目标检测的PTQ与QAT量化

1. 量化的基础知识

1.1 什么是模型量化?

模型指卷积神经网络，用于提取图像/视频视觉特征。

量化指将信号的连续取值近似为有限多个离散值的过程，可理解信息压缩的方法。INT8（8位的**定点整数**）。

1.2 什么是定点数?

数字包括整数，又包括小数，如果想在计算机中，即能表示整数，也能表示小数，关键就在于这个小数点如何表示?

约定计算机中小数点的位置，且这个位置固定不变，小数点前、后的数字，分别用二进制表示，然后用组合起来就可以把这个数字在计算机中存储起来，这就做定点表示法。用这种方法表示的数字叫做定点数

就理解为 【定】是指固定位置的意思，【点】为小数点，所以小数点位置固定的数为 【定点数】

定点数表示方式如下：

$$S_{n.m}$$

- S 有符号
- n 整数
- m 小数位数

考虑到二进制的补码形式表示负数，所以总的位数 n + m + 1, 比如 $S_{2.13}$比对应 n = 2, m = 13, 二进制的长度就为 n + m + 1 = 16。也就是说用的16位的定点化表示。

1.2.1 定点数转换

给定一个 Sn.m 格式的定点数二进制形式，那么他对应的数值为：

$$x = a \times 2^{-5} a = Binary \rightarrow Dec$$

示例1：2.71875

用 $S_{10.5}$ 说明

$$2.71875 = a \times 2^{-5}$$

$$a = Round(\frac{2.71875}{2^{-5}}) = Round(\frac{2.71875}{0.03125}) = 87$$

Round 向上取整

可以得到 a = 87 的二进制值为： 1010111

要补全 10 + 5 + 1 = 16 位定点表示，可以知道 2.71875 的 $S_{10.5}$ 结果为 0,0000000010,10111

示例2：-0.499878

这里用$S_{2.13}$ 定点化表示：

$$-0.499878 = a \times 2^{-13}$$

$$a = Round(\frac{-0.499878}{2^{-13}}) = Round(\frac{-0.499878}{0.0001220703125}) = -4095$$

可以得到 a = -4095, 4095 的二进制值为 0,00,0111111111111 ， 因为是有符号表示， “-” 号要对4095的二进制结果进行反码 +1 。 最终 -4095 的二进制为 1,11,1000000000001。

将 1,11,1000000000001 转换为浮点值：

由于第一位为1，因此判断是个负数，因此需要先对二进制 -1，然后取反码可以得到：

$$(1, 11, 1000000000001 - 1) = 0, 00, 0111111111111$$

二进制转换为十进制，为 4095， 同时再把 “-1”乘回来，便可以得到结果 -4095，然后再按照如下计算：

$$x = -4095 \times 2^{-13} = -0.4998779$$

可以看到定点化后得到的精度还是比较高的。

示例3： 2

这里用$S_{7.0}$定点化表达：

$$2 = a \times 2^0$$
$$a = Round(\frac{2}{2^{-0}}) = 2$$

这样就可以得到 a = 2, 2 的二进制值为： 0,0000010 。

这里也将 0,0000010 转换为浮点数值：

二进制转换十进制，为 2, 进行如下计算

$$x = 2 \times 2^{-0} = 2$$

可以看到定点化后整型的定点化是没有任何精度损失的。也是后面为什么要映射到 int8 整型进行计算的原因。

示例4: 2.71875

用$S_{7.0}$定点化表达

$$2.71875 = a \times 2^{-0}$$
$$a = \frac{2.71875}{2^{-0}} = Round(\frac{2.71857}{1}) = 2$$

这样就可以得到 a = 2, 2 的二进制值为： 0,0000010 。

这里也将 0,0000010 转换为浮点数值：

二进制转换十进制，为 2, 进行如下计算

$$x = 2 \times 2^{-0} = 2$$

可以看到定点化后随着定点位置越小精度就越大。这里因为$S_{7.0}$定点分辨率就是1，因此小数点后的结果就会被直接舍弃。后面的线性映射便可以解决这样的问题。

线性映射

首先直接把2.71875作为浮点的最大值，这里依旧采用$S_{7.0}$的定点计算方式（也就是int8的计算方式），其表示范围也就是[-128,127]之间，这里使用线性映射来进行映射

1. 计算线性映射的缩放值Scale

$$Scale = 2.71875/127 = 0.0214075$$

2. 根据映射关系计算整型结果：

$$Float = Scale \times Quant$$
$$Quant = Float / Scale$$
$$Quant = 2.71875 / 0.0214075 = 127$$

3. 将整型127进行 $S_{7.0}$ 定点化：

$$127 ==> S_{7.0} ==> 01111111 ==> Dec ==> 127$$

4. 整个流程（量化的原理示意）：

$$\text{量化：} 2.71875 \times 0.0214075 ==> 127 ==> S_{7.0} ==> 01111111$$
$$\text{反量化：} 01111111 ==> Dec ==> 127 / 0.0214075 ==> 2.71875$$

5.

1.3 量化问题及解决方法

量化截断带来的精度损失比较大?

进行一个数组的映射，这里数据为 [-0.52, 0.3, 1.7]

量化

$$Q = Round\left(\frac{R}{Scale}\right)$$
$$Scale = \frac{R_{\max} - R_{\min}}{Q_{\max} - Q_{\min}}$$

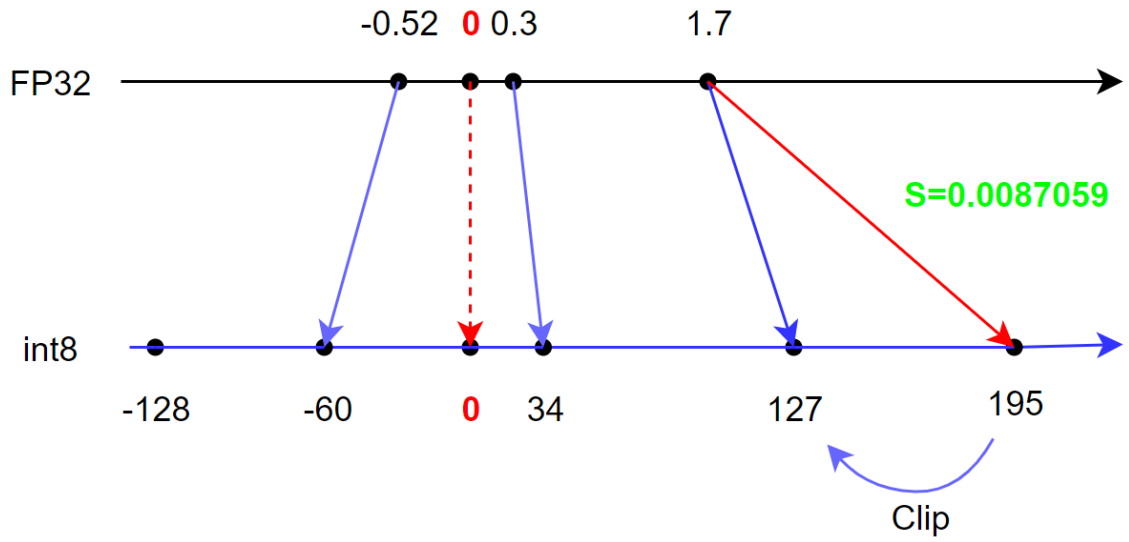
反量化

$$R = Q \times Scale$$
$$Scale = \frac{R_{\max} - R_{\min}}{Q_{\max} - Q_{\min}}$$

- 经过数组的最大最小值之差然后除以 以 `int8` 的表示范围可以得到

$$Q = Round(\frac{R}{Scale})$$
$$Scale = \frac{R_{max} - R_{min}}{Q_{max} - Q_{min}} = 0.0087059$$

- [-0.52, 0.3, 1.7] 根据 Scale 和上式的量化映射方法，可以得到映射后的 int8 数组为 [-60, 34, 195]
- 最后由于 int8 的定点范围为 [-128, 127], 因此对于前面得到的结果需要进行 Clip 截断操作，将 [60, 34, 195] 截断为 [-60, 34, 127]



前面得到映射后数组为： [-60, 34, 127] , 这里根据反量化表达式与Scale的值进行反量化计算可以得到，上述结果反量化后的结果： [-0.52236, 0.296, 1.1065], 二原始的数据为 [-0.52, 0.3, 1.7] 。两者一比较，由于截断带来的精度损失还是比较大的。

