# 1 低精度量化研究

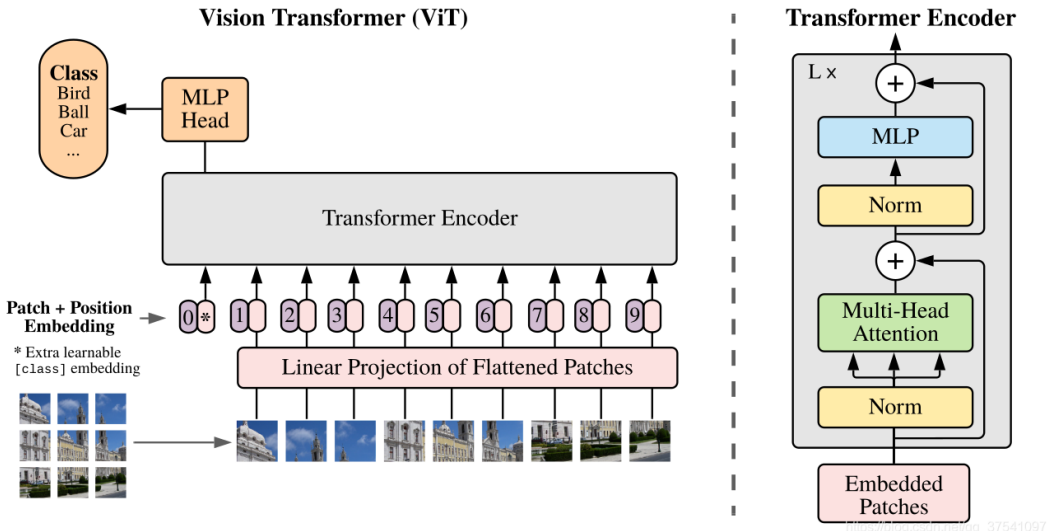
## 1.1 业界研究情况

### 1.1.1 背景综述

业界在训练后量化和量化感知训练中都取得了长足进步，对于深度学习的每一个模块都有相应的量化策略，可以实现全网络从输入起一直采用定点整数进行计算直到输出。一些研究旨在探讨通过非线性量化、更加合理的舍入等来达到更高的精度。量化后的模型更加轻量，计算更快，常常用于移动端设备的使用。

## 1.2 研究内容

研究基于 ViT 网络进行：




基于 Github 上的开源项目 Cifar10 上的 ViT 进行量化。

本项目的网络结构如上图，其中 Transformer Encoder 的块数 L 为 6，本实现不需要卷积层，矩阵型计算存在于 Multi-Head Attention 模块中（每个模块有两次非叶子节点之间的矩阵乘）和 Linear 层中（$y = xA^T + b$，主要存在于 MLP 模块中）。在一次前向传播中，发生 $1 + 6 \times (4 + 2) + 1 = 38$次矩阵型计算。

### 1.2.1 量化中 INT4 类型的比例

分别统计推理准确率损失2%，5%，10%的情况下，最多可以多少比例的计算（针对矩阵型计算，统计乘法次数）采用 INT4 类型。

#### 1.2.1.1 研究内容

训练后量化（PTQ），即先在全精度下训练模型，再在推理时进行量化。

如果我们希望将矩阵乘法在更低的 bit 下完成，需要对数据进行量化，即将浮点实数的乘数映射到$0 \sim 2^n - 1$（n为量化的 bit 数）范围的整数：

$$r = s(q - z)$$

$$q = \text{round}\left(\frac{r}{s} + z\right)$$

其中r为浮点实数，q为量化后的定点整数；S为缩放因子，表示实数和整数之间的比例关系，Z为零点，表示实数中的 0 经过量化后对应的整数。对于S和Z的决定，为了避免截断误差，此处采用最大最小值法：

$$s = \frac{r_{\max} - r_{\min}}{q_{\max} - q_{\min}}$$

$$z = \text{round}\left(q_{\max} - \frac{r_{\max}}{s}\right)$$

其中$q_{max} = 2^n - 1$，$q_{min} = 0$，零点取整的原因是保证 0 的换算不存在精度损失。
对一个矩阵乘法：

$$C = AB$$

有

$$A = s_A(Q_A - Z_A)$$

$$B = s_B(Q_B - Z_B)$$

代入得：

$$C = s_A(Q_A - Z_A)s_B(Q_B - Z_B)$$

$$= s_A s_B(Q_A Q_B - Q_A Z_B - Z_A Q_B + Z_A Z_B)$$

$$= s_A s_B(Q_A Q_B - z_B Q_A \mathbf{1} - z_A \mathbf{1} Q_B + z_A z_B \mathbf{1})$$

