

# Quantization\_Notes

---

## 量化(Quantization)

### 1. 简介

### 2. 量化

#### 2.1 后量化

##### 2.1.1 直接计算

##### 2.1.2 直方图截断

##### 2.1.3 滑动平均

##### 2.1.4 均值和方差

##### 2.1.5 自动搜索

#### 2.2 伪量化

#### 2.3 推理

by pass

batch normalization

### 3. 代码实现

基础量化函数

Pytorch 实现

### 4.量化工具

#### 4.1 SNPE

##### 4.1.1 简介

##### 4.1.2 量化算法

##### 4.1.3 量化模式

##### 4.1.4 量化影响

##### 4.1.5 安装配置

#### 4.2 MTK

#### 4.3 AIMET

简介

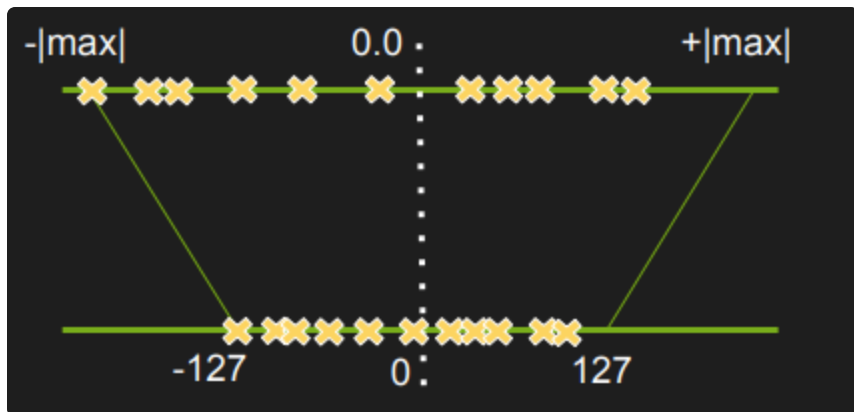
安装配置

模型压缩

# 量化(Quantization)

## 1. 简介

量化就是在神经网络前向计算过程中将浮点数运算量化为整数运算，以达到计算加速的目的，通常是将 float32 转为 int8 进行运算，经实践部署 dsp 上能提升 2.5–3 倍的推理速度。对浮点数的量化操作是将其从一个高维度映射到低维度的转换过程，如下图所示：



实际上，我们在对图像预处理的时候就有用到量化，将 0–255 范围的图片归一化为 0.0~1.0 范围的 float32 类型的张量，就是一个反量化的过程，而反过来，将网络输出范围在 0.0~1.0 之间的张量调整为数值是 0~255，uint8 类型的图片数据，这就是一个量化的过程。

量化的主要流程如下：

1. 统计出网络某一层的最大值和最小值：

$$x_{float} \in [x_{float}^{max}, x_{float}^{min}]$$

2. 计算 scale 和 zero\_point，scale 表示整数和实数之间的比例关系，而 zero\_point 表示实数中的 0 经过量化后对应的整数，其计算方法如下

$$x_{scale} = \frac{x_{float}^{max} - x_{float}^{min}}{x_{quantized}^{max} - x_{quantized}^{min}}$$

$$x_{zeropoint} = round(x_{quantized}^{max} - \frac{x_{float}^{max}}{x_{scale}})$$

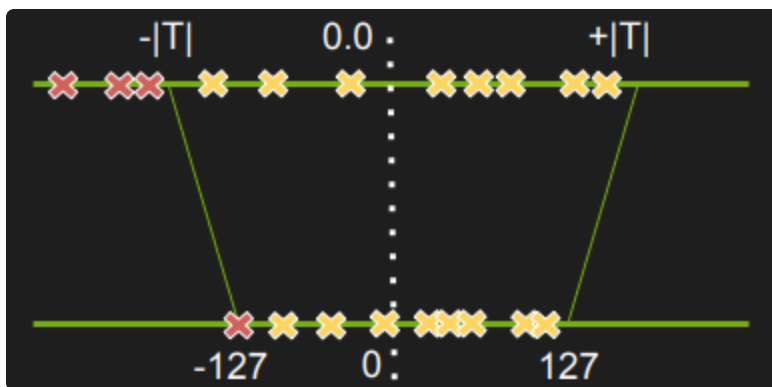
3. 通过以下公式计算出任意 float32 量化后的 int8 结果

$$x_{quantized} = round(\frac{x_{float}}{x_{scale}} + x_{zeropoint})$$

注意定点整数的 zero\_point 就代表浮点实数的 0，二者之间的换算不存在精度上的损失，这样做的目的是为了在 padding 的时候保证浮点数值 0 和定点整数的 zero\_point 完全等价，保证定点和浮点之间的表征能够一致。

从上述公式看，量化中精度的损失是不可避免的，当浮点数分布比较均匀的时候，精度损失较少，但浮点数分布不均匀的时候，按照最大值最小值映射，实际有效的 int8 动态范围就更小了，精度损失就变大了；

这里，Nvidia 提出了通过寻找最优阈值进行非饱和截取思路改善精度损失，如下图所示，具体参考链接：<https://on-demand.gputechconf.com/gtc/2017/presentation/s7310-8-bit-inference-with-tensorrt.pdf>



目前量化其实分为以下几种方式

1. 后训练量化 (post training quantization, PTQ)，在不重新训练的前提下，直接找到 scale 和 zero\_point，操作起来相对方便友好；
2. 量化感知训练 (quantization aware training, QAT)，在网络中加入伪量化节点进行量化训练，操作相对复杂，但效果会更好，一般来说 QAT 的效果决定了 PTQ 的上界。

## 2. 量化

### 2.1 后量化

正如同简介里介绍的，量化里最重要的四个参数分别是  $S$ ,  $Z$ ,  $r_{max}$ ,  $r_{min}$ ，即分别代表 Scale, Zero\_point, 以及浮点数值的最大值和最小值，其计算公式如下（对比前面说的略有修改）：

$$S = \frac{r_{max} - r_{min}}{q_{max} - q_{min}}$$
$$Z = clip(round(q_{max} - \frac{r_{max}}{S}), 0, 255)$$

其中略有修改的地方是  $Z$  的计算增加了 clip 操作，主要是限制  $z$  的范围在 0~255 之间。

那么对于后量化来说，就需要找到合适的量化参数，对于权重 weight 来说，网络训练完后就基本确定了，但对于每层的 feature map 是不知道的，所以通常会用一批矫正数据（一般是训练集的一小部分，即一些代表性数据）跑一遍网络，来统计每层 feature map 的数值范围，也就是浮点数值的范围；