1.3.1 偏移法

量化

$$Q = Round\left(\frac{R}{Scale} + Z\right)$$

$$Scale = \frac{R_{max} - R_{min}}{Q_{max} - Q_{min}}$$

$$Z = Q_{max} - Round\left(\frac{R_{max}}{Scale}\right)$$

反量化

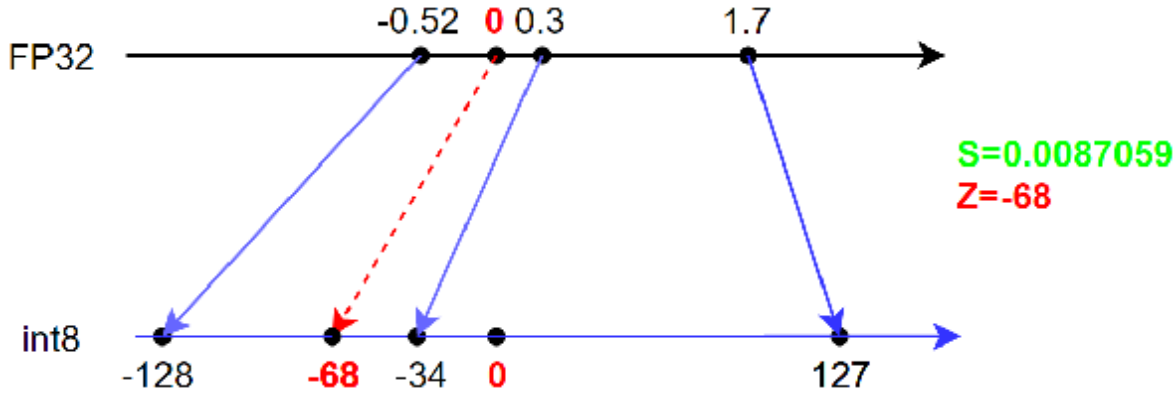
$$R = Q \times Scale + Z$$

$$Scale = \frac{R_{max} - R_{min}}{Q_{max} - Q_{min}}$$

$$Z = Q_{max} - Round\left(\frac{R_{max}}{Scale}\right)$$

Clip 截断的部分其实就是浮点最大值量化后经过 int8 最大表示范围的值，因此我们直接向左平移68，就可以得到无需截断的结果。

通过偏移68， 得到 int8 结果是： [-60, 34, 195] - 68 = [-128, -34, 127]



这个结果就是所谓的 Z-Point, 也就是浮点值的 0 点通过偏移在 int8 量化表示范围中的位置：

Z-Point 计算方式如下：

$$Z = Q_{max} - Round(\frac{R_{max}}{Scale})$$

1.3.2 最大绝对值对称法

量化

$$Q = Round\left(\frac{R}{Scale}\right)$$

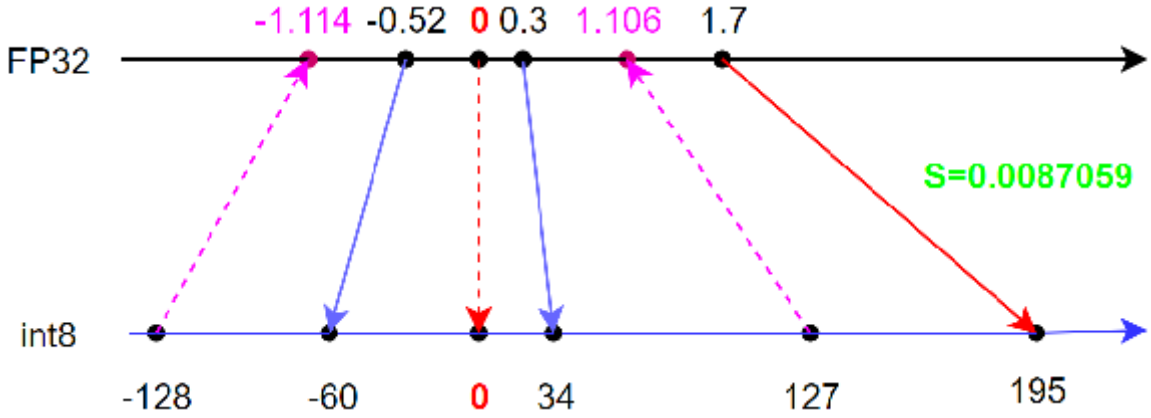
$$Scale = \frac{|R_{max}|}{|Q_{max}|}$$

反量化

$$R = Q \times Scale + Z$$

$$Scale = \frac{|R_{max}|}{|Q_{max}|}$$

为什么会被截断，通过将 127 和 -128与Scale 进行反量化得知，这保证正浮点数不大于 1.106， 负浮点数不小于-1.114。



而 1.7 则是大于这个边界。而上面偏移法就拉回 $Round((1.7-1.106)/S)=68$

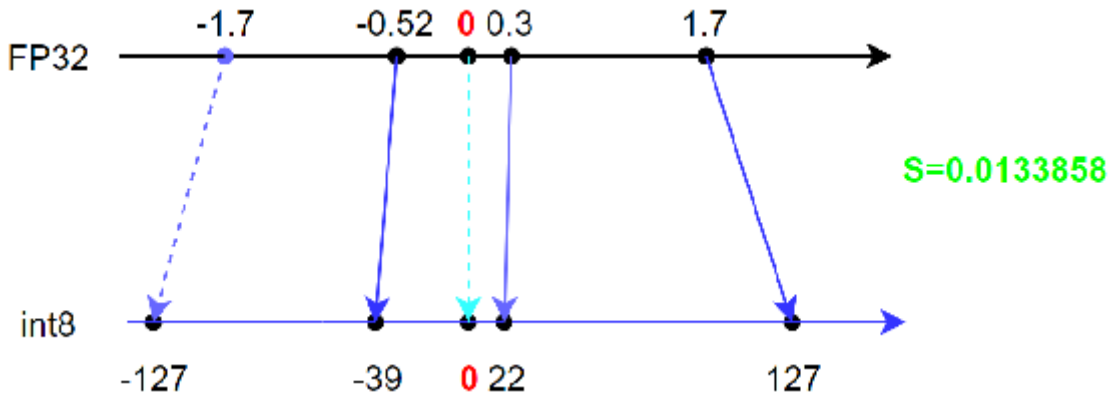
最大绝对值法：

如果将 -128 舍弃采用对称结构，就不会出现对不齐的问题。

直接去 1.7 为最大值，然后对称到负边界，而此时的 int8 负边界已经舍弃 -128， 因此

$$Scale = \frac{2 * 1.7}{254}$$

然后直接计算 int8 的结果为： [-39, 22, 127]



1.4 量化有什么优缺点?

作用： 将FP32的浮点计算转化为低bit位的计算，从而达到模型压缩和运算加速的目的。比如int8量化，就是让原来32bit存储的数字映射到 8 bit存储。 int8范围是[-128, 127] (实际工程量化用的时候，不会考虑-128)，uint8范围是[0, 255]

模型量化优点：

1. 加快推理速度，访问一次32位浮点型可以访问4次int8整型，整型运算比浮点型运算更快。
2. 减少存储空间，在边缘侧存储空间不足时更具有意义；
3. 减少设备功耗，内存耗用少了推理速度快了自然减少了设备功耗；
4. 易于在线升级，模型更小意味着更加容易传输；
5. 减少内存占用，更小的模型大小意味着不再需要更多的内存；

模型量化缺点：

1. 模型量化增加了操作复杂度，在量化时需要做一些特殊的处理，否则精度损失更严重；
2. 模型量化会损失一定的精度，虽然在微调后可以减少精度损失，但推理精度确实下降；

1.5 定点计算

- 对随机生成的浮点数据进行量化，转换为定点数据；
- 将原始输出，量化后的数据，反量化后的数据进行对比并计算误差；
- 对比量化前后的数据内存占用情况；
- 通过数据自身求和 n 次来测试运算速度

代码实现：

```
import sys
import time
import numpy as np

# 随机生成一些浮点数据（float32）
data_float32 = np.random.randn(10).astype('float32')


# 量化上下限
Qmin = 0
Qmax = 255

# 计算缩放因子（Scale）
S = (data_float32.max() - data_float32.min()) / (Qmax - Qmin)

# 计算零点（Zero Point）
Z = Qmax - data_float32.max() / S

# 将浮点数据（float32）量化为定点数据（UInt8）
data_uint8 = np.round(data_float32 / S + Z).astype('uint8')

# 将定点数据（UInt8）反量化为浮点数据（Float32）
data_float32_ = ((data_uint8 - Z) * S).astype('float32')


######## 误差计算
# 使用均方误差计算误差
```

