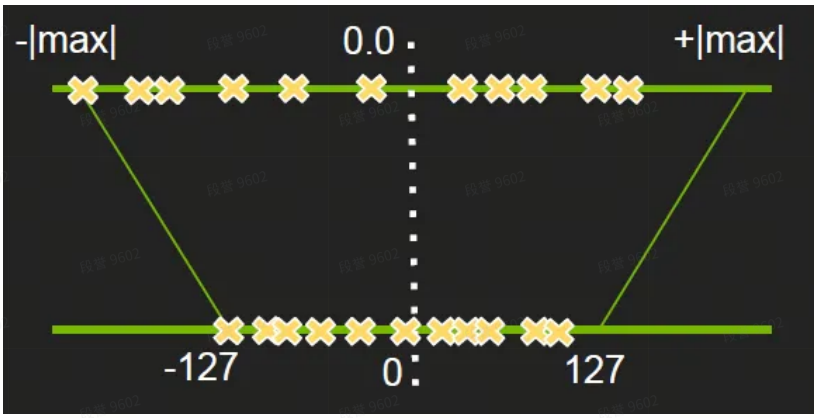
量化（1）：概念介绍

量化的原理

简单说，就是把 float 或者 half 数据映射到 int8 数据，最简单的映射如下图：



这是**对称量化**。很简单是吧！就是把你一个layer的激活值范围的给圈出来，然后按照**绝对值最大值作为阀值**，然后把这个范围按照**某种映射关系**映射到正负127的范围内来。

最简单的映射关系就是**线性映射。即：**

$$R = s * Q + z$$

这不就是初中学的线性函数 `y=kx+z` 吗，k、z 怎么求？ 带入两个点不就完了？因为是对称量化，所以 float 取0时，期望映射到int也是0，所以第一个点是（0，0），这样z不就是0，第二个点就是（|max|，127），故

```
s = |T| / 127.0  #(T = |max|)
```

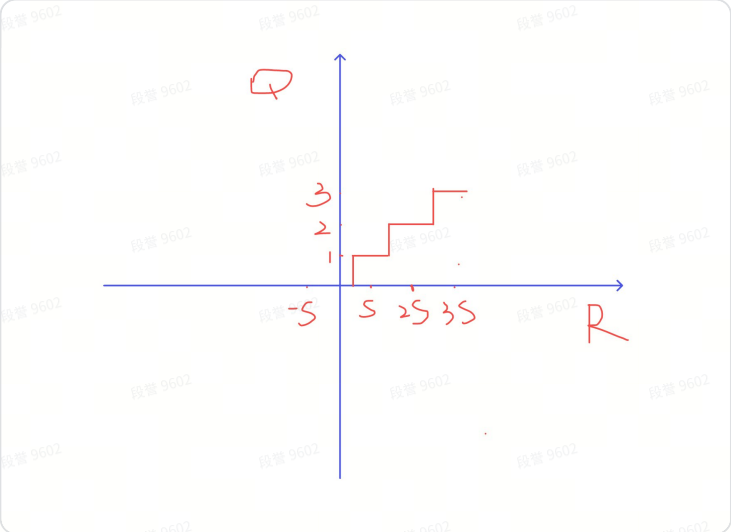
所以映射关系为：

$$\text{量化：} Q = round(\frac{R}{s})$$

$$\text{反量化：} R = s * Q$$

配合下面的图，体会下这个公式：

比如对于区间（0.5s 到 1.5 s）之间的浮点数，都量化为了 1， 反量化时，最后都是 s，你看，误差不就来了。



不妨写一段 code 来看看量化和反量化，直观的感受一下：

```python
import torch

a = torch.empty(10).uniform_(-3, 3)
print(a)

scale = a.amax() / 127.0
print(f"scale: {scale}")

# quant
a_quant = (a.clone() / scale).round().clamp(-127, 127).to(torch.int8)
print(f"a_quant: {a_quant}")

# dequant
a = a_quant * scale
print(f"a: {a}")
```

