytes)
```

ImageClassifierActivity.kt

Preparing the input

- Resize the bitmap to 224 x 224

```
Bitmap.createScaledBitmap(bitmap, INPUT_SIZE, INPUT_SIZE, false)
```

- Convert bitmap to bytebuffer

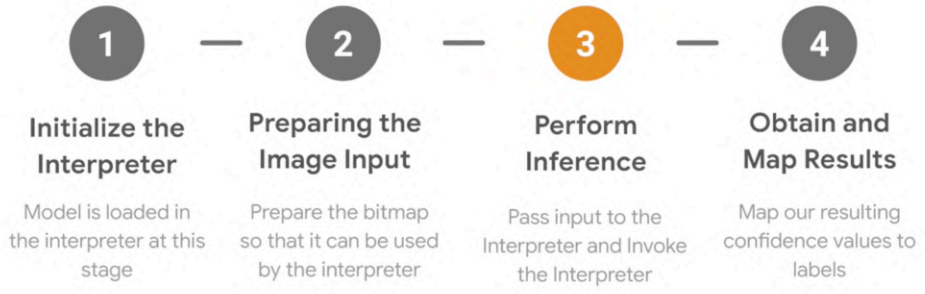
```
// Batch size is 1 (Quantized model)
val byteBuffer = ByteBuffer.allocateDirect(BATCH_SIZE *
                                           INPUT_SIZE * INPUT_SIZE *
                                           PIXEL_SIZE)
byteBuffer.order(ByteOrder.nativeOrder())
```

Classifier.kt

Preparing the input

```
byteBuffer.put((intValue.shr(16) and 0xFF).toByte())
byteBuffer.put((intValue.shr(8) and 0xFF).toByte())
byteBuffer.put((intValue and 0xFF).toByte())
```

Steps Involved in Performing Inference



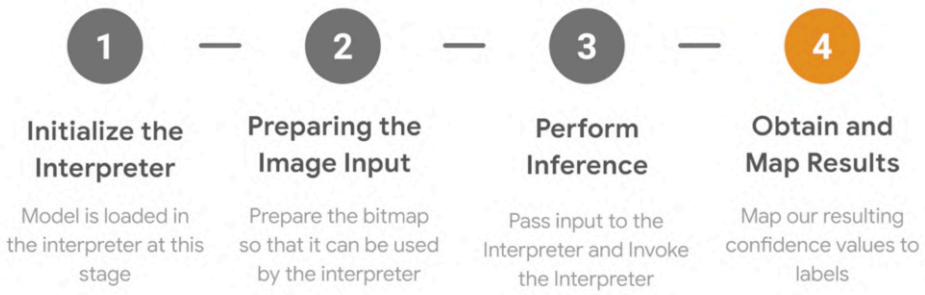
Classifier.kt

Running inference and accumulating the results

Feed the byte buffer and the labels probability array to the interpreter to get the result

```
val result = Array(1) { ByteArray(labelList.size) }
interpreter.run(byteBuffer, result)
```

Steps Involved in Performing Inference



Classifier.kt

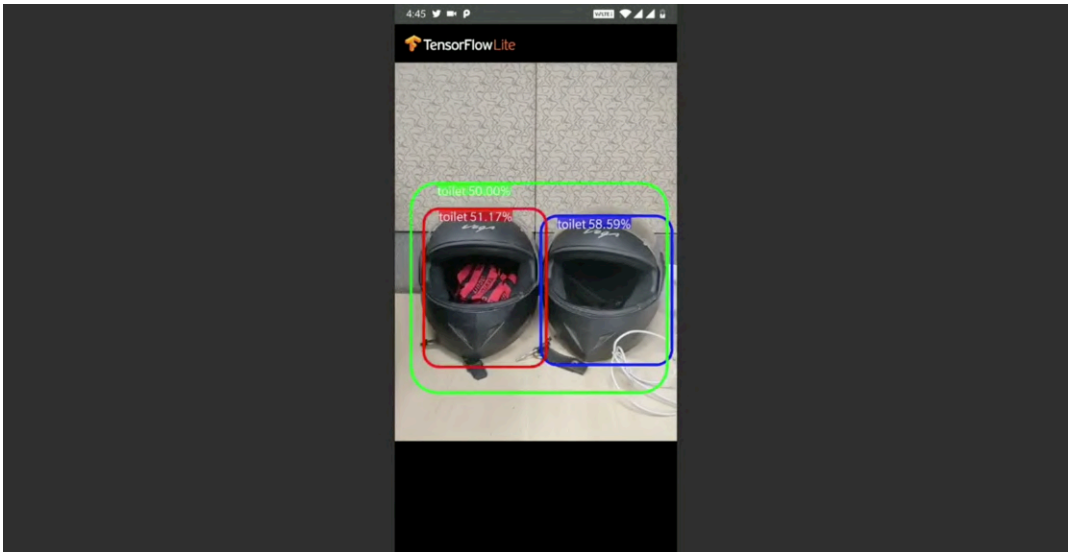
Sorting the results

- Here we instantiate a queue to accumulate the results with its size indicating the number of results to be shown