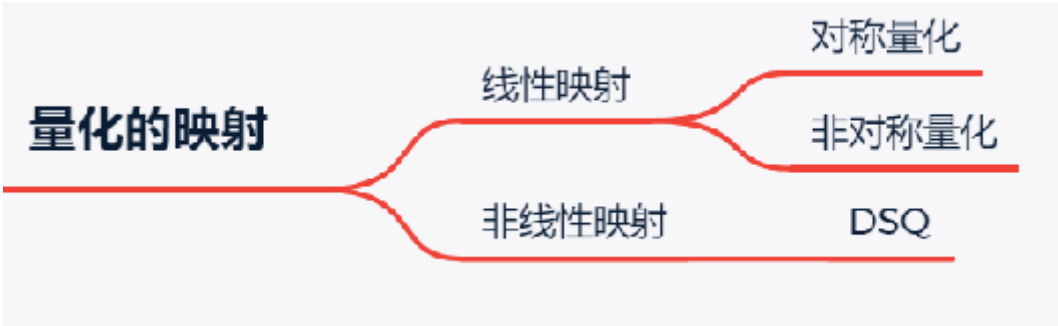
```
mse = ((data_float32 - data_float32_)**2).mean()

print("原始数据： \n", data_float32)
print("\n")
print("反量化后数据：\n", data_float32_)
print("\n")
print("量化后数据: \n", data_uint8)
print("\n")
print("原始数据和反量化数据的均方误差：", mse)
print("\n")

######## 内存对比
print("\n内存对比\n")
# 空数组的内存占用
empty_size = sys.getsizeof(np.array([]))

float32_size = (sys.getsizeof(data_float32) - empty_size)
uint8_size = (sys.getsizeof(data_uint8) - empty_size)

print("原始数据内存占用： %d Bytes " % float32_size)
print("量化后数据内存占用： %d Bytes " % uint8_size)
print("量化后数据与原始数据内存占用之比： ", uint8_size / float32_size)
print("\n")

######## 速度对比
print("\n速度对比\n")
# 重复次数
repeat = 10000

# 速度测试
start = time.time()
sum_float32 = 0
for i in range(repeat):
    sum_float32 += data_float32
float32_time = time.time() - start

start = time.time()
sum_uint8 = 0
for i in range(repeat):
    sum_uint8 += data_uint8
uint8_time = time.time() - start

print("原始数据求和 %d 次耗时： %f s" % (repeat, float32_time))
print("量化后数据求和 %d 次耗时： %f s" % (repeat, uint8_time))
print("量化后数据与原始数据计算耗时之比： ", uint8_time / float32_time)
```

随机对比结果如下

```
原始数据：
 [-0.48557758  0.18193954 -0.5339216  -1.4918683  -0.09712425 -0.08384836
 -0.9099724   1.0239322  -0.9450555  -1.0603378 ]


反量化后数据：
 [-0.48554808  0.18533206 -0.5348775  -1.4918683  -0.10077858 -0.08104681
 -0.9097811   1.0239322  -0.9492446  -1.0577693 ]


量化后数据：
 [102 170  97   0 141 143  59 255  55  44]


原始数据和反量化数据的均方误差：  5.7809093e-06



内存对比

原始数据内存占用：  40 Bytes
量化后数据内存占用：  10 Bytes
量化后数据与原始数据内存占用之比：   0.25



速度对比

原始数据求和 10000 次耗时：  0.007247 s
量化后数据求和 10000 次耗时：  0.005045 s
量化后数据与原始数据计算耗时之比：   0.6961442295038821
```

2. 模型量化方法

2.1 放射映射量化



基于模型的量化映射方式，如上图所示可以分为线性映射和非线性映射，对于线性映射又可以分为对称量化和非对称量化。

由于实际项目中基本不会使用非线性量化，本次课程不对非线性量化进行介绍。

2.2 线性映射

线性映射分为对称量化和非对称量化

2.2.1 对称量化

对称量化即使使用一个映射公式将输入数据映射到[-127,127]的范围内映射公式需要保证原始的输入数据中的零点通过映射公式后仍然对应 [-127, 127]区间的零点。

量化

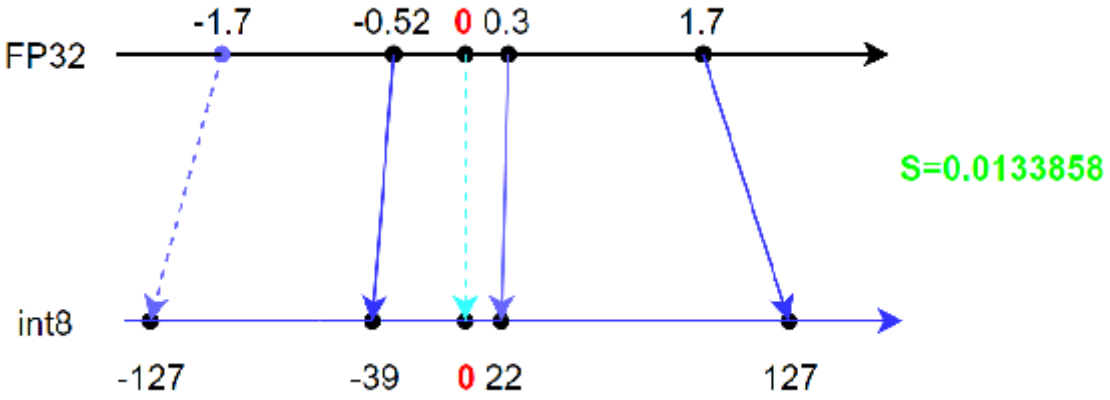
$$Q = Round\left(\frac{R}{Scale}\right)$$

$$Scale = \frac{|R_{max}|}{|Q_{max}|}$$

反量化

$$R = Q \times Scale + Z$$

$$Scale = \frac{|R_{max}|}{|Q_{max}|}$$



2.2.2 非对称量化

即使用一个映射公式将输入数据映射到[-128, 127] 的范围内，但是原始的输入数据中的零点通过映射公式后对应的位置并不是原点。

量化

$$Q = Round\left(\frac{R}{Scale} + Z\right)$$

$$Scale = \frac{R_{max} - R_{min}}{Q_{max} - Q_{min}}$$

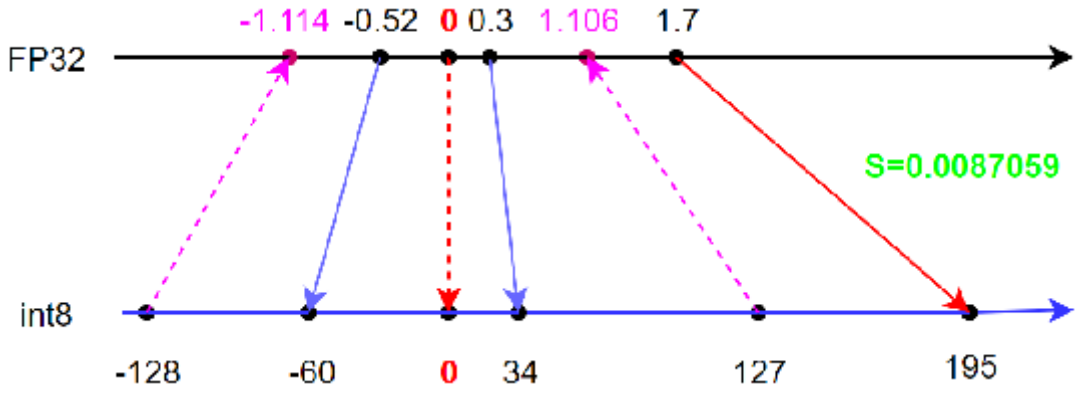
$$Z = Q_{max} - Round\left(\frac{R_{max}}{Scale}\right)$$

反量化

$$R = Q \times Scale + Z$$

$$Scale = \frac{R_{max} - R_{min}}{Q_{max} - Q_{min}}$$

$$Z = Q_{max} - Round\left(\frac{R_{max}}{Scale}\right)$$



2.3 逐层量化、逐组量化和逐通道量化

根据量化的粒度可以分为 逐层量化（per-Tensor）、逐组量化（per-Group）和逐通道量化（per-Channel）。

- 逐层量化：以一个层为单位，整个layer 的权重共用一组缩放因子 S 和偏移量 Z
- 逐组量化：以组为单位，每个 Group 使用一组 S 和 Z。
- 逐通道量化：则以通道为单位，每个channels单独使用一组 S 和 Z。

当 Group = 1 时，逐组量化与逐层量化等价；当Group = num_filters (即深度可分离卷积) 时，逐组量化逐通道量化等价。

2.4 在线量化和离线量化

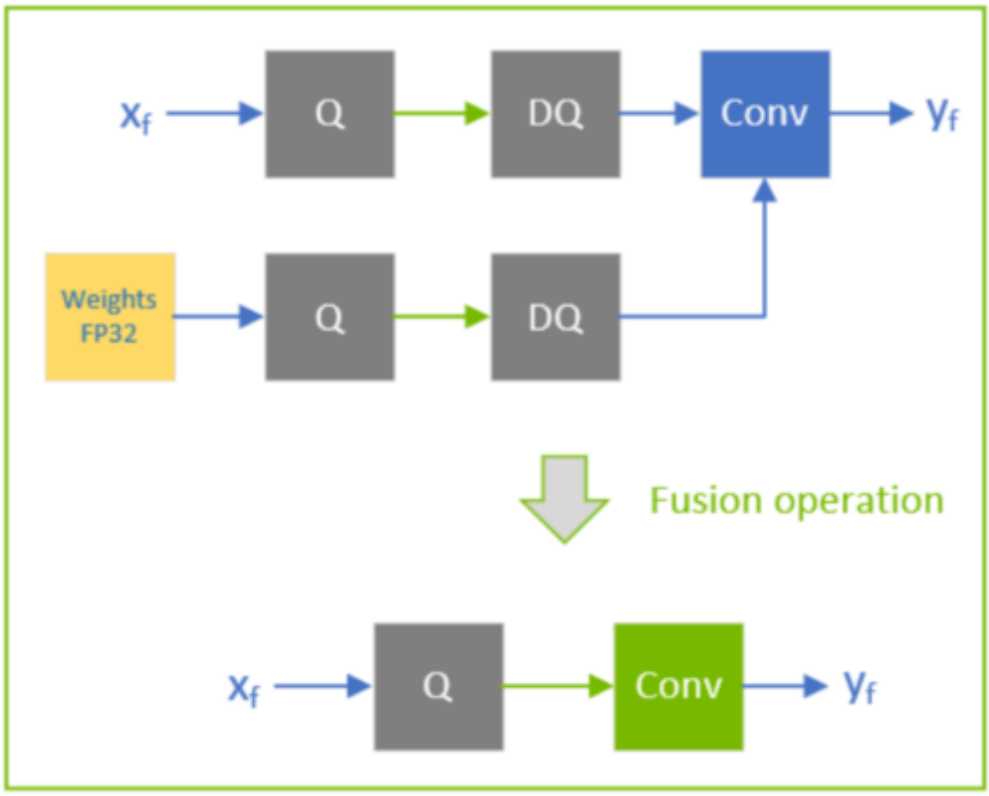
根据激活值的量化方式，可以分为在线（online) 量化和 离线（offline) 量化。

- 在线量化： 指激活值的Scale 和 Z-point 在实际推断过程中根据实际的激活值动态计算；
- 离线量化：即指提前确定好激活值的Scale 和 Z-point.

