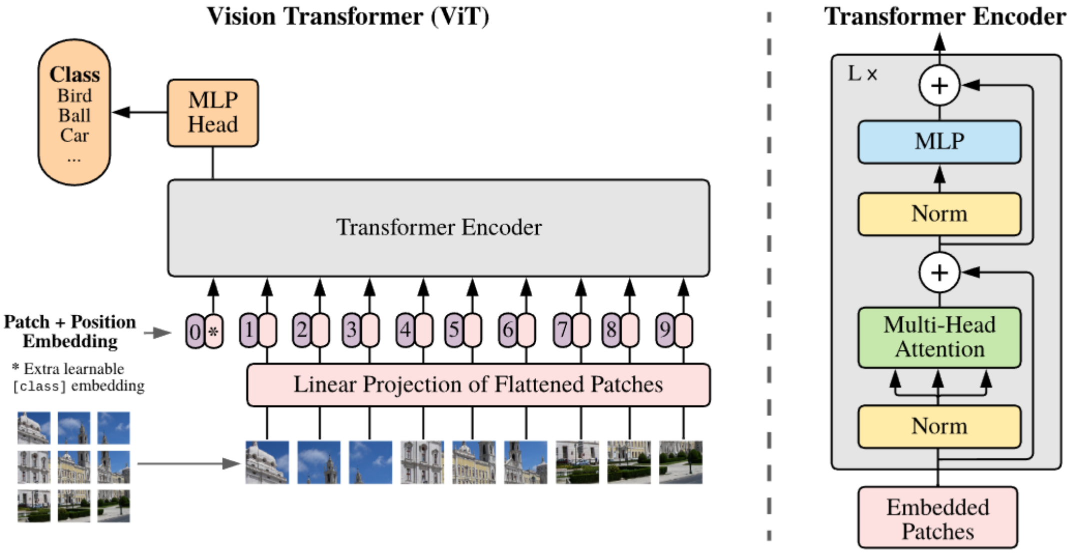
1 ViT架构



由上图可知，ViT 由 Patch Embedding, Transformer Encoder 和 MLP Head 三部分组成。具体如下:

- Patch Embedding: 一层 Conv2d 并加上位置编码, 其中位置编码在实现时作为可学习参数参与训练获得
- Transformer Encoder: 具体结构如上图，由L个 Block 组成，每个Block 则由 一个带跳连结构的 MHA 和 一个带跳连结构的 MLP 组成，同时在 MHA 和 MLP 之前需进行 LN 操作。MHA 和 MLP 主要是由矩阵乘积实现(MHA中还包含softmax操作, MLP中包含激活函数GELU操作, 这些运算是非线性的)。由于Transformer Encoder Block是ViT中的核心部分，也是量化的核心部分，因此下面对其操作进行形式化叙述。
  - 每个Block的计算过程可形式化表示如下:

$$\begin{aligned} \boldsymbol{Y}_{l-1} &= \operatorname{MHA}\left(\operatorname{LayerNorm}\left(\boldsymbol{X}_{l-1}\right)\right)+\boldsymbol{X}_{l-1} \\ \boldsymbol{X}_{l} &= \operatorname{MLP}\left(\operatorname{LayerNorm}\left(\boldsymbol{Y}_{l-1}\right)\right)+\boldsymbol{Y}_{l-1} \end{aligned}$$

    其中$\boldsymbol{X}_{l-1}$是上一层Block的输出(或者Patch Embedding的输出), $\boldsymbol{Y}_{l-1}$是MHA的输出，同时作为MLP的输入，$\boldsymbol{X}_{l}$则是MLP的输出，也是该Block的输出。

