20 Vision Transformer(ViT)网络详解

AN IMAGE IS WORTH 16X16 WORDS: TRANSFORMERS FOR IMAGE RECOGNITION AT SCALE

**Alexey Dosovitskiy*,†, Lucas Beyer*, Alexander Kolesnikov*, Dirk Weissenborn*,
Xiaohua Zhai*, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer,
Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, Neil Houlsby*,†**
*equal technical contribution, †equal advising
Google Research, Brain Team
{adosovitskiy, neilhoulsby}@google.com

20.1 前言

ViT 其原始论文为 An Image Is Worth 16x16 Words: Transformers For Image Recognition At Scale。首先看一下 ViT 模型的效果，在 ImageNet 1k 上最高能达到 88.55 的准确率，但是是现在自家的数据集上进行了预训练，总共三亿数据量。

	Ours-JFT (ViT-H/14)	Ours-JFT (ViT-L/16)	Ours-I21k (ViT-L/16)	BiT-L (ResNet152x4)	Noisy Student (EfficientNet-L2)
ImageNet	**88.55** ± 0.04	87.76 ± 0.03	85.30 ± 0.02	87.54 ± 0.02	88.4/88.5*
ImageNet ReaL	**90.72** ± 0.05	90.54 ± 0.03	88.62 ± 0.05	90.54	90.55
CIFAR-10	**99.50** ± 0.06	99.42 ± 0.03	99.15 ± 0.03	99.37 ± 0.06	–
CIFAR-100	**94.55** ± 0.04	93.90 ± 0.05	93.25 ± 0.05	93.51 ± 0.08	–
Oxford-IIIT Pets	**97.56** ± 0.03	97.32 ± 0.11	94.67 ± 0.15	96.62 ± 0.23	–
Oxford Flowers-102	99.68 ± 0.02	**99.74** ± 0.00	99.61 ± 0.02	99.63 ± 0.03	–
VTAB (19 tasks)	**77.63** ± 0.23	76.28 ± 0.46	72.72 ± 0.21	76.29 ± 1.70	–
TPUv3-core-days	2.5k	0.68k	0.23k	9.9k	12.3k

Table 2: Comparison with state of the art on popular image classification benchmarks. We report mean and standard deviation of the accuracies, averaged over three fine-tuning runs. Vision Transformer models pre-trained on the JFT-300M dataset outperform ResNet-based baselines on all datasets, while taking substantially less computational resources to pre-train. ViT pre-trained on the smaller public ImageNet-21k dataset performs well too. *Slightly improved 88.5% result reported in Touvron et al. (2020).

20.2. ViT 模型架构

动图展示：

其原始架构图如下。

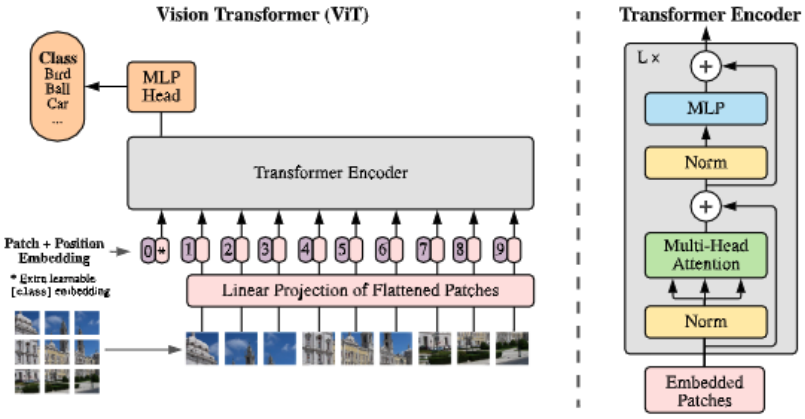


Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification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

可以看到：

1. 首先输入图片分为很多 patch，论文中为 16。
2. 将 patch 输入一个 Linear Projection of Flattened Patches 这个 Embedding 层，就会得到一个个向量，通常就称作 token。
3. 紧接着在一系列 token 的前面加上加上一个新的 token（类别token，有点像输入给 Transformer Decoder 的 START，就是对应着 * 那个位置），此外还需要加上位置的信息，对应着 0 ~ 9。
4. 然后输入到 Transformer Encoder 中，对应着右边的图，将 block 重复堆叠 L 次。
5. Transformer Encoder 有多少个输入就有多少个输出。
6. 最后只进行分类，所以将 class 位置对应的输出输入 MLP Head 进行预测分类输出。

