YOLOv5 tflite Android 部署

项目地址：**https://github.com/yyccR/yolov5-tflite-android**

一、 yolov5 tflite

（一）、demo演示
（二）、tflite量化原理
（三）、tflite量化导出, 模型文件细节

二、Android部署(以摄像头检测为例)

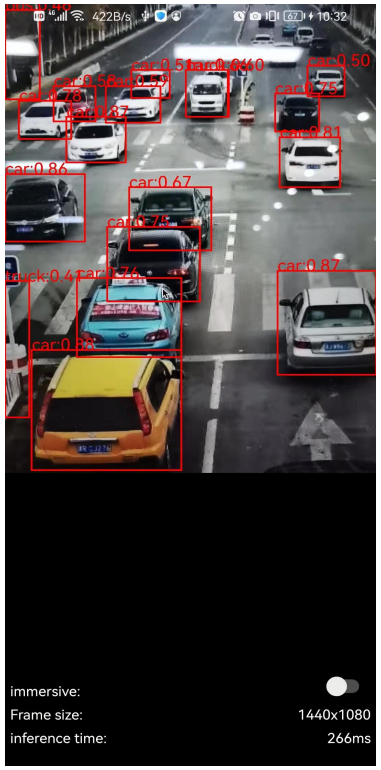
（零）、android部署整体流程
（一）、相关依赖
（二）、模型加载，输入与输出定义
（三）、NNAPI代理，gpu代理，多线程加速
（四）、tflite task和support的区别
（五）、布局文件示例
（六）、cameraX 摄像头数据细节
（七）、摄像头逐帧分析器
（八）、异步计算，避免UI刷新卡顿

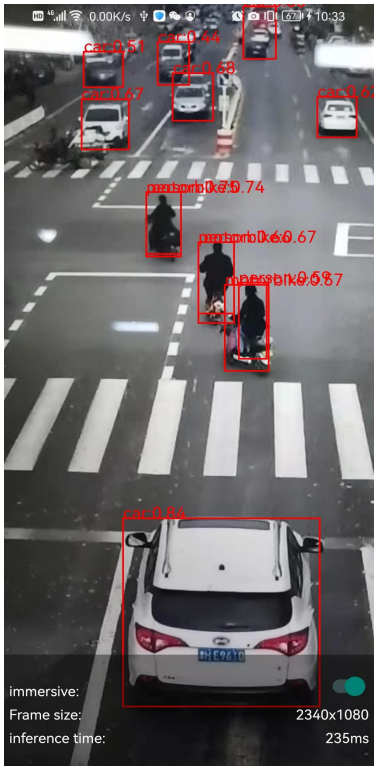
三、替换自己的yolov5模型

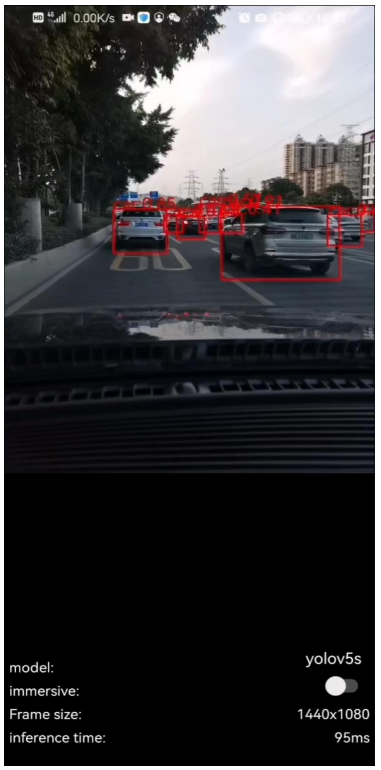
（一）、assert文件替换
（二）、输入与输出修改

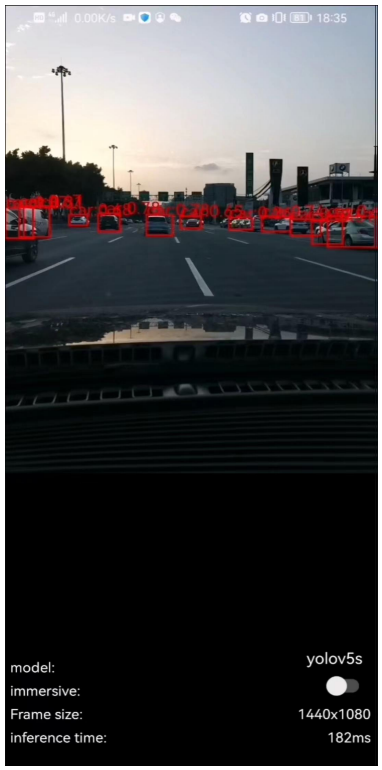
一、 yolov5 tflite

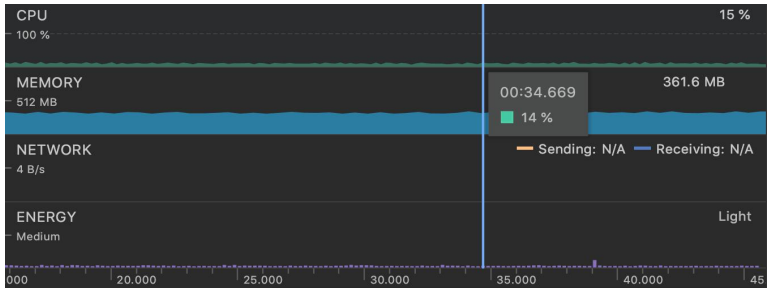
（一）、demo演示，yolov5-tflite-android cpu(kirin 980)



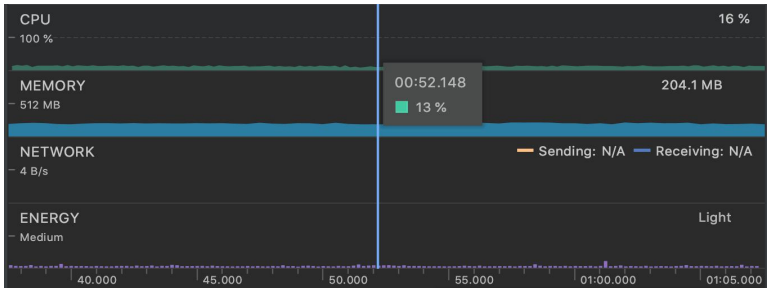








yolo5m-fp16：320x320
内存占用约360m, 推理时延200-260ms



yolo5s-fp16：320x320
内存占用约200m, 推理时延80-180ms

一、 yolov5 tflite

（二）、tflite量化原理-概念和角色

Q: 通常说的量化，是在量化什么？

A: 在tensorflow-lite的论文里面提到[1]，量化是将使用较高浮点数(通常是32位)的神经网络近似为一个低比特宽度的神经网络的过程。

Q: 量化在模型压缩是什么角色？

A: 目前常见的模型压缩技术有：剪枝、量化、蒸馏，低秩分解，权值共享等，量化属于模型压缩中的一环，但是也是效果比较明显的一环，下图展示了模型压缩的一些技术点，参考论文[2][3]。

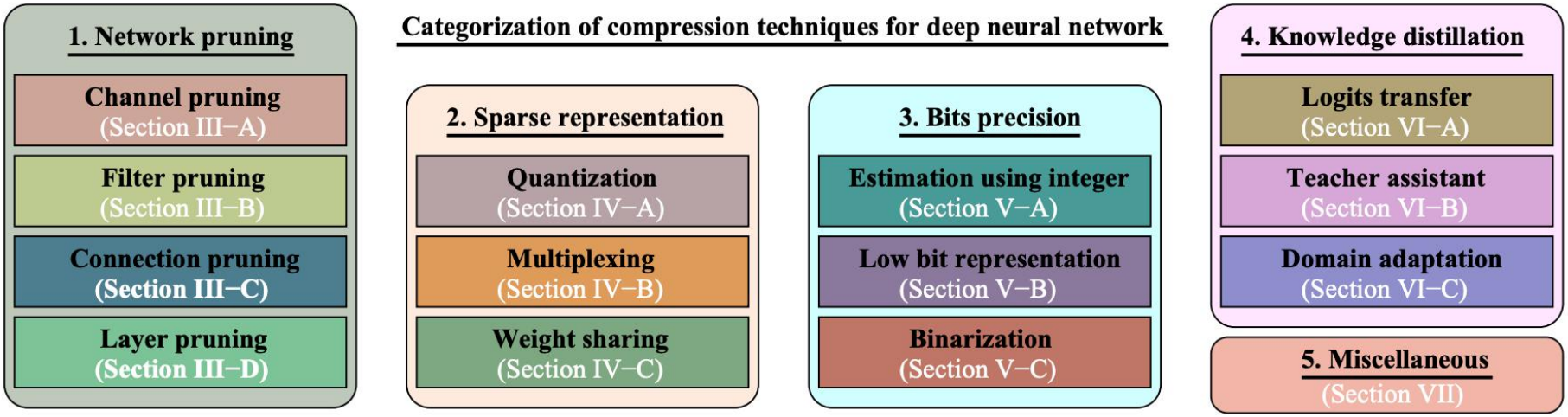


Fig. 1: Overview of different categories of compression techniques for deep neural network.

1. 《Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference》https://arxiv.org/pdf/2103.13630.pdf
2. 《An Survey of Neural Network Compression》https://arxiv.org/pdf/2006.03669.pdf
3. 《A Survey on Deep Neural Network Compression: Challenges, Overview, and Solutions》https://arxiv.org/pdf/2010.03954.pdf

一、 yolov5 tflite

(二) 、tflite量化原理-理论细节

量化有训练后量化(PTQ)，量化感知训练(QAT)，下面主要围绕PTQ里的推理部分：

tflite训练后量化推理大概流程（int8为例）：

1. 输入量化后的数据和权重

2. 通过反量化公式计算矩阵卷积:

$$S_3(q_3^{(i,k)} - Z_3) = \sum_{j=1}^{N} S_1(q_1^{(i,j)} - Z_1) S_2(q_2^{(j,k)} - Z_2)$$

3. 将int32 bias加到矩阵卷积结果,其中bias的量化参数为:

$$S_{\text{bias}} = S_1 S_2, \quad Z_{\text{bias}} = 0.$$

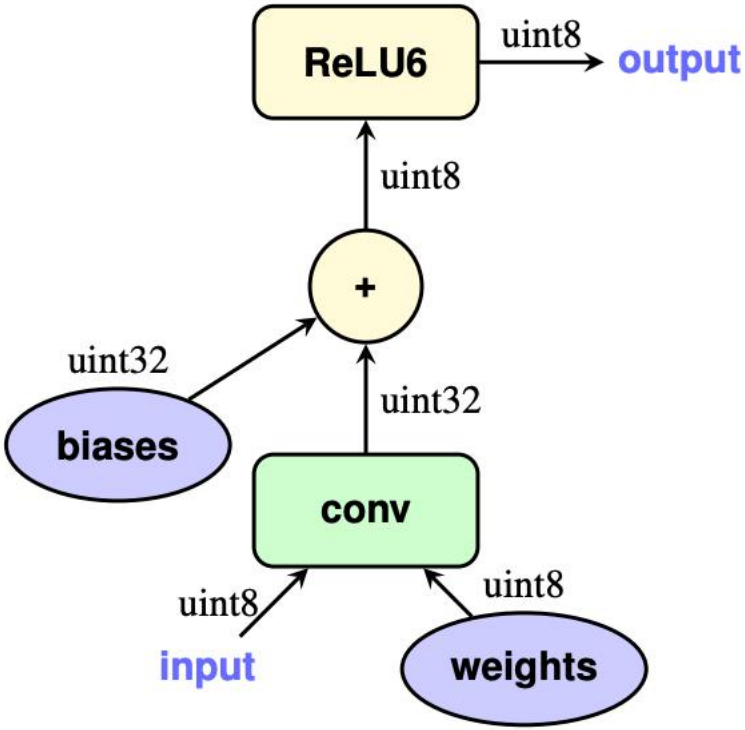
4. 如果卷积之后包含bn层，则将bn包含到卷积计算中:

$$w_{\text{fold}} := \frac{\gamma w}{\sqrt{EMA(\sigma_B^2) + \varepsilon}}.$$

5. 如果卷积/bn层之后包含激活层,比如ReLU,那么ReLU也会直接通过区间截断操作包含到对应的卷积计算中,如果不能包含的进去的, 则会做相应的定点计算近似逼近.

