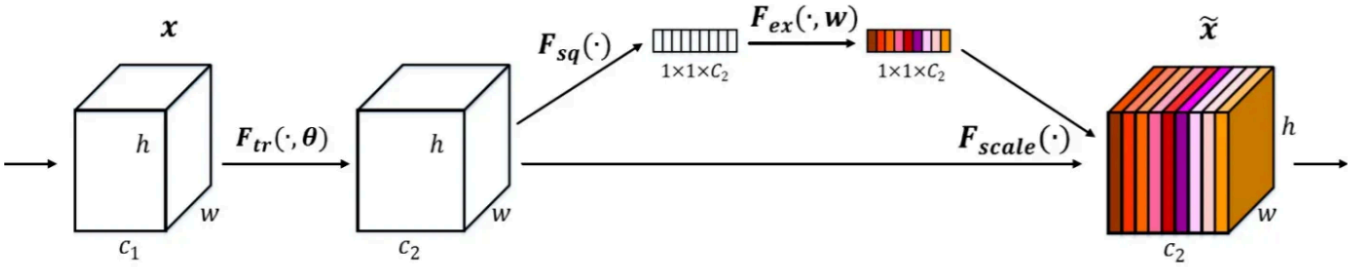
注意力机制和Transformer

人生来就有注意力机制，看任何画面，我们会自动聚焦到特定位置特定物体上。对于输入给网络的任何模态，不管是图像、文本、点云还是其他，我们都希望网络通过训练能够自动聚焦到有意义的位置，例如图像分类和检测任务，网络通过训练能够自动聚焦到待分类物体和待检测物体上。

Transformer并不是最早提出注意力机制的算法。例如，视觉算法如SENet中，利用 Squeeze-and-Excitation 模块计算注意力权重概率分布，然后作用于特征图上实现对每个通道重加权功能。



主要思想：通过建模channel之间的关系来矫正channel的特征，以此提升神经网络的表征能力。

具体分为两步：squeeze和excitation:

1. squeeze压缩每个channel的特征作为该channel的descriptor，采用的方法是均值池化，对channel里面的特征求均值：\

$$z_c = \mathbf{F}_{sq}(\mathbf{u}_c) = \frac{1}{H \times W} \sum_{i=1}^{H} \sum_{j=1}^{W} u_c(i,j) \tag{1}$$

2. excitation捕捉channel之间关系。论文使用的是一个gate网络，通过两层神经网络学习每个channel的权重，激活函数依次选为ReLU和Sigmoid。：

$$\mathbf{s} = \mathbf{F}_{ex}(\mathbf{z}, \mathbf{W}) = \sigma(g(\mathbf{z}, \mathbf{W})) = \sigma(\mathbf{W}_2 \delta(\mathbf{W}_1 z)) \tag{2}$$

$W_1 = R^{\frac{C}{r} \times C}$

有维度递减的作用，将C维的特征进行压缩，充分捕捉channel之间的关系。\

$W_2 = R^{C \times \frac{C}{r}}$

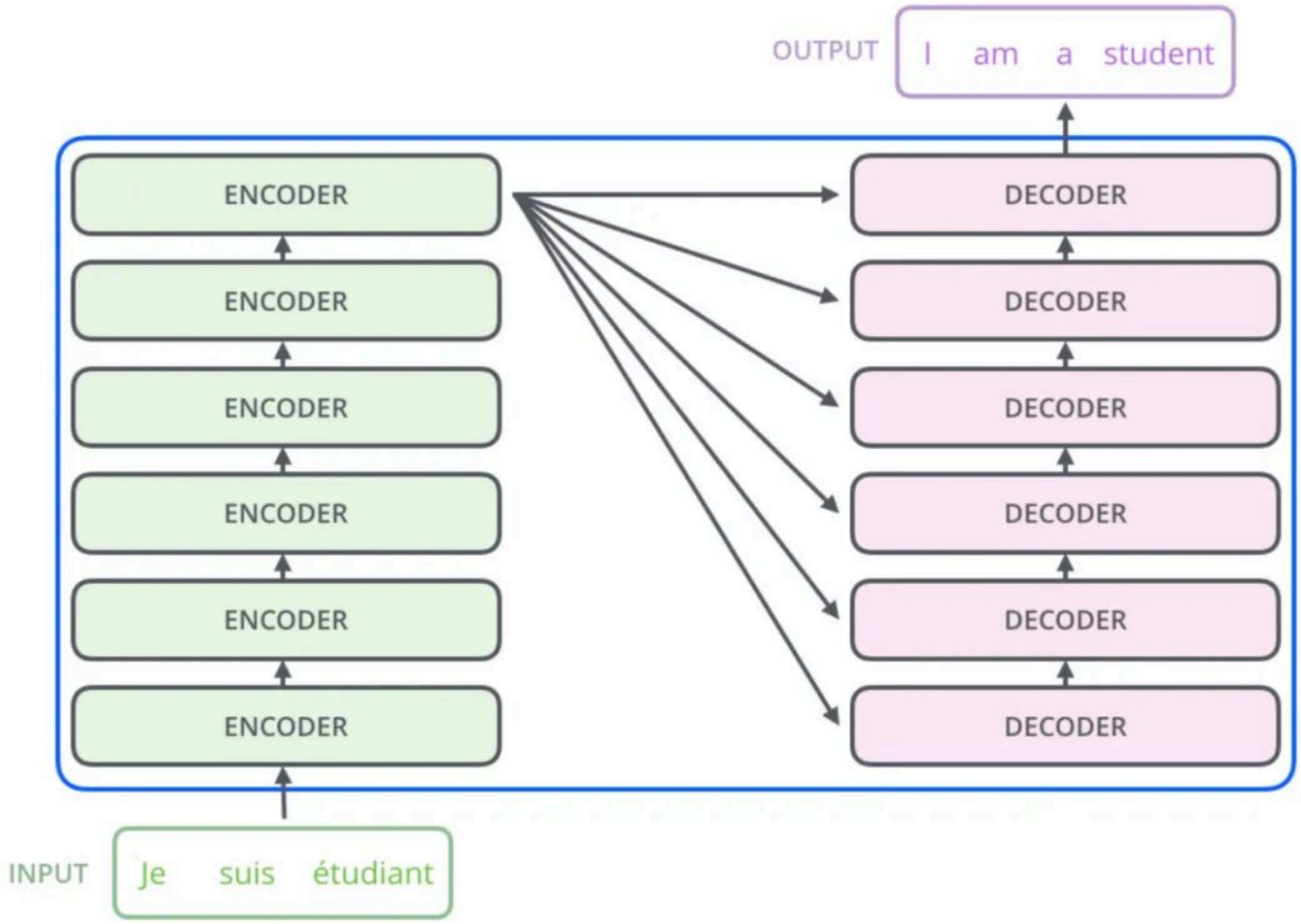
还原维度。gate网络输出s是C维的向量，每个值是对应的channel的权重。输入的特征最后会乘上对应的权重，无用的特征会被趋近于0：