```
val pq = PriorityQueue(
                MAX_RESULTS,
               Comparator<Classifier.Recognition> {...})
```

- We assume the minimum score value to be 40% or above for a result to be considered as a recognition

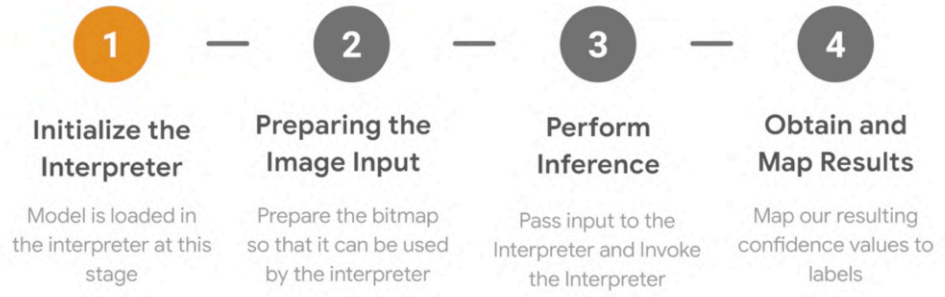
```
for (i in labelList.indices) {
    val confidence = labelProbArray[0][i]
    if (confidence >= THRESHOLD) {
        pq.add(Classifier.Recognition("" + i,
                if (labelList.size > i) labelList[i] else "Unknown", confidence))
    }
}
```



Object Detection Model

- Identifies classes of objects along with localizing them
- MobileNet SSD trained on COCO dataset
- COCO dataset has 80 classes
- Labels file is used to list COCO classes and map to output confidences

Steps Involved in Performing Inference



Classifier.kt

```
val tfliteOptions = Interpreter.Options()
tfliteOptions.setNumThreads(5)
tfliteOptions.setUseNNAPI(true)

d.tfLite = Interpreter(
               loadModelFile(assetManager, "detect.tflite"),
               tfliteOptions)
```

Classifier.kt

```
labels = assetManager.open("labelmap.txt").bufferedReader()
               .useLines { it.toList() }
```

Steps Involved in Performing Inference

Initialize the Interpreter

Model is loaded in the interpreter at this stage

Preparing the Image Input

Prepare the bitmap so that it can be used by the interpreter

Perform Inference

Pass input to the Interpreter and Invoke the Interpreter

Obtain and Map Results

Map our resulting confidence values to labels

Classifier.kt

Get pixels from the Bitmap

Signature

```
public void getPixels (int[] pixels,
                       int offset,
                       int stride,
                       int x,
                       int y,
                       int width,
                       int height)
```

In practice

```
bitmap.getPixels(intValues,
                 0,
                 bitmap.width,
                 0,
                 0,
                 bitmap.width,
                 bitmap.height)
```

```
// Dump the image data from the Bitmap to the integer pixel array based on the
// provided parameters.
```

Classifier.kt

Extract image data