6. 最后将输出结果量化到int8

其中4跟5，是根据具体情况在导出量化模型的时候已经做好，推理的时候直接无需再多额外参数合并。



一、 yolov5 tflite

（二）、tflite量化原理-理论细节

tflite训练后量化推理大概流程（int8为例）：

1. 输入量化后的数据和权重：导出

通常输入数据以及模型权重都是float32/ float64类型，量化模型会把float32/float64类型转成float16/int8格式，下面以int8为例：

```
# 模型转换时，提供represent_dataset方法帮助计算int8量化参数，该方法根据自己数据读取提供部分即可
def representative_dataset():
    for _ in range(100):
      data = np.random.rand(1, 244, 244, 3)
      yield [data.astype(np.float32)]

# 读取keras模型
converter = tf.lite.TFLiteConverter.from_keras_model(keras_model)
# 指定优化器，DEFAULT会自己权衡模型大小和延迟性能
converter.optimizations = [tf.lite.Optimize.DEFAULT]
# 指定代表数据样本
converter.representative_dataset = representative_dataset
# 指定模型量化类型为int8
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.target_spec.supported_types = []
# 指定输入和输出类型都是int8，所以才需要提供represent_dataset
converter.inference_input_type = tf.uint8  # or tf.int8
converter.inference_output_type = tf.uint8  # or tf.int8
converter.experimental_new_quantizer = False
tflite_model = converter.convert()
open(f, "wb").write(tflite_model)
```

一、 yolov5 tflite

（二）、tflite量化原理-理论细节

tflite训练后量化推理大概流程（int8为例）：

1. 输入量化后的数据和权重：导入

java示例代码，具体看：https://github.com/yyccR/yolov5-tflite-android/blob/master/app/src/main/java/com/example/yolov5tfliteandroid/detector/Yolov5TFLiteDetector.java#L125-L172

```
# 在具体部署推理阶段，需要对客户端接收的图片数据量化为int8再丢到模型里
imageProcessor =
                new ImageProcessor.Builder()
                        .add(new ResizeOp(INPNUT_SIZE.getHeight(), INPNUT_SIZE.getWidth(),ResizeOp.ResizeMethod.BILINEAR))
                        .add(new NormalizeOp(0, 255))
                        .add(new QuantizeOp(input5SINT8QuantParams.getZeroPoint(), input5SINT8QuantParams.getScale()))
                        .add(new CastOp(DataType.UINT8)) # 这里将数据转Int8
                        .build();
yolov5sTfliteInput = new TensorImage(DataType.UINT8);

# 同时对模型输出的int8，反量化为float
probabilityBuffer = TensorBuffer.createFixedSize(OUTPUT_SIZE, DataType.UINT8);
tflite.run(yolov5sTfliteInput.getBuffer(), probabilityBuffer.getBuffer());
TensorProcessor tensorProcessor = new TensorProcessor.Builder()
                # 这里Int8反量化为float
                .add(new DequantizeOp(output5SINT8QuantParams.getZeroPoint(), output5SINT8QuantParams.getScale()))
                .build();
probabilityBuffer = tensorProcessor.process(probabilityBuffer);

# 注意：
input5SINT8QuantParams.getZeroPoint(), input5SINT8QuantParams.getScale())
output5SINT8QuantParams.getZeroPoint(), output5SINT8QuantParams.getScale())
都是量化完模型已知的参数.
```

一、 yolov5 tflite

（二）、tflite量化原理-理论细节

tflite训练后量化推理大概流程（int8为例）：

2. 通过反量化公式计算矩阵卷积:

$$S_3(q_3^{(i,k)} - Z_3) = \sum_{j=1}^{N} S_1(q_1^{(i,j)} - Z_1) S_2(q_2^{(j,k)} - Z_2)$$

公式里是假定两个矩阵相乘，分别为量化后的input tensor q1和filter kernel q2，另外S1,S2,S3缩放因子，用来缩放值域，Z1,Z2,Z3为零点，用来为对齐浮点和量化值0值，在没量化前，矩阵的乘法为：$r_3 = r_1 r_2$ r表示浮点数，r(real value)和q(quantization value)的关系可以用下面公式表示：

$$r = S(q - Z)$$

进一步展开反量化公式可以得到最后卷积乘法的量化输出：

$$q_3^{(i,k)} = Z_3 + M \sum_{j=1}^{N} (q_1^{(i,j)} - Z_1)(q_2^{(j,k)} - Z_2)$$

其中 $M := \frac{S_1 S_2}{S_3}$

那么，到目前为止，除了还没加上bias，tflite中int8卷积计算大概就是上面公式所示，该公式中除了M的计算涉及浮点外，另外的所有计算都是在整数范围下，为了让整个计算过程能尽量减少浮点参与，tflite特地针对M的计算进行了优化，具体为，将M替换为：

$$M = 2^{-n} M_0$$

由于S1,S2,S3都是已知，且通过大量观察得到M通常都是位于区间(0,1)，这样就可以通过采用定点数 $M_0 \in [0.5, 1)$ 以及位运算近似得到M，而定点数运算在gemmlowp库已有高效的实现。

tflite 关于M的部分：
https://github.com/tensorflow/tensorflow/blob/4952f981be07b8bf508f8226f83c10cdafa3f0c4/tensorflow/contrib/lite/kernels/internal/reference/reference_ops.h#L36-L42
gemmlowp 关于M_0以及位移运算：
https://github.com/google/gemmlowp/blob/fcf32e7a0a4d2af46e63eccf0c8fa4d83d0311c5/fixedpoint/fixedpoint.h#L256-L287

一、 yolov5 tflite

（二）、tflite量化原理-理论细节

tflite训练后量化推理大概流程（int8为例）：

为了更加清晰了解M的近似计算过程，大概看一下源码，tflite中具体计算M是用下面内联函数：

```
inline int32 MultiplyByQuantizedMultiplierSmallerThanOne(int32 x, int32 quantized_multiplier, int right_shift) {
  using gemmlowp::RoundingDivideByPOT;
  using gemmlowp::SaturatingRoundingDoublingHighMul;
  return RoundingDivideByPOT(
      SaturatingRoundingDoublingHighMul(x, quantized_multiplier), right_shift);
}
```

其中：
int32 x是输入Tensor和卷积核的矩阵计算结果，对应上一页公式中的：$\sum_{j=1}^{N}(q_1^{(i,j)}-Z_1)(q_2^{(j,k)}-Z_2)$，再加上bias。
int32 quantized_multiplier就是M_0，用int32表示，这个在gemmlowp里面表示的是32位的定点小数。
int right_shift表示右移的位数，对应$M=2^{-n}M_0$ 中的n。

函数里面调用的gemmlowp库方法：
SaturatingRoundingDoublingHighMul为定点乘法，通过将32位x和quantized_multiplier表示到64位，再通过补充精度最后再降低到32位。
RoundingDivideByPOT为位运算操作，具体将上面的结果处理到跟原M大小。

一、 yolov5 tflite

（二）、tflite量化原理-理论细节

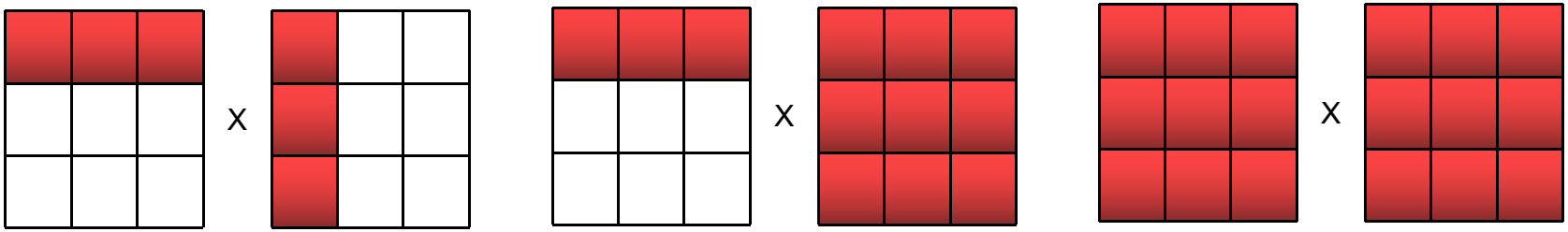
tflite训练后量化推理大概流程（int8为例）：

2. 通过反量化公式计算矩阵卷积，继续上两页的公式：

$$q_3^{(i,k)} = Z_3 + M\sum_{j=1}^{N}(q_1^{(i,j)} - Z_1)(q_2^{(j,k)} - Z_2)$$

为了更加高效得计算这个矩阵乘法，优化掉里面涉及的$2N^3$=2N*N*N次减法

2N次减法　　2N*N次减法　　2N*N*N次减法



为了降低这个复杂度，tflite里把上面公式进一步展开得到：

$$q_3^{(i,k)} = Z_3 + M\left(NZ_1Z_2 - Z_1a_2^{(k)} - Z_2\bar{a}_1^{(i)} + \sum_{j=1}^{N}q_1^{(i,j)}q_2^{(j,k)}\right)$$

其中：$a_2^{(k)} := \sum_{j=1}^{N} q_2^{(j,k)}$，$\bar{a}_1^{(i)} := \sum_{j=1}^{N} q_1^{(i,j)}$.，这里面$2N^3$次减法就被分解掉，$a_1$,$a_2$是N次求和，最终整个卷积计算关于$a_1$,$a_2$的加法运算是$2N^2$，另外公式中最后一部分是核心 $\sum_{j=1}^{N} q_1^{(i,j)}q_2^{(j,k)}$ ，这部分与传统卷积计算复杂度相同$2N^3$，区别在于这里是整型的矩阵乘法。

一、 yolov5 tflite

（二）、tflite量化原理-理论细节

tflite训练后量化推理大概流程（int8为例）：

3. 将int32 bias加到矩阵卷积结果：

$$S_3(q_3^{(i,k)} - Z_3) = \sum_{j=1}^{N} S_1(q_1^{(i,j)} - Z_1) S_2(q_2^{(j,k)} - Z_2)$$

上面公式的原型是：$r_3 = r_1 r_2$ ，是我们省略了bias的结果，实际应该是：$r_3 = r_1 r_2 + \beta$,代入上面公式可以得到：

$$S_3(q_3^{(i,k)} - Z_3) = \sum_{j=1}^{N} S_1(q_1^{(i,j)} - Z_1) S_2(q_2^{(j,k)} - Z_2) + S_{bias}(\beta - Z_{bias})$$

为了能让bias和前面的矩阵乘法在公式分解的时候共用一个权重M，这里tflite对bias的2个量化参数表示为：

$$S_{\text{bias}} = S_1 S_2, \quad Z_{\text{bias}} = 0.$$

得到最终的公式为：

$$q_3^{(i,k)} = Z_3 + M(\sum_{j=1}^{N}(q_1^{(i,j)} - Z_1)(q_2^{(j,k)} - Z_2) + \beta)$$

