5

第 5 章

用 TensorFlow Mobile 构建机器学习应用

本章将介绍怎样使用 TensorFlow 开发 Android 的机器学习应用。

5.1 准备工作

为了看到更多日志记录，可以在 tensorflow/examples/android/src/org/tensorflow/demo/env/Logger.java 中把 Log.DEBUG 换成 Log.VERBOSE：

```
private static final String DEFAULT_TAG = "tensorflow";
private static final int DEFAULT_MIN_LOG_LEVEL = Log.DEBUG;
```

在应用中要用到模型和标注表，通常我们希望开发者能自动下载编译，应用也能自动

读取，我们来看一下 Bazel 是怎样实现这个功能的。首先我们看一下 tensorflow/examples/android/下的 build 文件：

```
android_binary( name = "tensorflow_demo",
assets = [
    "//tensorflow/examples/android/assets:asset_files",
    ":external_assets",
],
```

build 文件里定义了编译目标 tensorflow_demo，以及编译要生成的应用 tensorflow_demo.apk。这个应用依赖于 asset 中的:extemal assets，它在 build 文件中的定义如下：

```
filegroup(
    name = "external_assets",
    srcs = [
        "@inception_v1//:model_files",
        "@mobile_ssd//:model_files",
        "@speech_commands//:model_files",
        "@stylize//:model_files",
    ],
)
```

filegroup 定义了一组文件，这些文件就是模型和标注文件。tensorflow_demo 这个应用会把其中定义的四个目标包含的文件全部保存到 APK 里的 assert 下面。由于模型文件占用存储空间较大，所以通常只保存所需模型即可。现在看看其中的一个构建目标“@inception_v1//:model_files”，它的定义可以在 workspace 文件里找到，workspace 文件代码如下：

```
http_archive(
    name = "inception_v1",
    build_file = "//:models.BUILD",
    sha256   =   "7efe12a8363f09bc24d7b7a450304a15655a57a7751929b2c1593a
71183bb105",
    urls = [
        "http://storage.googleapis.com/download.tensorflow.org/models
/inception_v1.zip",
        "http://download.tensorflow.org/models/inception_v1.zip",
    ],
)
```

Bazel 会自动从 http://storage.googleapis.com/download.tensorflow.org/models/inception_v1.zip 或 http://download.tensorflow.org/models/inception_v1.zip 下载解压文件，并检查

sha256 值。下载成功后，使用"//:models.BUILD"进行构建，具体实现代码如下：

```
    filegroup(
        name = "model_files",
        srcs = glob(
            [
                "**/*",
            ],
            exclude = [
                "**/BUILD",
                "**/WORKSPACE",
                "**/LICENSE",
                "**/*.zip",
            ],
        ),
    )
```

这段代码会自动忽略下载文件中的一些文件，比如 BUILD、WORKSPACE 等，而把其他文件作为构建目标，这样其他构建目标可以参照这些文件。

比如，我们可以通过下面的命令下载模型文件：

```
    $  wget  "http://storage.googleapis.com/download.tensorflow.org/models/
inception_v1.zip"
    $ unzip inception_v1.zip
    $ ls -all
    -r--r----- 1    10492 Nov 18  2015 imagenet_comp_graph_label_strings.txt
    -rw-r----- 1  49937249 Jan 22  2018 inception_v1.zip
    -r--r----- 1    11416 Nov 18  2015 LICENSE
    -rw-r----- 1  53881635 Sep 28  2017 tensorflow_inception_graph.pb
```

得到这些文件并成功构建 tensorflow_demo.apk 后，我们可以执行下面的命令：

```
    $ unzip bazel-bin/tensorflow/examples/android/tensorflow_demo.apk
    $ ls -all asserts
    -rw-rw-rw- 1      328 Jan  1  2010 BUILD.bazel
    -rw-rw-rw- 1      665 Jan  1  2010 coco_labels_list.txt
    -rw-rw-rw- 1   3771239 Jan  1  2010 conv_actions_frozen.pb
    -rw-rw-rw- 1       60 Jan  1  2010 conv_actions_labels.txt
    -rw-rw-rw- 1    10492 Jan  1  2010 imagenet_comp_graph_label_strings.txt
    -rw-rw-rw- 1    11416 Jan  1  2010 LICENSE
    -rw-rw-rw- 1  29083865 Jan  1  2010 ssd_mobilenet_v1_android_export.pb
    -rw-rw-rw- 1   563897 Jan  1  2010 stylize_quantized.pb
```

```
-rw-rw-rw- 1  53881635 Jan  1  2010 tensorflow_inception_graph.pb
drwxr-x--- 2     4096 Jan 21 11:01 thumbnails
-rw-rw-rw- 1       28 Jan  1  2010 WORKSPACE
```

我们定义的模型文件和标注目标文件都被保存在 asserts 下面了。请注意，tensorflow_inception_graph.pb 文件大小近 54MB，stylize_quantized.pb 文件大小不到 600KB。模型占用的存储空间还是不小的。APK 的大小，不仅影响了对设备存储的要求，而且，用户第一次下载要花费大量时间，这对用户体验也有很大影响。

5.2 图像分类（Image Classification）

图像分类是人工智能的一个主要应用，我们来看一下怎样在移动设备上实现图像的分类。

5.2.1 应用

下面的例子主要讲解 tensorflow/examples/android/src/org/tensorflow/demo 下的 TensorFlowImageClassifier.java 文件。

图像分类的 Activity 是 tensorflow/examples/android/src/org/tensorflow/demo/ ClassifierActivity.java。它的定义是：

```
public class ClassifierActivity extends CameraActivity implements OnImageAvailableListener {}
```

它继承了 CameraActivity，并实现了 OnImageAvailableListener。CameraActivity 实现了 Android 应用的基本生命周期的功能，比如 onStart、onCreate、onStop、onDestroy 等。

另外，它实现了相机的预览（Preview）。实现预览的主要原因是，我们要从相机里取得图像的数据。在 onCreate 里首先调用 setFragment，在这段代码里会生成 CameraConnectionFragment 的一个实例。

```
if (useCamera2API) {
  CameraConnectionFragment camera2Fragment =
      CameraConnectionFragment.newInstance(
          new CameraConnectionFragment.ConnectionCallback() {
            @Override
            public void onPreviewSizeChosen(final Size size, final int
```

```
rotation) {
                previewHeight = size.getHeight();
                previewWidth = size.getWidth();
                CameraActivity.this.onPreviewSizeChosen(size, rotation);
              }
            },
            this,
            getLayoutId(),
            getDesiredPreviewFrameSize());

      camera2Fragment.setCamera(cameraId);
      fragment = camera2Fragment;
    } else {
      fragment =
         new LegacyCameraConnectionFragment(this, getLayoutId(),
getDesiredPreviewFrameSize());
    }
```