其中$Q_A\mathbf{1}$与$\mathbf{1}Q_B$分别为$Q_A$按行作和和$Q_B$按列作和，则原式的计算时间复杂度集中在$Q_A Q_B$上，也就实现了矩阵乘法的量化。

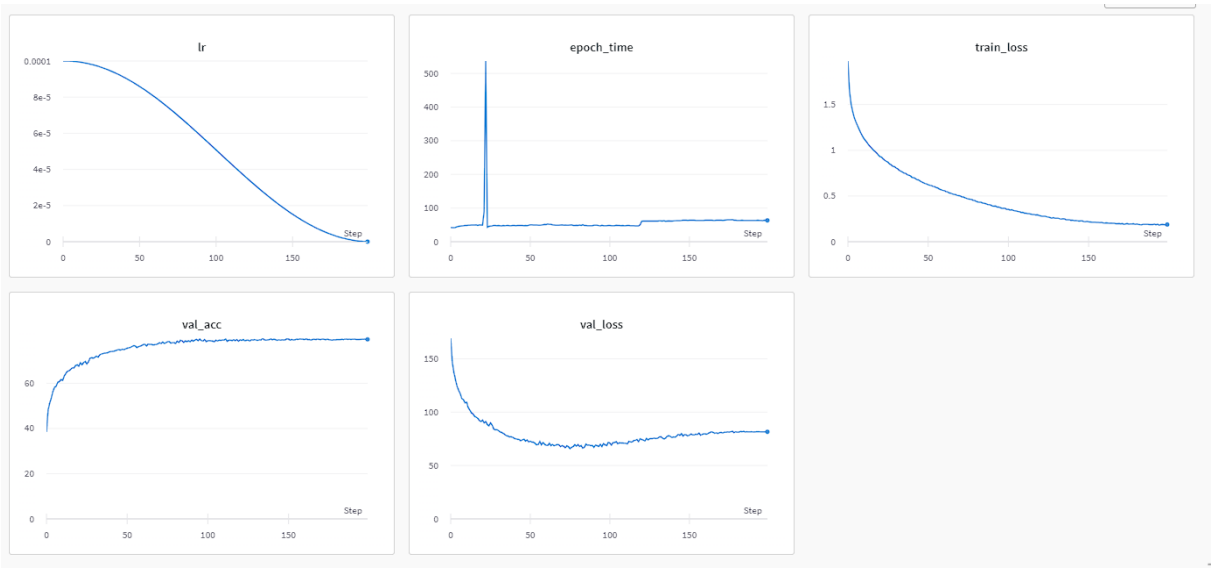
本次实验采用 per-tensor 量化、动态量化，即在每次矩阵乘法前，独立地计算两个乘数的$s$和$z$并量化，再进行矩阵运算。如原本为如下代码：

```
out = torch.matmul(attn, v)
```

则推理阶段变为：

```
a, self.sa, self.za = quantize(a, self.quat)
v, self.sv, self.zv = quantize(v, self.quat)
out = (torch.matmul(a, v) - v.sum(axis=2, keepdim=True) * self.za
- a.sum(axis=3,

keepdim=True) * self.zv + self.za * self.zv * a.size(
    3)) * self.sa * self.sv
```