具体实现时，由于q累乘最后为了避免数值溢出用int32表示，所以一开始 β 的存储也是用int32。

bias相关计算具体实现可以参考：
https://github.com/tensorflow/tensorflow/blob/4952f981be07b8bf508f8226f83c10cdafa3f0c4/tensorflow/contrib/lite/toco/graph_transformations/quantize.cc#L171-L197
最后量化后卷积计算的具体实现可以参考：
https://github.com/tensorflow/tensorflow/blob/4952f981be07b8bf508f8226f83c10cdafa3f0c4/tensorflow/contrib/lite/kernels/internal/reference/reference_ops.h#L248-L314

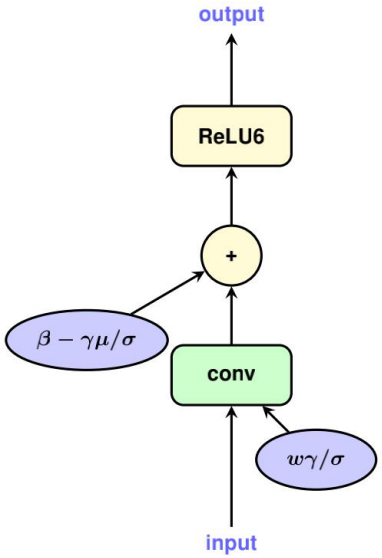
一、 yolov5 tflite

（二）、tflite量化原理-理论细节

tflite训练后量化推理大概流程（int8为例）：

4. 如果卷积之后包含bn层，则将bn包含到卷积计算中，论文里面的fold公式以及对应的流程图：

$$w_{\text{fold}} := \frac{\gamma w}{\sqrt{EMA(\sigma_B^2) + \varepsilon}}.$$



但是上面还少了个bias，我们再补充一下最后fold bn为：

$$y = \gamma_B \frac{[(\sum_{i=1}^{N} w_i x_i + \beta_{bias}) - \mu_B]}{\sqrt{EMA(\sigma_B^2) + \epsilon}} + \beta_B$$

最后得到：

$$w_{fold} = \frac{\gamma_B w}{\sqrt{EMA(\sigma_B^2) + \epsilon}}$$

$$\beta_{bias\ fold} = \frac{\gamma_B(\beta_{bias} - \mu_B)}{\sqrt{EMA(\sigma_B^2) + \epsilon}} + \beta_B$$

一、 yolov5 tflite

（二）、tflite量化原理-理论细节

tflite训练后量化推理大概流程（int8为例）：

5. 如果卷积/bn层之后包含激活层,比如ReLU,那么ReLU也会直接通过区间截断操作包含到对应的卷积计算中,如果不能包含的进去的, 则会做相应的定点计算近似逼近.

这里举两个个例子，ReLU和sigmoid：
首先我们看最开始卷积计算公式：

$$S_3(q_3^{(i,k)} - Z_3) = \sum_{j=1}^{N} S_1(q_1^{(i,j)} - Z_1) S_2(q_2^{(j,k)} - Z_2)$$

此时如果接一层relu，则有：

$$S_4(q_4 - Z_4) = ReLU(S_3(q_3 - Z_3))$$

$$S_4(q_4 - Z_4) = \begin{cases} S_3(q_3 - Z_3), & S_3(q_3 - Z_3) > 0 \\ 0, & S_3(q_3 - Z_3) <= 0 \end{cases}$$

在tflite里面，由于计算计算的时候用int32表示最后的结果，最后再丢到下一层时会转成unit8，所以上面公式应该再表示为：

$$S_4(q_4 - Z_4) = \begin{cases} clip(S_3(q_3 - Z_3), 0, 255), & S_3(q_3 - Z_3) > 0 \\ 0, & S_3(q_3 - Z_3) <= 0 \end{cases}$$

可以看到clip(*,0,255)方法就是把上一层的数据规约到[0,255]，这个操作就等于ReLU，而且这个操作在每一层卷积都会进行，所以当我们后面再接一层ReLU的时候，其实是可以直接省略的，最后直接用下面公式表示ReLU融合的结果：

$$S_4(q_4 - Z_4) = clip(S_3(q_3 - Z_3), 0, 255)$$

但是当用其他非线性激活函数就不能省略了，比如sigmoid, leakyReLU.

一、 yolov5 tflite

（二）、tflite量化原理-理论细节

tflite训练后量化推理大概流程（int8为例）：

5. 再看看sigmoid，

具体实现可以参考：
https://github.com/google/gemmlowp/blob/master/fixedpoint/fixedpoint.h#L877-L898
关于sigmoid近似逼近的几种方法：
http://umpir.ump.edu.my/id/eprint/21765/1/Sigmoid%20function%20implementation%20using%20the%20unequal%20segmentation.pdf

在gemmlowp里，sigmoid在量化计算主要是通过下面2个步骤实现：
(1). 将sigmoid拆成下面2部分，先计算D：

$$D = e^{-x}$$

$$sigmoid(x) = \frac{1}{1+D}$$

采用定点表示预先计算好的几个指数结果，再具体计算时利用查表法估计，预先计算如下值：

```
GEMMLOWP_EXP_BARREL_SHIFTER(-2, 1672461947);
GEMMLOWP_EXP_BARREL_SHIFTER(-1, 1302514674);
GEMMLOWP_EXP_BARREL_SHIFTER(+0, 790015084);
GEMMLOWP_EXP_BARREL_SHIFTER(+1, 290630308);
GEMMLOWP_EXP_BARREL_SHIFTER(+2, 39332535);
GEMMLOWP_EXP_BARREL_SHIFTER(+3, 720401);
GEMMLOWP_EXP_BARREL_SHIFTER(+4, 242);
```

举例GEMMLOWP_EXP_BARREL_SHIFTER(-2, 1672461947); 具体计算：

$$e^{-2^{-2}} = e^{-0.25} = 0.7788007830714049 \text{ 定点为: } e^{-0.25} \cdot 2^{31} \approx 1672461947$$

一、 yolov5 tflite

（二）、tflite量化原理-理论细节

那么定点查表 保存的常量可以表示为：

指数	浮点值	定点值
$e^{-0.25}$	0.7788007830714049	1672461947
$e^{-0.5}$	0.6065306597126334	1302514674
e^{-1}	0.36787944117144233	790015084
e^{-2}	0.1353352832366127	290630308
e^{-4}	0.018315638888734186	39332535
e^{-8}	0.00033546262790251196	720401
e^{-16}	1.1253517471925921e-07	242

如果目标是计算$e^{-5.5}$，可以换算成查表$e^{-4} \cdot e^{-1} \cdot e^{-0.5}$
那么如果是计算$e^{-5.1}$，可以换算成查表$e^{-4} \cdot e^{-1} \cdot e^{-0.1}$
但是上面表里没有$e^{-0.1}$,那么实际上在gemmlowp里面，如果x>-0.25，也就是|x|<0.25，则用4阶泰勒展开估计：
具体实现代码为：https://github.com/google/gemmlowp/blob/fcf32e7a0a4d2af46e63eccf0c8fa4d83d0311c5/fixedpoint/fixedpoint.h#L603-L625
(详见下一页PPT讲解)

同时地，在gemmlowp里面，如果指数位数>5，也就是指数x<-32，则直接用1表示e^{-x}，具体是现代码为：
https://github.com/google/gemmlowp/blob/master/fixedpoint/fixedpoint.h#L792-L800

一、 yolov5 tflite

（二）、tflite量化原理-理论细节

这里详细说下，如果x>-0.25，也就是|x|<0.25，例如$e^{-0.1}$，如何用4阶泰勒展开估计：

具体实现代码为：https://github.com/google/gemmlowp/blob/fcf32e7a0a4d2af46e63eccf0c8fa4d83d0311c5/fixedpoint/fixedpoint.h#L603-L625

代码片段为：

```
F x = a + F::template ConstantPOT<-3>();
F x2 = x * x;
F x3 = x2 * x;
F x4 = x2 * x2;
F x4_over_4 = SaturatingRoundingMultiplyByPOT<-2>(x4);
F x4_over_24_plus_x3_over_6_plus_x2_over_2 = SaturatingRoundingMultiplyByPOT<-1>(((x4_over_4 + x3) * constant_1_over_3) + x2);
return constant_term + constant_term * (x + x4_over_24_plus_x3_over_6_plus_x2_over_2);
```

$$
\begin{aligned}
x &= a + 0.125 \\
x_2 &= (a + 0.125)^2 \\
x_3 &= (a + 0.125)^3 \\
x_4 &= (a + 0.125)^4 \\
x_4\_over\_4 &= \frac{(a + 0.125)^4}{4} \\
x_4\_over\_24\_plus\_x_3\_over\_6\_plus\_x_2\_over\_2 &= \frac{(((x_4\_over\_4 + x_3) \cdot \frac{1}{3}) + x2)}{2} \\
&= \frac{(a + 0.125)^4}{4 \cdot 3 \cdot 2} + \frac{(a + 0.125)^3}{3 \cdot 2} + \frac{(a + 0.125)^2}{2} \\
\text{最终返回 } e^a &= e^{-0.125} + e^{-0.125}(a + 0.125 + \frac{(a + 0.125)^2}{2} + \frac{(a + 0.125)^3}{3 \cdot 2} + \frac{(a + 0.125)^4}{4 \cdot 3 \cdot 2})
\end{aligned}
$$

上面公式中a<0，在gemmlowp会预先存储$e^{-0.125}$和$\frac{1}{3}$供后续计算查表用：

指数	浮点值	定点值
$e^{-0.125}$	0.8824969025845955	1895147668
$\frac{1}{3}$	0.3333333333333333	715827883

一、 yolov5 tflite

（二）、tflite量化原理-理论细节

所以现在我们就能通过定点+查表计算0<|x|<32条件下的任何指数e^{-x}，比如$e^{-5.1}$：

$$e^{-5.1}=e^{-4}\cdot e^{-1}\cdot e^{-0.1}$$

现在$e^{-4}\cdot e^{-1}$可以通过查表法得到，$e^{-0.1}$则可以通过上面的4阶泰勒展开计算近似得到：

$$e^{-0.1}=e^{-0.125}+e^{-0.125}(-0.1+0.125+\frac{(-0.1+0.125)^2}{2}+\frac{(-0.1+0.125)^3}{3\cdot 2}+\frac{(-0.1+0.125)^4}{4\cdot 3\cdot 2})$$

接下来就是sigmoid计算的第二步，得到D，代入计算sigmoid(x)：

$$D=e^{-x}$$

$$sigmoid(x)=\frac{1}{1+D}$$

理论细节参考：https://en.wikipedia.org/wiki/Division_algorithm#Newton.E2.80.93Raphson_division
具体实现参考：https://github.com/google/gemmlowp/blob/master/fixedpoint/fixedpoint.h#L853-L874

在gemmlowp里面采用的Newton-Raphson迭代来近似估计sigmoid(x)，为求解$\frac{1}{1+D}$，即求解1+D的倒数，在Newton-Raphson方法中，通过$f(x)=\frac{1}{x}-(1+D)$，迭代得到x最终即为1+D的倒数。具体计算为：

$$x_{n+1}=x_n-\frac{f(x_n)}{f'(x_{n+1})}=x_n-\frac{\frac{1}{x_n}-(1+D)}{-\frac{1}{x_n^2}}$$

$$=x_n+x_n(1-(1+D)x_n)$$

我们通过初始化一个$x_0=\frac{48}{17}-\frac{32}{17}(1+D)$，经过三次迭代即可得到$x_2$，求解$\frac{x_2}{2}$就是最终sigmoid(x)的近似估计。其中为什么$x_0=\frac{48}{17}-\frac{32}{17}(1+D)$，这个在上面理论细节链接里，wiki文档指出该点是为了最小化近似值x的误差，同时取该点需要保证(1+D) 在区间[0.5, 1]，所以在gemmlowp中实际计算是采用公式：$sigmoid(x)=\frac{1}{0.5(1+D)}$，所以最后求得的$x_2$需要除以2，才是最终$sigmoid(x)=\frac{1}{(1+D)}$的近似结果。