tensorflow/examples/android/src/org/tensorflow/demo 下的 CameraConnectionFragment. Java 文件实现了 CameraConnectionFragment，这是 Android 的一个 Fragment。其关键的功能是由 setUpCameraOutputs 实现的，代码如下：

```
      private void setUpCameraOutputs() {
        final Activity activity = getActivity();
        final    CameraManager    manager    =    (CameraManager)    activity.
getSystemService(Context.CAMERA_SERVICE);
        try {
          final    CameraCharacteristics    characteristics    =    manager.
getCameraCharacteristics(cameraId);

          final StreamConfigurationMap map =
             characteristics.get(CameraCharacteristics.SCALER_STREAM_
CONFIGURATION_MAP);

          // 使用最大尺寸进行图像抓取
          final Size largest =
             Collections.max(
                Arrays.asList(map.getOutputSizes(ImageFormat.YUV_ 420_888)),
                new CompareSizesByArea());

          sensorOrientation    =    characteristics.get(CameraCharacteristics.
SENSOR_ORIENTATION);
```

```
        // 预览尺寸过大会超过相框
        // 垃圾捕获数据
        previewSize =
           chooseOptimalSize(map.getOutputSizes(SurfaceTexture.class),
              inputSize.getWidth(),
              inputSize.getHeight());

        // TextureView 的长宽比要与我们选择的预览尺寸相匹配
        final int orientation = getResources().getConfiguration().
orientation;
        if (orientation == Configuration.ORIENTATION_LANDSCAPE) {
          textureView.setAspectRatio(previewSize.getWidth(),
previewSize.getHeight());
        } else {
          textureView.setAspectRatio(previewSize.getHeight(),
previewSize.getWidth());
        }
      } catch (final CameraAccessException e) {
        LOGGER.e(e, "Exception!");
      } catch (final NullPointerException e) {
        // 当此代码运行的设备不支持 Camera2API 时，抛弃 NPE
        // 此代码运行的设备

        ErrorDialog.newInstance(getString(R.string.camera_error))
           .show(getChildFragmentManager(), FRAGMENT_DIALOG);
        throw new RuntimeException(getString(R.string.camera_error));
      }

      cameraConnectionCallback.onPreviewSizeChosen(previewSize,
sensorOrientation);
    }
```

在通过 getCameraCharacteristics(cameraId)获得相机的属性后，通过与图像的解像度进行对比，调用 chooseOptimalSize 找到最合适的预览 Preview 的尺寸。

```
   protected static Size chooseOptimalSize(final Size[] choices, final int
width, final int height) {
    final int minSize = Math.max(Math.min(width, height),
MINIMUM_PREVIEW_SIZE);
    final Size desiredSize = new Size(width, height);
```

```
      // 收集支持的分辨率，这些分辨率至少与预览图面一样大
      boolean exactSizeFound = false;
      final List<Size> bigEnough = new ArrayList<Size>();
      final List<Size> tooSmall = new ArrayList<Size>();
      for (final Size option : choices) {
        if (option.equals(desiredSize)) {
          // 设置尺寸，但不返回，以便记录剩余尺寸
          exactSizeFound = true;
        }

        if (option.getHeight() >= minSize && option.getWidth() >= minSize) {
          bigEnough.add(option);
        } else {
          tooSmall.add(option);
        }
      }

      LOGGER.i("Desired size: " + desiredSize + ", min size: " + minSize + "x"
+ minSize);
      LOGGER.i("Valid preview sizes: [" + TextUtils.join(", ", bigEnough) + "]");
      LOGGER.i("Rejected preview sizes: [" + TextUtils.join(", ", tooSmall) +
"]");

      if (exactSizeFound) {
        LOGGER.i("Exact size match found.");
        return desiredSize;
      }

      // 选择其中最小的一个图像
      if (bigEnough.size() > 0) {
        final    Size    chosenSize    =    Collections.min(bigEnough,    new
CompareSizesByArea());
        LOGGER.i("Chosen   size:   "   +   chosenSize.getWidth()   +   "x"   +
chosenSize.getHeight());
        return chosenSize;
      } else {
        LOGGER.e("Couldn't find any suitable preview size");
        return choices[0];
      }
    }
```

如果找到完全匹配的图像就返回，否则返回较小的图像。

到这里我们基本上就实现了把一个相机设定好并取得其预览的功能。由于我们在这里需要构造一个完整的应用，所以要通过大量代码来实现从一个实际的应用中获取图像程序的功能。在测试或者非应用的程序中可以选用一些静态的图像，代码也会简单很多。

回到 CameraActivity.java 的 setFragment，下面的代码会显示 Fragment：

```
getFragmentManager()
     .beginTransaction()
     .replace(R.id.container, fragment)
     .commit();
```

在 onResume()里面，主要做了两件事，一是建立一个后台线程，二是启动相机。启动相机就是调用 Android CameraManager 的 openCamera 函数，这个应用做了一个简单的封装，代码如下：

```
@Override
  public void onResume() {
    super.onResume();
    startBackgroundThread();

    // 当屏幕关闭并重新打开时，surfacetexture 为可用状态，并且不会调用“onsurfaceextureavailable”，我们可以打开一个摄像头进行预览
    if (textureView.isAvailable()) {
      openCamera(textureView.getWidth(), textureView.getHeight());
    } else {
      textureView.setSurfaceTextureListener(surfaceTextureListener);
    }
  }
```

建立后台线程。过程很简单，这里调用 Android 的 Handler 和 HandlerThread，生成并启动一个名为“ImageListener”的线程。

```
private void startBackgroundThread() {
   backgroundThread = new HandlerThread("ImageListener");
   backgroundThread.start();
   backgroundHandler = new Handler(backgroundThread.getLooper());
 }
```

启动相机。如果相机被打开和启动，那么下面的函数就会被调用，并启动 createCamera-

PreviewSession。

```
    private final CameraDevice.StateCallback stateCallback =
        new CameraDevice.StateCallback() {
          @Override
          public void onOpened(final CameraDevice cd) {
            // 当相机打开时，这个方法就会被调用，我们即可开始预览相机

            cameraOpenCloseLock.release();
            cameraDevice = cd;
            createCameraPreviewSession();
          }
```

使用相机预览 Preview 的应用，基本要实现两个功能，一是设定相机的预览大小，二是实现一个相机录入会话 CameraCaptureSession，它的实现在

private void createCameraPreviewSession()中。具体实现过程如下：

首先，建立一个肌理（Texture）和与其关联的 TextureView，作为图像输出的显示区，这样我们就可以在设备上看到预览图像：

```
    final SurfaceTexture texture = textureView.getSurfaceTexture();
    assert texture != null;

    // 我们将默认缓冲区的大小配置为所需的相机预览大小
    texture.setDefaultBufferSize(previewSize.getWidth(),
previewSize.getHeight());

    // 这是我们需要开始预览的输出曲面
    final Surface surface = new Surface(texture);

    // 我们用输出曲面设置了 CaptureRequest.Builder
    previewRequestBuilder                                                    =
cameraDevice.createCaptureRequest(CameraDevice.TEMPLATE_PREVIEW);
    previewRequestBuilder.addTarget(surface);
```