可见，这是一种自注意力机制，它自动学习各通道之间的重要性程度。\
但是，这种注意力机制没有关注到通道内的注意力问题。

Transformer 结构是 Google 在 2017 年为解决机器翻译任务(例如英文翻译为中文)而提出，从题目 Attention is All You Need 中可以看出主要是靠 Attention 注意力机制，其最大特点是抛弃了传统的 CNN 和 RNN，整个网络结构完全是由 Attention 机制组成。为此需要先解释何为注意力机制，然后再分析模型结构。

初代Transformer：用于语言翻译任务

初代Transformer是为了解决NLP中的翻译任务而提出的。



在原论文中，Transformer的网络结构如下：

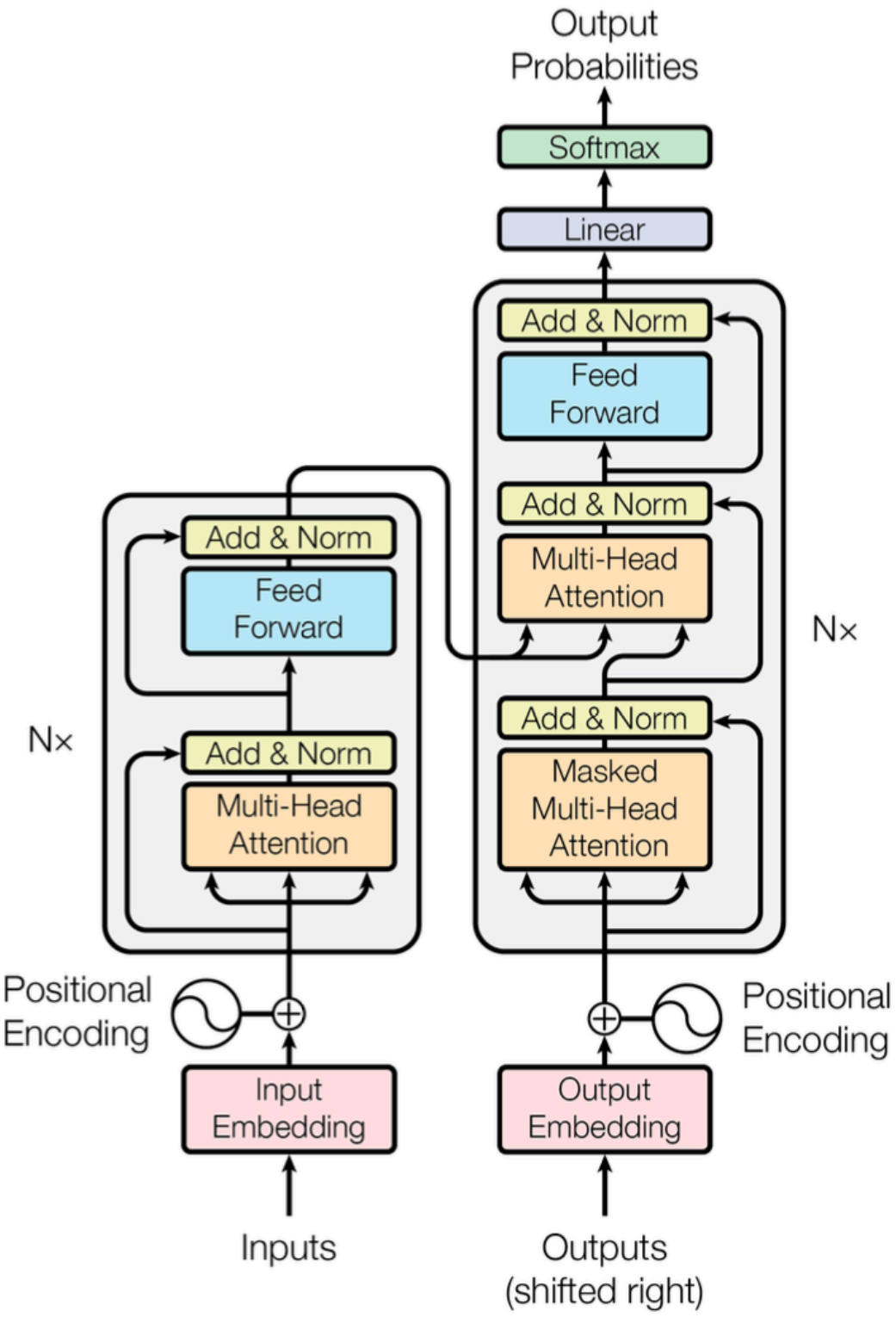


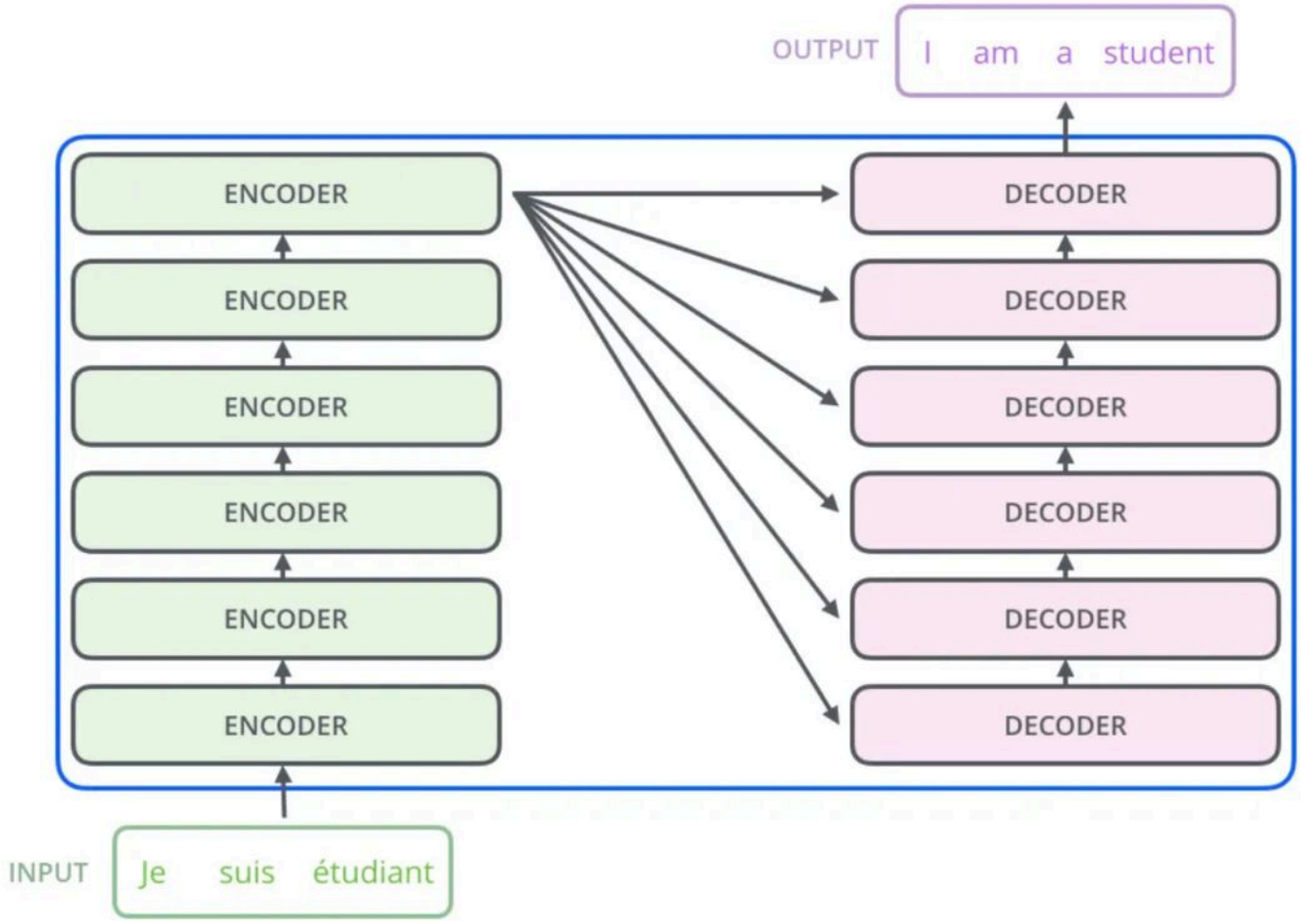
Figure 1: The Transformer - model architecture.

The **dominant** sequence transduction models are based on complex recurrent or convolutional neural networks that **include an encoder and a decoder**. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and

convolutions entirely (Vasvani et al., 2017). }

但这并不能直观反映Transformer的全局样貌，这主要是因为这个网络图使用了“Nx”标记来表示多个重复堆叠（stack）的相同模块。因此不太形象。

下面的图更好地反映了Transformer的全局结构：



如上图，Encoder部分是由6个相同的encoder组成，Decoder部分也是由6个相同的decoder组成，与encoder不同的是，每一个decoder都会接受最后一个encoder的输出。

注意到，Transformer不仅包含Encoder，还包含Decoder。为什么需要Decoder？这是由自然语言处理的特点决定的。道理很简单，翻译任务是典型的**Sequence-to-Sequence**任务——输入是一个序列，输出也是一个序列——因此，需要Encoder和Decoder。实际上，即使文本分类等其他任务，也要首先对输入的文本进行转译，变为一个可以被识别的token序列。

综上，标准的 Transformer 包括 6 个编码器和 6 个解码器串行。其中，数据的流向是：

1. 编码器内部接收源翻译输入序列，通过自注意力模块提取必备特征，通过前向网络对特征进行进一步抽象。\