#### 2.1.1 直接计算

那么比较直接的方法，就是根据上述公式，计算得到  $S$ ,  $Z$ ,  $r_{max}$ ,  $r_{min}$ ，但是这种方法的最大问题是容易受到噪声的影响。比如权重或者 feature map 中存在一些离群点，其数值很大，但对结果影响很小，如果也用来计算 minmax，就容易造成浪费。

比如某个 weight 的数值是  $[-0.1, 0.2, 0.3, 255.1]$ ，那么 min 和 max 分别是 -0.1 和 255.1，但 255.1 其实是个离群点，提供的信息有限，反而是更有信息的 0.2 和 0.3 就会映射到同个定点数，信息损失就很大，它们对结果的影响会远大于 255.1，这种情况下的选择应该是丢弃 255.1，保留 0.2 和 0.3。

所以，怎么改进上述办法呢？

#### 2.1.2 直方图截断

第一个改进方式就是根据 weight 和 feature map 的数值范围计算一个直方图，根据直方图来舍弃前后  $x\%$  的数值，用剩下的数值来确定 min 和 max，这样可以剔除掉离群点；

#### 2.1.3 滑动平均

第二种是对 feature map 比较有效的统计方法，来自 Google 论文《Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference》里提到的技巧。

具体的做法是：

1. 将矫正数据集分为几个 batch，依次输入到网络中统计数值；
2. 每次更新数值范围，按照  $r_{max}^t = r_{max}^t * (1 - \alpha) + r_{max}^{t-1} * \alpha$  来更新，其中  $r_{max}^{t-1}$  是上一次统计得到的最大值。

通过控制  $\alpha$  参数可以控制新数据对历史数据的影响，使得最终结果可以覆盖到大部分数值，又不会被一些离群点主导。

#### 2.1.4 均值和方差

第三种改进方法是基于一个前提：通常我们会假设 weight 和 feature 是呈正态分布的。

在此假设下，可以统计出 minmax 的均值和方差  $\mu, \sigma$ ，然后根据正态分布的性质，区间

$(\mu - 3\sigma, \mu + 3\sigma)$  内的数值就占了总数量的 99%，因此可以令  $r_{min} = \mu - 3\sigma, r_{max} = \mu + 3\sigma$ ，这样就基本包含了大部分数值，而且也能过滤掉一些离群点。

不过这个方法得确定数值分布是真的正态分布，否则如果是其他分布，那么就可能反而影响效果了；

#### 2.1.5 自动搜索

上述 3 种方法其实都类似于调参，比如直方图里舍弃的比例是多少，滑动平均中  $\alpha$  应该取值为多少比较合适，正态分布里是否一定要取  $3\sigma$ ，如果离群点其实很重要怎么办？

所以接下来介绍的就是更加 mathematic 的方法，其实就是让方法更加自动化，那最关键的就是评价标准，即衡量信息损失的方法，来判断选择的 minmax 是否合适，精度损失是否最小。比较常用的度量方法包括了欧式距离、L1 距离、KL 散度、余弦距离等；

##### 搜索 minmax

确定好度量方法后就可以自动化搜索最合适的数值范围了，最简单的思路就是在原来的 minmax 区间内，逐步搜索一个更小的数值范围，计算该数值范围的量化损失，损失越小，这个数值范围就更加的合适；

比较常用的 TensorRT 量化算法就是基于 KL 散度来搜索 minmax 的。它采用的是 8bit 对称量化，即范围是 $[-128, 127]$ ，量化过程如下：

1. 首先根据矫正数据集确定数值范围  $[r_{min}, r_{max}]$ ；
2. 把这个范围区间划分为 **2048 份**（相当于离散化成 2048 个 bin 的直方图，具体多少 bin 可以调整）；
3. 以最前面的 128 个 bin 作为基准，逐次向后搜索，每次扩增一个 bin 的长度，得到一个新的数值范围。然后把这个数值范围重新划分为一个 128 个 bin 的直方图 Q（这一步相当于舍弃了部分数值信息，并做了**量化**）；
4. 那要如何评价当前这个数值范围是否合适呢？这个时候 KL 散度就能派上用场了。我们把剩下那些没有搜索到的数值**压缩**到当前搜索到的 bin 上，得到一个信息**基本没有损失**的直方图 P，如果我们之前搜索到的 Q 跟 P 相比信息损失最小（即 KL 散度最小），那这个 Q 对应的数值范围就是最好的数值范围。不巧的是，KL 散度需要两个直方图的 bin 是一样的（L1 距离等也有这个要求），而 Q 之前已经被量化到 128 个 bin 了。为了解决这个问题，需要把 Q 再**反量化**到跟 P 的 bin 数相同，这样就可以计算信息损失了。
5. 重复步骤 3、4，记录每次搜索的 KL 散度大小，直到搜索完整个范围。KL 散度最小的范围，就是理论上信息损失最小的 minmax。

以上就是 TRT 量化的大致过程，基本套路就是：从一个小的搜索范围逐渐扩大出去，每次搜索都量化一遍信息（比如划分成固定 bin 数的直方图），然后用一种度量方式（KL 散度、L1 距离等）来衡量完整信息和量化信息之间的差异，差异最小的区间就是我们需要的 minmax。

对于 S 和 Z 其实也可以用类似的自动搜索办法来搜索到最合适的数值，这个可以参考论文《EasyQuant: Post-training Quantization via Scale Optimization》

---

## 2.2 伪量化

2017 年谷歌的论文《[Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference](#)》提出了一个新的量化框架，在训练过程中引入伪量化的操作，用来模拟量化过程带来的误差。

伪量化就是指将模拟量化操作引入训练过程中，在每个 weight 的输入后和输出 output 前进行伪量化，将浮点量化到定点，再反量化成浮点，用 round 过程中所产生的误差的浮点进行前向运算。