然后，新建一个 ImageReader，并设定预览大小和图像数据的回调：

```
    // 为预览帧创建读卡器
    previewReader =
        ImageReader.newInstance(
```

```
          previewSize.getWidth(), previewSize.getHeight(), ImageFormat.
YUV_420_888, 2);

    previewReader.setOnImageAvailableListener(imageListener,
backgroundHandler);
    previewRequestBuilder.addTarget(previewReader.getSurface());
```

最后，调用 cameraDevice.createCaptureSession 生成一个相机的图像抓取会话，关键的地方是生成一个会话的回调，代码如下：

```
       public       void       onConfigured(final       CameraCaptureSession
cameraCaptureSession) {
         // 相机已关闭
         if (null == cameraDevice) {
           return;
         }
          // 当会话准备就绪时，开始显示预览
          captureSession = cameraCaptureSession;
         try {
           // 在相机预览时，自动对焦是连续的
           previewRequestBuilder.set(
               CaptureRequest.CONTROL_AF_MODE,
               CaptureRequest.CONTROL_AF_MODE_CONTINUOUS_PICTURE);
           // 必要时自动启用闪存
           previewRequestBuilder.set(
               CaptureRequest.CONTROL_AE_MODE, CaptureRequest.CONTROL_AE_
MODE_ON_AUTO_FLASH);
            // 显示相机预览
           previewRequest = previewRequestBuilder.build();
           captureSession.setRepeatingRequest(
               previewRequest, captureCallback, backgroundHandler);
         } catch (final CameraAccessException e) {
           LOGGER.e(e, "Exception!");
         }
       }
```

至此，这个应用的基本准备功能和框架都有了，可以从相机看到图像，图像数据通过回调函数也可以得到。下面把应用和图像分类的模型联系起来。

ClassifierActivity 实现了 OnImageAvailableListener，也实现了 Camera.PreviewCallback，

当新的图像被相机生成后，onImageAvailable 会被调用。这个函数主要做两个工作，一是图像格式的转换，二是调用 processImage()进行图像处理。

```
@Override
public void onImageAvailable(final ImageReader reader)
```

下面是函数的定义。它的图像转换主要调用 ImageUtils 类里的函数。

```
Trace.beginSection("imageAvailable");
final Plane[] planes = image.getPlanes();
fillBytes(planes, yuvBytes);
yRowStride = planes[0].getRowStride();
final int uvRowStride = planes[1].getRowStride();
final int uvPixelStride = planes[1].getPixelStride();

imageConverter =
    new Runnable() {
      @Override
      public void run() {
        ImageUtils.convertYUV420ToARGB8888(
            yuvBytes[0],
            yuvBytes[1],
            yuvBytes[2],
            previewWidth,
            previewHeight,
            yRowStride,
            uvRowStride,
            uvPixelStride,
            rgbBytes);
      }
    };
```

然后，调用 processImage()函数：

```
Trace.beginSection("imageAvailable");
Trace.endSection();
```

该函数使用 Android 的 Trace 功能，可以记录图像处理所需要的时间。

请注意，在 ImageUtils 里有几个 YUV 和 RGB 转换的函数。YUV 在硬件图像处理里使用较多，多数相机的输出格式也支持 YUV。RGB 是一种历史很长的格式，它代表了红绿蓝在颜色里的构成，比较容易理解，很多应用会使用这种格式。所以，不同颜色的转换

是必要的，比如 convertYUV420ToARGB8888，相关转换代码如下：

```
int yp = 0;
for (int j = 0; j < height; j++) {
  int pY = yRowStride * j;
  int pUV = uvRowStride * (j >> 1);

  for (int i = 0; i < width; i++) {
    int uv_offset = pUV + (i >> 1) * uvPixelStride;

    out[yp++] = YUV2RGB(
        0xff & yData[pY + i],
        0xff & uData[uv_offset],
        0xff & vData[uv_offset]);
  }
 }
```

对于每一个像素，依据下面的公式进行转换：

```
private static int YUV2RGB(int y, int u, int v) {
  // nR = (int)(1.164 * nY + 2.018 * nU);
  // nG = (int)(1.164 * nY - 0.813 * nV - 0.391 * nU);
  // nB = (int)(1.164 * nY + 1.596 * nV);
}
```

细心的读者一定会注意到，这个函数很简单，是简单的 for 循环和加乘法的组合，也一定会消耗很多时间，如果是 512×512 的图像，会重复很多简单计算。在图像处理中，通过优化函数来提高性能的方法有几种，可以用机器原生语言 C 或者汇编来实现，也可以使用机器上的硬件加速来实现。

实际上，在实测中我们也发现了这个问题，但是这是一个演示应用，并不过多占有机器性能，因此就采用了现在这个方案。

这个应用会得到两个回调，一是相机本身产生的图像，二是预览 Preview 的图像，我们会根据应用采用不同的处理方式。

为了实现图像分类，在 ClassifierActivity 里重载了 processImage，并调用分类模型，输出分类的结果：

```
@Override
protected void processImage() {
  rgbFrameBitmap.setPixels(getRgbBytes(),  0,  previewWidth,  0,  0,
```

```
previewWidth, previewHeight);
       final Canvas canvas = new Canvas(croppedBitmap);
       canvas.drawBitmap(rgbFrameBitmap, frameToCropTransform, null);

       // 检查当前的 TF 输入
       if (SAVE_PREVIEW_BITMAP) {
         ImageUtils.saveBitmap(croppedBitmap);
       }
       runInBackground(
           new Runnable() {
             @Override
             public void run() {
               final long startTime = SystemClock.uptimeMillis();
               final  List<Classifier.Recognition>  results  =  classifier.
recognizeImage(croppedBitmap);
               lastProcessingTimeMs = SystemClock.uptimeMillis() - startTime;
               LOGGER.i("Detect: %s", results);
               cropCopyBitmap = Bitmap.createBitmap(croppedBitmap);
               if (resultsView == null) {
                 resultsView = (ResultsView) findViewById(R.id.results);
               }
               resultsView.setResults(results);
               requestRender();
               readyForNextImage();
             }
           });
     }
```

classifier.recognizeImage 实现了图像分类，它的输入是一个 BitMap，然后返回一个分类结果的队列：

```
    final List<Classifier.Recognition> results = classifier.recognizeImage
(croppedBitmap);
```

由于模型的预测需要耗费计算资源和时间，这个函数一定要运行在非主线程上，这里使用了 Runnable。分类的结果是一个类 Recognition，包含了分类的结果，它的定义如下：

```
    public class Recognition {
      // 已识别内容的唯一标识符。特定于类，而不是对象
     private final String id;
      // 显示识别名称
```

```
    private final String title;
     // 一个可排序的分数，表示相对于其他人的认可度，该认可度值越高越好
    private final Float confidence;
    // 源图像中可用于识别对象位置的可选位置
    private RectF location;
```

title 是被分类物体的名称，confidence 是返回的与分类物体相对应的信心数，这个数值越高越好。

下面是图像分类的实现过程，它由几部分构成。

第一步，先把图像的 RGB 数值转换为浮点值：

```
    bitmap.getPixels(intValues, 0, bitmap.getWidth(), 0, 0, bitmap.getWidth(),
bitmap.getHeight());
    for (int i = 0; i < intValues.length; ++i) {
      final int val = intValues[i];
      floatValues[i * 3 + 0] = (((val >> 16) & 0xFF) - imageMean) / imageStd;
      floatValues[i * 3 + 1] = (((val >> 8) & 0xFF) - imageMean) / imageStd;
      floatValues[i * 3 + 2] = ((val & 0xFF) - imageMean) / imageStd;
    }
```