一、 yolov5 tflite

（二）、tflite量化原理-理论细节

tflite训练后量化推理大概流程（int8为例）：

Q：上文关于推理部分的原理，都是建立在我们已知模型量化值的基础上，那么模型如何计算量化权重？

A：关于tflite在将浮点模型导出为int8模型时，权重值的计算，以及量化参数计算如下：

$$q = round(\frac{r}{S} + Z)$$

其中r和q和上文一样，表示real value和quantization value，S和Z就是量化参数，具体计算如下：

$$S = \frac{r_{max} - r_{min}}{2^n - 1}$$

$$Z = round(q - \frac{r}{S})$$

其中r_{max}和r_{min}表示real value的最大和最小值，n表示量化级别，例如int8量化时 n=2^8=256，Z的计算如果最终是均衡分布量化，那么可以等比取q和r的值进行计算，但是简单起见通常都是取q_{max}和r_{max}进行计算。
注意，S数据存储在具体实现里面是用浮点，而Z存储类型跟随q；

```
template<typename QType>
struct QuantizedBuffer {
  vector<QType> q;
  float S;
  QType Z;
};
```

举例例子：
当前r的取值范围为[-500,500]，量化为Int8，则：
S = (500 - (-500)) / (256 - 1) = 3.92156863
Z = round(255 - 500 / 3.92156863) = 128
当r=100时，q=round(100 / 3.92156863 + 128) = 153

一、 yolov5 tflite

（三）、tflite量化导出，模型文件细节

tflite官方一共提供了4种量化方法 https://www.tensorflow.org/lite/performance/model_optimization，分别如下：

Technique	Data requirements	Size reduction	Accuracy	Supported hardware
Post-training float16 quantization	No data	Up to 50%	Insignificant accuracy loss	CPU, GPU
Post-training dynamic range quantization	No data	Up to 75%	Smallest accuracy loss	CPU, GPU (Android)
Post-training integer quantization	Unlabelled representative sample	Up to 75%	Small accuracy loss	CPU, GPU (Android), EdgeTPU, Hexagon DSP
Quantization-aware training	Labelled training data	Up to 75%	Smallest accuracy loss	CPU, GPU (Android), EdgeTPU, Hexagon DSP

由上到下分别为：
1. float16量化，input/output都是float32，体积减小50%，这种能尽最大可能保留模型精度，同时又能减小模型体积。

2. 动态量化，input/output都是float32，模型参数为int8，过程输入输出都是float32，体积能减小75%.

3. 全整型量化，input/output，包括模型参数，过程输入输出都是int8，同样体积能减小75%，与方法2不同的是，全整型量化输入输出都是Int8，对于一些只能在整型上计算的板子，这是唯一的方法，同时这种方法需要提供小批量数据，用于标定input/output的量化参数scale/zero-point.

4. 量化感知训练，可以用于边量化模型边训练，提高量化后模型精度。

一、 yolov5 tflite

（三）、tflite量化导出，模型文件细节

1. float16量化：

yolov5导出：
python3 export.py --weights yolov5s.pt --include tflite --imgsz 320

```
# 实现细节
converter = tf.lite.TFLiteConverter.from_keras_model(keras_model)
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS]
converter.target_spec.supported_types = [tf.float16]
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_model = converter.convert()
open(f, "wb").write(tflite_model)
```

其中converter.target_spec.supported_ops指定的是使用哪些op作为量化时可以采用的，目前可以选择的有：
tf.lite.OpsSet.TFLITE_BUILTINS：只用tflite内置op，这是默认选择
tf.lite.OpsSet.SELECT_TF_OPS：采用tf本身的op，但是不是所有tf方法都支持，不建议用这种，除非是自己设计的比较复杂的结构
tf.lite.OpsSet.TFLITE_BUILTINS_INT8：只用tflite里面int8的op
tf.lite.OpsSet.EXPERIMENTAL_TFLITE_BUILTINS_ACTIVATIONS_INT16_WEIGHTS_INT8：实验接口，int8权重，int16激活值，int32bias，建议生产环境不用，这种设计可以在牺牲一定体积压缩下取得比单纯int8更高的精度

一、 yolov5 tflite

（三）、tflite量化导出，模型文件细节

2. 动态量化：

yolov5里面没有关于动态量化具体实现，可以将如下代码替换fp16量化代码，具体直接删除以下代码即可：
converter.target_spec.supported_types = [tf.float16]

yolov5导出，与fp16导出相同：
python3 export.py --weights yolov5s.pt --include tflite --imgsz 320

```
# 实现细节
converter = tf.lite.TFLiteConverter.from_keras_model(keras_model)
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS]
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_model = converter.convert()
open(f, "wb").write(tflite_model)
```

一、 yolov5 tflite

（三）、tflite量化导出，模型文件细节

3. 全整型量化：

yolov5导出：

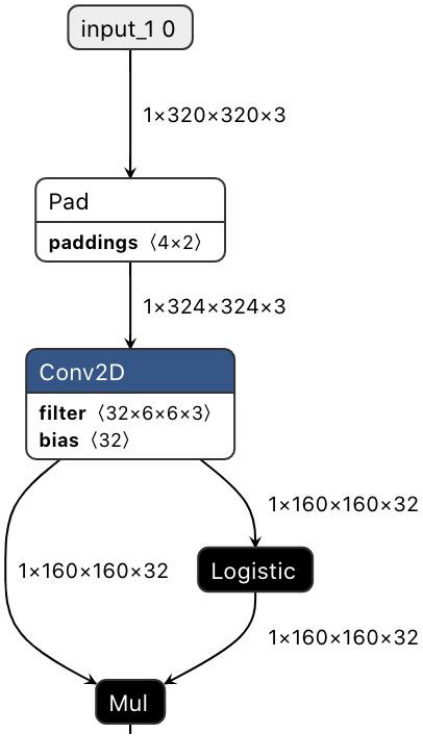
```
python3 export.py --weights yolov5s.pt --include tflite --imgsz 320 --int8
```

```
# 实现细节
dataset = LoadImages(check_dataset(data)['train'], img_size=imgsz, auto=False)
converter.representative_dataset = lambda: representative_dataset_gen(dataset, ncalib)
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.target_spec.supported_types = []
converter.inference_input_type = tf.uint8  # or tf.int8
converter.inference_output_type = tf.uint8  # or tf.int8
converter.experimental_new_quantizer = False
tflite_model = converter.convert()
open(f, "wb").write(tflite_model)
```

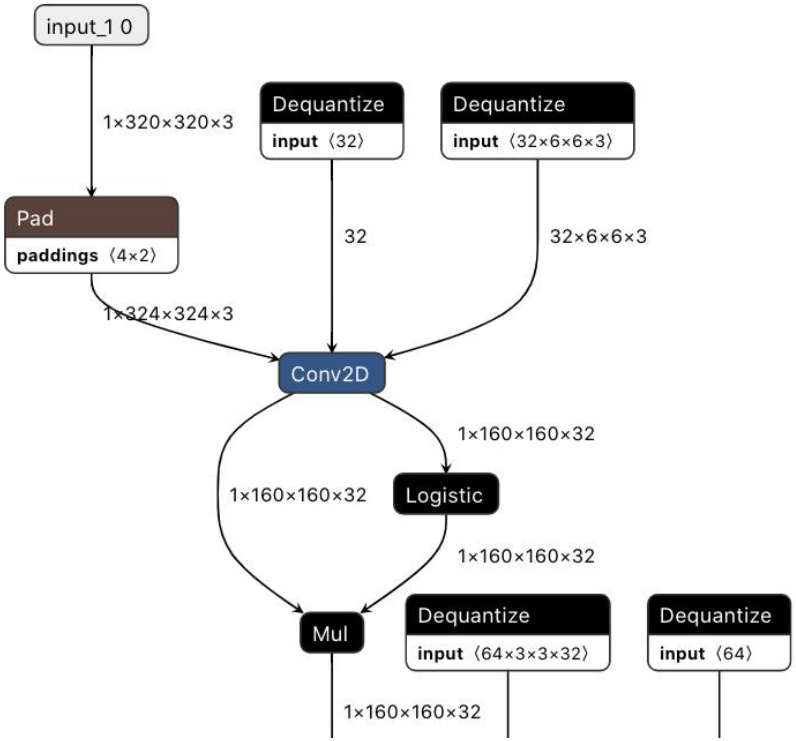
一、 yolov5 tflite

（三）、tflite量化导出，模型文件细节

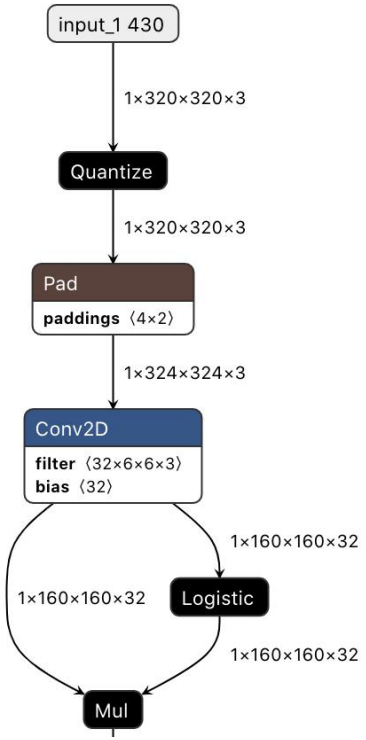
对比3种导出的结果，输入部分：



动态量化



fp16量化



int8量化

动态量化下，input/output都是float32，计算过程整型加速与浮点加速同时兼顾，模型参数为int8，过程输入输出都是float32

fp16量化下，input/output都是float32，同时当采用CPU计算时，模型权重w和bias会dequantize到float32，如果采用gpu计算，则不需要做此步dequantize，因为tflite的gpu代理支持fp16操作。

int8量化下，input/output都是int8，在quantize是量化方法，里面有关于input/output的量化参数scale/zero point，所有的计算如conv2D都遵循之前所讲的理论。

一、 yolov5 tflite

（三）、tflite量化导出，模型文件细节

```
# 加载模型
>>> yolov5s = "./yolov5s-int8.tflite"
>>> interpreter = tf.lite.Interpreter(model_path=yolov5s)
>>> interpreter.
interpreter.allocate_tensors(    interpreter.get_output_details(   interpreter.get_tensor_details(   interpreter.reset_all_variables(  interpreter.set_tensor(
interpreter.get_input_details(   interpreter.get_tensor(          interpreter.invoke(               interpreter.resize_tensor_input(  interpreter.tensor(
```

tflite里面Interpreter类提供了10个方法用于操作tflite模型文件，分别表示：

allocate_tensors()	加载模型权重到内存
get_output_details()	读取output数据细节，包含量化参数scale/zero-point，数据类型等
get_tensor_details()	读取所有数据细节，包含量化参数scale/zero-point，数据类型等
reset_all_variables()	重置所有变量
set_tensor()	set_tensor(input_index, input_data) 设置输入数据到对应Tensor
get_input_details()	读取input数据细节，包含量化参数scale/zero-point，数据类型等
get_tensor()	predictions = interpreter.get_tensor(output_index) 读取目标Tensor，在调用invoke()后使用为读取计算结果tensor
invoke()	执行计算流
resize_tensor_input()	interpreter.resize_tensor_input(input_index, [num_test_images, 224, 224, 3])
tensor()	返回给出当前张量缓冲区numpy视图的函数，无需调用allocate_tensors()即可拿到Tensor数据