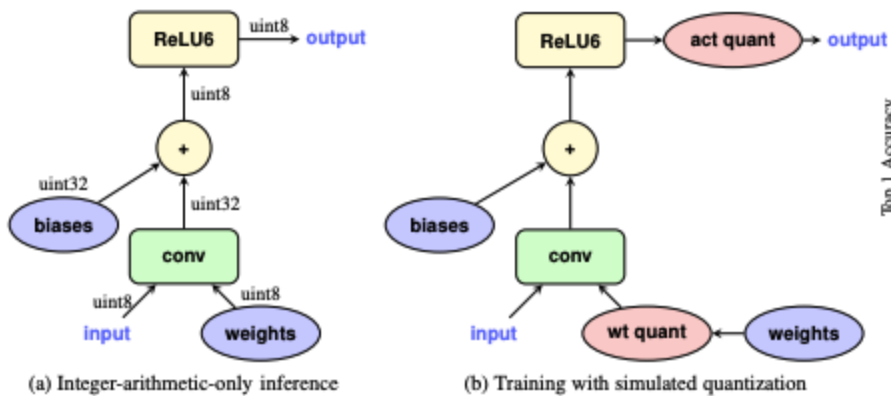
伪量化的优点有：

1. 可以使权值、激活值的分布更加均匀，也就是方差更小，相比直接进行后量化的精度损失能更小；
2. 能够控制每层的输出在一定范围内，对溢出处理更有帮助。

值得注意的是，量化训练中都是采用浮点运算来模拟定点运算，所以训练过程中的量化结果与真实量化结果是有差异的。

## 2.3 推理

量化主要是在推理时候发挥作用，因为训练过程中用的还是浮点训练，只是保存模型才转为 int8 类型，如下图所示展示了常规的量化以及伪量化的区别，a 图表示正常的量化过程，权值和输入数据在卷积之前都要转换为 int8 类型，然后激活函数的输出也是转换为 int8 再输出到下一层，而在伪量化过程中，实际上都是用浮点运算来进行计算的；



### by pass

by pass 结构在 resnet 结构中会遇到，激活值的量化在激活函数和 by pass 后进行，如下图所示，图也是来自谷歌的论文

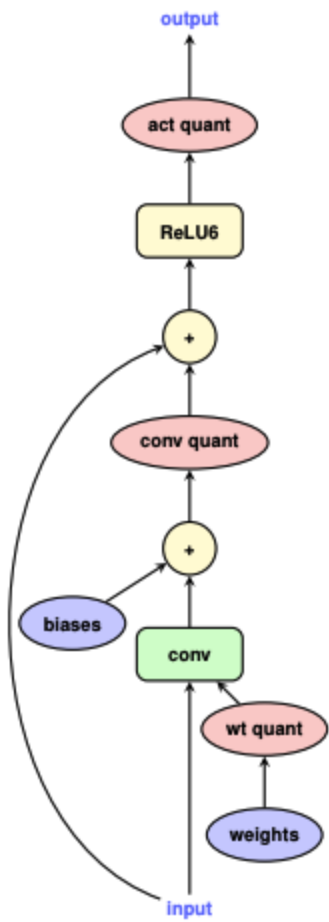


Figure C.4: Layer with a bypass connection: quantized

推理时候则如下所示：

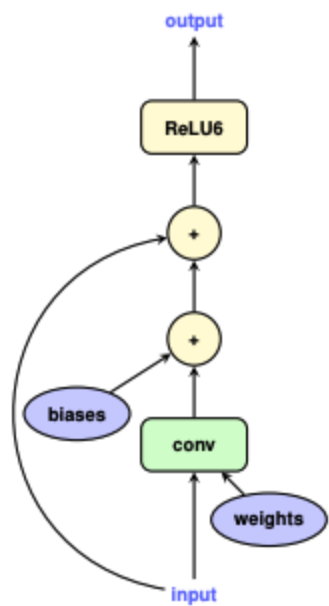


Figure C.3: Layer with a bypass connection: original



## batch normalization

如果有 BN 层，需要在量化前，将 BN 的参数折叠到权重中。

这里简单推导一下，卷积（假设 bias 为 0）的计算公式如下：

$$y = W * x$$

BN 的计算公式为：

$$y = \frac{\gamma(y - \mu)}{\sigma} + \beta$$

两者合并可以得到：

$$y = \frac{\gamma(W * x - \mu)}{\sigma} + \beta$$

如下图所示：

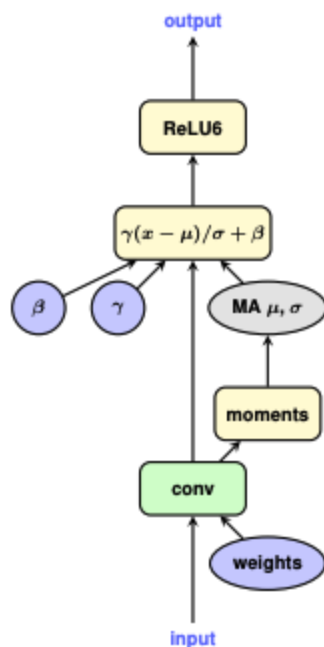


Figure C.5: Convolutional layer with batch normalization: training graph

而折叠是怎么做的呢，下图是伪量化的过程：

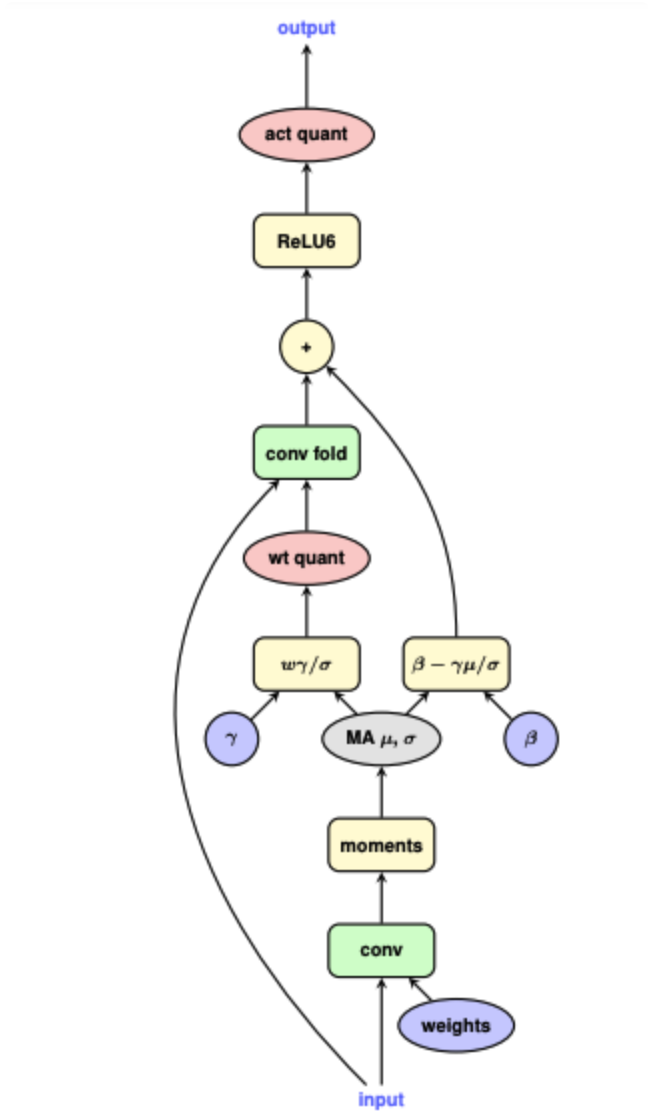


Figure C.8: Convolutional layer with batch normalization: training graph, folded and quantized