  - MHA可形式化表示如下:

$$\begin{aligned} \left[\boldsymbol{Q}_i, \boldsymbol{K}_i, \boldsymbol{V}_i\right] &= \boldsymbol{X}' \boldsymbol{W}^{qkv}+\boldsymbol{b}^{qkv} \quad i=1,2,\cdots,h \\ \operatorname{Attn}_i &= \operatorname{Softmax}\left(\frac{\boldsymbol{Q}_i \cdot \boldsymbol{K}_i^T}{\sqrt{D_h}}\right) \boldsymbol{V}_i \\ \operatorname{MHA}\left(\boldsymbol{X}'\right) &= \left[\operatorname{Attn}_1, \operatorname{Attn}_2, \ldots, \operatorname{Attn}_h\right] \boldsymbol{W}^o+\boldsymbol{b}^o \end{aligned}$$

    对MHA的输入进行投影获得对应的$\boldsymbol{Q}$、$\boldsymbol{K}$、$\boldsymbol{V}$,同时在嵌入维度上进行分组获得各个head的$\boldsymbol{Q}$、$\boldsymbol{K}$、$\boldsymbol{V}$,也即$\boldsymbol{Q}_i$, $\boldsymbol{K}_i$, $\boldsymbol{V}_i$, 接着各个head对应的$\boldsymbol{Q}$、$\boldsymbol{K}$运算并经过softmax运算获得各个head的score，将score和$\boldsymbol{V}$相乘获得各个head的$\boldsymbol{Attn}$,最后将各个head的$\boldsymbol{Attn}$拼接并做投影获得输出结果。

  - MLP可形式化表示如下:

$$\operatorname{MLP}\left(\boldsymbol{Y}'\right)=\operatorname{GELU}\left(\boldsymbol{Y}' \boldsymbol{W}^1+\boldsymbol{b}^1\right) \boldsymbol{W}^2+\boldsymbol{b}^2$$

    MLP主要对输入进行两层线性层运算，在两层之间使用GELU作为激活函数。

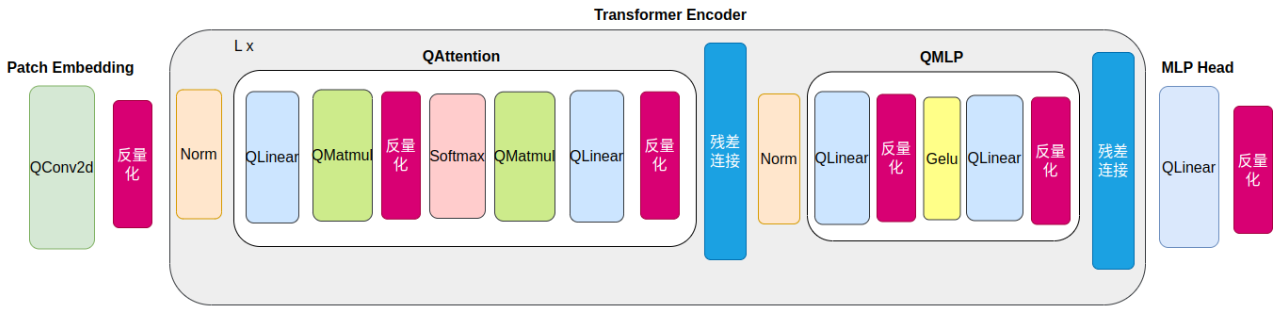
- MLP Head: 一层线性层, 在ViT原论文中因为面向分类任务是用于获得属于各个类别的概率。

2 量化方案

关于模型量化方案，需要考虑的是对哪些层进行量化，量化位宽的选择，量化方式的确定。下面依次进行介绍。

2.1 量化层

如前面展示的ViT架构中的各种组件，其中有**矩阵运算作为核心算子**的MHA，MLP中的矩阵运算部分都可以进行量化，但是对于模型中的softmax, gelu，LN(也就是Norm)不进行量化操作，需要注意的是残差连接也不进行量化。 在量化和非量化部分之间需要插入量化节点和非量化节点。具体见下图:



如图所示，在非线性层和残差连接以及最后的输出前进行了**反量化操作**,在`QConv2d`,`QLinear`和`QMatMul`的输入进行量化。

2.2 量化位宽

为较好的保持模型的效果同时支持ViT在FPGA上的高效推理，这里对量化的权重和激活值采用常用的`uint8`形式。具体来说：对于线性层(**矩阵运算作为核心算子**)的网络层对其输入进行`uint8`量化，其输出由于一般进行汇总操作，为防止溢出，采用`int32`；对于非线性层(softmax 和 gelu) 由于无法直接使用量化后的数据进行推理，采用定点数形式(**具体在FPGA上的小数位宽仍需分析这些层的输入输出分布确定**)，也就是说，在前面层量化得到的整数(**一般是**`int32`**,因为前面层的输出需要以**`int32`**存储**)需要进行反量化后输入到这些非线性层。

2.3 量化方式

确定了量化的位宽，接下来，就需要考虑量化的方式了，而为了尽可能保持原有模型的精度，**需要考虑模型推理过程中的数据分布来确定具体的量化方式**。

量化的方法在CNN领域已经得到充分的研究，一般来说，采用`weight`进行`per-channel量化`,各层的输入采用`per-tensor`的量化，一般都使用Uniform的量化。但是同样的配置在以Transformer为基础的ViT上效果并不好，一些研究工作发现ViT推理过程中经过`LN`层后的输出具有严重的跨通道差异，因此在进入下一层之前采用`per-channel`量化可以保持模型较高的精度。然而，对这些`激活值`采用`per-channel量化`会给推理过程带来较大的负担。

在这里，决定采用 [RepQ-ViT] 论文中的量化方案，其核心思想是将`LN`层输出的跨通道差异通过仿射变换转移到下一层的`weight`中，而`weight`本身就需要进行`per-channel`的量化, 而且这些调整是在软件端就可完成的，也不会带来`FPGA`推理上的负担，这样就可以实现`激活值`的`per-tensor`量化, 这一步在论文中被称为**Scale Reparam for LayerNorm Activations**。除此之外，论文也提到经过`softmax`计算得到的`attention`分数具有幂率分布特点，因此采用`log量化`比较合适，但是使用传统的 $log2$ 量化 间距过大，因而采用 $log\sqrt{2}$ 量化 比较合适，但是 $log2$量化 在硬件实现上很高效，直接使用位运算即可，因此，论文根据 $log2$ 函数和 $log\sqrt{2}$函数的关系，通过分离出一个系数，使用 $log2$ 完成 $log\sqrt{2}$,以达到硬件的高效实现, 这一步在论文中被称为**Scale Reparam for Softmax Activations**。具体的原理以下分别介绍。

2.3.1 Scale Reparam for LayerNorm Activations

$$\operatorname{LayerNorm}\left(\boldsymbol{X}_{n,:}\right)=\frac{\boldsymbol{X}_{n,:}-\mathrm{E}\left[\boldsymbol{X}_{n,:}\right]}{\sqrt{\operatorname{Var}\left[\boldsymbol{X}_{n,:}\right]+\epsilon}} \odot \boldsymbol{\gamma}+\boldsymbol{\beta}$$

其中 $n=1,2,\cdots,N, \mathrm{E}\left[\boldsymbol{X}_{n,:}\right]$ 和 $\operatorname{Var}\left[\boldsymbol{X}_{n,:}\right]$ 分别是均值和方差, 而 $\gamma \in \mathbb{R}^D$ 和 $\boldsymbol{\beta} \in \mathbb{R}^D$ 则是代表线性仿射变换的参数因子的行向量。另外, $\odot$ 代表 Hadamard product。

正如前面所说，论文中提到对于经过LN后的激活值，由于其具有严重的通道间差异，在量化时采用`per-channel`量化可在一定程度解决这个问题，但是对激活值进行`per-channel`的量化在硬件上效率不高。论文中采用首先进行`per-channel`的量化，接着将其转化为`per-tensor`的量化。