一、 yolov5 tflite

（三）、tflite量化导出，模型文件细节

```
# 查看输入输出细节
>>> interpreter = tf.lite.Interpreter(model_path=yolov5s)
>>> interpreter.get_input_details()
[{'name': 'input_1', 'index': 430, 'shape': array([  1, 320, 320,   3], dtype=int32), 'shape_signature': array([  1,
320, 320,   3], dtype=int32), 'dtype': <class 'numpy.uint8'>, 'quantization': (0.003921568859368563, 0),
'quantization_parameters': {'scales': array([0.00392157], dtype=float32), 'zero_points': array([0],
dtype=int32), 'quantized_dimension': 0}, 'sparsity_parameters': {}}]

>>> interpreter.get_output_details()
[{'name': 'Identity', 'index': 431, 'shape': array([   1, 6300,   85], dtype=int32), 'shape_signature': array([   1,
6300,   85], dtype=int32), 'dtype': <class 'numpy.uint8'>, 'quantization': (0.006305381190031767, 5),
'quantization_parameters': {'scales': array([0.00630538], dtype=float32), 'zero_points': array([5],
dtype=int32), 'quantized_dimension': 0}, 'sparsity_parameters': {}}]
```

可以看到关于input/output数据的一些细节：
index：input tensor的索引，后续获取计算都是通过这个索引来
shape：input tensor的shape
dtype：当前数据类型
quantization：量化参数，第一个是scale，第二个是zero point，后面会把这个传到Java代码里，当调用Java api识别图片时，需要利用这个参数量化和反量化图片数据。

一、 yolov5 tflite

（三）、tflite量化导出, 模型文件细节

```
# 执行计算流
>>> interpreter.allocate_tensors()
>>> input_index = interpreter.get_input_details()[0]["index"]
>>> output_index = interpreter.get_output_details()[0]["index"]

>>> interpreter.set_tensor(input_index, test_image)
>>> interpreter.invoke()
>>> predictions = interpreter.get_tensor(output_index)
```

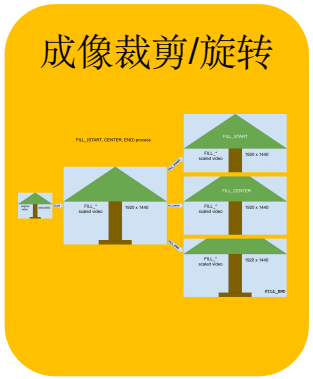
大概4个步骤:

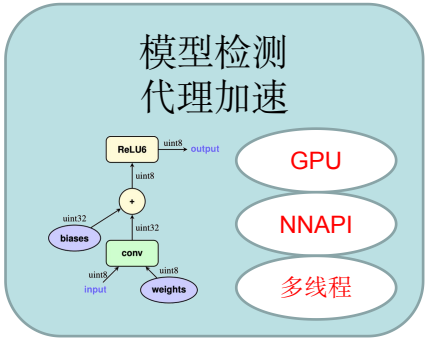
1. 在计算之前我们需要通过allocate_tensors()方法获取Tensor,
2. 通过set_tensor到input tensor
3. 通过invoke执行计算
4. 获取计算结果Tensor

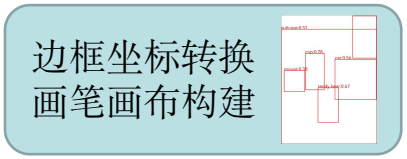
二、Android部署(以摄像头检测为例)

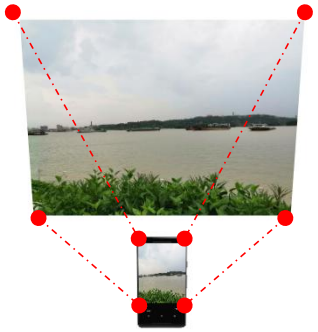
（零）、android部署整体流程

异步



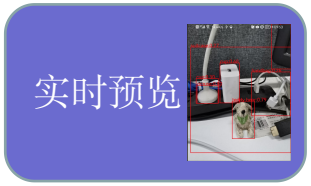






获取摄像头数据

主线程渲染



二、Android部署(以摄像头检测为例)

（一）、相关依赖

tf官网示例build.gradle，不建议用，已经很久没更新:

```
android {
    aaptOptions {
        noCompress "tflite"
    }
}
dependencies {
    // Import tflite dependencies
    implementation 'org.tensorflow:tensorflow-lite:0.0.0-nightly-SNAPSHOT'
    // The GPU delegate library is optional. Depend on it as needed.
    implementation 'org.tensorflow:tensorflow-lite-gpu:0.0.0-nightly-SNAPSHOT'
    implementation 'org.tensorflow:tensorflow-lite-support:0.0.0-nightly-SNAPSHOT'
}
```

推荐maven上查看最新稳定版本：https://mvnrepository.com/search?q=tensorflow-lite

```
android {
    aaptOptions {
        noCompress "tflite"
    }
}
dependencies {
    implementation 'org.tensorflow:tensorflow-lite:2.8.0'
    implementation 'org.tensorflow:tensorflow-lite-gpu:2.8.0'
    implementation 'org.tensorflow:tensorflow-lite-support:0.3.1'
    implementation 'org.tensorflow:tensorflow-lite-metadata:0.3.1'
}
```

其中tensorflow-lite为核心api库，管理模型加载和运行，tensorflow-lite-gpu为gpu代理库，如果当前gpu不支持则不必引入，tensorflow-lite-support为核心库api之外的支持库，包含一些常用的数据处理方法，tensorflow-lite-metadata为模型元数据管理相关api库。

二、Android部署(以摄像头检测为例)

(二)、模型加载，输入与输出定义

模型加载，单变量输入输出定义：https://www.tensorflow.org/lite/inference_with_metadata/lite_support

```
// 加载yolov5模型和标签文件
ByteBuffer tfliteModel = FileUtil.loadMappedFile(activity, MODEL_FILE);
tflite = new Interpreter(tfliteModel, options);
associatedAxisLabels = FileUtil.loadLabels(activity, LABEL_FILE);

// 定义输入，处理输入数据
TensorImage yolov5sTfliteInput;
ImageProcessor imageProcessor = new ImageProcessor.Builder()
                        .add(new ResizeOp(320, 320, ResizeOp.ResizeMethod.BILINEAR))
                        .add(new NormalizeOp(0, 255))
                        .build();
TensorImage yolov5sTfliteInput = new TensorImage(DataType.FLOAT32);
yolov5sTfliteInput.load(bitmap);
yolov5sTfliteInput = imageProcessor.process(yolov5sTfliteInput);

// 定义输出
TensorBuffer probabilityBuffer = TensorBuffer.createFixedSize(OUTPUT_SIZE, DataType.FLOAT32);

// 执行推断
tfLite.run(yolov5sTfliteInput.getBuffer(), probabilityBuffer.getBuffer());

// 输出数据被平铺了出来，从[1,6300,85]变成[1x6300x85]，这里再重新解析数据，具体看：
// https://github.com/yyccR/yolov5-tflite-android/blob/master/app/src/main/java/com/example/yolov5tfliteandroid/detector/Yolov5TFLiteDetector.java#L179-L216
float[] recognitionArray = probabilityBuffer.getFloatArray();
```

二、Android部署(以摄像头检测为例)

(二) 、模型加载，输入与输出定义

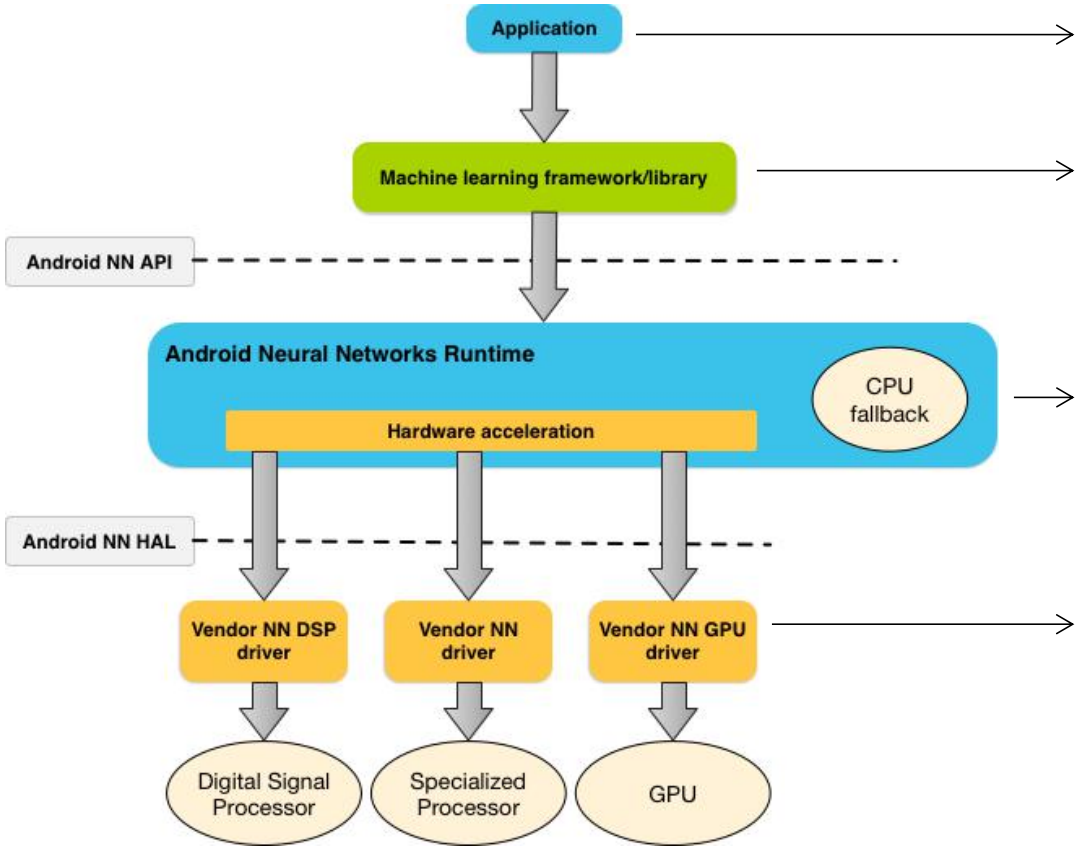
多变量输入输出定义：

```
// 加载模型和标签文件，与上文一样

// 假设输入图片数据，mask数据，数据已经处理到对应的input size，最后输入处理成object[]格式
TensorImage imageInput = new TensorImage(DataType.FLOAT32);
imageInput .load(bitmap);
TensorImage maskInput = new TensorImage(DataType.FLOAT32);
maskInput .load(bitmap2);
Object[] inputArray = {imageInput.getBuffer(), maskInput.getBuffer()};

// 假设输出类别标签数据，边框预测数据，mask分割数据，最后处理成Map<>格式
TensorBuffer classesOutput = TensorBuffer.createFixedSize(CLASS_SIZE, DataType.FLOAT32);
TensorBuffer locationsOutput = TensorBuffer.createFixedSize(LOCATIONS_SIZE, DataType.FLOAT32);
TensorBuffer maskOutput = TensorBuffer.createFixedSize(MASK_SIZE, DataType.FLOAT32);
Map<Integer, Object> outputMap = new HashMap<>();
outputMap.put(0, classesOutput.getBuffer() );
outputMap.put(1, locationsOutput.getBuffer() );
outputMap.put(2, maskOutput.getBuffer() );

// 执行推理，入口变了
tflite.runForMultipleInputsOutputs(inputArray, outputMap);

// 输出数据被平铺了出来，同样要一个一个解析
float[] classesOutputArray = classesOutput .getFloatArray();
float[] locationsOutputArray = locationsOutput .getFloatArray();
float[] maskOutputArray = maskOutput .getFloatArray();
```