那么推理过程如下

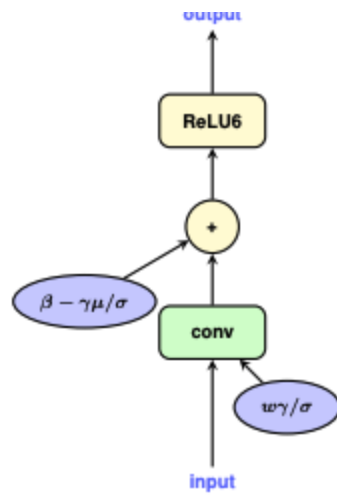


Figure C.6: Convolutional layer with batch normalization: inference graph

### 3. 代码实现

#### 基础量化函数

这里先给出基础量化函数的实现，即上述说的 `scale`, `zero_point` 以及量化和反量化操作：

```

1  def calcScaleZeroPoint(min_val, max_val, num_bits=8):
2      qmin = 0.
3      qmax = 2. ** num_bits - 1.
4      scale = float((max_val - min_val) / (qmax - qmin)) # S=(rmax-rmin)/(qmax-
qmin)
5
6      zero_point = qmax - max_val / scale    # Z=round(qmax-rmax/scale)
7
8      if zero_point < qmin:
9          zero_point = qmin
10     elif zero_point > qmax:
11         zero_point = qmax
12
13     zero_point = int(zero_point)
14
15     return scale, zero_point
16
17 def quantize_tensor(x, scale, zero_point, num_bits=8, signed=False):
18     if signed:
19         qmin = - 2. ** (num_bits - 1)
20         qmax = 2. ** (num_bits - 1) - 1
21     else:
22         qmin = 0.
23         qmax = 2.**num_bits - 1.
24
25     q_x = zero_point + x / scale
26     q_x.clamp_(qmin, qmax).round_()    # q=round(r/S+Z)
27
28     return q_x.float() # 由于pytorch不支持int类型的运算，因此我们还是用float来表示整
数
29
30 def dequantize_tensor(q_x, scale, zero_point):
31     return scale * (q_x - zero_point)    # r=S(q-Z)

```

可以将上述操作封装成一个 `QParam` 类：

```
1 class QParam:
2
3     def __init__(self, num_bits=8):
4         self.num_bits = num_bits
5         self.scale = None
6         self.zero_point = None
7         self.min = None
8         self.max = None
9
10    def update(self, tensor):
11        if self.max is None or self.max < tensor.max():
12            self.max = tensor.max()
13
14        if self.min is None or self.min > tensor.min():
15            self.min = tensor.min()
16
17        self.scale, self.zero_point = calcScaleZeroPoint(self.min, self.max,
18            self.num_bits)
19
20    def quantize_tensor(self, tensor):
21        return quantize_tensor(tensor, self.scale, self.zero_point,
22            num_bits=self.num_bits)
23
24    def dequantize_tensor(self, q_x):
25        return dequantize_tensor(q_x, self.scale, self.zero_point)
```

其中 `update` 函数就是用来统计 min 和 max 的。

折叠 BN 的代码实现如下所示，参考 [神经网络量化入门--Folding BN ReLU代码实现](#)

```

1  class QConvBNReLU(QModule):
2
3      def __init__(self, conv_module, bn_module, qi=True, qo=True, num_bits=8):
4          super(QConvBNReLU, self).__init__(qi=qi, qo=qo, num_bits=num_bits)
5          self.num_bits = num_bits
6          self.conv_module = conv_module
7          self.bn_module = bn_module
8          self.qw = QParam(num_bits=num_bits)
9
10     def forward(self, x):
11
12         if hasattr(self, 'qi'):
13             self.qi.update(x)
14             x = FakeQuantize.apply(x, self.qi)
15
16         if self.training: # 开启BN层训练
17             y = F.conv2d(x, self.conv_module.weight, self.conv_module.bias,
18                         stride=self.conv_module.stride,
19                         padding=self.conv_module.padding,
20                         dilation=self.conv_module.dilation,
21                         groups=self.conv_module.groups)
22             y = y.permute(1, 0, 2, 3) # NCHW -> CNHW
23             y = y.contiguous().view(self.conv_module.out_channels, -1) # CNHW
-> (C,NHW), 这一步是为了方便channel wise计算均值和方差
24             mean = y.mean(1)
25             var = y.var(1)
26             self.bn_module.running_mean = \
27                 self.bn_module.momentum * self.bn_module.running_mean + \
28                 (1 - self.bn_module.momentum) * mean
29             self.bn_module.running_var = \
30                 self.bn_module.momentum * self.bn_module.running_var + \
31                 (1 - self.bn_module.momentum) * var
32         else: # BN层不更新
33             mean = self.bn_module.running_mean
34             var = self.bn_module.running_var
35
36         std = torch.sqrt(var + self.bn_module.eps)
37
38         weight, bias = self.fold_bn(mean, std)
39
40         self.qw.update(weight.data)
41
42         x = F.conv2d(x, FakeQuantize.apply(weight, self.qw), bias,
43                     stride=self.conv_module.stride,
44                     padding=self.conv_module.padding,
45                     dilation=self.conv_module.dilation,
46                     groups=self.conv_module.groups)

```

```

47         x = F.relu(x)
48
49         if hasattr(self, 'qo'):
50             self.qo.update(x)
51             x = FakeQuantize.apply(x, self.qo)
52
53         return x
54
55     def fold_bn(self, mean, std):
56         if self.bn_module.affine:
57             gamma_ = self.bn_module.weight / std # 这一步计算gamma'
58             weight = self.conv_module.weight *
gamma_.view(self.conv_module.out_channels, 1, 1, 1)
59             if self.conv_module.bias is not None:
60                 bias = gamma_ * self.conv_module.bias - gamma_ * mean +
self.bn_module.bias
61             else:
62                 bias = self.bn_module.bias - gamma_ * mean
63         else: # affine为False的情况, gamma=1, beta=0
64             gamma_ = 1 / std
65             weight = self.conv_module.weight * gamma_
66             if self.conv_module.bias is not None:
67                 bias = gamma_ * self.conv_module.bias - gamma_ * mean
68             else:
69                 bias = -gamma_ * mean
70
71         return weight, bias