对于这些激活值的量化采用Uniform的量化，具体原理如下:

$$\begin{aligned}
& \text{Quant: } \boldsymbol{x}^{(\mathbb{Z})}=\operatorname{clip}\left(\left\lfloor\frac{\boldsymbol{x}}{s}\right\rceil+z, 0,2^b-1\right) \\
& \text{DeQuant : } \hat{\boldsymbol{x}}=s\left(\boldsymbol{x}^{(\mathbb{Z})}-z\right) \approx \boldsymbol{x} \\
& s=\frac{\max (\boldsymbol{x})-\min (\boldsymbol{x})}{2^b-1}, \quad z=\left\lfloor-\frac{\min (\boldsymbol{x})}{s}\right\rceil
\end{aligned}$$

以上给出了对一个tensor $\boldsymbol{x}$进行uniform 的量化和反量化的数学原理，对于LayerNorm Activations，如果是`per-tensor`的量化则$\boldsymbol{x}$是LN的整个输出, 而`per-channel`则是输出按各个通道划分出的tensor。

先进行`per-channel`的量化，会得到对应于各个通道的缩放因子$\boldsymbol{s}$和零点$\boldsymbol{z}$, 而`per-tensor`则只有一个缩放因子s和零点z, 这里一个自然也是论文中的方法是, 设置其量化参数为：`scale`为各通道`scale`的均值, `zero-point`为各通道`zero-point`的均值。由此逆推，如何对LN输出的激活值进行仿射变换，使得其分布能符合这个量化参数，也就是说各个通道上的量化参数都是一样的，因此也就等价变成了`per-tensor`的量化， 这里推导仿射变换的参数如下:

$$\begin{aligned}
& \tilde{\boldsymbol{z}}=\boldsymbol{z}-\boldsymbol{r}_2=\left\lfloor-\frac{\left[\min \left(\boldsymbol{X}_{:, d}^{\prime}\right)\right]_{1 \leq d \leq D}+\boldsymbol{s} \odot \boldsymbol{r}_2}{\boldsymbol{s}}\right\rceil \\
& \tilde{\boldsymbol{s}}=\frac{\boldsymbol{s}}{\boldsymbol{r}_1}=\frac{\left[\max \left(\boldsymbol{X}_{:, d}^{\prime}\right)-\min \left(\boldsymbol{X}_{:, d}^{\prime}\right)\right]_{1 \leq d \leq D} / \boldsymbol{r}_1}{2^b-1}
\end{aligned}$$

其中$\tilde{\boldsymbol{z}}$和$\tilde{\boldsymbol{s}}$分别代表$\boldsymbol{z}$和$\boldsymbol{s}$的均值。由于LN中也有进行仿射变换，因此可以将这里的$\boldsymbol{r1}$和$\boldsymbol{r2}$按如下公式合并到LN中的$\boldsymbol{\beta}$和$\boldsymbol{\gamma}$中。

$$\widetilde{\boldsymbol{\beta}}=\frac{\boldsymbol{\beta}+\boldsymbol{s} \odot \boldsymbol{r}_2}{\boldsymbol{r}_1}, \quad \tilde{\boldsymbol{\gamma}}=\frac{\boldsymbol{\gamma}}{\boldsymbol{r}_1}$$