20.2.1 Embedding 层

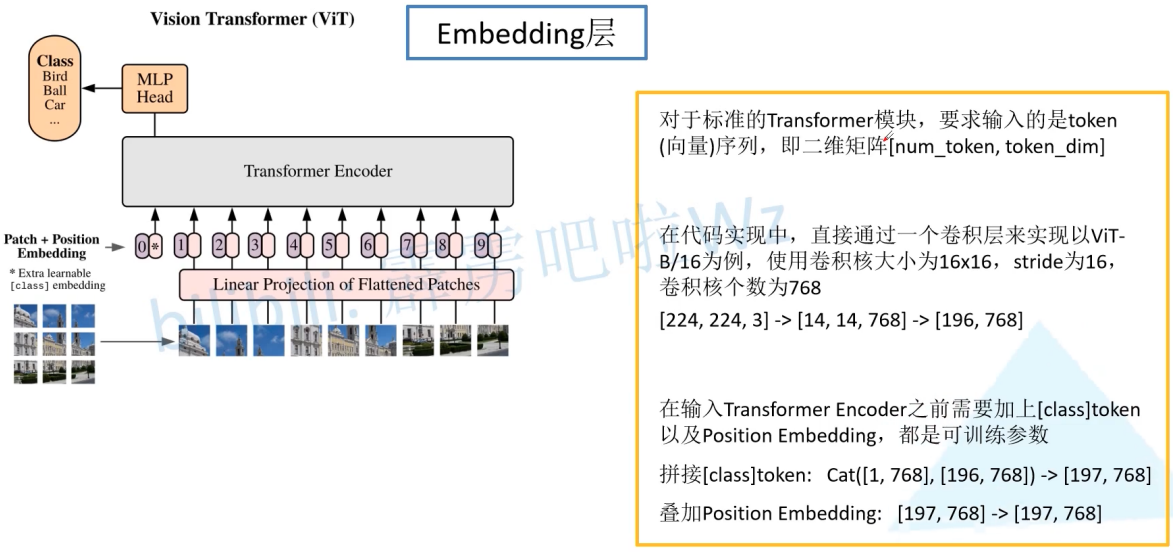
首先是 Embedding 层。

对于标准的 Transformer 模块，要求的输入是 token 向量的序列，即二维矩阵。在具体的代码实现过程中呢，我们实际是通过一个卷积层来实现。

以 ViT-B/16 为例，使用卷积核大小为 16×16, stride 为 16，卷积核个数为 768 ，即 [224,224,3] –> [14,14,768] –> [196, 768]。即一共 196 个token，每个 token 向量长度为 768。

此外我们还需要加上一个类别的 token，为此我们实际上是初始化了一个可训练的参数 [1, 768]，将其与 token 序列进行拼接得到 Cat([1, 768], [196,768]) –> [197, 768]。

然后再叠加上位置编码 Position Embedding: [197,768] –> [197, 768]。



再详细考虑下 Position Embedding。

不用 Position Embedding 得到的结果是 0.61382，使用一维的位置编码得到的结果是 0.64206，明显比不使用位置编码高了三个百分点。使用 2D 以及相对位置编码其实和 1D 效果差不多。论文中也提到说 `the difference in how to encoder spatial information is less important`，**即位置编码的差异其实不是特别重要。**

1D 的话，简单效果好参数少，所以默认使用 1D 的位置编码。

Pos. Emb.	Default/Stem	Every Layer	Every Layer-Shared
No Pos. Emb.	0.61382	N/A	N/A
1-D Pos. Emb.	0.64206	0.63964	0.64292
2-D Pos. Emb.	0.64001	0.64046	0.64022
Rel. Pos. Emb.	0.64032	N/A	N/A

Table 8: Results of the ablation study on positional embeddings with ViT-B/16 model evaluated on ImageNet 5-shot linear.