```

## Pytorch 实现

官方的 github 参考代码：

<https://github.com/pytorch/vision/tree/master/references/classification#quantized>

参考文章

- [Pytorch量化感知训练-代码示例](#)
- [Pytorch量化感知训练详解](#)
- [神经网络量化入门--后训练量化](#)

## 4.量化工具

### 4.1 SNPE

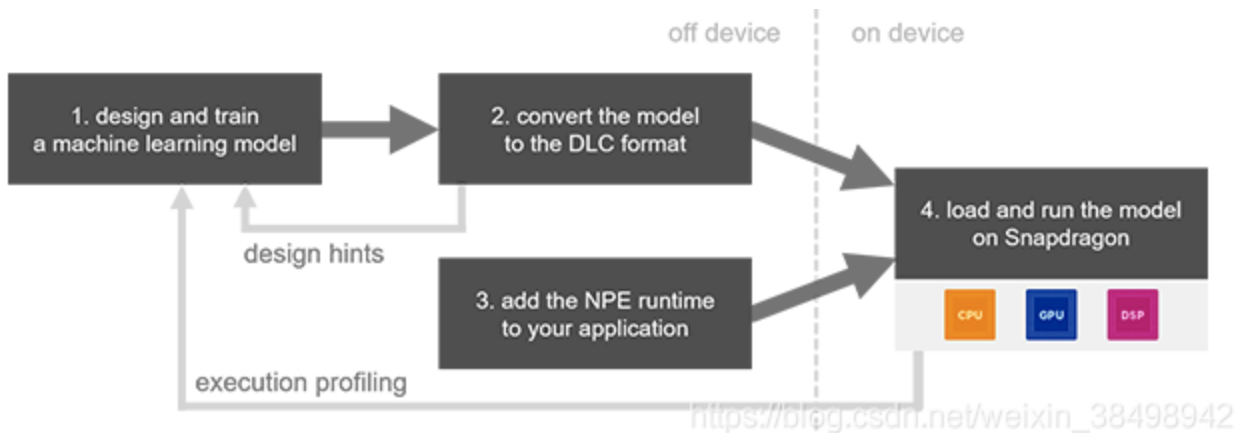
参考：

- [https://blog.csdn.net/weixin\\_38498942/category\\_8993279.html](https://blog.csdn.net/weixin_38498942/category_8993279.html)，这个专栏介绍了 SNPE 的相关内容，包括如何配置和使用；

#### 4.1.1 简介

Snapdragon神经处理引擎（SNPE）是Qualcomm Snapdragon软件加速的运行时，用于执行深度神经网络。借助SNPE，用户可以：1) 执行任意深度的神经网络。2) 在Snapdragon™ CPU，Adreno™ GPU或Hexagon™ DSP上执行网络。3) 在x86 Ubuntu Linux上调试网络执行。4) 将Caffe，Caffe2，ONNX™和TensorFlow™模型转换为SNPE深度学习容器（DLC）文件。5) 将DLC文件量化为8位固定点以在Hexagon DSP上运行。6) 使用SNPE工具调试和分析网络性能。7) 通过C++或Java将网络集成到应用程序和其他代码中。

对于 SNPE 的使用，首先是在深度学习框架（SNPE 支持 Caffe、Caffe2、ONNX 和 TensorFlow 模型）上训练完成后，将模型转为 DLC 文件，然后可以将其加载到 SNPE 运行时中，接着使用 Snapdragon加速的计算核心之一将此DLC文件用于执行前向推理过程。整个开发流程如下图所示：



#### 4.1.2 量化算法

SNPE 支持量化的模型有：

- FXP-CPU runtime 跑的就是 int8 量化的模型，但前提要对DLC格式的模型使用量化脚本来量化；
- DSP和AIP runtime也可以跑量化模型，虽然模型int8量化不是必须的，但是之后DSP加载模型初始化时，也会对加载的参数量化，但可能DSP自动加载量化会有精度不如用脚本对模型量化的方式。

#### 算法流程

1. 计算输入数据的浮点范围，得到最大和最小的浮点值；
2. 计算 encoding-min 和 encoding-max，这两个数值将用于第三步的量化



- encoding-min: 将以定点的形式表示, 表示input的浮点值的最小值;
- encoding-max: 将以定点的形式表示, 表示input的浮点值的最大值;
- 每一步的浮点值:  $\text{step\_size} := (\text{encoding\_max} - \text{encoding\_min}) / 255$ , 不过这里先临时用第一步得到的float\_max与float\_min 来表示 encoding\_max 与 encoding\_min。  
encoding\_max 与 encoding\_min 的范围区间不能小于 0.01。那么有:  $\text{encoding\_max} := \max(\text{float\_max}, \text{float\_min} + 0.01)$ , encoding\_min后面再调整;
- 0 的浮点值必须精确表示, 根据输入值分情况:
  - 全为正:  $\text{encoding\_min} = 0.0$ , 即最小的定点值0来表示浮点0;
  - 全为负:  $\text{encoding\_max} = 0.0$ , 即最大的定点值255来表示浮点0;
  - 有正有负: 如输入浮点范围为 $[-5.1, 5.1]$ :
    - $\text{encoding\_min} = -5.1$ ,  $\text{encoding\_max} = 5.1$ ;
    - encoding区间10.2,  $\text{step\_size} = 10.2 / 255 = 0.04$ ;
    - 先不表示0, 因为定点step\_idx区间是 $[0, 255]$ , 观察在给定最小值+步长情况下, 即  $\text{encoding\_min} + \text{step\_idx} * \text{step\_size}$  的值与0最近的值的step\_idx, 与0最近的step\_idx 必然一正一负, 发现刚好步数step\_idx为127和128时, 分别是-0.02与+0.02与0最近。因为这俩数 (的绝对值) 与0一样近, (127与128这两个数) 选哪个作为0都可以, 那就选小的吧, 那么step\_idx即第127步就是0点, 那么这样一来, 最小值和最大值就都被拉低了0.02与0的差即0.02, 新的 $\text{encoding\_min} = -5.12$ ,  $\text{encoding\_max} = 5.08$ ;
- 3. 对输入浮点值定点量化:  $\text{quantized\_value} = \text{round}(255 * (\text{float\_val} - \text{encoding\_min}) / (\text{encoding\_max} - \text{encoding\_min}))$ , 这样量化的值被夹在 $[0, 255]$ , 考虑两端的情况:
  - 当遇到最小值-5.10时,  $\text{quantized\_value} = \text{round}(255 * (-5.10 + 5.12) / (5.08 + 5.12)) = 1.0$ ;
  - 当遇到最大值+5.10时,  $\text{quantized\_vlaue} = \text{round}(255 * (5.10 + 5.12) / (5.08 + 5.12)) = 256$ , 这种情况是溢出了, 但是 SNPE 文档里规定了量化值在0-255 之间
- 4. 输出: 浮点的量化结果, 即 encoding\_min 和 encoding\_max