对于一个全精度下预训练 200 个 epochs 的 ViT 模型：




其推理 acc 达到79.58%。

现将全部矩阵型运算均进行 4bit 量化，推理 acc：

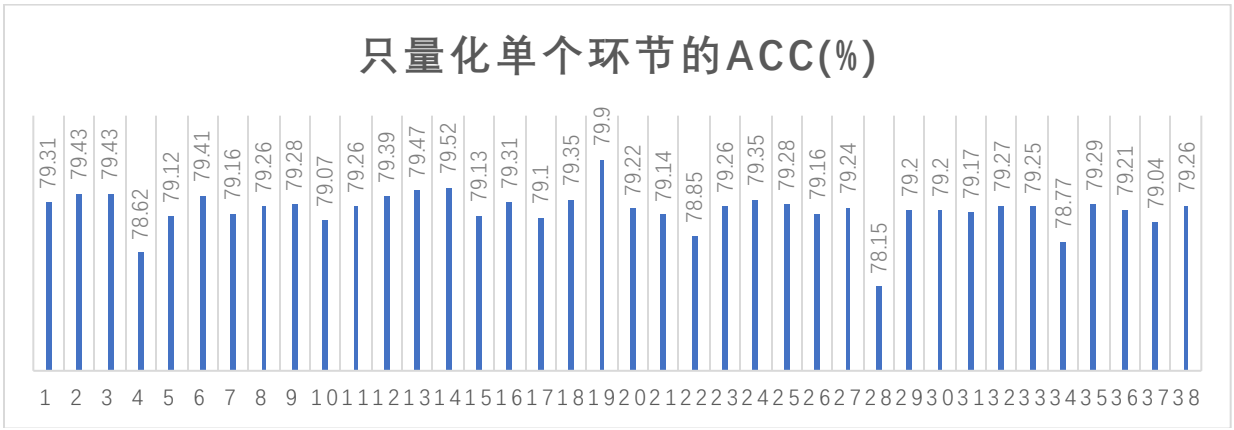
```
==> Resuming from checkpoint..
 [=============================================================>]  Step: 144ms | Tot:  100/100
Mon Jul 31 10:27:16 2023 Epoch 0, lr: 0.0001000, val loss: 99.06732, acc: 73.09000
```

即全部量化后 acc 下降6.49%。对于只量化矩阵型运算而非整个网络的 ViT，没有达到10%的准确率损失。

为了尽可能减少 acc 损失，现通过每次选择一个环节量化来判断优先选择哪些环节量化：

```
 [=============================================================>]  Step: 51ms | Tot: 100/100
Mon Jul 31 12:45:17 2023 Epoch 0, lr: 0.0001000, val loss: 76.49803, acc: 79.24000
 [=============================================================>]  Step: 52ms | Tot: 100/100
Mon Jul 31 12:45:22 2023 Epoch 0, lr: 0.0001000, val loss: 81.24986, acc: 78.15000
 [=============================================================>]  Step: 51ms | Tot: 100/100
Mon Jul 31 12:45:28 2023 Epoch 0, lr: 0.0001000, val loss: 76.50938, acc: 79.20000
 [=============================================================>]  Step: 52ms | Tot: 100/100
Mon Jul 31 12:45:33 2023 Epoch 0, lr: 0.0001000, val loss: 76.54466, acc: 79.20000
 [=============================================================>]  Step: 53ms | Tot: 100/100
Mon Jul 31 12:45:39 2023 Epoch 0, lr: 0.0001000, val loss: 77.63962, acc: 79.17000
 [=============================================================>]  Step: 54ms | Tot: 100/100
Mon Jul 31 12:45:45 2023 Epoch 0, lr: 0.0001000, val loss: 77.02307, acc: 79.27000
 [=============================================================>]  Step: 56ms | Tot: 100/100
```

得到如下条形图：

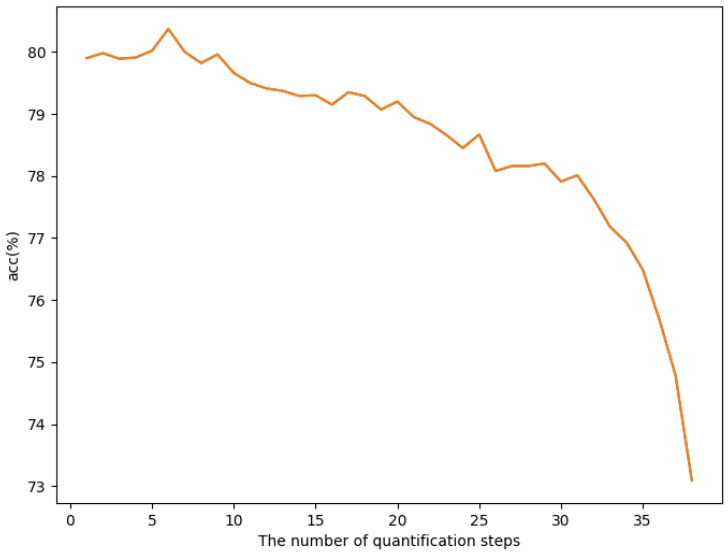


有趣的是有些环节的量化可能造成 acc 的提高。

按照 acc 由高到低的顺序，即

```
[18,13,12,1,2,5,11,17,23,0,15,34,8,24,31,7,10,22,37,32,26,19,35,28,29
,30,6,25,20,14,4,16,9,36,21,33,3,27]
```