由于不需要动态计算量化参数，通常离线量化的推断速度更快些。

2.5 权重量化和权重激活量化

- 权重量化：只将权重的精度从浮点型减低为 8 bit 整型。由于只有权重进行量化，所以无需验证数据集就可以实现。如果只是想为了方便传输和存储而减小模型大小，而不考虑在预测时浮点型计算的性能开销的话，这种量化方法是很有用的。
- 权重激活量化: 可以通过计算所有将要被量化的数据的量化参数，来将一个浮点型模型量化为一个8bit精度的整型模型。由于激活输出需要量化，这时我们就得需要标定数据了，并且需要计算激活输出的动态范围，一般使用100个小批量数据就足够估算出激活输出的动态范围了



Q 指的是 quantize layer(量化层），DQ 指的是 dequantize layer（反量化层）

2.6 量化一般步骤

1. 在输入数据（通常是权重或者激活值）中计算出相应的 min_value 和 max_value;
2. 选择合适的量化类型，对称量化（int8) 还是非对称量化 (uint8)
3. 根据量化类型，min_value 和 max_value 来计算 Z (Zero point) 和 S（Scale);
4. 根据标定数据对模型执行量化校准操作，即将预训练模型由FP32量化为INT8模型；
5. 验证量化后的模型性能，如果效果不好，尝试使用不同的方式计算S 和 Z, 重新执行上面的操作。
6. 如果步骤 5 未能满足性能要求，则尝试使用敏感层分析和量化感知训练。

3. 模型校准

3.1 什么是校准?

校准是为模型权重和激活选择边界的过程。为了简单起见，这里只描述了对称范围的校准（工业界一般对称量化即可满足需求），如对称量化所需。这里考虑三种校准方法：

- Max： 使用校准期间的最大绝对值；
- Histogram: 将范围设置为校准期间看到的绝对值分布的百分位。
- Entropy: 使用 **KL散度**来最小化原始浮点值和量化比特表示的值之间的信息损失。

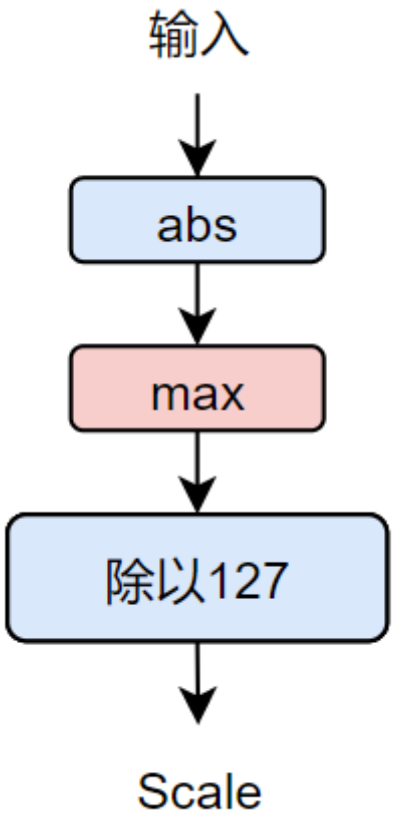
3.2 Max校准

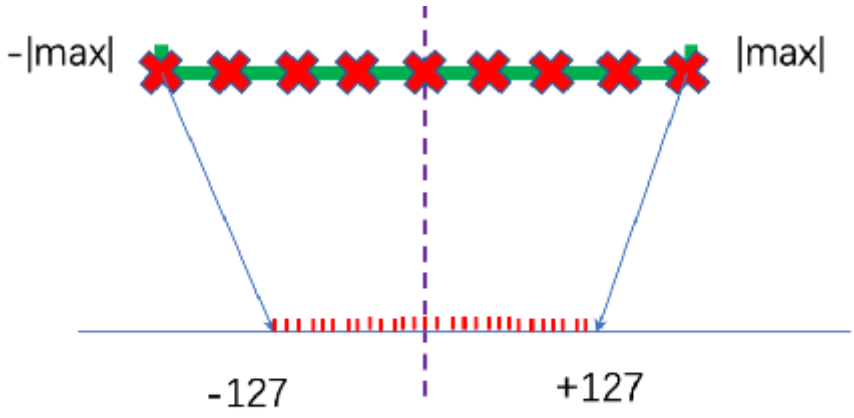
Max 校准主要使用校准期间的最大绝对值

校准过程如下所示：

1. 对输入求取绝对值；
2. 算出绝对值的最大值；
3. 以TensorRT 的int8 对称量化为例，这里如果想得到Scale 就要除以 127.

```
def maxq(value):
    dynamic_range = np.abs(value).max()
    scale = dynamic_range / 127.0
    return scale
```



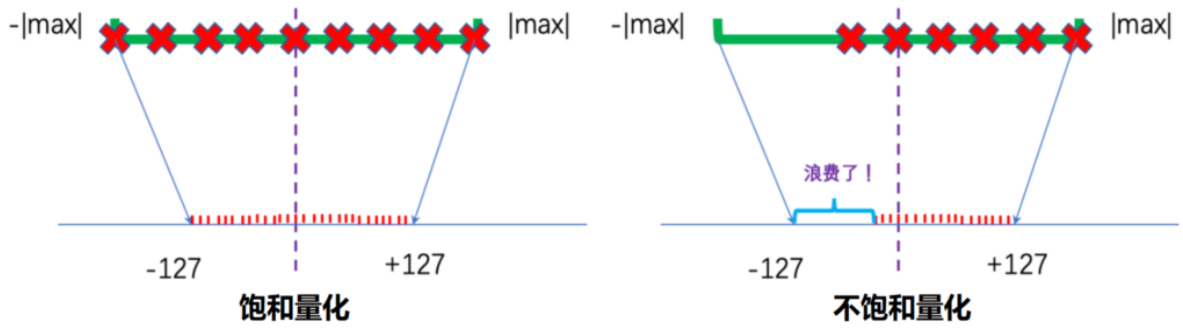


数据分布均匀

3.3 Histogram (直方图) 校准

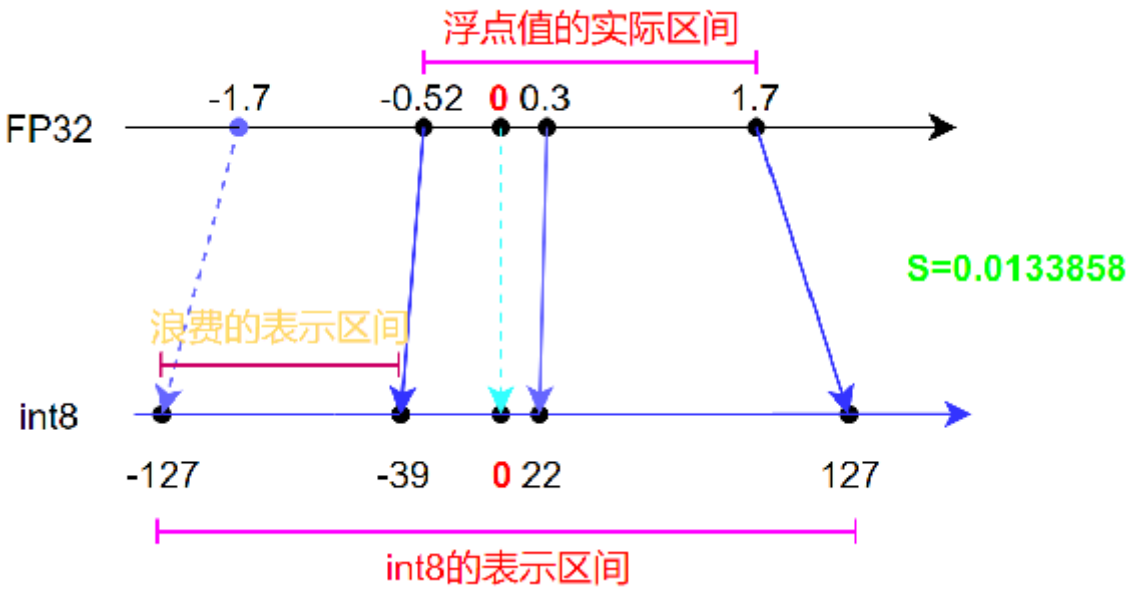
我们都知道量化的过程与数据的分布有关。如下图所示：当数据的直方图分布比较均匀时，高精度向低精度进行映射就会将表示空间利用比较充分；如果分布不均匀，就会浪