这里给出一个例子, 简单进行量化和反量化。

首先输入数据是  $[-1.8, -1.0, 0, 0.5]$ 。

## 量化

1.  $\text{encoding\_max} := 0.5$ ,  $\text{encoding\_min} := -1.8$ ;
2.  $\text{range} := \text{encoding\_max} - \text{encoding\_min} = 2.3$ ,  $\text{step\_size} := \text{range} / 255 = 0.009019607843137253$ ;
3. 有正有负, 根据 $\text{encoding\_min} + \text{step\_idx} * \text{step\_size}$ ,  $\text{step\_idx} \in [0, 255]$ 。观察到距离0最近的step\_idx分别为:
  - step\_idx为199时:  $\text{zero\_diff\_neg} = -1.8 + 199 * 2.3 / 255 = -0.005098039215686301$ ;
  - step\_idx为200时:  $\text{zero\_diff\_pos} = -1.8 + 200 * 2.3 / 255 = 0.003921568627450744$ ;
  - 从绝对值来说, step\_idx为200时距离0更近, 即 $\text{abs}(\text{zero\_diff\_neg}) > \text{abs}(\text{zero\_diff\_pos})$ 。那么就选200这个step\_idx作为定点与浮点0相对应的点,  $\text{zero\_step\_idx} := 200$ ;

4. 修正 encoding\_max 与 encoding\_min, 根据 step\_idx=200 时的 zero\_offset = 0.003921568627450744, 对二者修正, 都往右移:
  - encoding\_min := encoding\_min + zero\_offset = -1.7960784313725493;
  - encoding\_max := encoding\_max + zero\_offset = 0.5039215686274507; 注: 这里SNPE文档中, 算的zero\_offset=-0.003921568627450744, 是负数, 怀疑是否是SNPE搞错了正负号。
5. 量化输入为定点, 根据公式  $\text{quant\_value} = \text{round}(\frac{255 * (\text{float\_val} - \text{encoding\_min})}{(\text{encoding\_max} - \text{encoding\_min})})$ : 1) quant\_value(-1.8) = -0.0; 2) quant\_value(-1.0) = 88; 3) quant\_value(0) = 199; 4) quant\_value(0.5) = 255;
6. 输出量化结果和 encoding\_min 与 encoding\_max。

## 反量化

输入:

- 量化的定点值: [0, 88, 199, 255];
- encoding\_min = -1.7960784313725493;
- encoding\_max = 0.5039215686274507;
- step\_size = 0.009019607843137253。

反量化公式, 根据量化公式  $\text{quant\_value} = \text{round}(\frac{255 * (\text{float\_value} - \text{encoding\_min})}{(\text{encoding\_max} - \text{encoding\_min})})$  推导出来float\_value。反量化公式即:  $\text{float\_value} = \frac{\text{quant\_value} * (\text{encoding\_max} - \text{encoding\_min})}{255} + \text{encoding\_min}$ ; 则有反量化结果:

- float\_value(0) = -1.7960784313725493;
- float\_value(88) = -1.0023529411764711;
- float\_value(199) = -0.0011764705882355564;
- float\_value(255) = 0.5039215686274505。

### 4.1.3 量化模式

SNPE 支持两种 8bit 定点的模式, 区别在于量化参数的选择。

1. 默认量化方式, 前面量化算法介绍的。特点是使用真实数据的最大最小值来做量化, 有一个最小区间阈值是 0.01, 以及 0 要被精确表示 (而且也会影响到对最大最小值的修正)
2. 增强量化方式, 给量化命令 snpe-dlc-quantize 传入 use\_enhanced\_quantizer 从而调用。该模式会计算出一组更好的量化参数, 而且也要求最小范围range阈值为0.01, 确保0被精确量化 (类似默认量化方式)。但从表象上来说也有几点不同:
  - 该方式选出的最大最小值, 可能和默认量化方式的不同;
  - 设置的range范围可能权重或者激活值并没不会出现在这个范围内;
  - 更适用于具有长尾性质的权重或者激活值 (如大多数权重位于-100到1000之间, 但是有几一些值远远小于-100, 大于1000)。在某些情况下, 这种长尾值因为被忽略掉 (比用真实最大最小计算量化范围的量化方式, 如默认方式), 有更好的分类精度。

但第二种方式的计算精度, 在某些长尾情况下, 比简单直接使用真实最大最小的情况要好。

#### 4.1.4 量化影响

量化对模型对DSP是必不可少，因为DSP只能跑定点模型，而定点模型又会影响到精度，某些模型量化效果不好甚至出现错误结果。根据SNPE文档，评估量化模型的指标有：

1. Mean Average Precision
2. Top1误差：概率最高的类别不是真实类别；
3. Top5误差：真实类别不在概率前5类别中。

#### 4.1.5 安装配置

参考文章：

1. [高通SNPE开发环境搭建（一）](#)
2. [高通SNPE开发环境搭建（二）](#)

首先是基本的安装 python 的环境，这里推荐使用 anaconda 来创建专门的一个虚拟环境。

然后 SNPE SDK 的安装，下载 SDK 是在官网下载：


1. 访问Qualcomm 网站:(注册、登录 Qualcomm 账号):  
<https://developer.qualcomm.com/software/qualcomm-neural-processing-sdk>
2. 在 ‘Neural Processing SDK for AI -> Tools & Resources’ 页面, 选择 Qualcomm Neural Processing SDK for AI v1.35. 版本下载到本地,并解压文件。

#### 配置环境和依赖

1. 运行依赖项脚本以检查系统中的Ubuntu软件包依赖项。 它会要求安装缺少的那些。 安装缺少的软件包。

```
1 source snpe-X.Y.Z/bin/dependencies.sh
```