第二步，把浮点值传入模型中进行预测：

```
    inferenceInterface.feed(inputName, floatValues, 1, inputSize, inputSize, 3);
```

第三步，运行模型的会话：

```
    inferenceInterface.run(outputNames, logStats);
```

第四步，取出模型的预测结果：

```
    inferenceInterface.fetch(outputName, outputs);
```

以上几步，除了第一步，基本和用 Python 实现预测的步骤是一样的，只不过是用 Java 实现的。

下一步是对输出的结果进行比较，最后输出的处理比较简单，一是比较信心值，只显示几个比较有意义的结果，二是省去其他信心值比较低的结果。具体实现代码如下：

```
    // 找到最佳分类
    PriorityQueue<Recognition> pq =
        new PriorityQueue<Recognition>(
```

```
          3,
          new Comparator<Recognition>() {
            @Override
            public int compare(Recognition lhs, Recognition rhs) {
              // 故意颠倒，以在队列的最前面建立高度的信心
    return Float.compare(rhs.getConfidence(), lhs.getConfidence());
            }
          });
    for (int i = 0; i < outputs.length; ++i) {
      if (outputs[i] > THRESHOLD) {
        pq.add(
            new Recognition(
               "" + i, labels.size() > i ? labels.get(i) : "unknown",
outputs[i], null));
      }
    }
```

在程序中对每一个步骤都使用了 Trace，我们可以使用工具更好地了解每一步消耗的时间，并根据结果做优化。

```
    Trace.beginSection("feed");
    Trace.endSection();
```

5.2.2　模型

这个应用的模型是 tensorflow_inception_graph.pb，它由下面的代码定义。

```
    private static final String MODEL_FILE = "file:///android_asset/
tensorflow_inception_graph.pb";
      private static final String LABEL_FILE =
          "file:///android_asset/imagenet_comp_graph_label_strings.txt";
```

模型的源文件代码如下：

```
    http_archive(
        name = "inception_v1",
        build_file = "//:models.BUILD",
        sha256    =    "7efe12a8363f09bc24d7b7a450304a15655a57a7751929b2c1593
a71183bb105",
        urls = [
```

```
            "http://storage.googleapis.com/download.tensorflow.org/
models/inception_v1.zip",
            "http://download.tensorflow.org/models/inception_v1.zip",
        ],
    )
```

我们可以下载并解压这个模型，得到 tensorflow_inception_graph.pb，然后运行如下代码：

```
    $ python tensorflow/python/tools/import_pb_to_tensorboard.py --model_dir
tensorflow_inception_graph.pb --log_dir /tmp/log
    $ tensorboard --logdir /tmp/log
```

打开浏览器，输入地址 http://localhost:6006，可以看到 Inception 模型图，如图 5-1 所示。

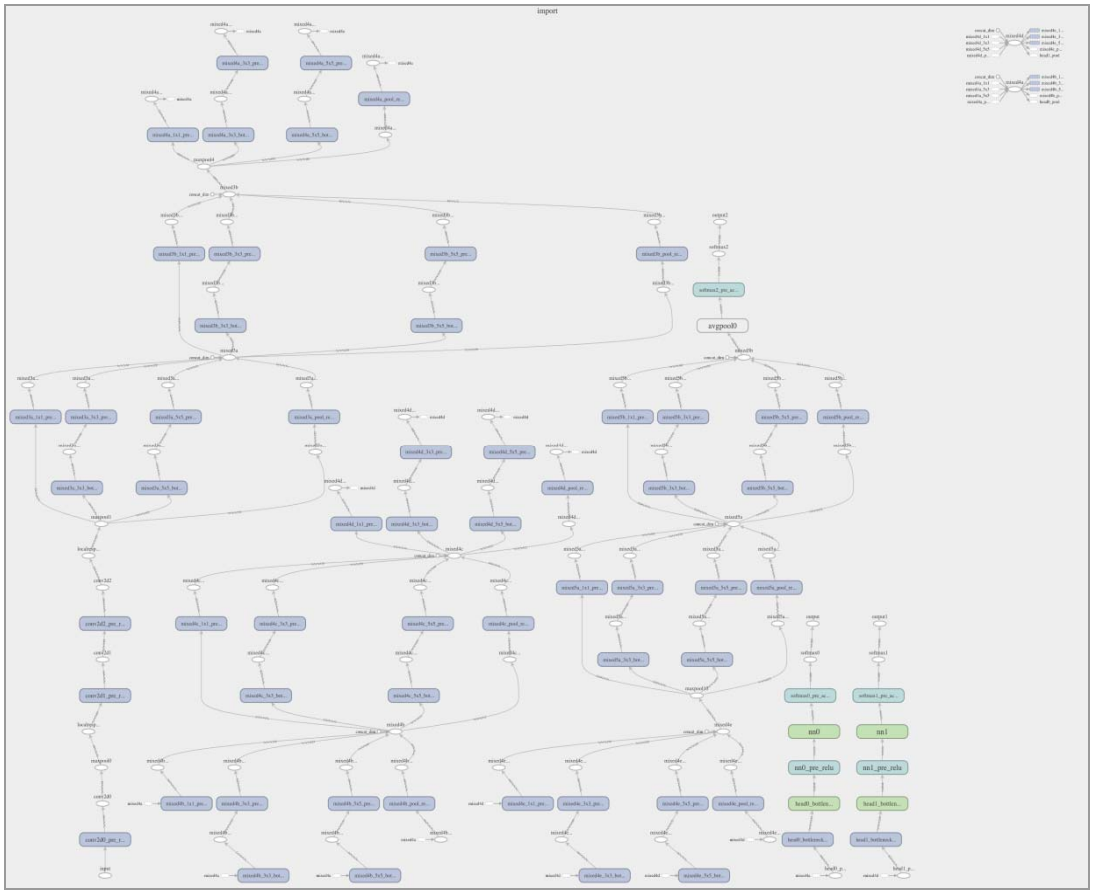

图 5-1　Inception 模型图

5.3 物体检测（Object Detection）

5.3.1 应用

DetectorActivity.java 实现了使用机器学习模型来进行物体检测。这个应用只是对相机的图像进行处理，所以只继承了 OnImageAvailableListener：

```
    public class DetectorActivity extends CameraActivity implements
OnImageAvailableListener
```

这个例子提供了三种模式，它们的定义如下：

```
    private enum DetectorMode {
      TF_OD_API, MULTIBOX, YOLO;
    }
```

