**PRACTICAL: 14**

**AIM:Create an application that uses the end-to-end process of training a machinelearning model that can recognize handwritten digit images with TensorFlow anddeploy it to an Android app.**

**Source Code:**

**Layout File/s:**

**activity\_main.xml**

|  |
| --- |
| *<?***xml version="1.0" encoding="utf-8"***?>* <**androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android" xmlns:app="http://schemas.android.com/apk/res-auto" xmlns:tools="http://schemas.android.com/tools" android:layout\_width="match\_parent" android:layout\_height="match\_parent" tools:context=".MainActivity"**>  <**com.divyanshu.draw.widget.DrawView android:id="@+id/draw\_view" android:layout\_width="match\_parent" android:layout\_height="0dp" app:layout\_constraintDimensionRatio="1:1" app:layout\_constraintTop\_toTopOf="parent"**/>  <**TextView android:id="@+id/predicted\_text" android:textStyle="bold" android:layout\_width="wrap\_content" android:layout\_height="wrap\_content" android:text="@string/prediction\_text\_placeholder" android:textSize="20sp" app:layout\_constraintBottom\_toTopOf="@id/clear\_button" app:layout\_constraintLeft\_toLeftOf="parent" app:layout\_constraintRight\_toRightOf="parent" app:layout\_constraintTop\_toBottomOf="@id/draw\_view"**/>  <**Button android:id="@+id/clear\_button" android:layout\_width="wrap\_content" android:layout\_height="wrap\_content" android:text="@string/clear\_button\_text" app:layout\_constraintBottom\_toBottomOf="parent" app:layout\_constraintLeft\_toLeftOf="parent" app:layout\_constraintRight\_toRightOf="parent"**/>  </**androidx.constraintlayout.widget.ConstraintLayout**> |

**File/s:**

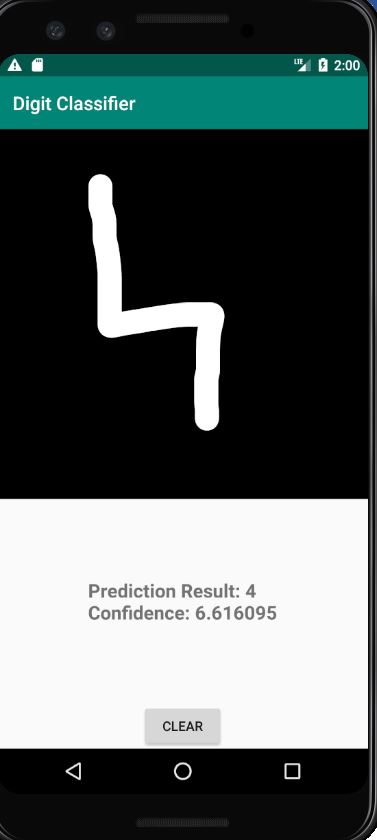
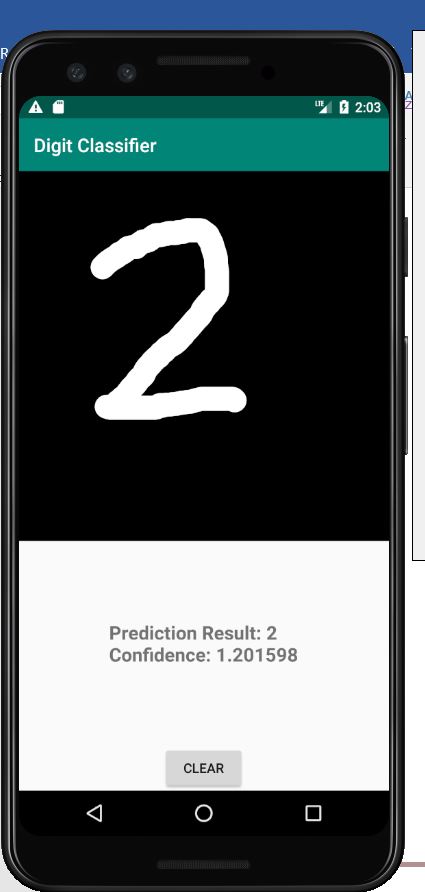
**MainActivity.kt**