当然，经过这样的仿射变换， LN输出激活值的分布发生了偏移，论文中采取的操作是对下一层的weight和bias进行仿射变换以抵消影响，因为weight本身就是进行`per-channel`量化的，因此量化方面不受影响。

$$\begin{aligned}
\boldsymbol{X}_{n,:}^{\prime} \boldsymbol{W}_{:, j}^{q k v}+\boldsymbol{b}_j^{q k v}= & \frac{\boldsymbol{X}_{n,:}^{\prime}+\boldsymbol{s} \odot \boldsymbol{r}_2}{\boldsymbol{r}_1}\left(\boldsymbol{r}_1 \odot \boldsymbol{W}_{:, j}^{q k v}\right) \\
& +\left(\boldsymbol{b}_j^{q k v}-(\boldsymbol{s} \odot \boldsymbol{r}_2) \boldsymbol{W}_{:, j}^{q k v}\right)
\end{aligned}$$

$$\begin{aligned}
\widetilde{\boldsymbol{W}}_{:, j}^{q k v} & =\boldsymbol{r}_1 \odot \boldsymbol{W}_{:, j}^{q k v} \\
\widetilde{\boldsymbol{b}}_j^{q k v} & =\boldsymbol{b}_j^{q k v}-(\boldsymbol{s} \odot \boldsymbol{r}_2) \boldsymbol{W}_{:, j}^{q k v}
\end{aligned}$$

以上给出了对下一层的weight和bias进行的仿射变换的参数推导。

2.3.2 Scale Reparam for Softmax Activations

由于softmax后的激活值具有明显的幂律分布特点，使用均匀分布对它效果并不好，这里对它采用log形式的量化，量化原理如下:

$$
\begin{aligned}
&\text{Quant: } \boldsymbol{x}^{(\mathbb{Z})}=\operatorname{clip}\left(\left\lfloor-\log_2 \frac{\boldsymbol{x}}{s}\right\rceil, 0, 2^b-1\right) \\
&\text{DeQuant }: \hat{\boldsymbol{x}}=s \cdot 2^{-\boldsymbol{x}^{(Z)}} \approx \boldsymbol{x}
\end{aligned}
$$

值得注意的是，这里的s实现时一般使用最大值或接近最大的百分位点对应的值(如95%位点对应的值)

论文的研究工作发现，使用$log2$在实践中效果并不好，它提供的quantization resolution不够，量化点之间的间隔较大，而使用$log\sqrt{2}$则能更准确的描述数据分布，但是$log\sqrt{2}$无法在硬件上高效的实现，仿照LN激活值的Scale Reparam思想，对Softmax的激活值也进行Scale Reparam，使其转换成$log2$量化，由于数字在硬件上都以二进制存储，对于$\lfloor\log_2\rceil$操作可以直接使用其最高位表示获得(如$\lfloor\log_2 34\rceil=\lfloor\log_2(32+2)\rceil=\lfloor\log_2(100010_2)\rceil=log32=5$)。

由于$log\sqrt{2}$量化和$log2$量化有如下关系:

$$
\begin{aligned}
\boldsymbol{A}^{(\mathbb{Z})} & =\operatorname{clip}\left(\left\lfloor-\log_{\sqrt{2}} \frac{\boldsymbol{A}}{s}\right\rceil, 0, 2^b-1\right) \\
& =\operatorname{clip}\left(\left\lfloor-2 \log_2 \frac{\boldsymbol{A}}{s}\right\rceil, 0, 2^b-1\right)
\end{aligned}
$$

$$
\begin{aligned}
\widehat{\boldsymbol{A}} & =s \cdot \sqrt{2}^{-\boldsymbol{A}^{(\mathbb{Z})}}=s \cdot 2^{-\frac{\boldsymbol{A}^{(\mathbb{Z})}}{2}} \\
& =\begin{cases}
s \cdot 2^{-\frac{A^{(\mathbb{Z})}}{2}} & \boldsymbol{A}^{(\mathbb{Z})}=2k, k \in \mathbb{Z} \\
s \cdot 2^{-\frac{\left(A^{(\mathbb{Z})}\right)+1}{2}} \cdot \sqrt{2} & \boldsymbol{A}^{(\mathbb{Z})}=2k+1, k \in \mathbb{Z}
\end{cases} \\
& =s \cdot 2^{\left\lfloor\frac{\boldsymbol{A}^{(\mathbb{Z})}}{2}\right\rfloor} \cdot\left[\mathbb{1}\left(\boldsymbol{A}^{(\mathbb{Z})}\right) \cdot(\sqrt{2}-1)+1\right]
\end{aligned}
$$