2. 解码器端输入包括两个部分：一是目标翻译序列经过自注意力模块提取的特征，二是编码器提取的全局特征。这两个输入特征向量会进行交叉注意力计算，抽取有利于目标序列分类的特征，然后通过前向网络对特征进行进一步抽象。\
3. 堆叠多个编码器和解码器，下一个编解码器接收来自上一个编解码的输出，构成串行结构不断抽取，最后利用解码器输出进行分类即可。

Transformer用于提取特征：仅需要Encoder

前面介绍了，原版的Transformer有解码器，这是由自然语言处理的特点决定的。

但诸如图片分类等其他分类任务中，通常不需要解码器模块，所以我们只需要关注编码器部分。这也就是Survival Analysis中并没有Decoder的原因。

因此，对于这类Transformer，它的主要作用是提取特征主要是位置编码模块 Positional Encoding、多头自注意力模块 Muti-Head Attention、前向网络模块 Feed Forward 以及必要的 Norm、Dropout 和残差模块等：

1. Patch Embedding 模块。把输入图像转为模型可以处理的Token。其中可以分成 Image to Token 模块 和 Token to Token 模块。 Image to Token 将图片转化为 Token，而 Token to Token 用于各个 Transformer 模块间传递 Token。\
2. Position Embedding 模块。Positional Encoding 用于给 Embedding 所得到的 Token 序列增加额外的位置信息。\
3. Attention 模块。Muti-Head Attention 用于计算全局空间注意力。\
4. 前向网络模块。Feed Forward 用于对通道维度信息进行混合。\
5. 必要的 Norm、Dropout 和残差模块提供了更好的收敛速度和性能。 \
6. 分类预测模块，通常有两种做法，额外引入 Class Token 和采用常规分类做法引入全局池化模块进行信息聚合。

下面以 Vision Transformer （ViT）为例，介绍这类Transformer的基本原理和各个模块。实际上，包括Survival Analysis任务在内的很多Transformer的网络，其结构都是相同的。

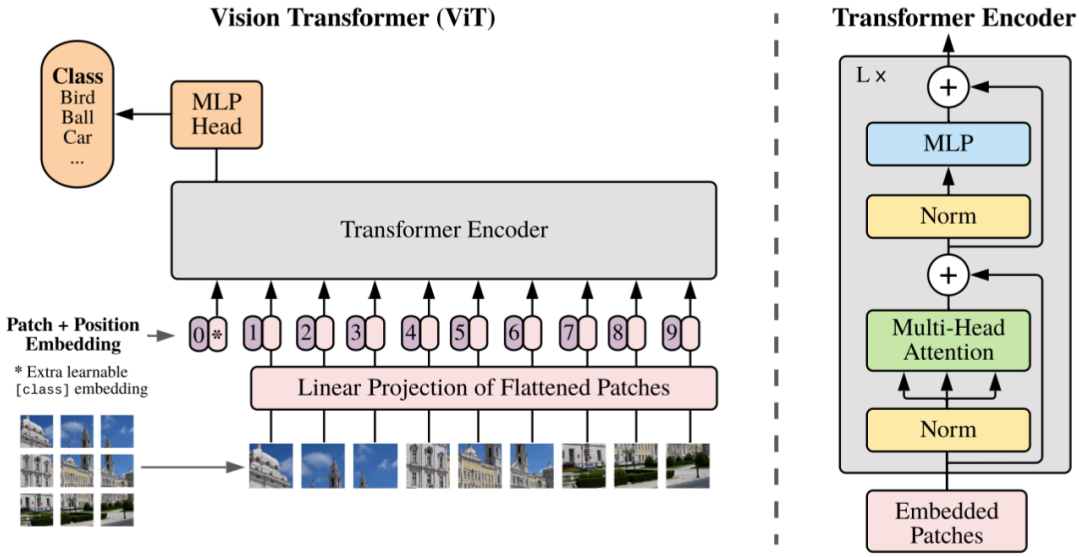


Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification token" to the sequence. The illustration of the Transformer encoder was inspired by Vaswani et al. (2017).