运行结果如下：

```
tensor([-1.6746,  1.2090,  1.8908, -0.5493, -2.8863, -1.3532,  2.5296,
0.3700,
         1.6904, -2.4225])
scale: 0.01991778053343296
a_quant: tensor([ -84,   61,   95,  -28, -127,  -68,  127,   19,   85, -122],
       dtype=torch.int8)
a: tensor([-1.6731,  1.2150,  1.8922, -0.5577, -2.5296, -1.3544,  2.5296,
0.3784,
         1.6930, -2.4300])
```

直观上看，反量化回去的精度还不错。

那上述的这种**线性对称**的量化方式，它的精度跟什么有关呢？（从公式上看，最直观的误差就是round， 四舍五入的误差。）

1. **数据的分布：**

上述demo中，randn 生成的是一个 -3 到 3 之间均匀的随机分布，假如数据分布严重不均匀呢，demo 如下：

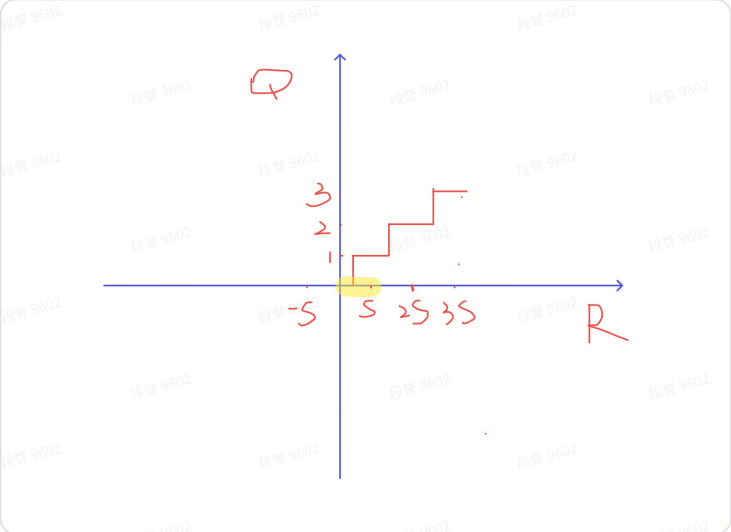
```
import torch
# a = torch.empty(10).uniform_(-3, 3)
a = torch.tensor([1, 5.89, 3.45, 1.66, 2.0, -0.99, -3.4, 1.9, 2.88, 999])
print(a)

# 其他部分和第一个demo 相同
```

如demo中所示，前 9 个数据分布在正负 6 以内， 而最后一个数值却为 999， 按照上面的量化方式，阈值 T = 999.

```
tensor([ 1.0000e+00,  5.8900e+00,  3.4500e+00,  1.6600e+00,  2.0000e+00,
        -9.9000e-01, -3.4000e+00,  1.9000e+00,  2.8800e+00,  9.9900e+02])
scale: 7.8661417961120605
a_quant: tensor([  0,   1,   0,   0,   0,   0,   0,   0,   0, 127],
dtype=torch.int8)
a: tensor([  0.0000,   7.8661,   0.0000,   0.0000,   0.0000,   0.0000,
0.0000,
          0.0000,   0.0000, 999.0000])
```

显然，反量化回去的误差要大多了。原因也非常简单，从结果上，也可以看出，前9个数对应的量化int8 数，要么为0， 要么为1。因为 T 值很大， 导致 scale 值也很大。

反量化回去的数值只能是 scale 的倍数（-127s, ..., 0, s, 2s, ... , 127s），如果int8 为 0了，反量化的结果也就是 0 了，画个数轴看看就知道了：



第二个 demo 中 scale 值为 7.8661417961120605， 前面9 个数都落在了 (0, s)区间内， 除以这个值，根据四舍五入，只有一个1，其他8个0，然后反量化回来，要么是 0， 要么就是 s 了。

这是一个很极端的例子，，由于正负分布很不均匀，如果按照对称最大值映射（原意是为了尽可能多地保留原信息）的话，那么+max那边有一块区域就浪费了，也就是说scale到int8后，int8的动态范围就更小了（就像上面demo中主要只有 0 和 1），举个极限的例子就是量化后原本int8的动态范围只剩1bit了（就是正的样本没有，负的全部扎堆在一个很小的值附近）。