默认值是 YOLO，读者可以改变这个值，重新编译运行这个应用，比较三个模型的差异。运行这三个模型的方式是基本一致的。

但是，对于信心（confidence）值，不同的模型会有不同处理，应用里分别定义了相应的数值：

```
    // 跟踪检测的最小置信度
    private static final float MINIMUM_CONFIDENCE_TF_OD_API = 0.6f;
    private static final float MINIMUM_CONFIDENCE_MULTIBOX = 0.1f;
    private static final float MINIMUM_CONFIDENCE_YOLO = 0.25f;
```

另外，模型对于输入图像有不同的要求，它们输入的变量名也不同。而且，应用把模型用不同的文件名保存起来，下面是各自的定义。这些常量其实可以封装起来，由 MODE 来决定，代码看起来会更简单一些。

```
    private static final int MB_INPUT_SIZE = 224;
    private static final int MB_IMAGE_MEAN = 128;
    private static final float MB_IMAGE_STD = 128;
    private static final String MB_INPUT_NAME = "ResizeBilinear";
    private static final String MB_OUTPUT_LOCATIONS_NAME = "output_
locations/Reshape";
    private static final String MB_OUTPUT_SCORES_NAME = "output_
scores/Reshape";
    private static final String MB_MODEL_FILE =
```

```
"file:///android_asset/multibox_model.pb";
    private static final String MB_LOCATION_FILE =
       "file:///android_asset/multibox_location_priors.txt";

    private static final int TF_OD_API_INPUT_SIZE = 300;
    private static final String TF_OD_API_MODEL_FILE =
       "file:///android_asset/ssd_mobilenet_v1_android_export.pb";
    private   static   final   String   TF_OD_API_LABELS_FILE   =   "file:
///android_asset/coco_labels_list.txt";

    private   static   final   String   YOLO_MODEL_FILE   =   "file:///android_
asset/graph-tiny-yolo-voc.pb";
    private static final int YOLO_INPUT_SIZE = 416;
    private static final String YOLO_INPUT_NAME = "input";
    private static final String YOLO_OUTPUT_NAMES = "output";
    private static final int YOLO_BLOCK_SIZE = 32;
```

以下代码生成了 MultiBoxTracker。MultiBoxTracker 负责追踪检测物体并把物体的外框（Box）表示出来。DetectorActivity 会把物体检测的结果传进来，并由 MultiBoxTracker 显示到设备上。

```
    tracker = new MultiBoxTracker(this);
```

下面的代码根据 MODE 生成了检测器的实例。这三个检测器都实现了接口 public interface Classifier，所以都被封装了起来，使用者也不用关心它们的实现细节。

```
    if (MODE == DetectorMode.YOLO) {
      detector =
         TensorFlowYoloDetector.create(
            getAssets(),
            YOLO_MODEL_FILE,
            YOLO_INPUT_SIZE,
            YOLO_INPUT_NAME,
            YOLO_OUTPUT_NAMES,
            YOLO_BLOCK_SIZE);
      cropSize = YOLO_INPUT_SIZE;
     } else if (MODE == DetectorMode.MULTIBOX) {
      detector =
         TensorFlowMultiBoxDetector.create(
            getAssets(),
            MB_MODEL_FILE,
            MB_LOCATION_FILE,
            MB_IMAGE_MEAN,
            MB_IMAGE_STD,
```

```
            MB_INPUT_NAME,
            MB_OUTPUT_LOCATIONS_NAME,
            MB_OUTPUT_SCORES_NAME);
      cropSize = MB_INPUT_SIZE;
    } else {
      try {
        detector = TensorFlowObjectDetectionAPIModel.create(
           getAssets(), TF_OD_API_MODEL_FILE, TF_OD_API_LABELS_FILE, TF_OD_
API_INPUT_SIZE);
        cropSize = TF_OD_API_INPUT_SIZE;
    }
```

在图像识别（Image Classification）的例子中，我们关心被识别的物体是什么，在这个应用里，我们也关心物体的位置，所以在接口 Classifier 里还定义了一个类型为 android.graphics.RectF 的位置变量：

```
    // 源图像中可用于识别对象位置的可选位置
    private RectF location;
```

Tracker 和 detector 完成了这个应用的主要功能。Detector 负责调用模型，返回检测到的物体名称和位置。Tracker 负责把检测到的物体显示到显示器上。在这个应用里，为了显示和追踪物体会花费大量的代码，包括 MultiBoxTracker.java、ObjectTracker.java 和位于 tensorflow/examples/android/jni/object_tracking 下的大量 C++代码。

模型会返回检测到的物体名称和位置。但是，返回的物体名称和位置不是连续的，如果我们只是按照模型返回的数值进行显示是绝对不够的，在现实中一般物体的移动速度是有限的，移动的位置是有关联性的。按照这个原理，我们可以通过代码实现简单的物体追踪。在机器学习的应用里，实际上编写大量的代码还是为了能实现应用的逻辑功能。

这个应用的主要功能是在 processImage()里实现的，分别调用了 Tracker 和 Detector 的函数。一个简化的实现过程大概有三个步骤：

```
    @Override
    protected void processImage() {
      // 第一步，清除已检测的物体，准备下一次显示
      ++timestamp;
      tracker.onFrame(
          previewWidth,
          previewHeight,
          getLuminanceStride(),
          sensorOrientation,
          originalLuminance,
          timestamp);
```

```
      trackingOverlay.postInvalidate();

      runInBackground(
          new Runnable() {
            @Override
            public void run() {
              // 第二步，调用模型，获取被识别物体的队列，包括物体的名称和位置
              LOGGER.i("Running detection on image " + currTimestamp);
              final long startTime = SystemClock.uptimeMillis();
              final       List<Classifier.Recognition>       results       =
detector.recognizeImage(croppedBitmap);
              lastProcessingTimeMs = SystemClock.uptimeMillis() - startTime;

              // 第三步，把已识别的物体显示出来
              tracker.trackResults(mappedRecognitions,         luminanceCopy,
currTimestamp);
              trackingOverlay.postInvalidate();

              requestRender();
              computingDetection = false;
            }
          });
     }
```

我们先看一下 detector 的实现。上面提到了，这个应用支持三个检测器，三个检测器可实现接口 classifier，它们的使用方法比较类似，都实现了 recognizeImage，但也有些不同。以下是三个检测器检测的具体实现方式，读者可以看看它们的不同。

TensorFlowMultiBoxDetector.java

```
    Trace.beginSection("preprocessBitmap");
    // 对图像数据进行预处理，转化为标准化浮点数
    bitmap.getPixels(intValues, 0, bitmap.getWidth(), 0, 0, bitmap.getWidth(),
bitmap.getHeight());

    for (int i = 0; i < intValues.length; ++i) {
      floatValues[i * 3 + 0] = (((intValues[i] >> 16) & 0xFF) - imageMean) /
imageStd;
      floatValues[i * 3 + 1] = (((intValues[i] >> 8) & 0xFF) - imageMean) /
imageStd;
      floatValues[i * 3 + 2] = ((intValues[i] & 0xFF) - imageMean) / imageStd;
    }
    Trace.endSection();
    // 位图预处理
```

```
    // 将输入数据复制到 TensorFlow 中
    Trace.beginSection("feed");
    inferenceInterface.feed(inputName, floatValues, 1, inputSize, inputSize, 3);
    Trace.endSection();

    // 运行推理调用
    Trace.beginSection("run");
    inferenceInterface.run(outputNames, logStats);
    Trace.endSection();

    // 将输出张量复制回输出数组
    Trace.beginSection("fetch");
    final float[] outputScoresEncoding = new float[numLocations];
    final float[] outputLocationsEncoding = new float[numLocations * 4];
    inferenceInterface.fetch(outputNames[0], outputLocationsEncoding);
    inferenceInterface.fetch(outputNames[1], outputScoresEncoding);
    Trace.endSection();
```