依次开启环节的量化，得到曲线：




acc 具体为：

```
[79.9, 79.98, 79.89, 79.91, 80.02, 80.37, 80.0, 79.82, 79.96, 79.66,
79.5, 79.41, 79.37, 79.29, 79.3, 79.15, 79.35, 79.29, 79.07, 79.2,
78.95, 78.84, 78.66, 78.45, 78.67, 78.08, 78.16, 78.16, 78.2, 77.91,
78.01, 77.63, 77.18, 76.93, 76.49, 75.7, 74.79, 73.09]
```

可见该曲线并非完全的单调下降，甚至相比无量化，一度带来 acc 的提升。大体上，损失5%的 acc（到74.58%）允许除了环节 27 以外全参与量化，即最多可以$\frac{37}{38}\times 100\%=$97.4%比例的计算采用 INT4 类型；损失2%的 acc（到77.58%）允许环节 16 及以前（按单个环节 acc 排序表）全参与量化，即最多可以$\frac{32}{38}\times 100\%=84.2\%$比例的计算采用 INT4 类型。

#### 1.2.1.2 研究结论

对于本例的小型 ViT，推理准确率损失2%的情况下，最多可以84.2%的矩阵型计算采用 INT4 类型；推理准确率损失5%的情况下，最多可以97.4%的矩阵计算采用 INT4 类型；推理准确率损失10%的情况下，全部矩阵型计算可以采用 INT4 类型（且不会达到10%的损失）。

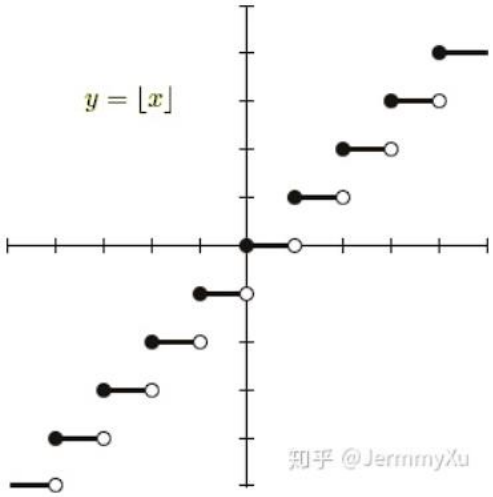
### 1.2.2 量化重训

如果矩阵型计算，全部采用 INT4，通过重训，可以达到多少精度。

#### 1.2.2.1 研究内容

量化重训，即量化感知训练（QAT），意为在模型训练阶段即使之感知量化噪声的存在，进而适应量化，通过训练取得更高的精度。

QAT 中能否直接使用 PTQ 的前向传播得到反向传播呢？答案是否定的。对于取整函数：




它的每一处求导不是未定义就是 0。在链式求导中，这会造成梯度无法反向传播下去。解决这个问题的方法是使用直通估计器（STE）来近似梯度，将取整函数的梯度近似为 1：

```
def ste_round(x):
    return torch.round(x) - x.detach() + x
```

在反向传播时，x.detach()清空 x 的梯度却保留 x 本身的值，则这种先减再加的操作使得取整操作本身的值不变，却拥有和 x 同样的梯度。

在进行 QAT 时，为了提高效率，我们不必解开方程式计算，可以对每个矩阵直接量化后再立即反量化：

```
def fake_quantize(q, quat):
    sq = ((q.max() - q.min()) / (2 ** quat - 1)).detach()
    zq = torch.round(2 ** quat - 1 - q.max() / sq).detach()
    return sq * (ste_round(q / sq + zq) - zq)
```