|  |
| --- |
| **package** org.tensorflow.lite.codelabs.digitclassifier  **import** android.annotation.SuppressLint **import** android.graphics.Color **import** android.os.Bundle **import** android.util.Log **import** android.view.MotionEvent **import** android.widget.Button **import** android.widget.TextView **import** androidx.appcompat.app.AppCompatActivity **import** com.divyanshu.draw.widget.DrawView  **class** MainActivity : AppCompatActivity() {  **private var drawView**: DrawView? = **null  private var clearButton**: Button? = **null  private var predictedTextView**: TextView? = **null  private var digitClassifier** = DigitClassifier(**this**)  @SuppressLint(**"ClickableViewAccessibility"**) **override fun** onCreate(savedInstanceState: Bundle?) { **super**.onCreate(savedInstanceState)  setContentView(R.layout.*activity\_main*)  *// Setup view instances.* **drawView** = findViewById(R.id.*draw\_view*) **drawView**?.setStrokeWidth(70.0f) **drawView**?.setColor(Color.*WHITE*) **drawView**?.setBackgroundColor(Color.*BLACK*) **clearButton** = findViewById(R.id.*clear\_button*) **predictedTextView** = findViewById(R.id.*predicted\_text*)  *// Setup clear drawing button.* **clearButton**?.setOnClickListener **{ drawView**?.clearCanvas() **predictedTextView**?.*text* = getString(R.string.*prediction\_text\_placeholder*) **}** *// Setup classification trigger so that it classify after every stroke drew.* **drawView**?.setOnTouchListener **{** \_, event **->** *// As we have interrupted DrawView's touch event,  // we first need to pass touch events through to the instance for the drawing to show up.* **drawView**?.onTouchEvent(event)  *// Then if user finished a touch event, run classification* **if** (event.*action* == MotionEvent.*ACTION\_UP*) {  classifyDrawing()  }  **true }** *// Setup digit classifier.* **digitClassifier** .initialize()  .addOnFailureListener **{** e **->**Log.e(**TAG**, **"Error to setting up digit classifier."**, e) **}** }  **override fun** onDestroy() { *// Sync DigitClassifier instance lifecycle with MainActivity lifecycle,  // and free up resources (e.g. TF Lite instance) once the activity is destroyed.* **digitClassifier**.close() **super**.onDestroy()  }  **private fun** classifyDrawing() { **val** bitmap = **drawView**?.getBitmap()  **if** ((bitmap != **null**) && (**digitClassifier**.**isInitialized**)) { **digitClassifier** .classifyAsync(bitmap)  .addOnSuccessListener **{** resultText **->predictedTextView**?.*text* = resultText **}** .addOnFailureListener **{** e **-> predictedTextView**?.*text* = getString(  R.string.*classification\_error\_message*,  e.*localizedMessage* )  Log.e(**TAG**, **"Error classifying drawing."**, e) **}** }  }  **companion object** { **private const val TAG** = **"MainActivity"** } } |