Plain Text

 复制代码

运行python依赖项检查器以检查系统中python软件包的依赖项，安装缺少的软件包。

```
1 source snpe-X.Y.Z/bin/check_python_depends.sh
```

Plain Text

 复制代码

经SNPE SDK测试的Python软件包版本为：

```
1  numpy v1.16.5
2  sphinx v2.2.1
3  scipy v1.3.1
4  matplotlib v3.0.3
5  skimage v0.15.0
6  protobuf v3.6.0
7  pyyaml v5.1
```

3. Define \$SNPE\_ROOT in .bashrc file.

```
1  export SNPE_ROOT=/home/user/SNPE-envri/snpe-1.35.0.698
```

参考高通官网网址: <https://developer.qualcomm.com/docs/snpe/setup.html>

接着则是安装所需的深度学习框架, Caffe, TensorFlow, 或者 PyTorch。

---

## 4.2 MTK

## 4.3 AIMET

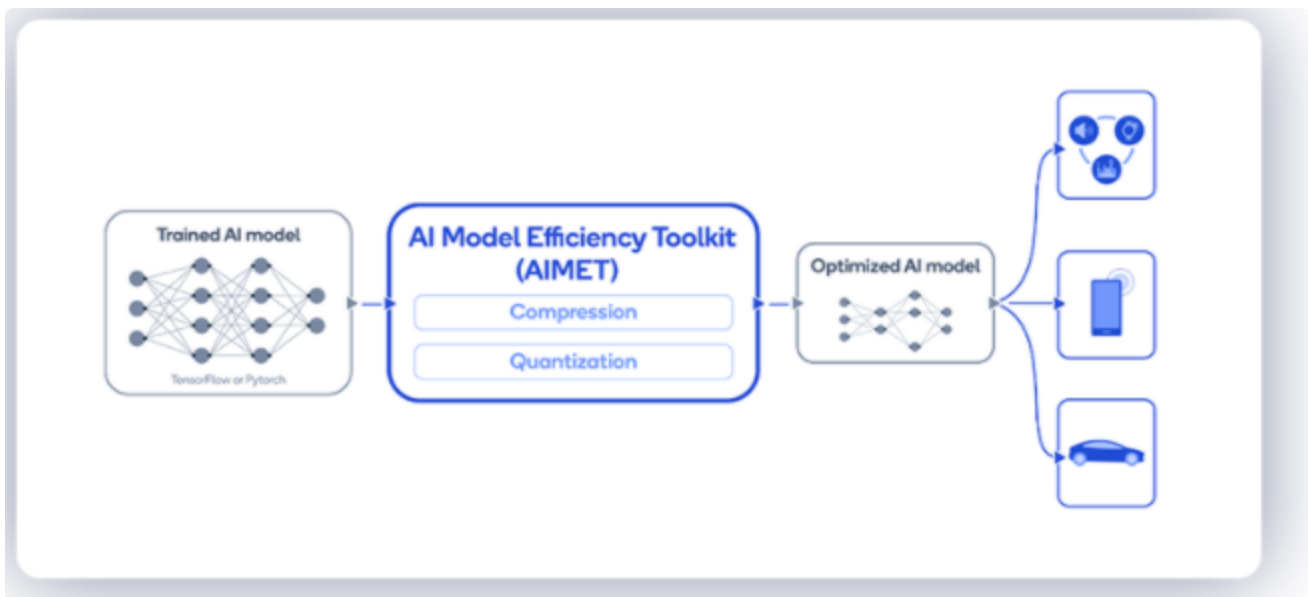
参考文章:

- [AIMET工具简介](#)
- [AIMET工具安装介绍\(1\)](#)
- [AIMET工具安装介绍\(3\)--在 Docker 中的安装和用法](#)

### 简介

AIMET, 全称是 AI Model Efficiency Toolkit, AI 模型效率工具包, 是一个为经过训练的神经网络模型提供高级模型量化和压缩技术的库。它提供的功能已被证明可以提高深度学习神经网络模型的运行时性能, 并具有较低的计算和内存要求, 并且对任务准确性的影响最小, 如下图所示, 它旨在和 PyTorch 以

及 TensorFlow 模型配合使用。



AIMET 的优势包括了：

- **支持高级量化技术：**使用整数运行时进行推理比使用浮点运行时快得多。例如，模型在 Qualcomm Hexagon DSP 上的运行速度比在 Qualcomm Kryo CPU 上快 5 到 15 倍。此外，8 位精度模型的占用空间比 32 位精度模型小 4 倍。然而，在量化 ML 模型时保持模型准确性通常具有挑战性。AIMET 使用无数据量化等新技术解决了这个问题，这些技术在几个流行模型上提供了最先进的 INT8 结果。
- **支持先进的模型压缩技术，**使模型在推理时运行得更快并需要更少的内存。
- **AIMET 旨在自动优化神经网络，避免耗时且繁琐的手动调整。** AIMET 还提供用户友好的 API，允许用户直接从他们的 TensorFlow 或 PyTorch 管道进行调用。

其支持的功能有：

### 量化

- 跨层均衡：均衡权重张量以减少跨通道的幅度变化
- 偏差校正：校正由于量化而引入的层输出偏移
- 自适应舍入：学习给定未标记数据的最佳舍入
- 量化模拟：模拟目标量化推理精度
- 量化感知训练：使用量化模拟进一步训练模型以提高准确性

### 模型压缩

- 空间 SVD：张量分解技术将一个大层分成两个较小的层
- 通道修剪：从层中删除冗余输入通道并重建层权重
- 每层压缩比选择：自动选择模型中每一层的压缩量

## 可视化

- 权重范围：目视检查模型是否适合应用跨层均衡技术。以及应用该技术后的效果
- 每层压缩敏感性：直观地获得有关模型中任何给定层对压缩敏感性的反馈

## 安装配置


转到 <https://github.com/quic/aimet/releases> 并确定要安装的软件包的发布标记。

根据所需的变体，将 设置为以下之一

- 对于 PyTorch GPU 变体，请使用“torch-gpu”
- 对于 PyTorch CPU 变体，请使用“torch-cpu”
- 对于 TensorFlow GPU 变体，请使用“tf-gpu”
- 对于 TensorFlow CPU 变体，请使用“tf-cpu”

```
1 export AIMET_VARIANT=<variant_string>
```