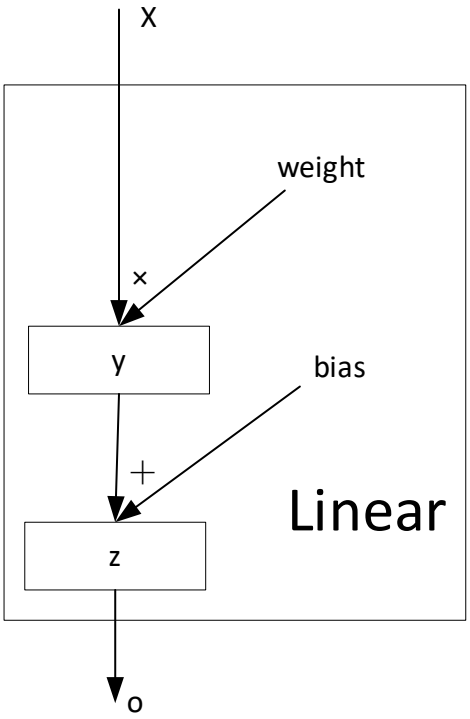
然后进行矩阵乘，理论上与解开等式结果相同。这样既引入量化噪声又使得代码简洁。

至此，我们始终使用最大-最小值算法在每一次乘法处确定 s 和 z。但是数据的分布可能是不均匀的，譬如一组数据 0,1,1,2,3,1,1000，我们把它们量化到 0~15 时，将会因为离群点 1000 的存在而使得所有数据集中在 0 和 15.除了采用计算 KL 散度等方法重新决定 s 和 z，还有一种方法来调整 s 和 z，也就是使这两个参数也处在网络的计算图中，使它们同样获得梯度：

```
def trainsz_quantize(q, sq, zq, quat):
    z = ste_round(zq)
    return sq * (torch.clamp(ste_round(q / sq) + z, 0, 2 ** quat - 1)
- z)
```

注意到公式发生了变化，首先为了便于零点 z 的训练，我们使之为浮点数，但是在前向传播时发生一次直通估计的取整。其次使用了 clamp 函数进行截断，因为调整 s 和 z 以后不能保证像使用最大-最小值算法一样每个点都在量化的范围内。这个操作实际上把离群点直接设置为最大/最小值。

最后，pytorch 中需要注意网络结构声明的机制。以一个 Linear 层为例：




在这个模块中，x 先乘以 weight（实际上是乘其转置），再加上 bias，最后得到输出 o。假如中间变量 y、z 真的存在，那么对于反向传播而言，显然 y、z、o 都需要求梯度，这样 weight、bias 才能获得梯度，进而求出自己的梯度。对于一个正常运转的网络，你将看到 y、z、o、weight、bias 这些 tensor 的 requires_grad 属性都为 True。

同时 y、z、o 的梯度用于反向传播后就没有用了，它们自己并不根据梯度更新值。只有 weight、bias 需要根据梯度更新值。以输出为根，反向传播就像一棵二叉树，其中非叶子节点（如 y、z、o）不需要更新，叶子节点（如 weight、bias）需要更新，因此必须保证所有参与训练的参数的 is_leaf 属性都为 True。本例中 y、z、o 的 is_leaf 将为 False，weight、bias 的 is_leaf 将为 True。

使用 `nn.Parameter` 函数声明的类的实例变量可以使得 requires_grad 和 is_leaf 均为 True，参数可以得到更新。此外，经实验验证，使用诸如 `self.mlp = nn.Linear(dim, hidden_dim)` 声明一个层，在前向传播中取出一个层所拥有的变量进行计算**而非**直接调用诸如 `x = mlp(x)`，即使变量经检查其 requires_grad 和 is_leaf 都为 True，它也获得不了梯度并且不能更新。因此对于原来的网络，读入 checkpoint 后，需要增加定义，如：

```
def trans(self):
    self.sw1, self.zw1 = sz_init(self.net[0].weight.data, self.quat)
    self.sx1 = nn.Parameter(torch.tensor(0.).to('cuda'))
    self.zx1 = nn.Parameter(torch.tensor(0.).to('cuda'))
    self.sw2, self.zw2 = sz_init(self.net[3].weight.data, self.quat)
    self.sx2 = nn.Parameter(torch.tensor(0.).to('cuda'))
    self.zx2 = nn.Parameter(torch.tensor(0.).to('cuda'))
    self.ww1 = nn.Parameter(self.net[0].weight.data.clone())
    self.ww2 = nn.Parameter(self.net[3].weight.data.clone())
    self.bb1 = nn.Parameter(self.net[0].bias.data.clone())
    self.bb2 = nn.Parameter(self.net[3].bias.data.clone())
```