Transformers were proposed by Vaswani et al. (2017) for machine translation, and have since be-
come the state of the art method in many NLP tasks. Large Transformer-based models are often
pre-trained on large corpora and then fine-tuned for the task at hand: BERT (Devlin et al., 2019)
uses a denoising self-supervised pre-training task, while the GPT line of work uses language mod-
eling as its pre-training task (Radford et al., 2018; 2019; Brown et al., 2020).

在 NLP 中，输入 Transformer 中的是一个序列，而在视觉领域，需要考虑如何将一个2d图片转化为一个1d的序列，最直观的想法就是将图片中的像素点输入到 transformer 中，但是这样会有一个问题，因为模型训练中图片的大小是224*224=50176。而一般的NLP任务中序列长度一般是512。处理对象扩大了100倍，复杂度太高了。对此，ViT 的解决方案是：先将图片切分成一个个patch，然后每一个patch作为一个token输入到 Transformer 中。

由于整个 Transformer 每个 token 之间都会做 attention，所以输入本身并不存在一个顺序问题。但是对于图片而言，每个patch之间是有顺序的，所以类比原版Transformer，给每个 patch embedding 加上一个 position embedding （两者直接相加）。

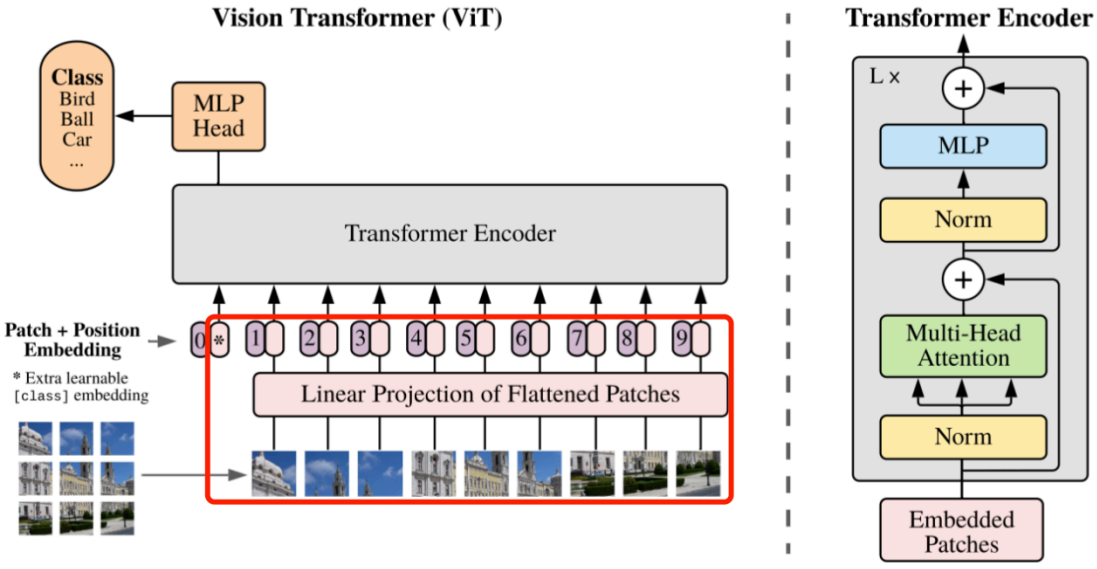
这时，我们已经得到了包含原始输入信息和位置信息的一系列 Tokens。这些Tokens接下来进入前面介绍的Encoder中，在Encoder中，输入信息依次流经6个encoders子层，每个子层中，又依次经过“标准化-多头注意力-标准化-全连接”四个模块（注意，其中还包含了两个残差结构的并联通路）。

这样，就网络就成功输出了提取到的特征。这些特征与一般非Transformer结构所提取的特征相比，优越性在于，在提取过程中很好地利用了注意力机制。

最终，把这些的特征用于分类等任务的损失函数，即得到了最终的输出。

1. Patch Embedding 模块

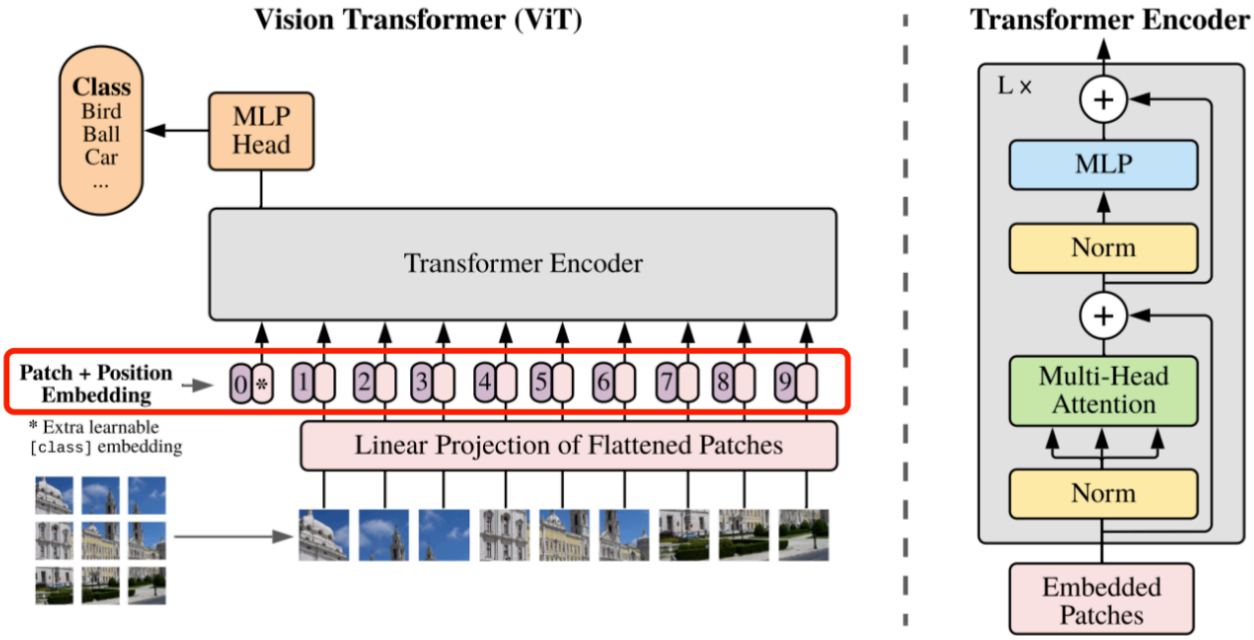
把输入图像转为模型可以处理的Token。其中可以分成 Image to Token 模块 和 Token to Token 模块。 Image to Token 将图片转化为 Token，而 Token to Token 用于各个 Transformer 模块间传递 Token。