```
for (i in 0 until inputSize) {
    for (j in 0 until inputSize) {
        val pixelValue = intValues[i * inputSize + j]
        if (isModelQuantized) {
            // Quantized model
            imgData.put((pixelValue shr 16 and 0xFF).toByte())
            imgData.put((pixelValue shr 8 and 0xFF).toByte())
            imgData.put((pixelValue and 0xFF).toByte())
        }
    }
}
```

Steps Involved in Performing Inference

Initialize the Interpreter

Model is loaded in the interpreter at this stage

Preparing the Image Input

Prepare the bitmap so that it can be used by the interpreter

Perform Inference

Pass input to the Interpreter and Invoke the Interpreter

Obtain and Map Results

Map our resulting confidence values to labels

Output Tensors

0	Bounding Boxes
1	Classes
2	Scores
3	Number of Results

Classifier.kt

Shapes of the outputs

```
d.outputLocations = Array(1) { Array(NUM_DETECTIONS) { FloatArray(4) } }
d.outputClasses = Array(1) { FloatArray(NUM_DETECTIONS) }
d.outputScores = Array(1) { FloatArray(NUM_DETECTIONS) }
d.numDetections = FloatArray(1)
```

Classifier.kt

Get the input and output arrays

```
val inputArray = arrayOf<Any>(imgData)

// The result is a map of the outputs
val outputMap = HashMap<Int, Any>()
outputMap[0] = outputLocations
outputMap[1] = outputClasses
outputMap[2] = outputScores
outputMap[3] = numDetections

// Run the inference call.
tfLite.runForMultipleInputsOutputs(inputArray, outputMap)
```

Getting And Processing the Result

Initialize the Interpreter

Model is loaded in the interpreter at this stage

Preparing the Image Input

Input image pixel buffer is converted to the format recognized by the model

Perform Inference

Pass input to the Interpreter and Invoke the Interpreter

Obtain and Map Results

Map our resulting confidence values to labels

TFLiteObjectDetectionAPIModel.kt

Obtain And Map Result

```
Each Detected objects location is
    val detection = RectF(
                    outputLocations[0][i][1] * inputSize,
                    outputLocations[0][i][0] * inputSize,
                    outputLocations[0][i][3] * inputSize,
                    outputLocations[0][i][2] * inputSize)

And Each detected result is
Classifier.Recognition( "" + i,
                        labels[outputClasses[0][i].toInt() + labelOffset],
                        outputScores[0][i],
                        detection)
```

TFLiteObjectDetectionAPIModel.kt

Obtain And Map Result

```
Each Detected objects location is
     val detection = RectF(
                    outputLocations[0][i][1] * inputSize,
                    outputLocations[0][i][0] * inputSize,
                    outputLocations[0][i][3] * inputSize,
                    outputLocations[0][i][2] * inputSize)

And Each detected result is
Classifier.Recognition( "" + i,
                          labels[outputClasses[0][i].toInt() + labelOffset],
                          outputScores[0][i],
                          detection)
```

TFLiteObjectDetectionAPIModel.kt

Obtain And Map Result

```
for (result in results) {
    val location = result.location
    if (location != null && result.confidence >= minimumConfidence) {
        canvas.drawRect(location, paint)
        cropToFrameTransform.mapRect(location)
        result.location = location
        mappedRecognitions.add(result)
    }
}
```

MultiBoxTracker.kt

Show the results on the screen

Process the result with respect to each individual component

(score, name, coordinates and color of a detected object)

```
val trackedRecognition = TrackedRecognition()
trackedRecognition.detectionConfidence = potential.first
trackedRecognition.location = RectF(potential.second.location)
trackedRecognition.title = potential.second.title
trackedRecognition.color = COLORS[trackedObjects.size]
trackedObjects.add(trackedRecognition)
```

Then show them in screen with

```
canvas.drawRoundRect(trackedPos, cornerSize, cornerSize, boxPaint)
borderedText.drawText(canvas, // The bounding box
                    trackedPos.left + cornerSize,
                    trackedPos.top, "$labelString%",   boxPaint) // Display predicted class
```