二、Android部署(以摄像头检测为例)

（三）、NNAPI代理，gpu代理，多线程加速

NNAPI：Android Neural Networks API (NNAPI) 是一个 Android C API，专为在 Android 设备上运行密集型运算而设计的。NNAPI 旨在为更高层级的机器学习框架（如 TensorFlow Lite 和 Caffe2）提供一个基本功能层，用来建立和训练神经网络。



NNAPI架构从上到下大致分为四层：

1. 应用层：指具体APP层，这一层就是整个ppt要讲的。

2. 框架层：这一层是tflite/caffe2等具体框架，我们在这一层引入自己的tflite模型，通过框架api实现对nnapi的调用。

3. NNAPI逻辑层：这是原生C层，包含了原始的模型创建，编译，内存共享，线程计算等，通常我们不直接用这一层接口。

4. DSP/GPU/加速器等硬件层：
这一层是具体到各个处理单元机器计算，nnapi通过具体的驱动调用对应的硬件接口，此部分具体参考：
https://source.android.google.cn/devices/neural-networks?hl=zh-cn

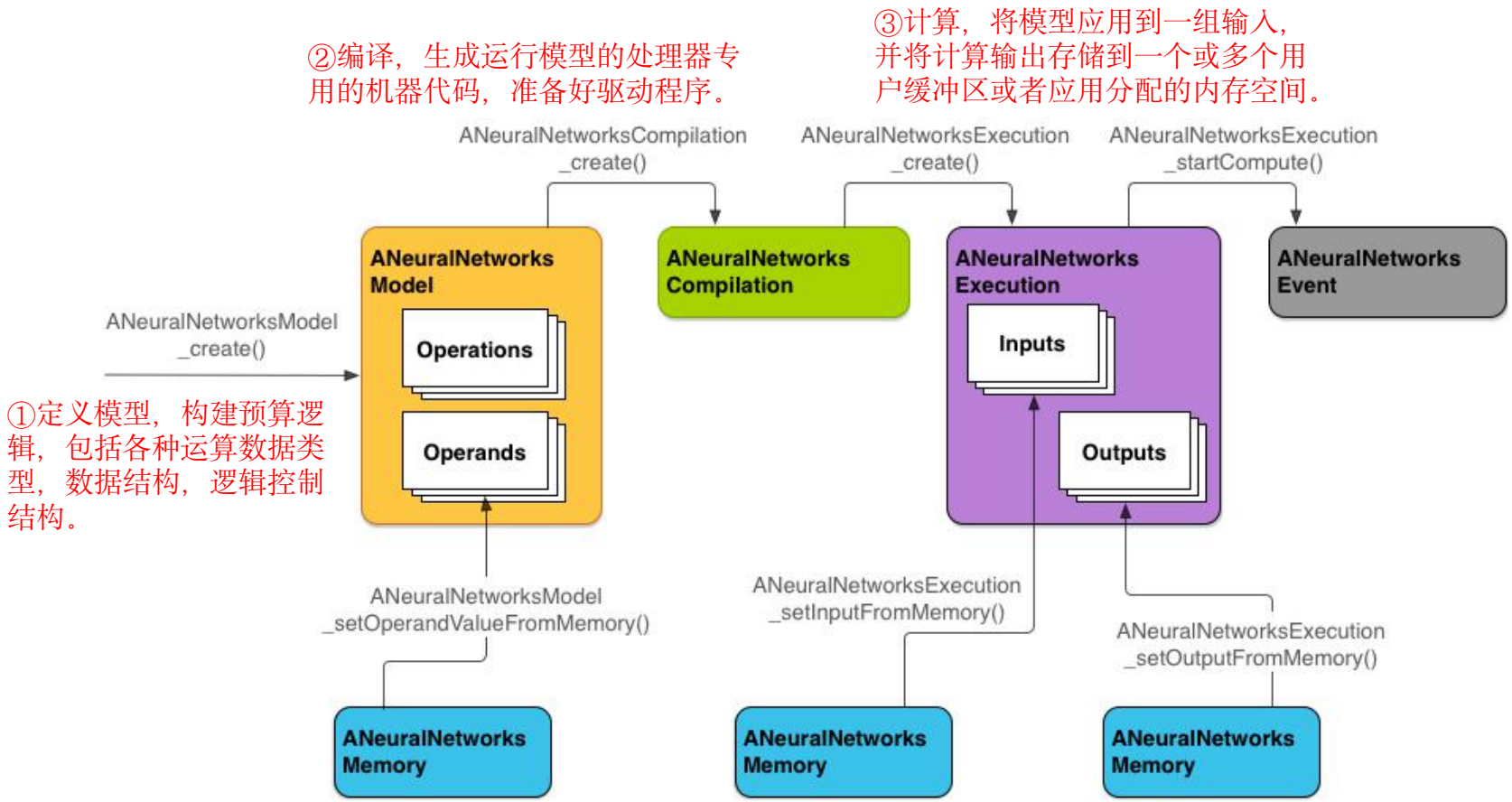
更多细节参考：

1. https://developer.android.com/ndk/guides/neuralnetworks#model
2. https://www.tensorflow.org/lite/android/delegates/nnapi

二、Android部署(以摄像头检测为例)

（三）、NNAPI代理，gpu代理，多线程加速

NNAPI逻辑层：原生C层，包含了原始的模型创建，编译，计算等。



ANeuralNetworksMemory为内存缓冲实例，可让 NNAPI 运行时更高效地将数据传输到驱动程序，存放定义模型所需的每个张量，还可以使用缓冲区来存储执行实例的输入和输出。

更多细节参考：

1. https://developer.android.com/ndk/guides/neuralnetworks#model
2. https://www.tensorflow.org/lite/android/delegates/nnapi

二、Android部署(以摄像头检测为例)

（三）、NNAPI代理，gpu代理，多线程加速

NNAPI代理在tflite中的具体使用：

```
import org.tensorflow.lite.Interpreter;
import org.tensorflow.lite.nnapi.NnApiDelegate;

Interpreter.Options options = (new Interpreter.Options());
NnApiDelegate nnApiDelegate = null;
// Initialize interpreter with NNAPI delegate for Android Pie or above
if(Build.VERSION.SDK_INT >= Build.VERSION_CODES.P) {
    nnApiDelegate = new NnApiDelegate();
    options.addDelegate(nnApiDelegate);
}
```

其中几个注意点：

1. sdk版本>=29，可以避免一些NNAPI输出与tflite输出不匹配问题，具体看:https://developer.android.com/ndk/guides/neuralnetworks#model
2. nnapi代理与gpu代理会有资源竞争问题，同时使用时并不会带来2倍的提升，建议只用一个

二、Android部署(以摄像头检测为例)

（三）、NNAPI代理，gpu代理，多线程加速

GPU代理细节：https://www.tensorflow.org/lite/performance/gpu

```
import org.tensorflow.lite.Interpreter;
import org.tensorflow.lite.gpu.CompatibilityList;
import org.tensorflow.lite.gpu.GpuDelegate;

// Initialize interpreter with GPU delegate
Interpreter.Options options = new Interpreter.Options();
CompatibilityList compatList = CompatibilityList();

if(compatList.isDelegateSupportedOnThisDevice()){
    // if the device has a supported GPU, add the GPU delegate
    GpuDelegate.Options delegateOptions = compatList.getBestOptionsForThisDevice();
    GpuDelegate gpuDelegate = new GpuDelegate(delegateOptions);
    options.addDelegate(gpuDelegate);
} else {
    // if the GPU is not supported, run on 4 threads
    options.setNumThreads(4);
}
```

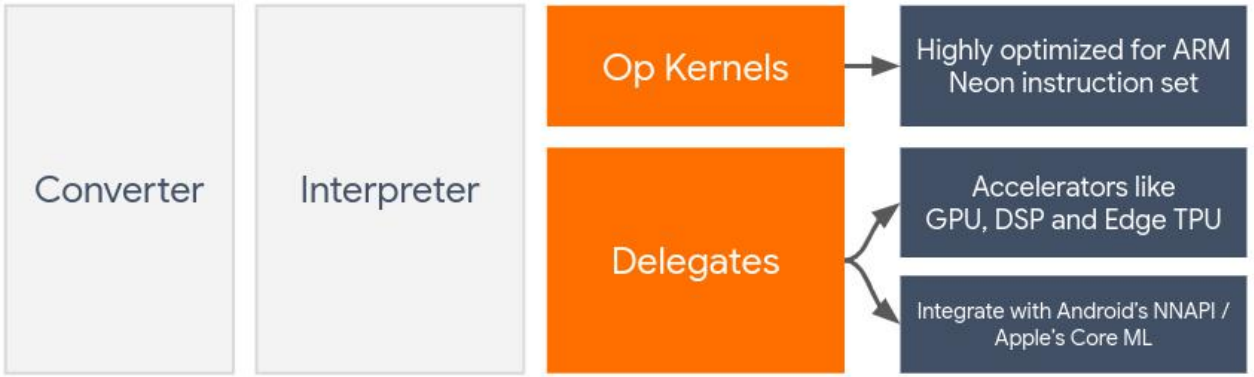
其中几个注意点：

1. 同样的，gpu代理与nnapi代理会有资源竞争问题，同时使用时并不会带来2倍的提升，建议只用一个
2. gpu代理与多线程设置一个即可
3. 目前tflite gpu代理支持的op还不是很多，具体可以参考：https://www.tensorflow.org/lite/performance/gpu_advanced
4. android12/sdk31如果遇到opencl无法使用问题，可以参考：https://github.com/tensorflow/tensorflow/issues/48001

二、Android部署(以摄像头检测为例)

（三）、NNAPI代理，gpu代理，多线程加速

GPU代理 与 NNAPI代理 异同点：



从代理角度看：

1. 相同之处在于，无论是gpu代理，nnapi代理，Hexagon代理，或者是coreML代理，都是tflite对一些模型计算方法的再封装，目的就是为了让模型或者模型中的部分节点能在GPU/TPU/DSP等加速器硬件上运行。
2. 不同之处在于，gpu代理，Hexagon代理属于tflite中对硬件驱动的封装，nnapi代理和coreML代理属于是在Android和ios系统上对自身库的封装，而自身库里已包含了对各种加速器硬件的支持。

模型类型	GPU	NNAPI	Hexagon	CoreML
浮点（32 位）	是	是	否	是
训练后 float16 量化	是	否	否	是
训练后动态范围量化	是	是	否	否
训练后整数量化	是	是	是	否
量化感知训练	是	是	是	否

从支持的量化类型看：

1. gpu代理支持所有量化类型，nnapi不支持半浮点(float16)量化类型。

二、Android部署(以摄像头检测为例)

（四）、tflite task和support的区别

tflite support提供了任意模型自定义输入输出，计算，数据处理方法。
tflite task是tflite封装了一些具体任务的库方法，需要提供满足输入输出要求的模型。

tflite里面大概提供了7种具体任务封装，分别如下：https://www.tensorflow.org/lite/inference_with_metadata/task_library/overview