需要删除无用的层，如：

```
def delet(self):
    del self.net[0]
    del self.net[2]
```

再保存即为一个可用于新的网络定义的 checkpoint。

在重新定义的网络中，对于 Linear 层：

```
if self.sx1 <= 0:
    self.sx1, self.zx1 = sz_init(x, self.quat)#避免缩放因子为 0 或负数造成的错误

x = F.linear(input=trainsz_quantize(x, self.sx1, self.zx1, self.quat),weight=trainsz_quantize(self.ww1, self.sw1, self.zw1,self.quat),bias=self.bb1)
```

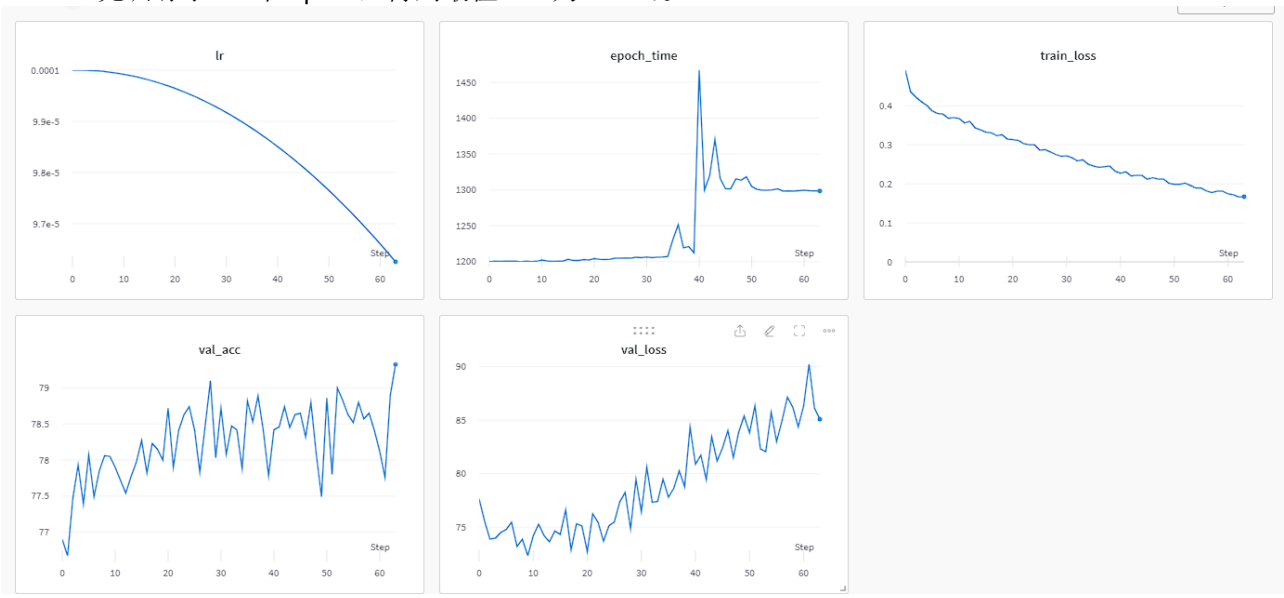
对于纯矩阵乘：

```
if self.sa <= 0:
    self.sa, self.za = sz_init(a, self.quat)

if self.sv <= 0:
    self.sv, self.zv = sz_init(v, self.quat)

out = torch.matmul(trainsz_quantize(a,self.sa,self.za, self.quat), trainsz_quantize(v,self.sv,self.zv, self.quat))
```

现在我们可以进行训练了。

训练结果：

先训练了 63 个 epoch，得到最佳 acc 为79.33%：




再以更低的学习率训练，在第 15 个 epoch 处达到更高的 acc 为 79.9%：


### 1.2.2.2 研究结论

如果矩阵型计算，全部采用 INT4，对于一个原本精度为 79.54%的全精度网络，量化重训后可以达到 79.9%的精度。

### 1.2.3 量化层的选择

如果不重训，用什么方法判断该层适合低精度运算。

### 1.2.3.1 研究内容与结论

如 3.2.1 的研究所示，通过每次只选择一处矩阵型运算进行量化推理，可以得知哪些层量化后精度损失小，进而得知哪些层适合低精度运算。本例中，环节 18,13,12,1,2,5,11,17,23 就格外适合低精度运算（尤其是因为它们同时开启量化时精度更高）。