TensorFlowObjectDetectionAPIModel.java：

```
    Trace.beginSection("preprocessBitmap");
    // 预处理图像数据，从 0x00rrggbb 格式的 int 中提取 r、g 和 b 字节
    bitmap.getPixels(intValues, 0, bitmap.getWidth(), 0, 0, bitmap.getWidth(),
bitmap.getHeight());

    for (int i = 0; i < intValues.length; ++i) {
      byteValues[i * 3 + 2] = (byte) (intValues[i] & 0xFF);
      byteValues[i * 3 + 1] = (byte) ((intValues[i] >> 8) & 0xFF);
      byteValues[i * 3 + 0] = (byte) ((intValues[i] >> 16) & 0xFF);
    }
    Trace.endSection();
    // 位图预处理
    // 将输入数据复制到 TensorFlow 中
    Trace.beginSection("feed");
    inferenceInterface.feed(inputName, byteValues, 1, inputSize, inputSize, 3);
    Trace.endSection();

    // 运行推理调用
    Trace.beginSection("run");
    inferenceInterface.run(outputNames, logStats);
    Trace.endSection();
```

TensorFlowYoloDetector.java：

```
    Trace.beginSection("preprocessBitmap");
    // 对图像数据进行预处理，转化为标准化浮点数
```

```
    bitmap.getPixels(intValues, 0, bitmap.getWidth(), 0, 0, bitmap.getWidth(),
bitmap.getHeight());

    for (int i = 0; i < intValues.length; ++i) {
      floatValues[i * 3 + 0] = ((intValues[i] >> 16) & 0xFF) / 255.0f;
      floatValues[i * 3 + 1] = ((intValues[i] >> 8) & 0xFF) / 255.0f;
      floatValues[i * 3 + 2] = (intValues[i] & 0xFF) / 255.0f;
    }
    Trace.endSection(); //位图预处理
    // 将输入数据复制到 TensorFlow 中
    Trace.beginSection("feed");
    inferenceInterface.feed(inputName, floatValues, 1, inputSize, inputSize, 3);
    Trace.endSection();

    timer.endSplit("ready for inference");

    // 运行推理调用
    Trace.beginSection("run");
    inferenceInterface.run(outputNames, logStats);
    Trace.endSection();
```

在 TensorFlowObjectDetectionAPIModel 里，模型的数据输入类型是 byte，而 TensorFlowYoloDetector 和 TensorFlowMultiBoxDetector 的输入类型是 float，一个是定点数，另一个是浮点数。

5.3.2　模型

这个应用用到了三个模型，其中 inception 模型在上面已经介绍了。现在来看看另外两个，一个是 Mobile Net。

```
    http_archive(
        name = "mobile_ssd",
        build_file = "//:models.BUILD",
        sha256 = "bddd81ea5c80a97adfac1c9f770e6f55cbafd7cce4d3bbe15fbeb041e6b8
f3e8",
        urls = [
            "http://storage.googleapis.com/download.tensorflow.org/models/
object_detection/ssd_mobilenet_v1_android_export.zip",
            "http://download.tensorflow.org/models/object_detection/ssd_
mobilenet_v1_android_export.zip",
        ],
    )
```

使用前面介绍过的过程，模型结构的视图如图 5-2 所示。

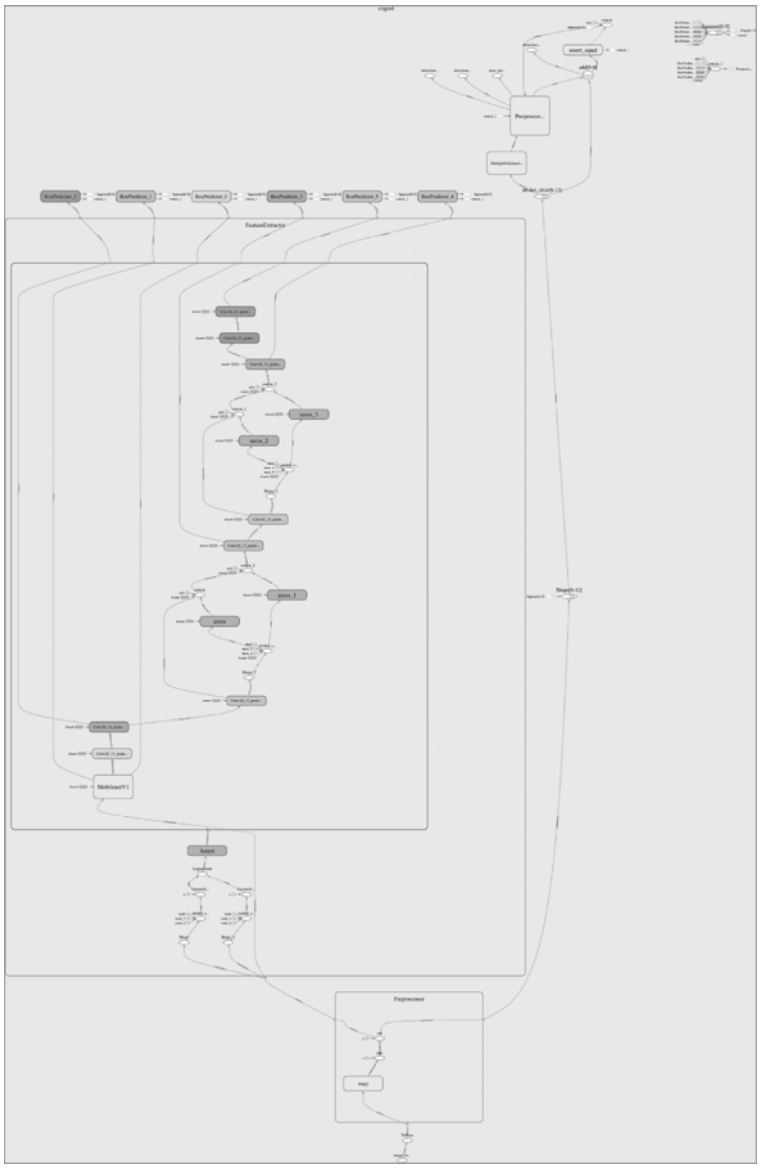

图 5-2　SSD Mobile Net 模型图

另外一个是 Mobile Multibox 模型，代码如下：

```
http_archive(
    name = "mobile_multibox",
    build_file = "//:models.BUILD",
```

```
        sha256                                                                    =
"859edcddf84dddb974c36c36cfc1f74555148e9c9213dedacf1d6b613ad52b96",
        urls = [
            "http://storage.googleapis.com/download.tensorflow.org/models/
mobile_multibox_v1a.zip",
            "http://download.tensorflow.org/models/mobile_multibox_v1a.zip",
        ],
    )
```