从分布上来说， 这种线性映射的量化方式是 **“均匀分布量化”** （ integer uniform quantization ）。所以原来的分布越均匀就越好。

2. 饱和截取

从1 中的demo，很自然的就会想到，10个数中，只有一个是999，这完全可以看成是一个特殊的点（outliers），他对阈值 T 的影响太大了，我们得把它去掉。

```
# a = torch.empty(10).uniform_(-3, 3)
a = torch.tensor([1, 5.89, 3.45, 1.66, 2.0, -0.99, -3.4, 1.9, 2.88, -999])
print(a)

# scale = a.amax() / 127.0
scale = 5.89 / 127.0
print(f"scale: {scale}")
```

结果如下：

```
tensor([ 1.0000e+00,  5.8900e+00,  3.4500e+00,  1.6600e+00,  2.0000e+00,
        -9.9000e-01, -3.4000e+00,  1.9000e+00,  2.8800e+00, -9.9900e+02])
scale: 0.04637795275590551
a_quant: tensor([ 22, 127,  74,  36,  43, -21, -73,  41,  62, -127],
dtype=torch.int8)
```

```
5   a: tensor([ 1.0203,  5.8900,  3.4320,  1.6696,  1.9943, -0.9739, -3.3856,
    1.9015,
6            2.8754,  -5.8900])
```

可以看出，scale 数值小很多，前9个数在 R 轴上分布更广，量化值 a_quant 不再都是 0 和 1 了。但是**最后一个值** -999 的误差就很大了，它量化值只能是 -127（截断）， 所以反量化回去就是 `-127 * s = -5.8900`

正如这个图，如果有些值超过了阈值 (-|T|, |T|), 量化时，被截断量化为 -127 或者 127。那么这些被截断的数值，误差就会很大。

像上图这样，先找一个阀值T，然后低于最低阀值的就全部都饱和映射到-127上，如上图的左边的三个红色的点就是这么处理的。

假如一个数据集，一半分布在0 到 10 之间，一半分布在 1000 左右，像这种分布的数据，普通的线性量化肯定是没办法的。如果说绝大部份数据都均匀分布在0 到 10 内，只有个别数据在 1000 左右，像这种情况，就可以把这些个别数据当作异常值，考虑阈值 T 的时候过滤掉。

总而言之，非饱和截取的问题是当数据分布极不均匀的时候，有很多动态范围是被浪费的，而饱和截取就是弥补这个问题的。当你数据分布很不均匀的时候，如图左边比右边多，那么我把原始信息在影射之前就截断一部分，然后构成对称且分布良好的截断信息，再把这个信息映射到int8上去，那么就不会有动态范围资源被浪费了。

（这也就是一个很自然的思路对吧～把无关的高频细节给去掉，从而获取性能上的好处！网络图像压缩技术不就是这么整的么！PCA主成分、傅立叶分解的思路不都是这样的么！抓住事物的主要矛盾，忽略细节，从而提高整体性能！）