费很大的表示空间。



关于上面这种直接将量化阈值设置为 |max| 的方法，它的显著特点是int8的表示空间没有充分利用，因此称为不饱和量化（no saturation quantization)。

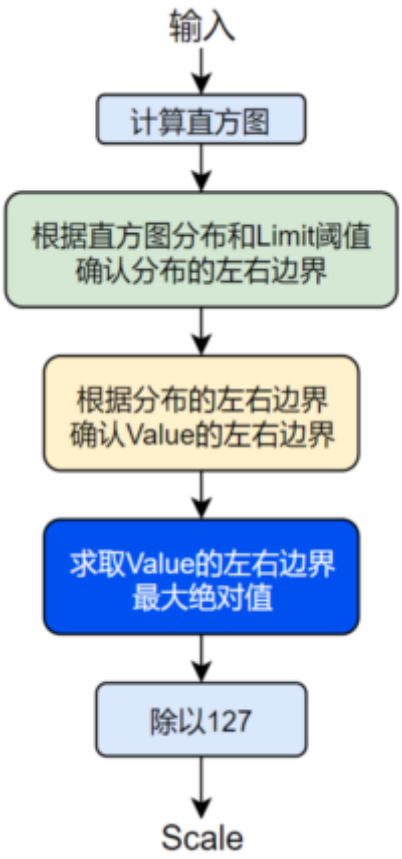
针对这种情况，可以选择一个合适的量化阈值（threshold), 舍弃那些超出范围的数进行量化，这种量化方式充分利用了int8 的表示空间，因此称为饱和量化（saturation quantization)。



对比两种量化方式可以发现，它们各有优缺点：不饱和量化方式的量化范围大，但是可能会浪费一些低比特的表示空间从而导致量化精度低；

饱和量化方式虽然充分利用了低比特表示空间，但是会舍弃一些量化范围。因此这两种方式其实是一个量化精度和量化范围之间的平衡。那么问题来了，对于数据流的饱和量化，怎么在数据流中找到这个最佳阈值(threshold) ?

最佳阈值的选择主要有2种方式，分别是Histogram 与Entropy的方式，关于Histogram 方法的流程如下：



python代码实现

```
def histogramq(value):
    # 计算直方图
    hist, bins = np.histogram(value, 100)
    total = len(value)
    left, right = 0, len(hist)
    limit = 0.99
    while True:
        nleft = left + 1
        nright = right + 1
        left_cover = hist[nleft:right].sum() / total
        right_cover = hist[left:nright].sum() / total
        # 判断是否 left 和 right 都小于limit 的限度，True 退出
        if left_cover < limit and right_cover < limit:
            break
        if left_cover > right_cover:
            left += 1
        else:
            right -= 1

    # 根据直方图占比和limit 计算的left 和right 边界，确定value 中的数值边界
    low, high = bins[left], bins[right - 1]
    # 计算最大绝对值边界
    dynamic_range = max(abs(low), abs(high))

    # 计算scale
    scale = dynamic_range / 127.0
    return scale
```

3.4 Entropy(熵) 校准

TensorRT 采用的校准就是 KL 散度校准。选择使FP32 和int8 的激活值分布的KL距离最小的阈值，校准过程如下所示：

- 统计输入 value 的 max(|x|) 值;
- 统计value 的 Histogram;
- 遍历直方图分布，stride 为 bin 的长度，默认为 128：

  生成 p 分布；

  计算 q 分布；

  归一化p分布和 q 分布；

  计算 p 分布和 q 分布的 KL 散度；
- 求取 KL 散度的最小值对应的位置；
- 求取 threshold；
- 计算 scale 的结果；

举个简单的栗子：

- 假设reference_distribution_P 包含 8 个bins（这里一个bin就只包含一个数据）：

  P = [1, 0, 2, 3, 5, 3, 1, 7]

- 我们想把它映射为 2 个bins，于是 4个一组合并：

  [1 + 0 + 2 + 3 , 5 + 3 + 1 + 7] = [6, 16]

- 然后要成比例的 扩展回到 8个组，保留原来是0的组：

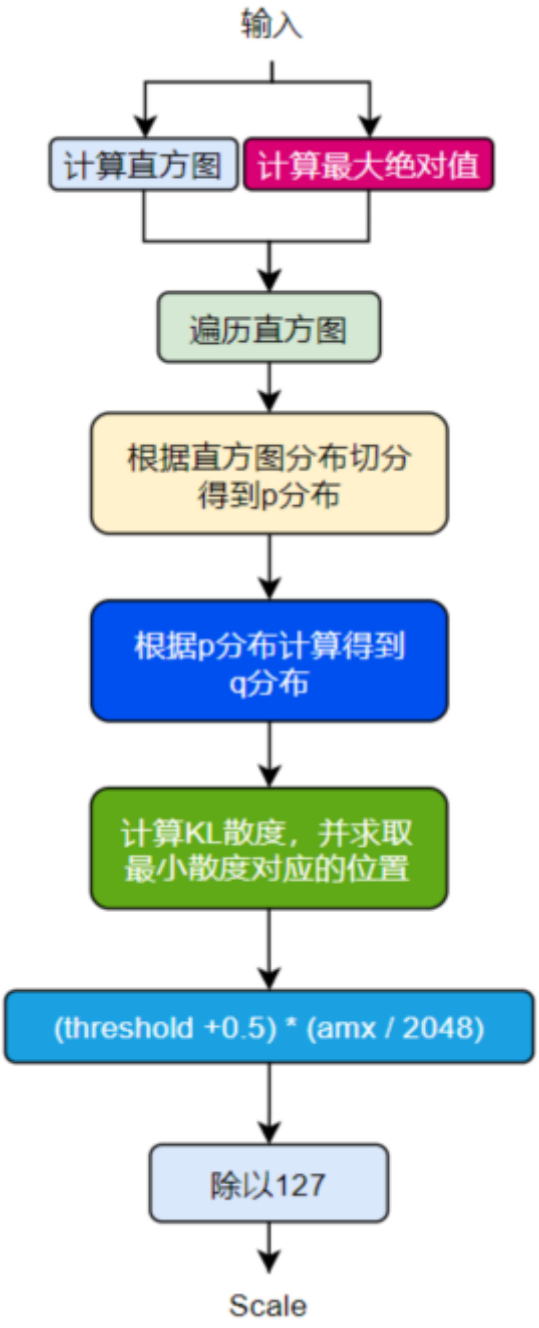
  Q = [6/3, 0, 6/3, 6/3, 16/4, 16/4, 16/4, 16/4] = [2, 0, 2, 2, 4, 4, 4, 4]

- 然后对 P和Q进行标准化：

  P /= sum(P) 、 Q /= sum(Q)

- 最后计算散度：

  result = KL_divergence(P, Q)



python 实现方式

```
# KL 散度校准
def KL(p, q):
    pk = 1.0 * p / np.sum(p)
    qk = 1.0 * q / np.sum(q)
    t = 0
    for i in range(pk.shape[0]):
        t += pk[i] * np.log(pk[i]) - pk[i] * np.log(qk[i])
    return t

def entropy(value, target_bin =  128):
    # 计算最大绝对值
    amax = np.abs(value).max()
    # 计算直方图分布
    distribution, _ = np.histogram(value, bins=2048, range = (0, amax))
    # 遍历直方图分布
    distribution = distribution[1:]
    length = distribution.size
    # 定义KL散度
    kl_divergence = np.zeros(length - target_bin)
    # 遍历 [128:2047]
    for threshold in range(target_bin, length):
        sliced_nd_hist = copy.deepcopy(distribution[:threshold])
        # 复制切分分布为：p
        p = sliced_nd_hist.copy()
        threshold_sum = sum(distribution[threshold:])

        # 边界外的组加到边界p[i-1]上，没有直接
        p[threshold - 1] += threshold_sum
        is_nonzeros = (p != 0).astype(np.int64)

        # 合并bins，步长为： num_merged_bins = sliced_nd_hist.size // target_bin =
16
        quantized_bins = np.zeros(target_bin, dtype=np.int64)
        num_merged_bins = sliced_nd_hist.size // target_bin

        for j in range(target_bin):
            start = j * num_merged_bins
            stop = start + num_merged_bins
            quantized_bins[j] = sliced_nd_hist[start:stop].sum()
        quantized_bins[-1] += sliced_nd_hist[target_bin *
num_merged_bins:].sum()

        # 定义分布: q ，这里的size 要和p分布一致， 也就是和sliced_hd_hist 分布一致
        q = np.zeros(sliced_nd_hist.size, dtype=np.float64)

        # 根据步长结合p的非零以及 quant_p，来以步长填充 q
        for j in range(target_bin):
            start = j * num_merged_bins
            stop = -1 if j == target_bin - 1 else start + num_merged_bins
            norm = is_nonzeros[start:stop].sum()
            q[start:stop] = float(quantized_bins[j]) / float(norm) if norm != 0
else q[start:stop]

        p = p / sum(p)
        q = q / sum(q)
```

```
        # 计算KL散度
        kl_divergence[threshold - target_bin] = KL(p, q)

    min_kl_divergence = np.argmin(kl_divergence)
    threshold_value = min_kl_divergence + target_bin

    scale = (threshold_value + 0.5) * (amax / 2048) / 127.0
    return scale
```