**Digitclassifier.kt**

|  |
| --- |
| **package** org.tensorflow.lite.codelabs.digitclassifier  **import** android.content.Context **import** android.content.res.AssetManager **import** android.graphics.Bitmap **import** android.util.Log **import** com.google.android.gms.tasks.Task **import** com.google.android.gms.tasks.Tasks.call **import** org.tensorflow.lite.Interpreter **import** java.io.FileInputStream **import** java.io.IOException **import** java.nio.ByteBuffer **import** java.nio.ByteOrder **import** java.nio.channels.FileChannel **import** java.util.concurrent.Callable **import** java.util.concurrent.ExecutorService **import** java.util.concurrent.Executors   **class** DigitClassifier(**private val context**: Context) { *//* ***TODO: Add a TF Lite interpreter as a field.* private var interpreter**: Interpreter? = **null  var isInitialized** = **false  private set** */\*\* Executor to run inference task in the background. \*/* **private val executorService**: ExecutorService = Executors.newCachedThreadPool()  **private var inputImageWidth**: Int = 0 *// will be inferred from TF Lite model.* **private var inputImageHeight**: Int = 0 *// will be inferred from TF Lite model.* **private var modelInputSize**: Int = 0 *// will be inferred from TF Lite model.* **fun** initialize(): Task<Void> { **return** call( **executorService**,  Callable<Void>**{** initializeInterpreter() **null }** )  }  @Throws(IOException::**class**) **private fun** initializeInterpreter() { *//* ***TODO: Load the TF Lite model from file and initialize an interpreter.* val** assetManager = **context**.*assets* **val** model = loadModelFile(assetManager, **"mnist.tflite"**)  *// Initialize TF Lite Interpreter with NNAPI enabled.* **val** options = Interpreter.Options()  options.setUseNNAPI(**true**) **val** interpreter = Interpreter(model, options)  *//* ***TODO: Read the model input shape from model file.* val** inputShape = interpreter.getInputTensor(0).shape() **inputImageWidth** = inputShape[1] **inputImageHeight** = inputShape[2] **modelInputSize** = **FLOAT\_TYPE\_SIZE** \* **inputImageWidth** \* **inputImageHeight** \* **PIXEL\_SIZE  this**.**interpreter** = interpreter  **isInitialized** = **true** Log.d(**TAG**, **"Initialized TFLite interpreter."**)  }  @Throws(IOException::**class**) **private fun** loadModelFile(assetManager: AssetManager, filename: String): ByteBuffer { **val** fileDescriptor = assetManager.openFd(filename) **val** inputStream = FileInputStream(fileDescriptor.*fileDescriptor*) **val** fileChannel = inputStream.*channel* **val** startOffset = fileDescriptor.*startOffset* **val** declaredLength = fileDescriptor.*declaredLength* **return** fileChannel.map(FileChannel.MapMode.*READ\_ONLY*, startOffset, declaredLength)  }  **private fun** classify(bitmap: Bitmap): String { *check*(**isInitialized**) **{ "TF Lite Interpreter is not initialized yet." }** *//* ***TODO: Add code to run inference with TF Lite.*** *// Preprocessing: resize the input image to match the model input shape.* **val** resizedImage = Bitmap.createScaledBitmap(  bitmap, **inputImageWidth**, **inputImageHeight**, **true** ) **val** byteBuffer = convertBitmapToByteBuffer(resizedImage)  *// Define an array to store the model output.* **val** output = Array(1) **{** FloatArray(**OUTPUT\_CLASSES\_COUNT**) **}** *// Run inference with the input data.* **interpreter**?.run(byteBuffer, output) *// Post-processing: find the digit that has the highest probability // and return it a human-readable string.* **val** result = output[0] **val** maxIndex = result.*indices*.*maxBy* **{** result[**it**] **}** ?: -1 **val** resultString = **"Prediction Result: %d\nConfidence: %2f"**. *format*(maxIndex, result[maxIndex])  **return** resultString  }  **fun** classifyAsync(bitmap: Bitmap): Task<String> { **return** call(**executorService**, *Callable*<String>**{** classify(bitmap) **}**)  }  **fun** close() {  call( **executorService**, *Callable*<String>**{** *//* ***TODO: close the TF Lite interpreter here* interpreter**?.close()   Log.d(**TAG**, **"Closed TFLite interpreter."**) **null }** )  }  **private fun** convertBitmapToByteBuffer(bitmap: Bitmap): ByteBuffer { **val** byteBuffer = ByteBuffer.allocateDirect(**modelInputSize**)  byteBuffer.order(ByteOrder.nativeOrder())  **val** pixels = IntArray(**inputImageWidth** \* **inputImageHeight**)  bitmap.getPixels(pixels, 0, bitmap.*width*, 0, 0, bitmap.*width*, bitmap.*height*)  **for** (pixelValue **in** pixels) { **val** r = (pixelValue shr 16 and 0xFF) **val** g = (pixelValue shr 8 and 0xFF) **val** b = (pixelValue and 0xFF)  *// Convert RGB to grayscale and normalize pixel value to [0..1].* **val** normalizedPixelValue = (r + g + b) / 3.0f / 255.0f byteBuffer.putFloat(normalizedPixelValue)  }  **return** byteBuffer  }  **companion object** { **private const val TAG** = **"DigitClassifier"  private const val FLOAT\_TYPE\_SIZE** = 4 **private const val PIXEL\_SIZE** = 1  **private const val OUTPUT\_CLASSES\_COUNT** = 10 } } |

**OUTPUT:**

** **