对于量化部分，乘上系数2再进行clip即可，而对于反量化则可以将$\mathbb{1}\left(\boldsymbol{A}^{(\mathbb{Z})}\right) \cdot(\sqrt{2}-1)+1$合并到系数$s$中，这个过程可以在软件端完成。

$$
\tilde{s}=s \cdot\left[\mathbb{1}\left(\boldsymbol{A}^{(\mathbb{Z})}\right) \cdot(\sqrt{2}-1)+1\right]
$$

3 量化细节(软件模拟)

具体的量化只涉及以矩阵运算作为核心算子的线性层，以下先分析`Linear`,`Matmul`和`Conv2d`的量化，然后分析量化层与非量化层之间的反量化操作。

3.1 Linear

weight采用`per-channel`的uniform量化,对于input采用`per-tensor`的uniform量化,bias和输出一样采用`int32`保存, 缩放因子采用 input 和 weight 的缩放因子乘积，量化的偏移为0。具体代码如下(**freeze对weight和bias进行量化，这些可在推理前固化，输入量化的过程和量化推理过程在inference中体现**):

```
def freeze(self):
    weight_int = torch.round(self.weight.data / self.weight_uniform_channel_delta) + self.weight_uniform_channel_zero_point
    weight_quant = torch.clamp(weight_int, 0, 2 **self.weight_uniform_channel_nbits - 1)
    self.weight.data = weight_quant - self.weight_uniform_channel_zero_point

    bias_int = torch.round(self.bias.data / (self.weight_uniform_channel_delta.reshape(-1)*self.input_uniform_delta.reshape(-1)))
    bias_quant = torch.clamp(bias_int, -2**31, 2**31-1)
    self.bias.data = bias_quant
```

```python
def inference(self, x, delta=None):
    if delta is None:
        x_int = torch.round(x / self.input_uniform_delta) + self.input_uniform_zero_point
        x_quant = torch.clamp(x_int, 0, 2 **self.input_uniform_nbits - 1) - self.input_uniform_zero_point

        out = F.linear(x_quant, weight=self.weight, bias=self.bias)

        return out, self.weight_uniform_channel_delta.reshape(-1) *self.input_uniform_delta.reshape(-1)
    else:
        x_int = torch.round(x * delta/ self.input_uniform_delta) + self.input_uniform_zero_point
        x_quant = torch.clamp(x_int, 0, 2 **self.input_uniform_nbits - 1) - self.input_uniform_zero_point

        out = F.linear(x_quant, weight=self.weight, bias=self.bias)

        return out, self.weight_uniform_channel_delta.reshape(-1) *self.input_uniform_delta.reshape(-1)
```

3.2 Matmul

Matmul层出现在Attention计算中的$attn = Matmul(K, Q)$ 和 $O = Matmul(attn, V)$，对于 attn 计算的Matmul，其输入都采用`uint8`的uniform量化，而对于 O 计算的Matmul，对attn 输入采用 $log\sqrt{2}$ 量化，对V输入采用`uint8`的uniform量化。(**freeze对weight和bias进行量化，这些可在推理前固化，输入量化的过程和量化推理过程在inference中体现**):

```python
def freeze(self):
    pass
```

```python
def inference(self, A, B, delta_A, delta_B):
    if self.A_log_quant:
        B_int = torch.round(B*delta_B/self.B_uniform_delta) + self.B_uniform_zero_point
        B_quant = torch.clamp(B_int, 0, 2 **self.B_uniform_nbits-1) - self.B_uniform_zero_point

        out = A @ B_quant

        return out, self.B_uniform_delta
    else:
        A_int = torch.round(A*delta_A/self.A_uniform_delta) + self.A_uniform_zero_point
        A_quant = torch.clamp(A_int, 0, 2 **self.A_uniform_nbits-1) - self.A_uniform_zero_point
        B_int = torch.round(B*delta_B/self.B_uniform_delta) + self.B_uniform_zero_point
        B_quant = torch.clamp(B_int, 0, 2 **self.B_uniform_nbits-1) - self.B_uniform_zero_point

        out = A_quant @ B_quant

        return out, self.A_uniform_delta * self.B_uniform_delta
```