如何描述误差

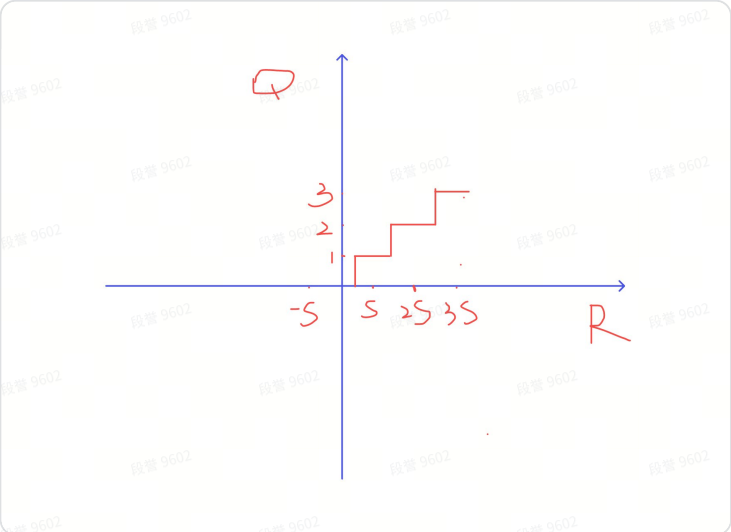
实际数据，肯定不是最理想的均匀分布，因此，如何选取阈值 T 就成了误差的关键。

那么， 如何定量的描述 T 对误差的影响呢（上面的 demo 只是我们定性的感受一下）。

这时候，就要掏出一个数学工具了。

> 说句题外话，数学只是描述一个问题的工具，在使用这个工具之前，一定要透彻明白你的需求，你想要用这个数学工具表达什么，现在我们的需求就是，T 到底是如何影响误差的，误差到底是多大，如何描述误差。

我们先想想这个量化的问题，本质是使用 8位编码来替代 16位或者32为编码来表达一个 tensor，我们继续掏出这个图：



假如说哈，我们的阈值选择合理，且数据分布均衡，这个 tensor 一共有255个 float 数，且每一个数都在不同的区间，比如 （0，s), (s, 2s),(2s, 3s) 等等每个区间都有一个数 ，因此量化后的数正好也覆盖了 -127 到 127。

可以看出，选择 8 位编码，或者16位编码，都能够对这255个数进行**区分，** 也可以说成，为了区分这255个数值信息，我们只需要最少的8位即可编码。

最直观的评价误差，就是得到量化的数值后，再反量化回去，如果数值和之前的绝对误差越小，不就说明精度越高。

或者至少原本不同的两个float 数，量化后也能够区分出来，即量化为了两个不同的int8数。

假如说原本的数据分布是A， 量化后的int8数据分布为B， 反量化回去的数据分布为C。

如果分布C 和 A 是一模一样的，那就没有误差了，也就是 A C两个分布的接近程度。

所以，如何**描述两个分布的相似程度**呢?

> 描述两个分布的相似程度有很多种数学方法，感兴趣的可以自行百度并深入研究

NVIDIA选择的是KL-divergence （KL散度），KL散度的物理意义是：一个分布（B）相比另一个分布（A）的信息损失（information loss）。

对于 KL 散度的计算方式和理解，这篇博客有非常好的解释和demo：

Kullback-Leibler Divergence Explained

一文直观理解KL散度

KL 散度在形式上的定义如下：

$$D_{KL}(p||q) = \sum_{i=1}^{n} p(x_i) log \frac{p(x_i)}{q(x_i)}$$

其中 q(x) 是近似分布，p(x) 是我们想要用 q(x) 匹配的真实分布。直观地说，这衡量的是给定任意分布偏离真实分布的程度。如果两个分布完全匹配，那么：

$$D_{KL}(p||q) = 0$$

，否则它的取值应该是在 0 到无穷大（inf）之间。**KL 散度越小，真实分布与近似分布之间的匹配就越好**。

KL 散度的直观解释

让我们看看 KL 散度各个部分的含义。首先看看

$$log(\frac{p(x_i)}{q(x_i)})$$

项。如果 q(x_i) 大于 p(x_i) 会怎样呢？此时这个项的值为负，因为小于 1 的值的对数为负。另一方面，如果 q(x_i) 总是小于 p(x_i)，那么该项的值为正。如果 p(x_i)=q(x_i) 则该项的值为 0。然后，为了使这个值为期望值，你要用 p(x_i) 来给这个对数项加权。也就是说，**p(x_i) 有更高概率的匹配区域比低 p(x_i) 概率的匹配区域更加重要。**
直观而言，优先正确匹配近似分布中真正高可能性的事件是有实际价值的。从数学上讲，这能让你自动忽略落在真实分布的支集（支集（support）是指分布使用的 X 轴的全长度）之外的分布区域。另外，这还能避免计算 log(0) 的情况——如果你试图计算落在真实分布的支集之外的任意区域的这个对数项，就可能出现这种情况。