但是按照单个量化精度损失从小到大的顺序依次开启量化，得到的曲线并不是单调下降的，环节之间的相互作用也可能使得精度上升。因此我们可能需要遍历所有可能的情况，比如，如果我们要选择 10 个矩阵型运算环节进行量化，可以进行$C_{38}^{10} = 472733756$次推理来选择最高 acc 的一个方案，耗时 82,071 天。或者选择单个环节测试中效果较好的前 15 个作为选择的范围，进行$C_{15}^{10} = 3003$次推理，共花费约 12.5 小时。

### 1.2.4 更低 bit 的量化

是否可以采用更低精度比如 INT2。

### 1.2.4.1 研究内容与结论

不量化重训，选择上面单个环节量化排序的前六个进行 INT2 的量化，其余环节不量化，得到 acc 为72.28%，尚未达到10%的精度损失，可以接受：

```
ifq = [0] * 38
ifq[18] = ifq[13]= ifq[12]= ifq[1]= ifq[2]= ifq[5]= 2
net.module.updat(ifq)
val_loss, acc = test(epoch)
```

```
 [==============================================================>]  Step: 63ms | Tot: 100/100
Mon Jul 31 17:05:13 2023 Epoch 0, lr: 0.0001000, val loss: 101.31642, acc: 72.28000
```

全部采用 INT2 的量化，结果接近随机：

```
ifq = [2] * 38
#ifq[18] = ifq[13]= ifq[12]= ifq[1]= ifq[2]= ifq[5]= 2
```

```
net.module.updat(ifq)
val_loss, acc = test(epoch)
```

```
==> Resuming from checkpoint..
 [==============================================================>]  Step: 187ms | Tot: 100/100
Mon Jul 31 17:10:54 2023 Epoch 0, lr: 0.0001000, val loss: 301.27392, acc: 11.40000
```

因为训练后量化结果接近随机，没必要转换模型，直接尝试从头进行量化感知训练：

```
Epoch: 0
 [==============================================================>]  Step: 5s106ms | T 98/98
 [==============================================================>]  Step: 1s487ms |  100/100
Saving..
Mon Jul 31 18:21:15 2023 Epoch 0, lr: 0.0001000, val loss: nan, acc: 10.00000
[nan]

Epoch: 1
/home/hihidhihi/miniconda3/envs/test/lib/python3.8/site-packages/torch/optim/lr_scheduler.py:139:
  warnings.warn("Detected call of `lr_scheduler.step()` before `optimizer.step()`. "
/home/hihidhihi/miniconda3/envs/test/lib/python3.8/site-packages/torch/optim/lr_scheduler.py:152:
  warnings.warn(EPOCH_DEPRECATION_WARNING, UserWarning)
 [==============================================================>]  Step: 5s102ms | T 98/98
 [==============================================================>]  Step: 1s474ms |  100/100
Mon Jul 31 18:42:35 2023 Epoch 1, lr: 0.0001000, val loss: nan, acc: 10.00000
[nan, nan]

Epoch: 2
 [==============================================================>]  Step: 5s105ms | T 98/98
 [==============================================================>]  Step: 1s474ms |  100/100
Mon Jul 31 19:02:39 2023 Epoch 2, lr: 0.0001000, val loss: nan, acc: 10.00000
[nan, nan, nan]

Epoch: 3
 [==============================================================>]  Step: 5s110ms | T 98/98
 [==============================================================>]  Step: 1s469ms |  100/100
Mon Jul 31 19:22:47 2023 Epoch 3, lr: 0.0001000, val loss: nan, acc: 10.00000
[nan, nan, nan, nan]

Epoch: 4
 [==============================================================>]  Step: 5s104ms | T 98/98
 [==============================================================>]  Step: 1s488ms |  100/100
Mon Jul 31 19:42:50 2023 Epoch 4, lr: 0.0000999, val loss: nan, acc: 10.00000
[nan, nan, nan, nan, nan]

Epoch: 5
```

可见一直为随机状态。

另外，目前正在进行由 4bit 量化逐环节向 2bit 退化的尝试，已达到所有 38 个环节 2bit，acc 达到74.5%。