3.5 校准方法对比

python全部代码实现

```
import numpy as np
import copy


# Max 校准
def maxq(value):
    dynamic_range = np.abs(value).max()
    scale = dynamic_range / 127.0
    return scale

# 直方图校准
def histogramq(value):
    # 计算直方图
    hist, bins = np.histogram(value, 100)
    total = len(value)
    left, right = 0, len(hist)
    limit = 0.99
    while True:
        nleft = left + 1
        nright = right + 1
        left_cover = hist[nleft:right].sum() / total
        right_cover = hist[left:nright].sum() / total
        # 判断是否 left 和 right 都小于limit 的限度，True 退出
        if left_cover < limit and right_cover < limit:
            break
        if left_cover > right_cover:
            left += 1
        else:
            right -= 1

    # 根据直方图占比和limit 计算的left 和right 边界，确定value 中的数值边界
    low, high = bins[left], bins[right - 1]
    # 计算最大绝对值边界
    dynamic_range = max(abs(low), abs(high))
    # 计算scale
    scale = dynamic_range / 127.0
    return scale

# KL 散度校准
def KL(p, q):
    pk = 1.0 * p / np.sum(p)
```

```
    qk = 1.0 * q / np.sum(q)
    t = 0
    for i in range(pk.shape[0]):
        t += pk[i] * np.log(pk[i]) - pk[i] * np.log(qk[i])
    return t

def entropy(value, target_bin =  128):
    # 计算最大绝对值
    amax = np.abs(value).max()
    # 计算直方图分布
    distribution, _ = np.histogram(value, bins=2048, range = (0, amax))
    # 遍历直方图分布
    distribution = distribution[1:]
    length = distribution.size
    # 定义KL散度
    kl_divergence = np.zeros(length - target_bin)
    # 遍历 [128:2047]
    for threshold in range(target_bin, length):
        sliced_nd_hist = copy.deepcopy(distribution[:threshold])
        # 复制切分分布为：p
        p = sliced_nd_hist.copy()
        threshold_sum = sum(distribution[threshold:])

        # 边界外的组加到边界p[i-1]上，没有直接
        p[threshold - 1] += threshold_sum
        is_nonzeros = (p != 0).astype(np.int64)

        # 合并bins，步长为： num_merged_bins = sliced_nd_hist.size // target_bin =
16
        quantized_bins = np.zeros(target_bin, dtype=np.int64)
        num_merged_bins = sliced_nd_hist.size // target_bin

        for j in range(target_bin):
            start = j * num_merged_bins
            stop = start + num_merged_bins
            quantized_bins[j] = sliced_nd_hist[start:stop].sum()
        quantized_bins[-1] += sliced_nd_hist[target_bin *
num_merged_bins:].sum()

        # 定义分布: q ，这里的size 要和p分布一致， 也就是和sliced_hd_hist 分布一致
        q = np.zeros(sliced_nd_hist.size, dtype=np.float64)

        # 根据步长结合p的非零以及 quant_p, 来以步长填充 q
        for j in range(target_bin):
            start = j * num_merged_bins
            stop = -1 if j == target_bin - 1 else start + num_merged_bins
            norm = is_nonzeros[start:stop].sum()
            q[start:stop] = float(quantized_bins[j]) / float(norm) if norm != 0
else q[start:stop]

        p = p / sum(p)
        q = q / sum(q)

        # 计算KL散度
        kl_divergence[threshold - target_bin] = KL(p, q)
```

```
    min_kl_divergence = np.argmin(kl_divergence)
    threshold_value = min_kl_divergence + target_bin

    scale = (threshold_value + 0.5) * (amax / 2048) / 127.0
    return scale

# int8截断， 注意，-128去调不要
def saturate(x):
    return np.clip(np.round(x), -127, +127)

class Quant:
    def __init__(self, value, s='max') -> None:
        if s == 'max':
            self.scale = maxq(value)
        if s == 'histogram':
            self.scale = histogramq(value)
        if s == 'entropy':
            self.scale = entropy(value)

    def __call__(self, f):
        return saturate(f / self.scale)

def Quant_Conv(x, w, b, iq, wq, oq=None):
    alpha = iq.scale * wq.scale
    out_int32 = iq(x) * wq(w)

    if oq is None:
        return out_int32 * alpha + b

    else:
        return saturate((out_int32 * alpha + b) / oq.scale)


if __name__ == '__main__':
    np.random.seed(222)
    nelem = 1000

    for s in ['entropy', 'histogram', 'max']:
        # 生成随机权重、输入与偏置向量
        x = np.random.randn(nelem)
        weight1 = np.random.randn(nelem)
        bias1 = np.random.randn(nelem)


        # 计算第一层卷积计算的结果输出（fp32)
        t = x * weight1 + bias1
        weight2 = np.random.randn(nelem)
        bias2 = np.random.randn(nelem)

        # 计算第二层卷积计算的结果输出（fp32)
        y = t * weight2 + bias2
        # 分别对输入、权重以及中间层输出（也是下一层的输入）进行量化校准
        xQ = Quant(x, s)
        w1Q = Quant(weight1, s)
```

```
        tQ = Quant(t, s)
        w2Q = Quant(weight2, s)
        qt = Quant_Conv(x, weight1, bias1, xQ, w1Q, tQ)
        # int8计算的结果输出
        y2 = Quant_Conv(qt, weight2, bias2, tQ, w2Q)
        # 计算量化计算的均方差
        y_diff = (np.abs(y-y2) ** 2).mean()

        print(s, " mse error: ", y_diff)
```

实现结果：

```
entropy  mse error:  2.026332470525213
histogram  mse error:  14.295293418403482
max  mse error:  28.958707956776518
```

对比Max、Histogram以及KL这3种校准方法，量化误差依次是：Max > Histogram > KL。

4. Pytorch-Quantization简介

PyTorch Quantization 是一个工具包，用于训练和评估具有模拟量化的PyTorch模型。

PyTorch Quantization API支持将 PyTorch 模块自动转换为其量化版本。

转换也可以使用 API 手动完成，这允许在不想量化所有模块的情况下进行部分量化。

一些层可能对量化比较敏感，对其不进行量化可提高任务精度。

PyTorch Quantization的量化模型可以直接导出到ONNX，并由TensorRT 8.0或者更高版本导入进行转换Engine。

pip安装 Pytorch Quantization

```
pip install nvidia-pyindex
pip install pytorch-quantization
```

4.1 量化函数

tensor_quant 和fake_tensor_quant 是量化张量的 2 个基本函数；

- fake_tensor_quant 返回伪量化张量（浮点值）。
- tensor_quant 返回量化后的张量（整数值）以及其对应的缩放值Scale

```
from pytorch_quantization import tensor_quant
import torch

# 固定种子
torch.manual_seed(1212)
x = torch.rand(10)
print(x)
# tensor([0.7935, 0.1974, 0.8881, 0.9816, 0.6679, 0.1026, 0.1975, 0.0733, 0.7243,
0.9567])
```

```
# 伪张量为 x:
fake_quant_x = tensor_quant.fake_tensor_quant(x, x.abs().max())
print(fake_quant_x)
# tensor([0.7961, 0.2010, 0.8889, 0.9816, 0.6647, 0.1005, 0.2010, 0.0696, 0.7265,
0.9584])

# 量化张量 x, scale=128.0057
quant_x, scale = tensor_quant.tensor_quant(x, x.abs().max())
print(quant_x)
# tensor([103.,  26., 115., 127.,  86.,  13.,  26.,   9.,  94., 124.])
```