另外需要注意，这里在 patch 的维度之外加入了一个 cls_token 维度，使 Tokens 的维度比 Patchs 数量多1。可以这样理解，其他的embedding表达的都是不同的patch的特征，而cls_token是要综合所有patch的信息，产生一个新的embedding，来表达整个图的信息。

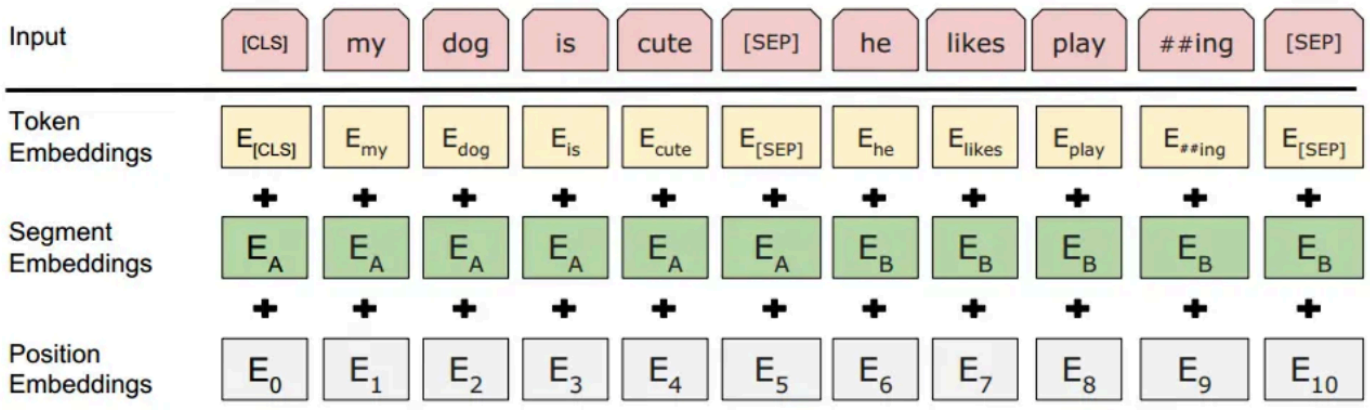
2. Position Encoding 模块

Positional Encoding 用于给 Embedding 所得到的 Token 序列增加额外的位置信息。



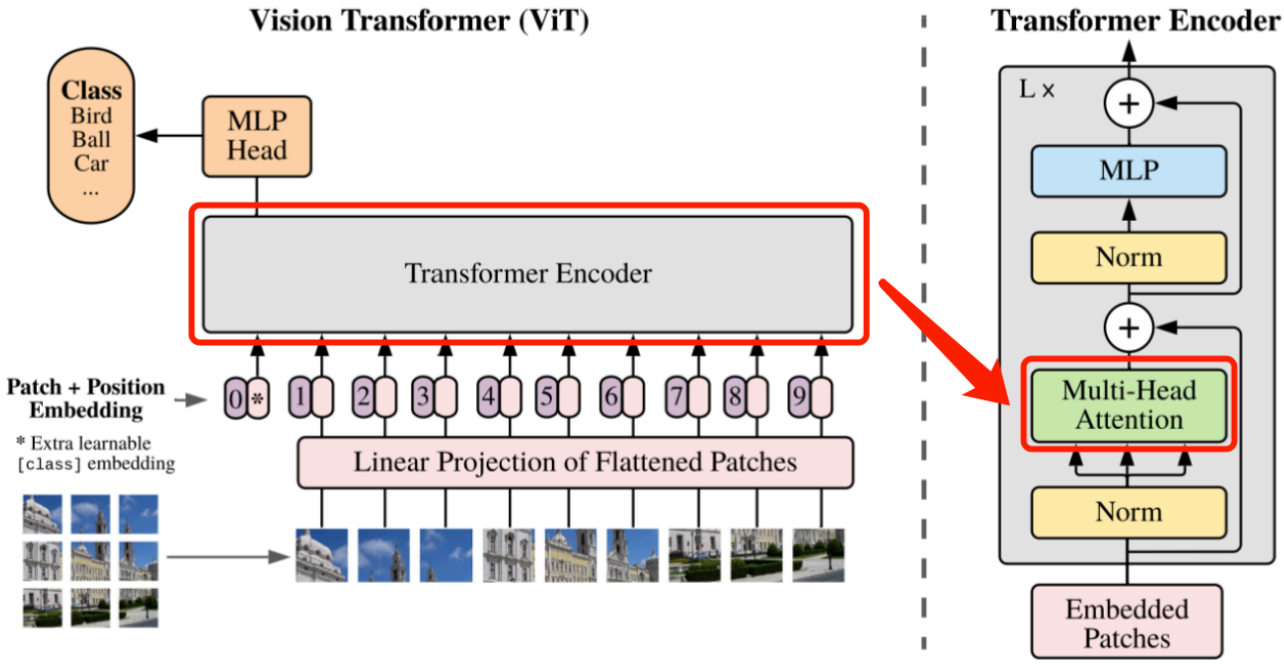
具体计算公式：

$$
\begin{aligned}
PE_{(pos,2i)} &= \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) \\
PE_{(pos,2i+1)} &= \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)
\end{aligned}
\tag{3}
$$