模型的可视化视图结构如图 5-3 所示。

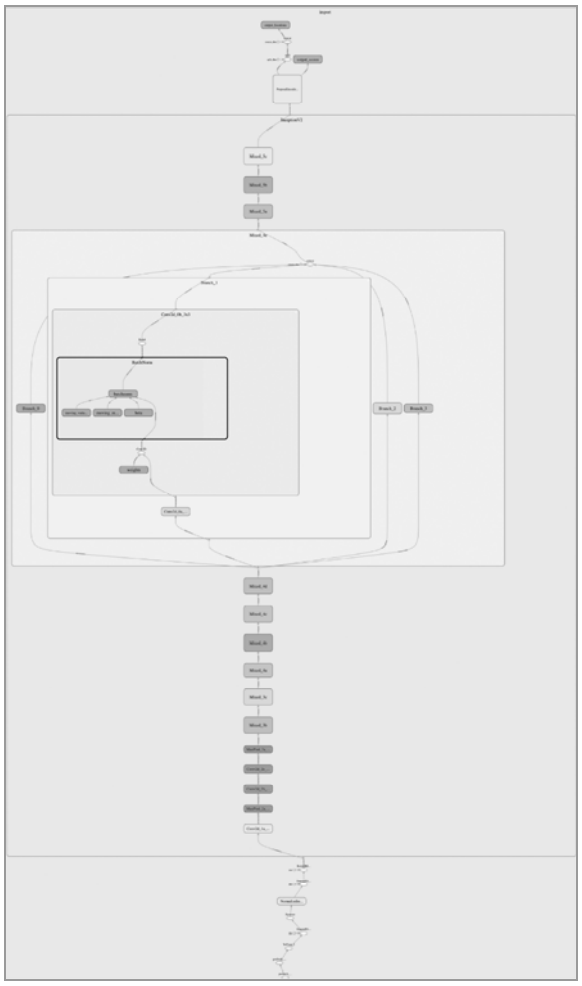

图 5-3　Mobile Multibox 结构图

5.4 时尚渲染（Stylization）

5.4.1 应用

时尚渲染的应用是由 StylizeActivity.java 实现的。主要的功能也是由 processImage()调用 stylizeImage()来实现的。

```
    ++frameNum;
    bitmap.getPixels(intValues, 0, bitmap.getWidth(), 0, 0, bitmap.getWidth(),
bitmap.getHeight());

    for (int i = 0; i < intValues.length; ++i) {
      final int val = intValues[i];
      floatValues[i * 3] = ((val >> 16) & 0xFF) / 255.0f;
      floatValues[i * 3 + 1] = ((val >> 8) & 0xFF) / 255.0f;
      floatValues[i * 3 + 2] = (val & 0xFF) / 255.0f;
    }

    // 将输入数据复制到 TensorFlow
    LOGGER.i("Width: %s , Height: %s", bitmap.getWidth(), bitmap.getHeight());
    inferenceInterface.feed(
        INPUT_NODE, floatValues, 1, bitmap.getWidth(), bitmap.getHeight(), 3);
    inferenceInterface.feed(STYLE_NODE, styleVals, NUM_STYLES);

    inferenceInterface.run(new String[] {OUTPUT_NODE}, isDebug());
    inferenceInterface.fetch(OUTPUT_NODE, floatValues);

    for (int i = 0; i < intValues.length; ++i) {
      intValues[i] =
          0xFF000000
              | (((int) (floatValues[i * 3] * 255)) << 16)
              | (((int) (floatValues[i * 3 + 1] * 255)) << 8)
              | ((int) (floatValues[i * 3 + 2] * 255));
    }

    bitmap.setPixels(intValues, 0, bitmap.getWidth(), 0, 0, bitmap.getWidth(),
bitmap.getHeight());
```

实现的过程也相对简单，通过调用 inferenceInterface 的 feed、run、fetch 后得到像素

的浮点值，然后转换成对应的 RGB 的定点数值即可。

5.4.2　模型

时尚模型源文件的定义如下：

```
    http_archive(
        name = "stylize",
        build_file = "//:models.BUILD",
        sha256   =   "3d374a730aef330424a356a8d4f04d8a54277c425e274ecb7d9c83aa
912c6bfa",
        urls = [
            "http://storage.googleapis.com/download.tensorflow.org/models/
stylize_v1.zip",
            "http://download.tensorflow.org/models/stylize_v1.zip",
        ],
    )
```

我们把模型文件下载下来，按照上面的例子，转换成一个可视化的图形，代码如下：

```
    $ python tensorflow/python/tools/import_pb_to_tensorboard.py --model_dir
stylize_quantized.pb --log_dir /tmp/log
```

被细化的 Style 模型如图 5-4 所示。

5.5　声音识别（Speech Recognization）

5.5.1　应用

以上的几个例子都是和图像处理有关的，本节这个例子是和声音有关的，也是典型的机器学习的例子。这个应用会识别声音，并把命令显示出来。由于要使用麦克风，所以要把权限加上：

```
    <uses-permission android:name="android.permission.RECORD_AUDIO" />
```

这个应用由 tensorflow/examples/android/src/org/tensorflow/demo/SpeechActivity.java 实现。当应用启动的时候，同时启动麦克风的录音和声音识别，代码的实现方法如下：

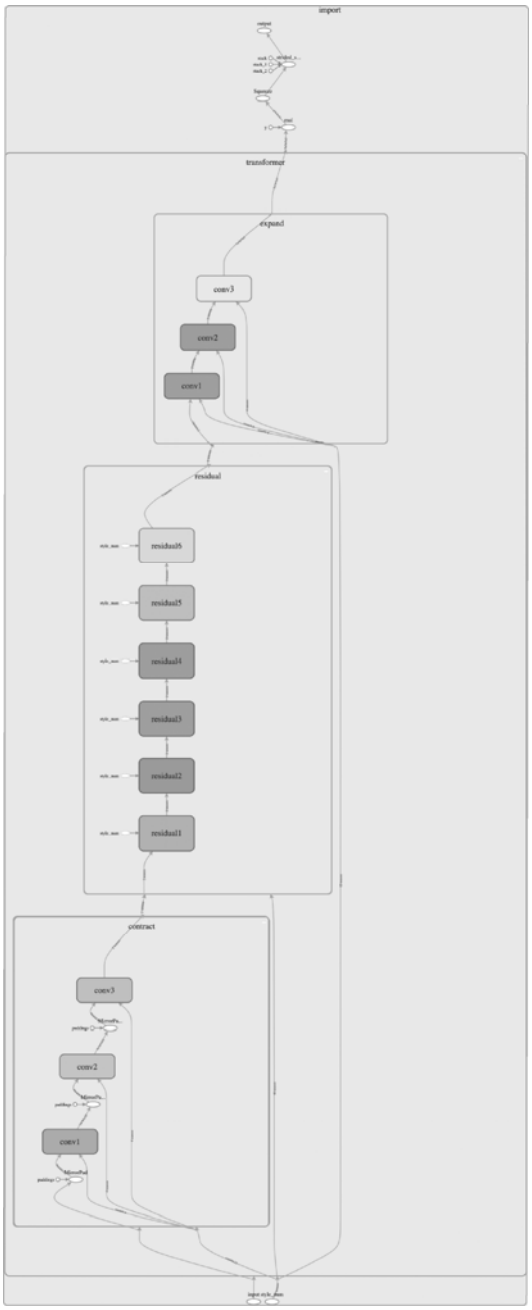

图 5-4　被细化的 Style 模型图

```
    protected void onCreate(Bundle savedInstanceState) {
      // 加载 TensorFlow 模型
      inferenceInterface = new TensorFlowInferenceInterface(getAssets(),
MODEL_FILENAME);

      // 启动录制并识别线程
      requestMicrophonePermission();
      startRecording();
      startRecognition();
    }
```