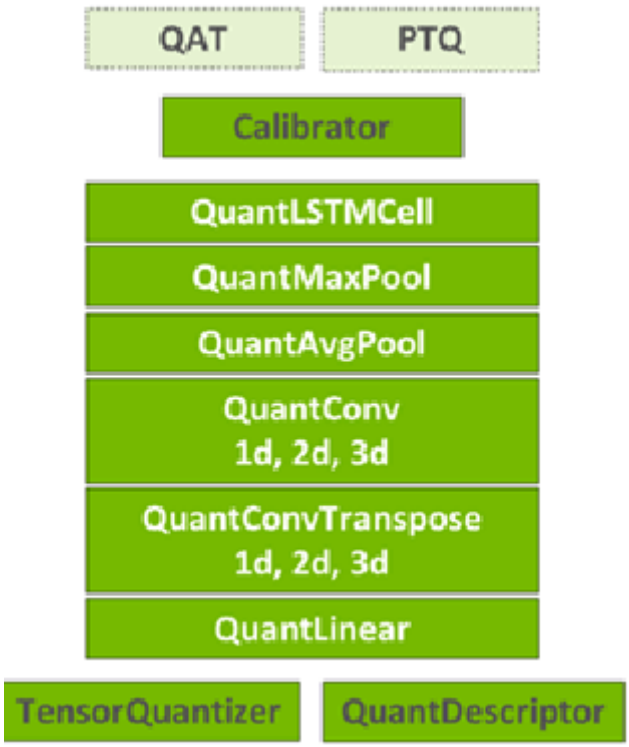
4.2 描述符和量化器

QuantDescriptor 是用来定义张量应如何量化；PyTorch Quantization 提供了一些预定义的 QuantDescriptor,例如：

1. QUANT_DESC_8BIT_PER_TENSOR
2. QUANT_DESC_8BIT_CONV2D_WEIGHT_PER_CHANNEL

TensorQuantizer 可以量化、伪量化或收集张量的统计信息。它与QuantDescriptor 一起使用，后者描述了如何量化张量。

在 TensorQuantizer 之上分层的是量化模块，这些模块被设计为 Pytorch 全精度模块的替代品。这些是使用TensorQuantizer 对模块的权重和输入进行伪量化或收集统计信息的方便模块。



4.3 量化模块

有两种主要类型的模块，Conv 和 Linear 。两者都可以取代torch.nn 版本， 并对权重和激活应用量化。

```
from torch import nn
from pytorch_quantization import tensor_quant
import pytorch_quantization.nn as quant_nn
```

```
# PyTorch 模块
fc1 = nn.Linear(in_features, out_features, bias=True)
conv1 = nn.Conv2d(in_channels, out_channels, kernel_size)

# 量化版本的模块
quant_fc1 = quant_nn.Linear(
    in_features, out_features, bias = True,
    quant_desc_input = tensor_quant.QUANT_DESC_8BIT_PER_TENSOR,
    quant_desc_weight= tensor_quant.QUANT_DESC_8BIT_LINEAR_WEIGHT_PER_ROW
)
quant_conv1 = quant_nn.Conv2d(
    in_channels, out_channels, kernel_size,
    quant_desc_input = tensor_quant.QUANT_DESC_8BIT_PER_TENSOR,
    quant_desc_weight = tensor_quant.QUANT_DESC_8BIT_CONV2d_WEIGHT_PER_CHANNEL
)
```

量化工具提供了以下几个量化模块：QuantConv1d、QuantConv2d、Quant Conv3d、QuantConvTranspose1d、quantConvTranspose 2d、QuantiConvTransbose 3d、QuantiLinear、QuantAvgPool1d、QuantAvg Pool2d、Quant AvgPool 3d、QuantMaxPool1d、QuantMax Pool2D、Quant MaxPool3d。

```
包含量化模块映射的全部模块
_DEFAULT_QUANT_MAP = [_quant_entry(torch.nn, "Conv1d", quant_nn.QuantConv1d),
                      _quant_entry(torch.nn, "Conv2d", quant_nn.QuantConv2d),
                      _quant_entry(torch.nn, "Conv3d", quant_nn.QuantConv3d),
                      _quant_entry(torch.nn, "ConvTranspose1d",
quant_nn.QuantConvTranspose1d),
                      _quant_entry(torch.nn, "ConvTranspose2d",
quant_nn.QuantConvTranspose2d),
                      _quant_entry(torch.nn, "ConvTranspose3d",
quant_nn.QuantConvTranspose3d),
                      _quant_entry(torch.nn, "Linear", quant_nn.QuantLinear),
                      _quant_entry(torch.nn, "LSTM", quant_nn.QuantLSTM),
                      _quant_entry(torch.nn, "LSTMCell",
quant_nn.QuantLSTMCell),
                      _quant_entry(torch.nn, "AvgPool1d",
quant_nn.QuantAvgPool1d),
                      _quant_entry(torch.nn, "AvgPool2d",
quant_nn.QuantAvgPool2d),
                      _quant_entry(torch.nn, "AvgPool3d",
quant_nn.QuantAvgPool3d),
                      _quant_entry(torch.nn, "AdaptiveAvgPool1d",
quant_nn.QuantAdaptiveAvgPool1d),
                      _quant_entry(torch.nn, "AdaptiveAvgPool2d",
quant_nn.QuantAdaptiveAvgPool2d),
                      _quant_entry(torch.nn, "AdaptiveAvgPool3d",
quant_nn.QuantAdaptiveAvgPool3d),]
```

4.4 训练后量化

只需调用quant_modules.initialize(), 就可以对模型进行训练后量化。如果模型不完全由模块定义，则手动创建TensorQuantizer 并将其添加到模型中的相应位置。

```
from pytorch_quantization import quant_modules
quant_modules.initialize()
model = torchvision.models.resnet50()
```

支持3 种校准方法：

[max] : 只需使用全局最大绝对值

[entropy] : TensorRT 的KL校准；

[Histogram] : 基于mse 的校准；

percentile 的目的是根据给定的百分位数排除异常值；

```
def calibrate_model(model, model_name, data_loader, num_calib_batch, calibrator,
hist_percentile, out_dir, device):
    """
        Feed data to the network and calibrate.
        Arguments:
            model: detection model
            model_name: name to use when creating state files
            data_loader: calibration data set
            num_calib_batch: amount of calibration passes to perform
            calibrator: type of calibration to use (max/histogram)
            hist_percentile: percentiles to be used for historgram calibration
            out_dir: dir to save state files in
    """

    if num_calib_batch > 0:
        print("Calibrating model")
        with torch.no_grad():
            collect_stats(model, data_loader, num_calib_batch, device)

        if not calibrator == "histogram":
            compute_amax(model, method="max")
            calib_output = os.path.join(out_dir, F"{model_name}-max-
{num_calib_batch * data_loader.batch_size}.pth")
            torch.save(model.state_dict(), calib_output)
        else:
            for percentile in hist_percentile:
                print(F"{percentile} percentile calibration")
                compute_amax(model, method="percentile")
                calib_output = os.path.join(out_dir, F"{model_name}-percentile-
{percentile}-{num_calib_batch * data_loader.batch_size}.pth")
                torch.save(model.state_dict(), calib_output)

            for method in ["mse", "entropy"]:
                print(F"{method} calibration")
                compute_amax(model, method=method)
                calib_output = os.path.join(out_dir, F"{model_name}-{method}-
{num_calib_batch * data_loader.batch_size}.pth")
```

```
                torch.save(model.state_dict(), calib_output)
```

4.5 量化感知训练

量化感知训练基于直通估计（STE）导数近似，通常大家都叫QAT。校准完成后，量化感知训练只需选择一个训练机制并继续训练校准模型。通常，它不需要微雕很长时间。

QAT 通常使用大约10%的原始训练计划，从初始训练学习率的1%开始，以及余弦退火学习率计划，该计划遵循余弦周期的递减一半，下降到初始微调学习率（初始训练学习速率的0.01%）的1%。

量化感知训练（本质是一个离散的数值优化问题）不是数学上解决的问题。根据经验，给出以下建议：

1. 为了使STE近似效果良好，最好使用较小的学习速率。大的学习速率更可能放大STE 近似引入的方差，并破坏训练的网络。
2. 在训练期间不要改变量化表示（Scale尺度），至少不要太频繁。每一步都改变尺度，实际上就像每一步改变数据格式一样，这很容易影响收敛。

4.6 导出ONNX

导出到ONNX 的目标是通过TensorRT 。因此，只将伪量化模型导出伪TensorRT 将采用的形式。伪量化将被分解伪一对QuantizeLinear/DequantizeLinean ONNX 算子。

TensorRT 将获取该图，并在int8中以最优化的方式执行该图。

首先将TensorQuantizer 的静态成员设置为使用Pytorch 自己的伪量化函数：

quant_nn.TensorQuantizer.use_fb_fake_quant = True

此时的伪量化模型便可以像其他Torch模型一样导出到ONNX，例如：

```
quant_nn.TensorQuantizer.use_fb_fake_quant = True
quant_modules.initialize()
model = torchvision.model.resnet50()

# 加载量化校准后的权重
state_dict = torch.load("quant_resnet50-entropy-1024.pth", map_location="cpu")
model.load_state_dict(state_dict)
model.cuda()

dummy_input = torch.randn(128, 3, 224, 224, device='cuda')

input_names = ['actual_input_1']
output_names = ['output1']

# enable_onnx_checker needs to be disabled. See notes below.
torch.onnx.export(model, dummy_input, "quant_resnet50.onnx", verbose=True,
opset_version=13, enable_onnx_chcker=False)
```

5. Pytorch的准备工作hook函数

5.1 hook 函数

为了节省显存（内存），PyTorch 会自动舍弃图计算的中间结果，所以想要获取这些数值就需要使用hook函数。hook函数在使用后应及时删除（remove), 以避免每次都运行钩子增加运行负载。

hook 方法有4种，Tensor.register_hook、torch.nn.Module.register_forward_hook、torch.nn.Module.register_backward_hook、torch.nn.Module.register_forward_pre_hook