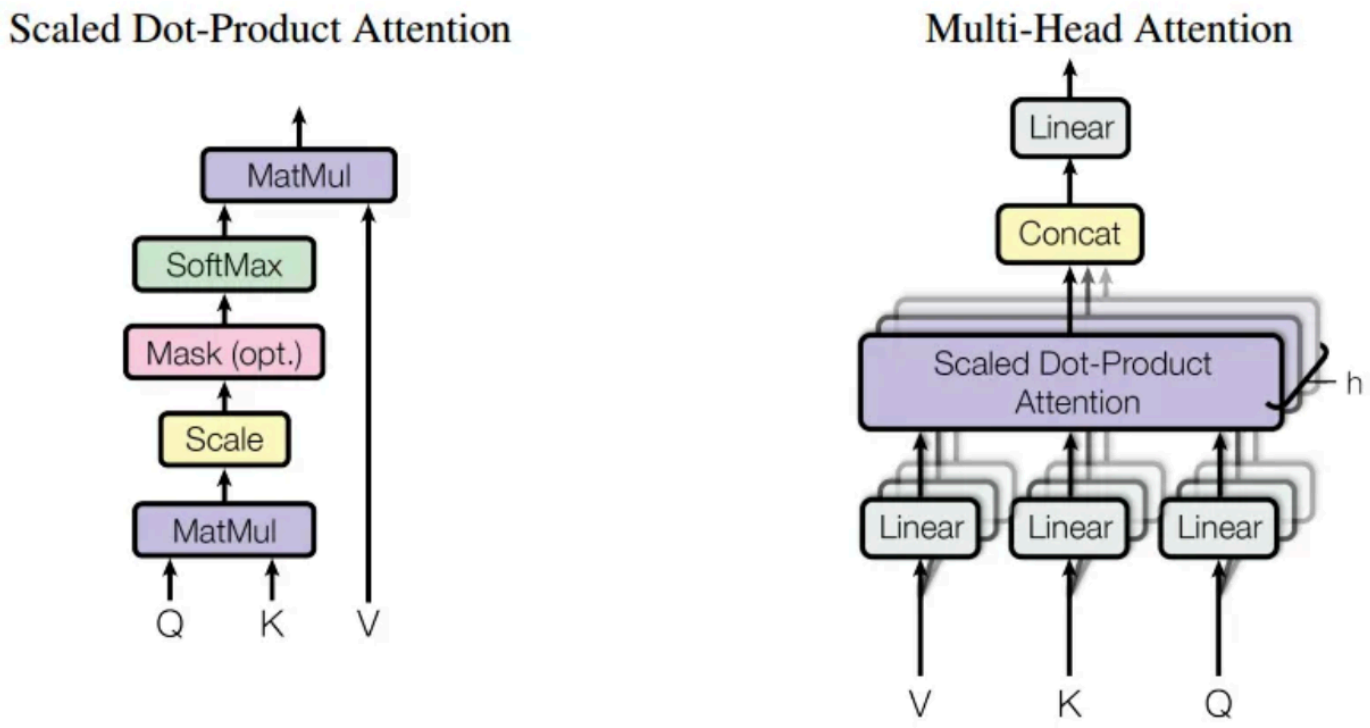


3. Attention 模块

Muti-Head Attention 用于计算全局空间注意力。同时还解决了RNN、LSTM等常用于处理序列化数据的网络结构无法在GPU中并行加速计算的问题

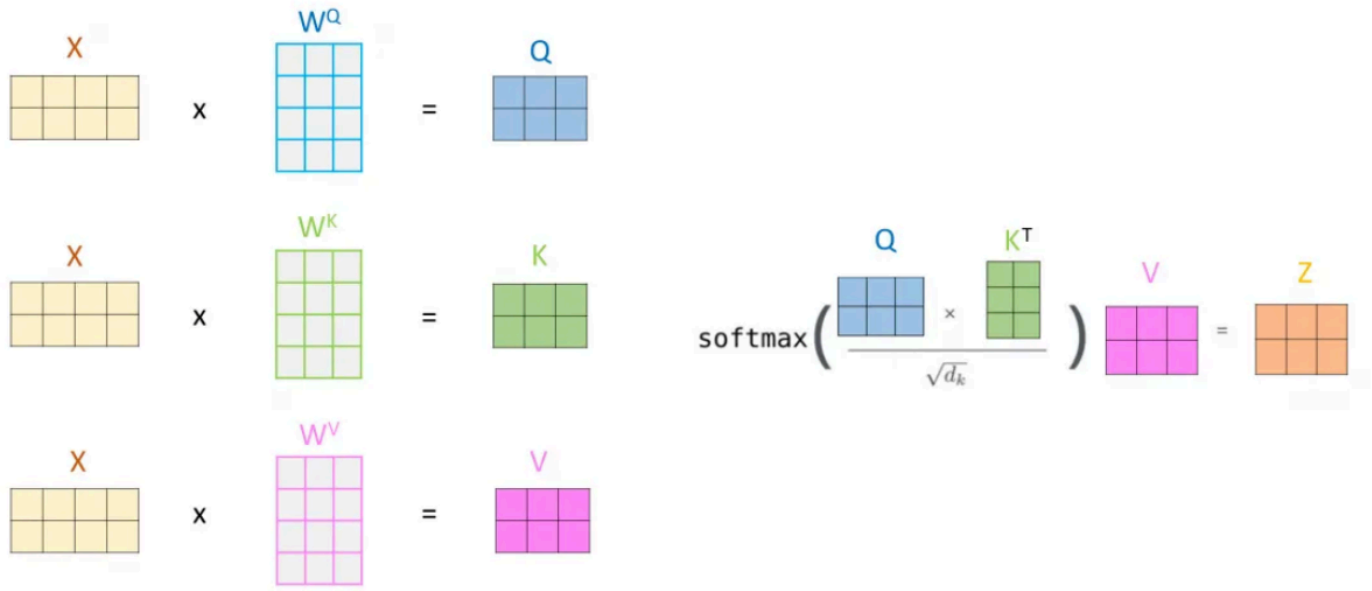


其中，有两种注意力机制：单头和多头。



结合下面的公式，网络结构图不难理解。先从基本的单头注意力开始：

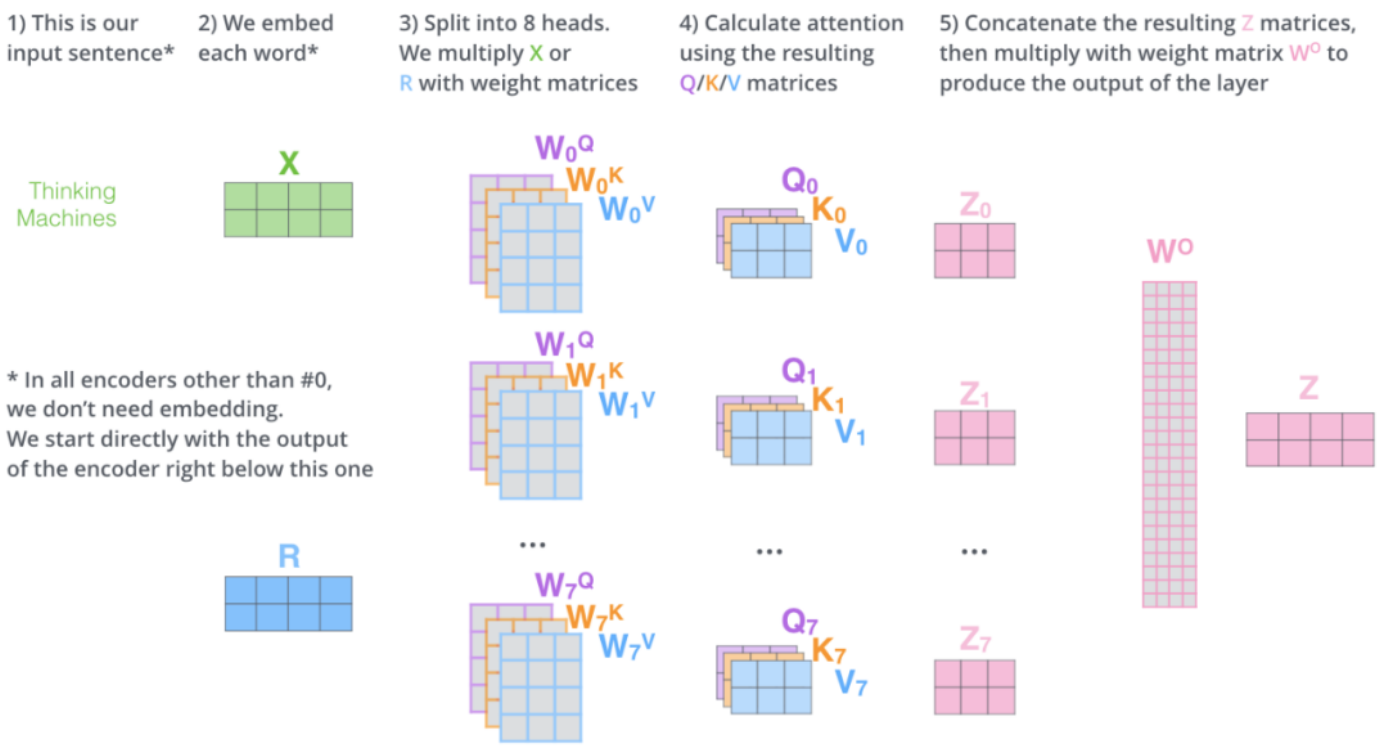
$$Attention(Q,K,V)=soft\max(\frac{QK^T}{\sqrt{d_k}})V$$



其实就相当于用了三个全连接层，分别把输入映射到三个不同的通道(Q,K,V)。然后把其中两个做内积，表示特征图中任意一个查询（Q）和索引（K）的关联程度，最后把这个图层解释为所谓的“注意力”。再用这个注意力与被查询的值相乘，来计算出带有注意力的特征。当然，这样的乘法结果与维度正相关，很容易导致结果随规模增大而过大，因此，在计算注意力这一步除以了维度的二次方根$\sqrt{d_k}$。

当然，这个注意力在一开始肯定是随机而盲目的，基本上不会有任何方向性，这时，把三个随机初始化的矩阵分别叫做Q,K,V似乎是令人费解的。但是，随着训练的进行，损失函数控制着参数向有利于损失下降的方向更新，这当然也包括(W_Q,W_K,W_V)的更新。因此，理想的情形是，(Q,K,V)会不断收敛到有效地注意目标的地步。这时再考虑Q,K,V的含义就很好理解了。

理解了单头注意力，多头注意力就不难理解了。它只是多个单头注意力叠在一起，各自并行地计算，最后再把输出$Z_1,Z_2,\ldots,Z_n$拼接（concatenate）起来，乘以一个对应的拼接的权重矩阵W^O，得到最后的输出Z。



以下是Attention部分的代码实现，只要理解了上述过程，其实也并不复杂：

```python
import torch.nn as nn

class Attention(nn.Module):
    def __init__(self, dim, num_heads=8, qkv_bias=False, attn_drop=0., proj_drop=0.):
        super().__init__()
        self.num_heads = num_heads
        # q,k,v向量长度
        head_dim = dim // num_heads
        self.scale = head_dim ** -0.5
        self.qkv = nn.Linear(dim, dim * 3, bias=qkv_bias)
        self.attn_drop = nn.Dropout(attn_drop)
        self.proj = nn.Linear(dim, dim)
        self.proj_drop = nn.Dropout(proj_drop)

    def forward(self, x):
        # 这里C对应上面的E，向量的长度
        B, N, C = x.shape
        # (B, N, C) -> (3, B, num_heads, N, C//num_heads), //是向下取整的意思。
        qkv = self.qkv(x).reshape(B, N, 3, self.num_heads, C // self.num_heads).permute(2, 0, 3, 1, 4)
        # 将qkv在0维度上切成三个数据块，q,k,v:(B, num_heads, N, C//num_heads)
```

```
        # 这里的效果是从每个向量产生三个向量，分别是query，key和value
        q, k, v = qkv.unbind(0)   # make torchscript happy (cannot use
tensor as tuple)
        # @矩阵相乘获得score (B,num_heads,N,N)