1. 图像分类，目标检测，图像分割：
   implementation 'org.tensorflow:tensorflow-lite-task-vision:0.3.1'

2. 文本分类，基于bert的文本分类，基于bert的只能问答
   implementation 'org.tensorflow:tensorflow-lite-task-text:0.3.1'

3. 音频分类
   implementation 'org.tensorflow:tensorflow-lite-task-audio:0.3.1'

以上的tflite task库提供了对特定模型的封装，只需要提供对应模型输入输出既可以，举个例子，图像分类：

```
// 加载模型
ImageClassifierOptions options = ImageClassifierOptions.builder().setMaxResults(1).build();
ImageClassifier imageClassifier = ImageClassifier.createFromFileAndOptions(context, modelFile, options);
// 推理计算
List<Classifications> results = imageClassifier.classify(image);
```

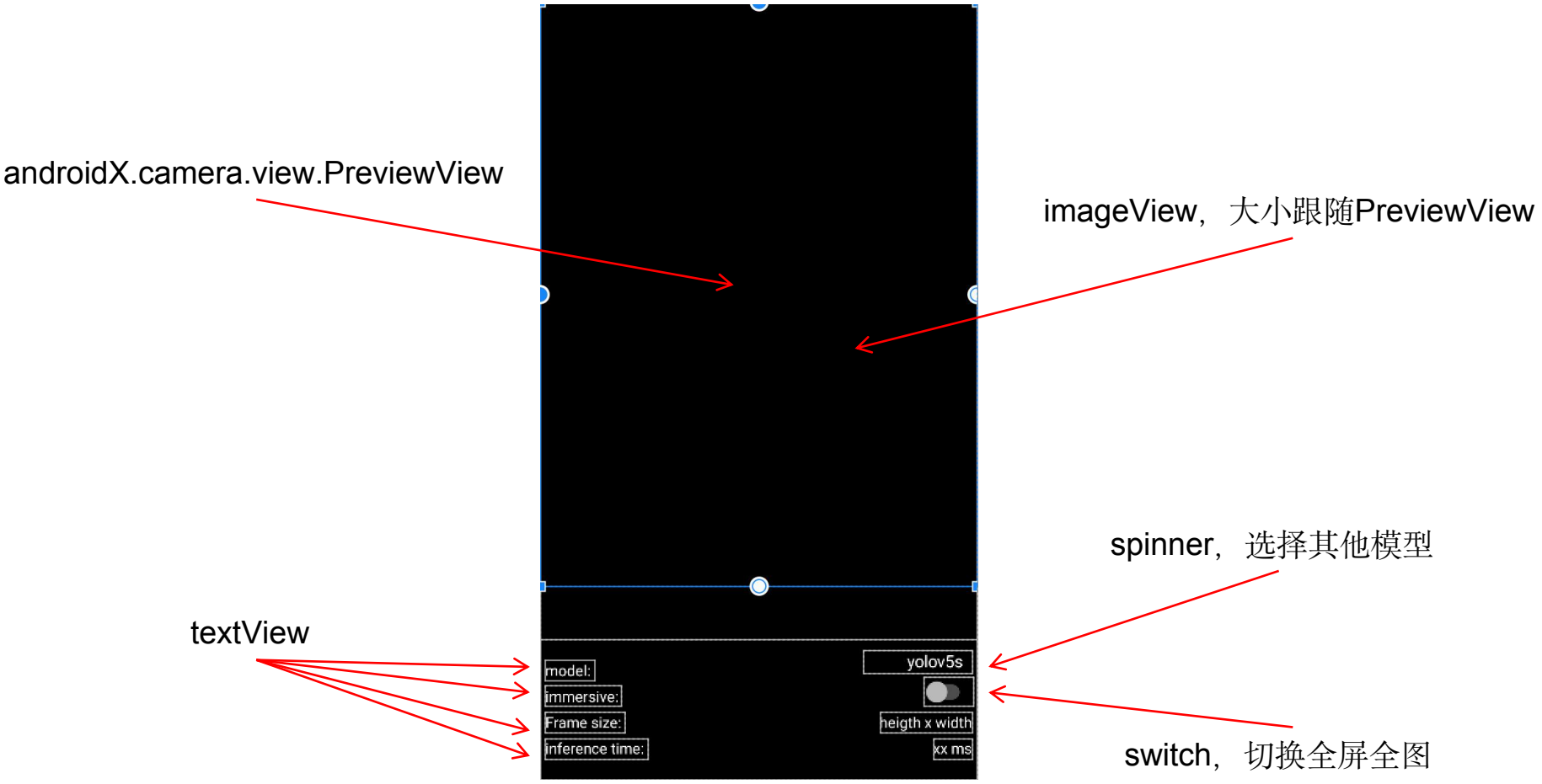
只需要提供模型满足如下即可：
输入：[batch=1,h,w,channel=3]，可为float32/uint8，float32格式需归一化，channel为rgb格式。
输出：[1,N]或者[1,1,1,N]，可为float32/uint8，N为类别数

二、Android部署(以摄像头检测为例)

（五）、布局文件示例

主要有以下几点：

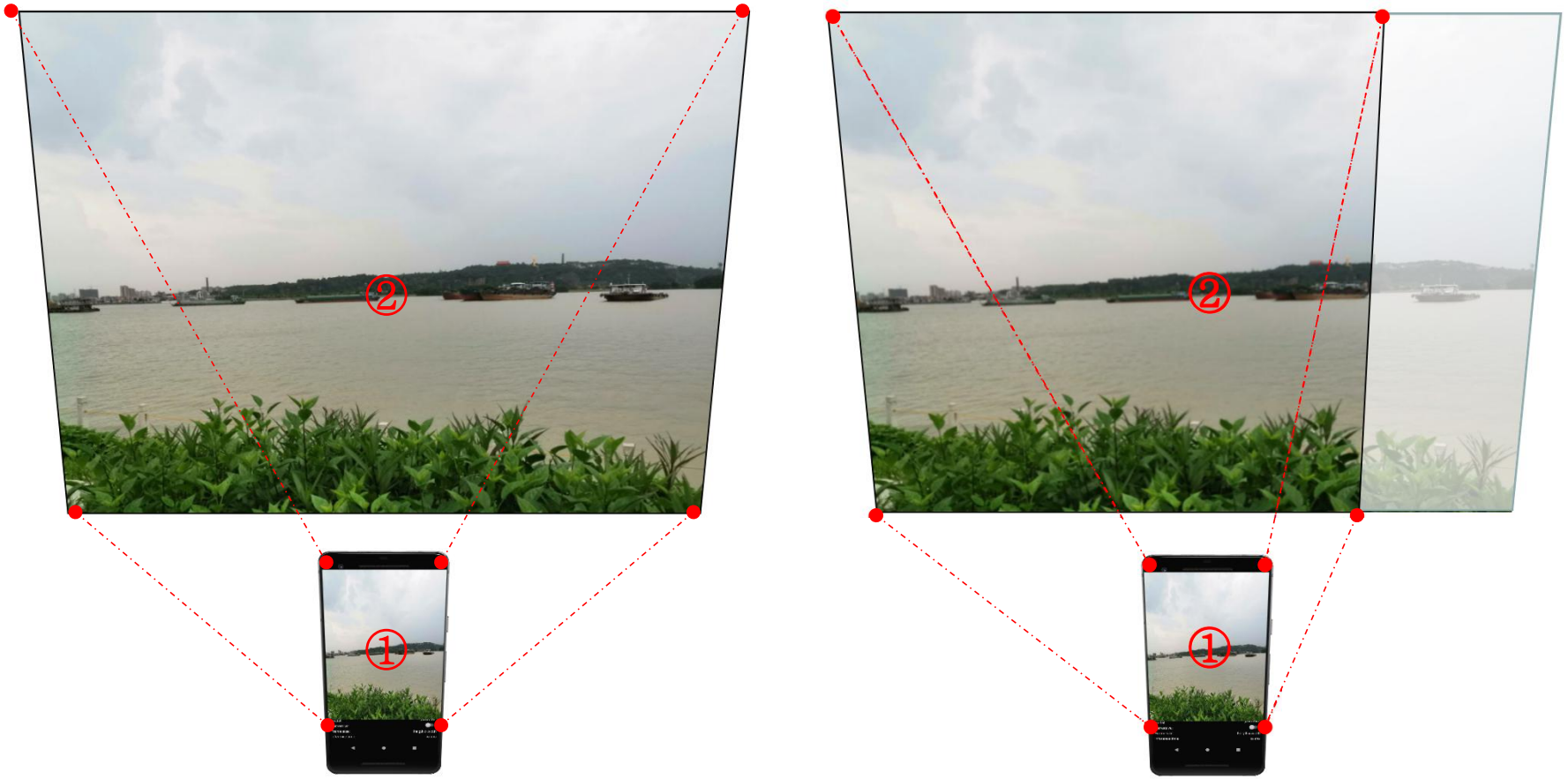
1. 采用constraint layout限制布局
2. 采用androidx.camera.view.PreviewView作为摄像头预览，包含全屏和全图两种
3. 采用imageView作为canvas画笔容器
4. 其他简单按钮，textView，spinner，switch。



二、Android部署(以摄像头检测为例)

（六）、cameraX 摄像头数据细节

cameraX里关于摄像头成像，有2种画面：①. 预览画面，②. 拍摄画面



当预览和拍摄的AspectRatio(宽高比)一致时，预览画面就与拍摄画面是等同的，如上左图，或者说通过缩放可以达到重合的效果，当画面比例设置不同时，就会出现裁剪或者黑边的情况，如上右图。

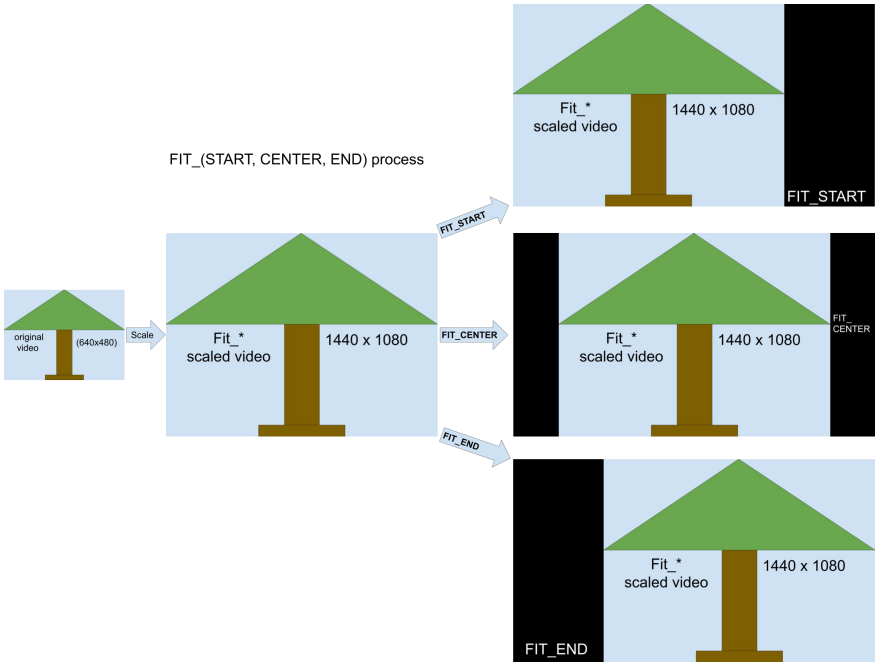
在设计app的时候，全图模式就是左边，会保证拍摄跟预览画面一致；
全屏模式就是右边，需要对拍摄画面进行裁剪，才能得到跟预览画面一致大小；