startRecording 实现的主要功能是，启动一个线程。在这个线程里，首先设定录音设备，然后启动录音。在录音开始后，把数据存到一个缓存数组里，供声音识别使用。代码的简单实现方法如下：

```
      private void record() {
        android.os.Process.setThreadPriority(android.os.Process.THREAD_
PRIORITY_AUDIO);

        // 预估设备需要的缓冲区大小
        int bufferSize =
            AudioRecord.getMinBufferSize(
                SAMPLE_RATE, AudioFormat.CHANNEL_IN_MONO, AudioFormat.ENCODING_
PCM_16BIT);
        if (bufferSize == AudioRecord.ERROR || bufferSize == AudioRecord.
ERROR_BAD_VALUE) {
          bufferSize = SAMPLE_RATE * 2;
        }
        short[] audioBuffer = new short[bufferSize / 2];

        AudioRecord record =
            new AudioRecord(
                MediaRecorder.AudioSource.DEFAULT,
                SAMPLE_RATE,
                AudioFormat.CHANNEL_IN_MONO,
                AudioFormat.ENCODING_PCM_16BIT,
                bufferSize);

        record.startRecording();

        // 循环并收集音频数据并将其复制到循环缓冲区
          while (shouldContinue) {
          int numberRead = record.read(audioBuffer, 0, audioBuffer.length);
          int maxLength = recordingBuffer.length;
          int newRecordingOffset = recordingOffset + numberRead;
          int secondCopyLength = Math.max(0, newRecordingOffset - maxLength);
          int firstCopyLength = numberRead - secondCopyLength;
```

```
          //存储所有数据，以便识别线程访问
          //线程将从这个缓冲区被复制到自己的缓冲区中，这个过程是加锁的
          recordingBufferLock.lock();
          try {
            System.arraycopy(audioBuffer, 0, recordingBuffer, recordingOffset,
firstCopyLength);
            System.arraycopy(audioBuffer, firstCopyLength, recordingBuffer, 0,
secondCopyLength);
            recordingOffset = newRecordingOffset % maxLength;
          } finally {
            recordingBufferLock.unlock();
          }
        }
      }
```

注意，代码设定的声音格式是 SAMPLE_RATE = 16000，声音取样的频率是 16k，采用的一个声道是 CHANNEL_IN_MONO，声音样本的数据格式是 16bit 和 AudioFormat.ENCODING_PCM_16BIT。声音识别主要是由 startRecognition()和 recognize()实现的，也是在另外一个线程上实现的。实现的主要代码如下：

```
    // 输入介于-1.0f 和 1.0f 之间的浮点值
      for (int i = 0; i < RECORDING_LENGTH; ++i) {
        floatInputBuffer[i] = inputBuffer[i] / 32767.0f;
      }

      // 运行模型
      inferenceInterface.feed(SAMPLE_RATE_NAME, sampleRateList);
      inferenceInterface.feed(INPUT_DATA_NAME,  floatInputBuffer,  RECORDING_
LENGTH, 1);
      inferenceInterface.run(outputScoresNames);
      inferenceInterface.fetch(OUTPUT_SCORES_NAME, outputScores);
```

首先，要做数据类型的转换，把 16 位的定点数转换为浮点数。然后，还是在调用 feed、run 和 fetch 后，把预测的结果转换为命令显示出来。

5.5.2 模型

时尚模型源文件的定义如下：

```
    http_archive(
        name = "speech_commands",
        build_file = "//:models.BUILD",
        sha256 = "c3ec4fea3158eb111f1d932336351edfe8bd515bb6e87aad4f25dbad0a
600d0c",
```

```
        urls = [
            "http://storage.googleapis.com/download.tensorflow.org/models/
speech_commands_v0.01.zip",
            "http://download.tensorflow.org/models/speech_commands_
v0.01.zip",
        ],
    )
```

我们把模型文件下载下来，按照上面的例子，转换成一个可视化的图形，代码如下：

```
    $ python tensorflow/python/tools/import_pb_to_tensorboard.py --model_dir
conv_actions_frozen.pb --log_dir /tmp/log
```

模型的视图如图 5-5 所示。

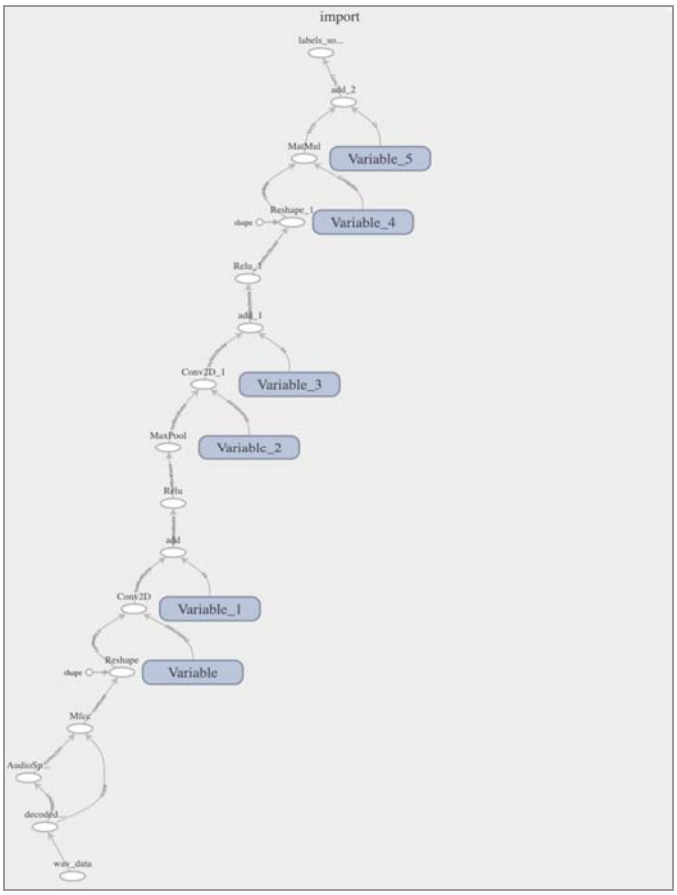

图 5-5　Speech 模型图