        attn = (q @ k.transpose(-2, -1)) * self.scale
        attn = attn.softmax(dim=-1)
        attn = self.attn_drop(attn)
        # (B,num_heads,N,N)@(B,num_heads,N,C//num_heads)->
(B,num_heads,N,C//num_heads)
        # (B,num_heads,N,C//num_heads) ->(B,N,num_heads,C//num_heads)
        # (B,N,num_heads,C//num_heads) -> (B, N, C)
        x = (attn @ v).transpose(1, 2).reshape(B, N, C)
        # (B, N, C) -> (B, N, C)
        x = self.proj(x)
        x = self.proj_drop(x)
        return x
```

有了以上的Attention模块，接下来自然是把它组装到每一个encoder子层里。

Encoder 中的每一个encoder子层（layer normalization -> multi-head attention -> drop path -> layer normalization -> mlp -> drop path）在ViT中的实现细节如下（Block就是encoder，只不过换了个名字）：

```
class Block(nn.Module):

    def __init__(self, dim, num_heads, mlp_ratio=4., qkv_bias=False,
drop=0., attn_drop=0.,
                 drop_path=0., act_layer=nn.GELU,
norm_layer=nn.LayerNorm):
        super().__init__()

        # 将每个样本的每个通道的特征向量做归一化
        # 也就是说每个特征向量是独立做归一化的
        # 我们这里虽然是图片数据，但图片被切割成了patch，用的是语义的逻辑
        self.norm1 = norm_layer(dim)
        self.attn = Attention(dim, num_heads=num_heads,
qkv_bias=qkv_bias, attn_drop=attn_drop, proj_drop=drop)
        # NOTE: drop path for stochastic depth, we shall see if this
is better than dropout here
        self.drop_path = DropPath(drop_path) if drop_path > 0. else
nn.Identity()
        self.norm2 = norm_layer(dim)
```

```
        mlp_hidden_dim = int(dim * mlp_ratio)
        # 全连接，激励，drop，全连接，drop,若out_features没填，那么输出维度不变。
        self.mlp = Mlp(in_features=dim, hidden_features=mlp_hidden_dim, act_layer=act_layer, drop=drop)

    def forward(self, x):
        # 最后一维归一化，multi-head attention, drop_path
        # (B, N, C) -> (B, N, C)
        x = x + self.drop_path(self.attn(self.norm1(x)))
        # (B, N, C) -> (B, N, C)
        x = x + self.drop_path(self.mlp(self.norm2(x)))
        return x
```