Plain Text

 复制代码

将以下步骤中的 替换为适当的标签：

```
1 release_tag=<release_tag>
```

Plain Text

 复制代码

按照下面指定的顺序安装 AIMET 包：

注意：

- Python 依赖项将自动安装。
- 根据发布页面上的实际轮文件名，将 py3-none-any 替换为 cp36-cp36m-linux\_x86\_64 或 cp37-cp37m-linux\_x86\_64。

```

1  release_tag=<release_tag>
2  python3 -m pip install
   https://github.com/quic/aimet/releases/download/${release_tag}/AimetCommon-${
   AIMET_VARIANT}_${release_tag}-py3-none-any.whl
3
4  # Install ONE of the following depending on the variant
5  python3 -m pip install
   https://github.com/quic/aimet/releases/download/${release_tag}/AimetTorch-${A
   IMET_VARIANT}_${release_tag}-py3-none-any.whl -f
   https://download.pytorch.org/whl/torch_stable.html
6  # OR
7  python3 -m pip install
   https://github.com/quic/aimet/releases/download/${release_tag}/AimetTensorflo
   w-${AIMET_VARIANT}_${release_tag}-py3-none-any.whl
8
9  python3 -m pip install
   https://github.com/quic/aimet/releases/download/${release_tag}/Aimet-${AIMET_
   VARIANT}_${release_tag}-py3-none-any.whl

```

设置常用环境变量如下：

```

1  source /usr/local/lib/python3.6/dist-packages/aimet_common/bin/envsetup.sh

```

将 AIMET 包位置添加到环境路径中，如下所示：

```

1  export LD_LIBRARY_PATH=/usr/local/lib/python3.6/dist-
   packages/aimet_common/x86_64-linux-gnu:/usr/local/lib/python3.6/dist-
   packages/aimet_common:$LD_LIBRARY_PATH
2
3  if [[ $PYTHONPATH = "" ]]; then export
   PYTHONPATH=/usr/local/lib/python3.6/dist-packages/aimet_common/x86_64-linux-
   gnu; else export PYTHONPATH=/usr/local/lib/python3.6/dist-
   packages/aimet_common/x86_64-linux-gnu:$PYTHONPATH; fi

```

## 模型压缩

参考文章：

- [AIMET工具 压缩介绍\(1\)](#)

AIMET 提供了一个模型压缩库，可用于以最小的精度下降降低模型的 MAC 和内存成本。AIMET 支持各种压缩方案，如权重 SVD、空间 SVD 和通道修剪。AIMET 压缩的过程如下所示，AIMET 允许用户采用经过训练的模型并将其压缩到所需的压缩率，然后可以进一步微调并导出到目标。AIMET 中的所有压缩方案都使用两步过程：

- 压缩比选择
- 模型压缩



## 压缩比选择

**贪婪压缩率选择：**在此阶段，分析原始模型的各个层以确定每层的最佳压缩率。目前 AIMET 支持贪婪压缩比选择方法。

**手动压缩率选择：**作为 AIMET 自动选择每层最佳压缩率的替代方法，用户可以选择手动指定每层压缩率。建议的程序是首先使用贪婪压缩比选择方法来获得一组标称的压缩比。然后以此为起点手动更改一层或多层的压缩比。

## 模型压缩

在此阶段，AIMET 将应用每层的压缩率来创建压缩模型。目前，AIMET 支持以下模型压缩算法。

- 权重SVD
- 空间SVD
- 通道修剪

## 模型量化

参考文章：

- [AIMET工具模型量化介绍（1）](#)

## 用途

1. **预测目标精度：**AIMET 使用户能够模拟量化的效果，从而在量化目标上运行时获得模型精度的一阶估计。这对于在不需要实际目标平台的情况下估计目标精度很有用。请注意，为了创建模拟模型，



AIMET 使用代表性数据样本来计算每层量化编码。

2. **计算每层编码的微调模型**：AIMET 使用户能够使用代表性数据样本计算每层量化编码。并且它进一步使用户能够使用训练管道来微调模型，以在给定这些计算编码的情况下提高量化精度。
3. **训练后量化（无微调）**：在某些情况下，用户可能不希望进一步训练模型以提高量化精度。这可能是  
一些原因
  - 可能无法访问此模型的训练管道
  - 可能无法访问用于微调的标记训练数据
  - 可能想要避免选择正确的超参数和训练模型所需的时间和精力

在上述场景中，AIMET 提供了一组训练后量化技术，可以改变模型参数以实现更好的量化精度。这些技术旨在解决模型中的特定均衡问题，可能不适用于所有模型。

## 量化功能

**量化模拟**：AIMET 使用户能够修改模型以添加量化模拟操作。当使用这些量化模拟操作在模型上运行评估时，用户可以观察量化硬件上预期精度的一阶模拟。

**量化可视化**：AIMET 提供可视化工具，帮助指导用户确定 AIMET 训练后量化技术是否对给定模型有用

**应用自适应舍入**：确定权重张量的最佳舍入以提高量化性能

**应用跨层均衡**：训练后量化技术可帮助模型提高量化精度，而无需重新训练。跨层均衡均衡了连续层中的权重范围。

**应用偏差校正**：偏差校正可校正由于量化噪声引起的层输出偏移

**Quantization-aware Fine-tuning**：用户可以使用 AIMET 生成的 Quantization Sim 模型进行训练，以提高模型的量化精度。这是在量化硬件上模拟模型的微调。

## 量化感知微调的建议

以下是一些通用指南，可帮助通过量化感知训练（QAT）提高性能或加快收敛速度：

- **初始化**：通常，在应用 QAT 之前首先应用训练后量化（跨层均衡（CLE）和偏差校正）可能是有益的。如果 INT8 性能与 FP32 基线相比大幅下降，这尤其有益。
- **超参数**：
  - epoch数：一般15–20个epoch就足够收敛了

- 学习率：与 FP32模型在收敛时的最终学习率相当（或高一阶）。 AIMET 中的结果是学习  $1e-6$  阶。
- 学习率计划：每 5-10 个时期将学习率除以 10

---

## 参考文章

1. [量化训练：Quantization Aware Training in Tensorflow（一）](#)
2. [量化训练：Quantization Aware Training（二）](#)
3. [Pytorch量化感知训练-代码示例](#)
4. [Pytorch量化感知训练详解](#)
5. [神经网络量化入门--后训练量化](#)
6. [神经网络量化入门--量化感知训练](#)
7. [神经网络量化入门--Folding BatchNorm ReLU](#)
8. [神经网络量化入门--基本原理](#)
9. [神经网络量化--per-channel量化](#)
10. [盘点一下后训练量化的基本操作](#)