5.1.1 Tensor.register_hook()

用来导出指定张量的梯度，或修改这个梯度值。

代码示例：

```
import torch

def grad_hook(grad):
    grad *= 2

x = torch.tensor([2., 2., 2., 2.], requires_grad=True)
y = torch.pow(x, 2)
z = torch.mean(y)
h = x.register_hook(grad_hook)
z.backward()

print(x.grad)

# 删除 hook
h.remove()

## tensor
```

注意：

1. 上述代码是有效的，但如果写成 grad = grad * 就失效了，因为此时没有对grad进行本地操作，新的grad 值没有传递给指定的梯度。保险起见，最好在def语句中写明return grad。即：return grad
2. 可以用remove()方法取消hook。

   注意remove()必须在backward()之后，因为只有在执行 backward()语句时，pytorch才开始计算梯度，而在 x.register_hook(grad_hook)时它仅仅是"注册"了一个grad的 钩子，此时并没有计算，而执行remove就取消了这 个钩子，然后再backward()时钩子就不起作用了
3. 如果在类中定义钩子函数，输入参数必须先加上self，即 def grad_hook(self, grad):xxxxx

5.1.1 torch.nn.Module.register_forward_hook():

用来导出指定子模块（可以是层、模块等nn.Module类型）的输入输出张量，但只可修改输出，常用来导出或修改卷积特征图

```
inps, out = [], []
def layer_hook(module, inp, out):
    inps.append(inp[0].data.cpu().numpy())
    outs.append(out.data.cpu().numpy())

hook = net.layer1.register_forward_hook(layer_hook)
output = net(input)
hook.remove()
```

注意：

（1）因为模块可以是多输入的，所以输入是tuple型的，需要先提取其中的Tensor再操作；输出是Tensor型　　的可直接用。

（2）导出后不要放到显存上。

（3）只能修改输出out的值，不能修改输入inp的值（不能返回，本地修改也无效），修改时最好用return形式返回。

5.1.2 torch.nn.Module.register_backward_hook

用来导出或修改指定子模块的输入张量。

```
def pre_hook(module, inp):
    inp0 = inp[0]
    inp0 = inp0 * 2
    inp = tuple([inp0])
    return inp

hook = net.layer1.register_forward_pre_hook(pre_hook)
output = net(input)
hook.remove()
```

注意：

（1）inp值是个tuple类型，所以需要先把其中的张量提取出来，再做其它操作，然后还要再转化为tuple 返回。

（2）在执行outpu = net(input) 时才会调用此句，remove() 可放在调用后用来取消钩子。

5.1.3 torch.nn.Module.register_forward_pre_hook()

用来导出指定子模块的输入输出张量的梯度，但只可修改输入张量的梯度（即只能返回gin），输出张量梯度不可修改。

```
gouts = []
def backward_hook(module, gin, gout):
    print(len(gin), len(gout))
    gouts.append(gout[0].data.cpu().numpy())
    gin0, gin1, gin2 = gin
    gin1 = gin1 * 2
    gin2 = gin2 * 3
    gin = tuple([gin0, gin1, gin2])
    return gin

hook = net.layer1.register_backward_hook(backward_hook)
loss.backward()
hook.remove()
```

注意：

（1）其中的grad_in和grad_out都是tuple，必须要先解开，修改时执行操作后再重新放回tuple返回。

（2）这个钩子函数在backward()语句中被调用，所以remove()要放在backward()之后用来取消钩子

6. PTQ（训练后量化）YOLOv5-n的流程

Post-Training-Quantization (PTQ) 是目前常用的模型量化方法之一。

以INT8量化为例，PTQ处理流程如下：

1. 首先在数据集上以 FP32 精度进行模型训练，得到训练好的baseline 模型；
2. 使用小部分数据对 FP32 baseline 模型进行calibration(校准)，这一步主要是得到网络各层weights以及activation的数据分布特性（比如统计最大最小值）；
3. 根据2.中的数据分布特性，计算出网络各层、量化参数；
4. 使用3.中的量化参数对FP32 baseline 进行量化得到INT8模型，并将其部署至推理框架进行推理；

PTQ 方式会使用小部分数据集来估计网络各层 weights 和activation 的数据分布，找到合适的Scale，从而一定程度上降低模型精度的损失。

然而， PTQ方式虽然在大模型上效果较好（例如ResNet101），但是在小模型上经常会有较大的精度损失（例如MobileNet), 同时不同层对于精度的影响也比较大。

PTQ 后的提升

模型	AP50	AP50:95	FPS (A4000显卡)
Torch-FP16	45.7	27.7	-
TRT-FP16	45.7	27.7	592
TRT-int8	41.7	24.9	884
TRT-PTQ-int8	**44.0**	**26.6**	**826**

代码实现文件： quant_flow_ptq_int8.py

7. YOLOv5-n 量化敏感层分析

```
Sensitive summary:
Top0: Using fp16 model.24, map_calibrated = 0.26550
Top1: Using fp16 model.2, map_calibrated = 0.26129
Top2: Using fp16 model.6, map_calibrated = 0.26015
Top3: Using fp16 model.23, map_calibrated = 0.26013
Top4: Using fp16 model.8, map_calibrated = 0.25961
Top5: Using fp16 model.7, map_calibrated = 0.25958
Top6: Using fp16 model.0, map_calibrated = 0.25937
Top7: Using fp16 PTQ, map_calibrated = 0.25937
Top8: Using fp16 model.17, map_calibrated = 0.25934
Top9: Using fp16 model.10, map_calibrated = 0.25925
```

代码实现文件 quant_flow_ptq_sensitive_int8.py

8. YOLOv5-n 的QAT（训练时量化）实现与分析

QAT(Quantization-Aware-Training)

PTQ中的模型训练和量化是分开的，而QAT则是在模型训练时加了伪量化节点，用于模拟模型量化时引起的误差。

QAT处理流程如下：

1. 首先在数据集上以FP32精度进行模型训练，得到训练好的 baseline 模型；
2. 在 baseline 模型中插入伪量化节点。
3. 进行PTQ，得到PTQ后的模型。
4. 进行量化感知训练。
5. 导出ONNX模型。

代码实现文件 quant_flow_qat_int8.py

QAT 的提升：

模型	AP50	AP50:95	FPS (A4000显卡)
Torch-FP16	45.7	27.7	-
TRT-FP16	45.7	27.7	592
TRT-int8	41.7	24.9	884
TRT-PTQ-int8	44.0	26.6	826
TRT-QAT-int8	**45.1**	**27.1**	**826**

PTQ	QAT
不需要重新训练模型	需要重新训练模型
训练与量化过程没有联系，通常难以保证量化后模型的精度	由于量化参数是通过finetune过程学习得到，通常能够保证精度损失较小

9. 模型量化注意点与模型设计思想

9.1 模型量化的注意事项

1. 量化检测器时，尽量不要对 Detect Head 进行量化，一旦量化可能会引起比较大的量化误差；
2. 量化模型时，模型的 First & Second Layer 也尽可能不进行量化（精度损失具有随机性）；
3. TensorRT只支持对称量化，因此 Zero-Point 为 0；
4. PTQ 的结果一般比TensorRT 的结果好，同时更具有灵活性，可以进行局部量化（因为TensorRT时性能优先）；
5. Resize 和 add 操作可能会引人混合精度节点，这可能也是后面TensorRT 改进和优化的点；
6. 激活函数的量化对精度没有影响（ReLU/ReLU6等），因此使用per_tensor粗粒度量化即可；
7. 随机量化并不能提高精度；
8. 量化粒度越精细越高，相应计算复杂度也就越高，速度越慢；一般使用per_channels即可；
9. Depth-wise Conv 可能会破坏截断误差与舍入误差的平衡，进而影响精度；
10. BN 层与Conv的融合可能会对per_tensor量化有所影响，但是对于per_channel没有影响；
11. MobileNet系列中的深度可分离卷积的量化误差比较大，需要进行局部量化；
12. 卷积/转置卷积 的量化选择 per_tensor 与 per_channel 均可。
13. 参数量大的网络模型对于量化加速根据鲁棒；
14. 非对称的 per_channel 量化能够提高精度（TensorRT 不支持，其它框架支持）；
15. 语义分割一般PTQ即可满足量化精度的要求，因为本质是逐像素分类；
16. pytorch-quantization 本身的 initialize 不建议使用，最好使用本次实践中的方法更为灵活；
17. 多分支结构并不利于QAT 的训练，QAT办法缓解PTQ的精度损失。

9.2 模型设计原则

1. 模型涉及和改进避免多分支结构，如果项目中使用了多分支结构，建议使用结构重参思想；
2. 如果使用了结构重参，同时PTQ不能满足要求，QAT也不能缓解，建议使用梯度先验的形式 RepOpt
3. YOLOv6中的上采样使用TransposeConv比YOLOv5中使用的Upsample更适合进行量化，因为使用Upsample在转为Engine的时候，TensorRT会模型将其转为混合精度的Resize，影响性能；
4. 在自己设计Block的时候，应该更多考虑Block中的算子尽可能进行算子融合，YOLOv6这方面就是典范，值得多多学习。
5. 如果模型中涉及到Plugin，使用局部量化跳过该层即可；