3.3 Conv2d

这里与Linear一致，weight采用`per-channel`的uniform量化,对于input采用`per-tensor`的uniform量化,bias和输出一样采用`int32` 保存, 缩放因子采用 input 和 weight 的缩放因子乘积，量化的偏移为0。(**freeze对weight和bias进行量化，这些可在推理前固化，输入量化的过程和量化推理过程在inference中体现**):

```python
def freeze(self):
    weight_int = torch.round(self.weight.data / self.weight_uniform_channel_delta) + self.input_uniform_zero_point
    weight_quant = torch.clamp(weight_int, 0, 2 **self.weight_uniform_channel_nbits - 1)
    self.weight.data = weight_quant - self.input_uniform_zero_point

    bias_int = torch.round(self.bias.data / (self.weight_uniform_channel_delta.reshape(-1) *self.input_uniform_delta.reshape(-1)))
    bias_quant = torch.clamp(bias_int, -2**31, 2**31-1)
    self.bias.data = bias_quant
```

```python
def inference(self, x):

    x_int = torch.round(x / self.input_uniform_delta) + self.input_uniform_zero_point
    x_quant = torch.clamp(x_int, 0, 2 **self.input_uniform_nbits - 1) - self.input_uniform_zero_point

    out = F.conv2d(
        x_quant,
        self.weight,
        self.bias,
        self.stride,
        self.padding,
        self.dilation,
        self.groups
    )

    return out, self.weight_uniform_channel_delta.reshape(-1) *self.input_uniform_delta.reshape(-1)
```

4 效果验证

使用 timm 库中开源的**ViT-tiny**模型进行验证, 具体是使用自定义的`QConv2d`,`QLinear`,`QMatmul`提取原有模型被量化的权重参数，并通过前面的`inference`实现量化推理,供高层的`QAttention`,`QMLP`,`QVisionTransformer`等调实现ViT模型的量化推理(**其中也涉及前面介绍量化推理过程中的在非线性算子前的反量化**)。使用**ImageNet-1k**数据集进行验证，由于train部分数据集过大，下载较慢，直接使用验证集(val_images.tar.gz 6.7G)中的数据进行验证。效果如下:

```
╭─  ~/Desktop/本科毕设/实践/VIT加速/qvit
╰─ python3 test_quant_inference.py
Namespace(qmodel_path='/home/jack/Desktop/本科毕设/实践/VIT加速/qvit/qvit.pt', dataset='/home/jack/Desktop/本科毕设/实践/
, num_workers=16, device='cuda', print_freq=100, seed=0)
Building dataloader ...
Loading qmodel ...
Validating ...
Test: [0/60]    Time 8.475 (8.475)      Loss 0.4342 (0.4342)    Prec@1 88.500 (88.500)  Prec@5 96.000 (96.000)
 * Prec@1 75.908 Prec@5 92.916 Time 40.995
Freezing ...
Inference ...
Test: [0/60]    Time 5.150 (5.150)      Loss 0.4768 (0.4768)    Prec@1 87.500 (87.500)  Prec@5 97.000 (97.000)
 * Prec@1 73.465 Prec@5 91.660 Time 52.249
Over !!!
╭─  ~/Desktop/本科毕设/实践/VIT加速/qvit
```

经模拟，量化后的精度损失较小。

5 后续改进

计划尝试Softmax的量化推理方式以及GeLU函数的近似操作。