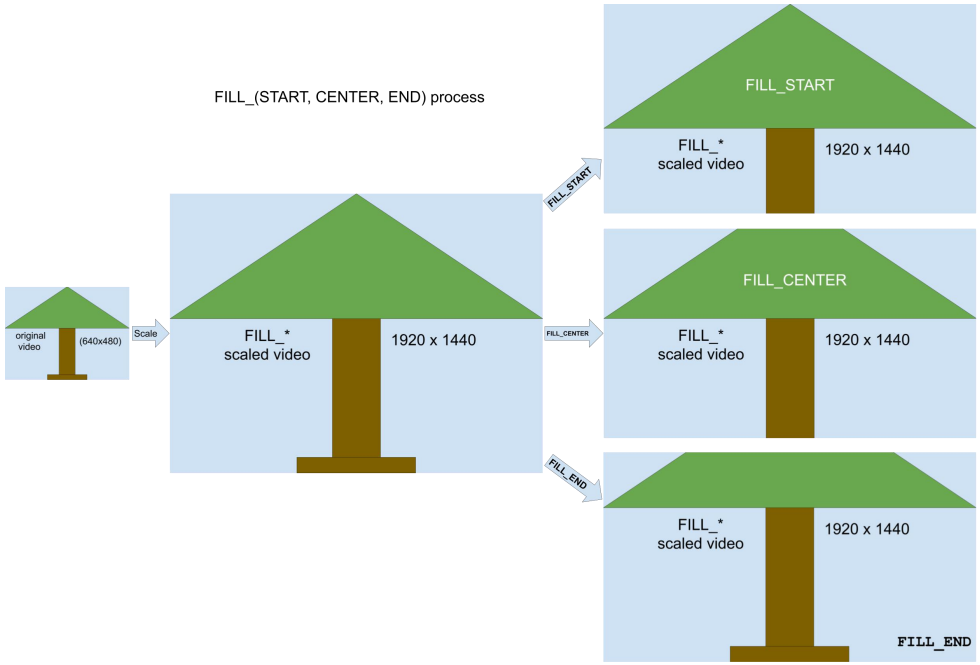
二、Android部署(以摄像头检测为例)

（六）、cameraX 摄像头数据细节

那么，拍摄画面如何裁剪出对应的预览画面？

首先需要了解android画面的缩放规则：https://developer.android.com/training/camerax/preview#scale-type





左图是Android摄像头画面缩放规则: FIT_*
这种保证屏幕能显示下整个图片，对于屏幕多出来图片的部分，用黑边填充。

右图是Android摄像头画面缩放规则: FILL_*
这种保证整个屏幕能被图片填充满，对于图片超过屏幕的部分，会被裁剪掉。

由于黑边对于体验效果比较差，在设计的采用右边缩放规则：FILL_*

```
cameraPreviewMatch = findViewById(R.id.camera_preview_match);
cameraPreviewMatch.setScaleType(PreviewView.ScaleType.FILL_START);
```

二、Android部署(以摄像头检测为例)

（七）、摄像头逐帧分析器

实现调用手机摄像头进行逐帧分析大概需要下面4步(详细看代码utils/CameraProcess.java)：

1. 引入cameraX相关库：

```
def camerax_version = "1.0.0-beta07"
implementation "androidx.camera:camera-camera2:$camerax_version"
implementation "androidx.camera:camera-lifecycle:$camerax_version"
implementation "androidx.camera:camera-view:1.0.0-alpha14"
```

2. 申请摄像头权限：

```
private int REQUEST_CODE_PERMISSIONS = 1001;
private final String[] REQUIRED_PERMISSIONS = new
String[]{"android.permission.CAMERA","android.permission.WRITE_EXTERNAL_STORAGE"};
for (String permission : REQUIRED_PERMISSIONS) {
    if (ContextCompat.checkSelfPermission(context, permission) != PackageManager.PERMISSION_GRANTED) {
        ActivityCompat.requestPermissions(activity, REQUIRED_PERMISSIONS, REQUEST_CODE_PERMISSIONS);
    }
};
```

3. 继承ImageAnalysis.Analyzer类， 重写public void analyze(@NonNull ImageProxy image)方法：

```
public class FullImageAnalyse implements ImageAnalysis.Analyzer {
    @Override
    public void analyze(@NonNull ImageProxy image) {
        // 这里面image就是每一帧拍摄画面，在这里实现自己的任何操作
        .....
    }
}
```

4. 绑定摄像头生命周期

```
ProcessCameraProvider cameraProvider = cameraProviderFuture.get();
cameraProvider.bindToLifecycle((LifecycleOwner) context, cameraSelector, imageAnalysis, previewBuilder);
```

二、Android部署(以摄像头检测为例)

（八）、异步计算，避免UI刷新卡顿

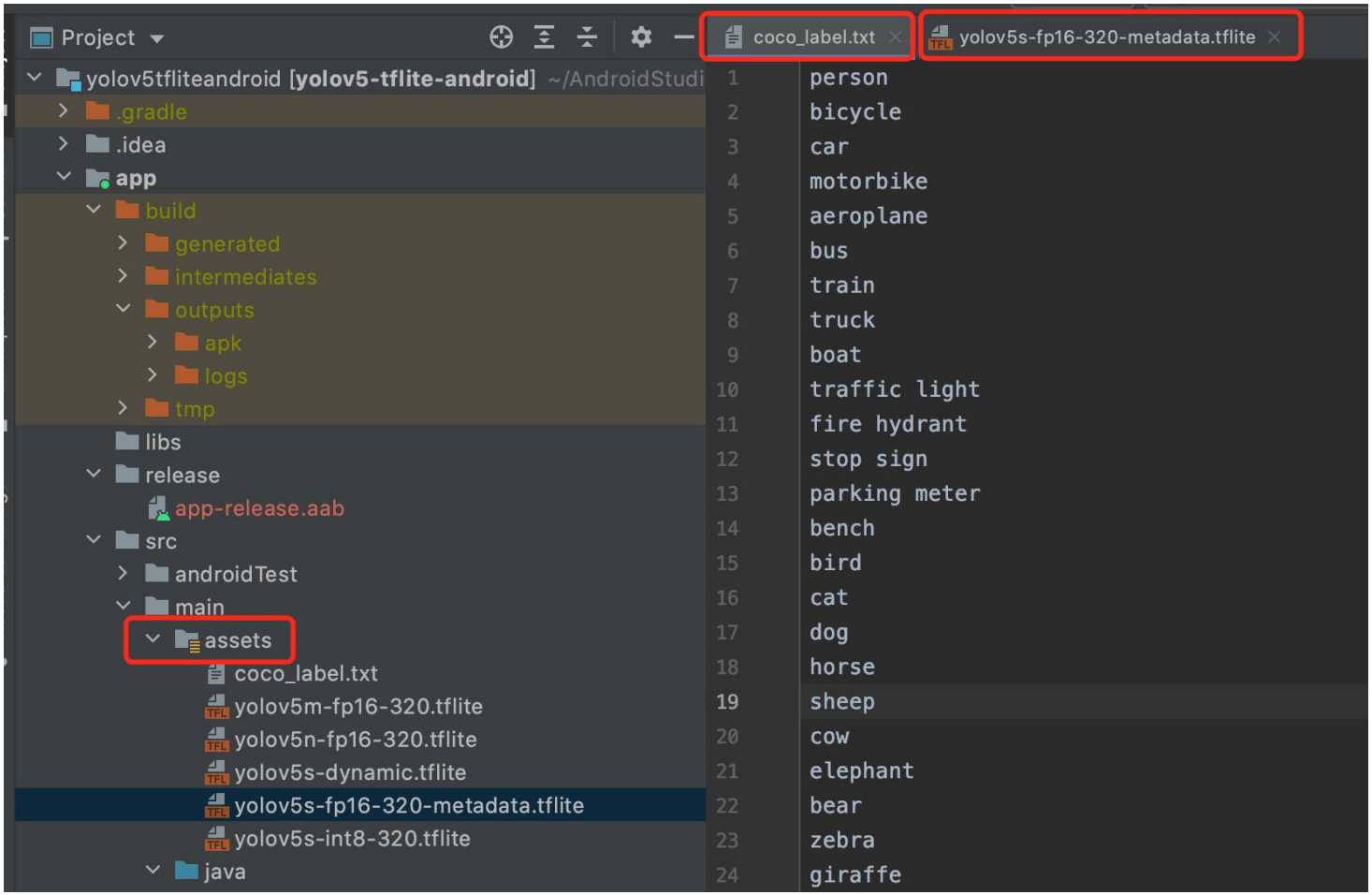
关于继承ImageAnalysis.Analyzer类，重写public void analyze(@NonNull ImageProxy image)方法，详细展开：

由于Android UI只有一个线程，把帧计算和UI渲染放到同个主线程会造成UI的卡顿，所以修改成如下(详细看代码analysis/FullImageAnalysis.java)：

```
// 引入rxjava3库
implementation 'io.reactivex.rxjava3:rxandroid:3.0.0'
implementation 'io.reactivex.rxjava3:rxjava:3.0.0'

public class FullImageAnalyse implements ImageAnalysis.Analyzer {
    @Override
    public void analyze(@NonNull ImageProxy image) {
        // 这里Observable将image analyse的逻辑放到子线程计算, 渲染UI的时候再拿回来对应的数据, 避免前端UI卡顿
        Observable.create( (ObservableEmitter<Result> emitter) -> {
            // 这里面image就是每一帧拍摄画面，在这里实现自己的任何操作
            .....
        }).subscribeOn(Schedulers.io()) // 这里定义被观察者,也就是上面代码的线程, 如果没定义就是主线程同步, 非异步
                // 这里就是回到主线程, 观察者接受到emitter发送的数据进行处理
                .observeOn(AndroidSchedulers.mainThread())
                // 这里就是回到主线程处理子线程的回调数据.
                .subscribe((Result result) -> {
                    // result  数据使我们自定义的类，可以在这里回到主线程处理计算结果，canvas作画等
                });
    }
}
```

三、替换自己的yolov5模型

（一）、assert文件替换



1. 导出自己的yolov5.tflite模型，建议用yolov5s，同时量化为fp16或者动态范围
2. 定义自己的label.txt文件，每行为一个类名
3. 将label.txt和yolov5.tflite放到assert目录下

三、替换自己的yolov5模型

（二）、输入与输出修改

修改以下这4个地方(detetor/Yolov5TFLiteDetector.java)：

1. 修改input size和output size:

```
private final Size INPNUT_SIZE = new Size(320, 320);
private final int[] OUTPUT_SIZE = new int[]{1, 6300, 85};
```

2. 修改模型名/标签文件名：

```
private final String MODEL_YOLOV5S = "yolov5s-fp16-320-metadata.tflite";
private final String LABEL_FILE = "coco_label.txt";
```

3. 如果你的模型是int8格式，则需要修改input/output的量化参数，如果不懂怎么拿到这几个参数，回去看上文<模型文件细节>章节：

```
 MetadataExtractor.QuantizationParams input5SINT8QuantParams = new
MetadataExtractor.QuantizationParams(0.003921568859368563f, 0);
    MetadataExtractor.QuantizationParams output5SINT8QuantParams = new
MetadataExtractor.QuantizationParams(0.0063053811900317 67f, 5);
```

4. 如果你的模型是int8格式，还需要修改setModelFile()方法里面：

```
case "yolov5s":
    IS_INT8 = true;
```

总结：

如果你的模型是fp16或者动态量化的，只需要执行1，2步骤

如果你的模型是int8的，需要执行1，2，3，4步骤

完结

欢迎关注b站：薛定谔的AI