Ok, 纸上得来终觉浅，我们写一点 code 计算下 KL 散度，下面是一个计算两个tensor kl 散度的demo

```
import torch
import torch.nn.functional as F

a = torch.tensor([1.0, 2.0]).softmax(dim=-1)
```

```
5
6   b = torch.tensor([1.0, 2.0]).softmax(dim=-1)
7
8   kl_sum = F.kl_div(a.log(), b, reduction='sum')
9   print(kl_sum)
```

其中 kl_div 接收三个参数，第一个为预测分布，第二个为真实分布，第三个为reduction。(其实还有其他参数，只是基本用不到)

这里有一些细节需要注意，第一个参数与第二个参数都要进行softmax(dim=-1)，目的是使两个概率分布的所有值之和都为1，若不进行此操作，如果x或y概率分布所有值的和大于1，则可能会使计算的KL为负数。softmax接收一个参数dim，dim=-1表示在最后一维进行softmax操作。除此之外，第一个参数还要进行log()操作（至于为什么，大概是为了方便pytorch的代码组织，pytorch定义的损失函数都调用handle_torch_function函数，方便权重控制等），才能得到正确结果。

第三个参数reduction有三种取值，为 none 时，各点的损失单独计算，输出损失与输入（x）形状相同；为 mean 时，输出为所有损失的平均值；为 sum 时，输出为所有损失的总和。

Ok， 回到最开始的 demo， 一个float矩阵，一个量化后然后反量化回去的矩阵，他们的 KL 散度是多少呢。请看下面的 demo：

```
1   import torch
2   import torch.nn.functional as F
3
4   def compute_histogram(tensor, bins=10, range=(-5, 5), epsilon=1e-5):
5       origin_histogram = torch.histc(tensor, bins=bins, min=range[0],
    max=range[1])
6
7       # Add a small epsilon to avoid division by zero
8       histogram =  origin_histogram + epsilon
9
10      # Normalize the histogram to obtain probabilities with a sum of 1
11      histogram /= histogram.sum()
12
13      return origin_histogram, norm_histogram
14
15  def kl_divergence(p, q, epsilon=1e-5):
16      p = torch.clamp(p, min=epsilon)  # Clip probabilities to avoid log(0)
17      q = torch.clamp(q, min=epsilon)
18
19      # Normalize the distributions to ensure the sum is 1
20      p /= p.sum()
21      q /= q.sum()
22
```

```
        return torch.sum(p * (torch.log(p) - torch.log(q)))

a = torch.empty(10).uniform_(-3, 3)
print(f"a: {a}")

T = a.amax()
scale = T / 127.0

# quant
a_quant = (a.clone() / scale).round().clamp(-127, 127).to(torch.int8)
print(f"a_quant: {a_quant}")

# dequant
a_2 = a_quant * scale
print(f"a_2: {a_2}")

# 计算概率分布(直方图)
o_p, p = compute_histogram(a) # 真实分布 p
o_q, q = compute_histogram(a_2) # 反量化的分布 q

print(f"真实分布 op: {o_p}")
print(f"量化分布 pq: {o_q}")
print(f"归一化 p: {p}")
print(f"归一化 q: {o_q}")

# 计算 KL 散度
kl_sum = kl_divergence(p, q)
print(f"kl 散度: {kl_sum}")
```

这里要说的一点是 bins 的选择，为了更好的区分两个数值，bins当然是大一点好，NVIDAIA给的是2048个bin（maxnet代码里面给的是8000bins），上述demo用的10个bins，读者可以自行调整演示观察。

（到这里就可以感受到了数学之美了）

如何寻找 scale

ok，到了这里，我们的目标很明确了：

寻找一个能让 KL 散度最小的 scale

最简单最暴力的方式不就是 for 循环遍历很多候选的 scale，比如 `for scale in range(0, max, 0.01)`

记录每个 scale 所对应的 KL 散度，然后把最小的找出来，不就 OK 了吗， nv tensorrt 中就是这么干的。