在ViT中这样的block会有好几层（像原版的Transformer即是有6层这样的block），它们串行地联接（stack），组成blocks：

```

class Blocks(Block):

    def __init__(self, drop_path_rate, depth):
        super().__init__()
        self.drop_path_rate = drop_path_rate
        self.depth = depth
        self.dpr = [x.item() for x in torch.linspace(0, drop_path_rate, depth)]

        self.blocks = nn.Sequential(*[
            Block(
                dim=embed_dim, num_heads=num_heads, mlp_ratio=mlp_ratio, qkv_bias=qkv_bias, drop=drop_rate,
                attn_drop=attn_drop_rate, drop_path=dpr[i], norm_layer=norm_layer, act_layer=act_layer
            )
            for i in range(depth)
        ])

    def forward(self, x):
        return blocks(x)

```

4. 前向网络模块

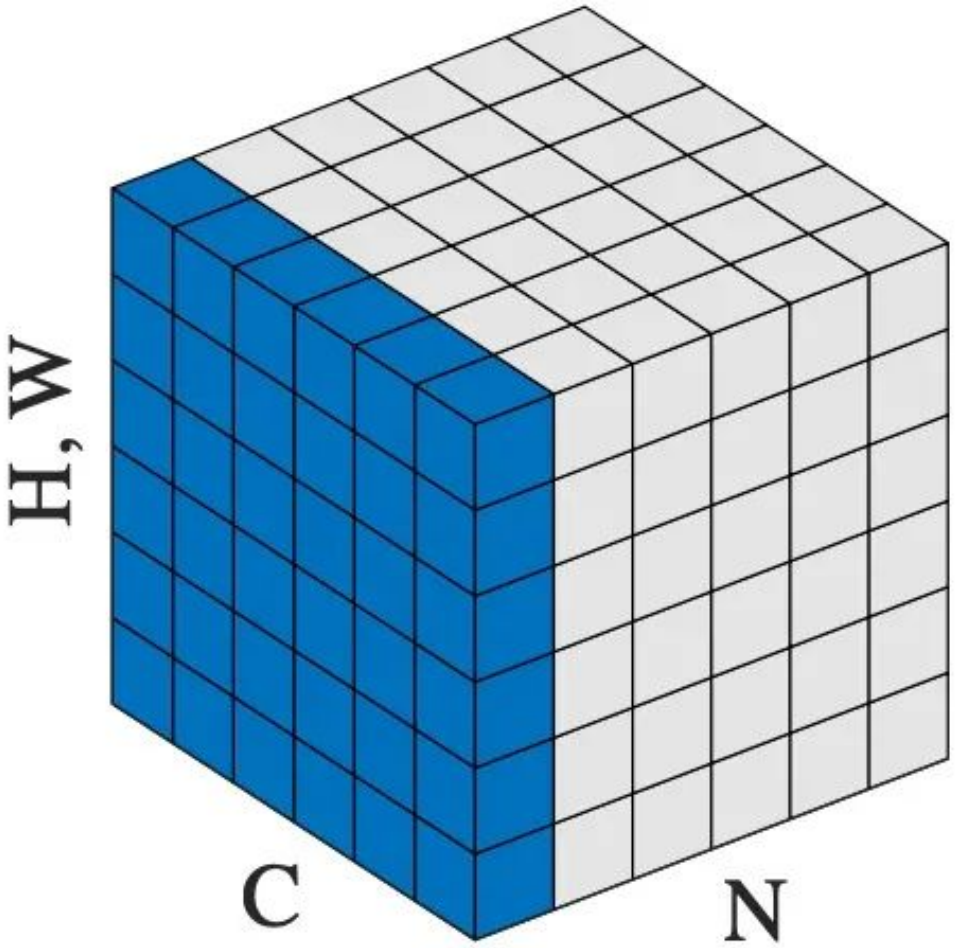
Feed Forward 用于对通道维度信息进行混合。

```
def forward_features(self, x):
    # x由 (B, C, H, W) -> (B, N, E)
    x = self.patch_embed(x)
    # stole cls_tokens impl from Phil Wang, thanks
    # cls_token由(1, 1, 768)->(B, 1, 768), B是batch_size
    cls_token = self.cls_token.expand(x.shape[0], -1, -1)
    # dist_token是None,DeiT models才会用到dist_token。
    if self.dist_token is None:
        # x由(B, N, E)->(B, 1+N, E)
        x = torch.cat((cls_token, x), dim=1)
    else:
        # x由(B, N, E)->(B, 2+N, E)
        x = torch.cat((cls_token, self.dist_token.expand(x.shape[0],
-1, -1), x), dim=1)
    # +pos_embed:(1, 1+N, E)，再加一个dropout层
    x = self.pos_drop(x + self.pos_embed)
    x = self.blocks(x)
    # nn.LayerNorm
    x = self.norm(x)
    if self.dist_token is None:
        # 不是DeiT，输出就是x[:,0]，(B, 1, 768)，即cls_token
        return self.pre_logits(x[:, 0])
    else:
        # 是DeiT，输出就是cls_token和dist_token
        return x[:, 0], x[:, 1]
```

5. 必要的 Norm、Dropout 和残差模块

提供了更好的收敛速度和性能。

其中，Norm主要用的是Layer normalization。与Batch Normalization根本的不同在于，Layer normalization是对每个样本的所有特征进行归一化，而Batch Normalization是对每个通道的所有样本进行归一化。




```python
import torch

# NLP Example
batch, sentence_length, embedding_dim = 20, 5, 10
embedding = torch.randn(batch, sentence_length, embedding_dim)
# 指定归一化的维度
layer_norm = nn.LayerNorm(embedding_dim)
# 进行归一化
layer_norm(embedding)

# Image Example
N, C, H, W = 20, 5, 10, 10
input = torch.randn(N, C, H, W)
# Normalize over the last three dimensions (i.e. the channel and spatial dimensions)
# as shown in the image below
layer_norm = nn.LayerNorm([C, H, W])
output = layer_norm(input)
```

6. 分类预测模块

通常有两种做法，额外引入 Class Token 和采用常规分类做法引入全局池化模块进行信息聚合。