论文中有给这样一个图，我们训练得到的位置编码与其他位置编码之间的余弦相似度。这里的 patches 大小是 32×32 的，224/32=7，所以这里的大小是 7×77×7。这张图怎么理解呢？我们会在每个 token 上叠加一个位置编码，中间那个图的 49 个小图中，每个小图其实也是 7×7 的。左上角第一行第一个 patch 的位置编码与自己的位置编码是一样的，所以余弦相似度是1，所以左上角是黄色。然后在与其他位置编码进行计算。就得到了左上角的小图。其他的也都是类似的规律。注意，**这个是学出来的。**

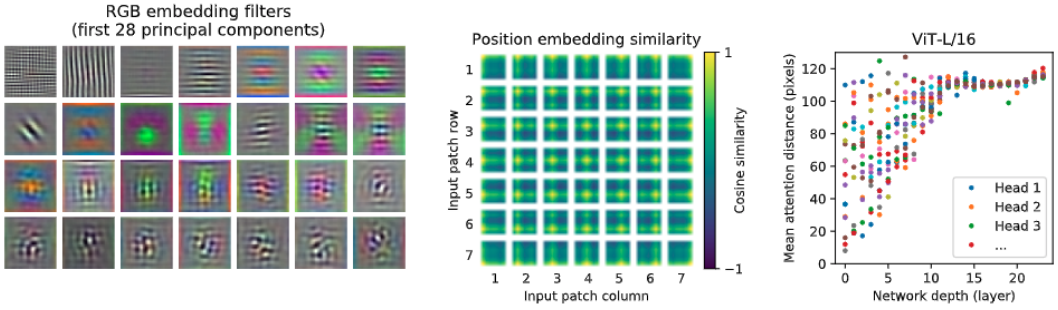
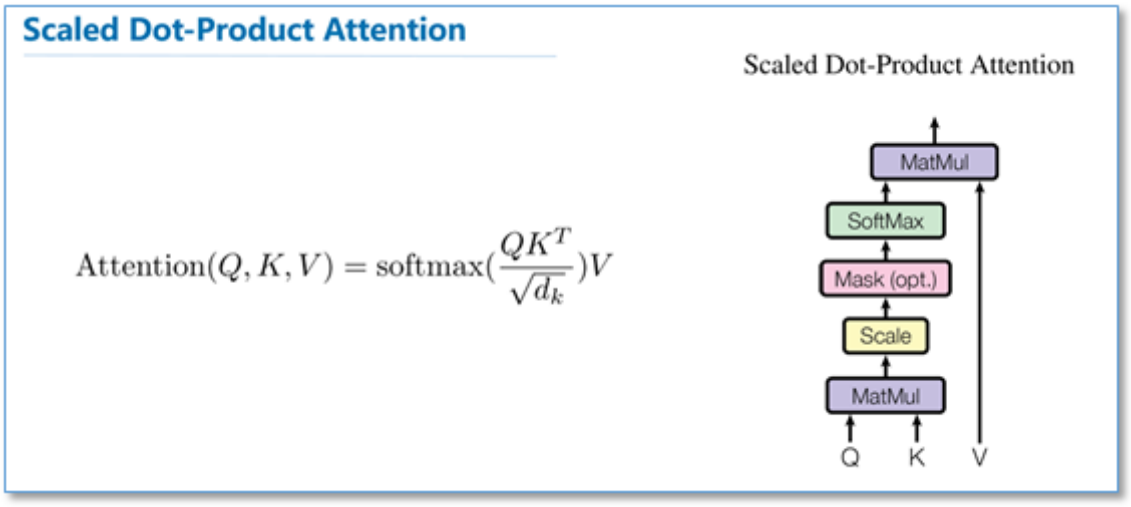


Figure 7: **Left:** Filters of the initial linear embedding of RGB values of ViT-L/32. **Center:** Similarity of position embeddings of ViT-L/32. Tiles show the cosine similarity between the position embedding of the patch with the indicated row and column and the position embeddings of all other patches. **Right:** Size of attended area by head and network depth. Each dot shows the mean attention distance across images for one of 16 heads at one layer. See Appendix D.7 for details.

20.2.2 Transformer Encoder 层

Transformer 编码器 由交替的 **多头自注意力层** (MSA, 附录 A) 和 **多层感知机块** (MLP, 等式 2, 3) 构成。在每个块前应用 **层归一化 (Layer Norm)**，在每个块后应用 **残差连接 (Residual Connection)**。



img

在 Transformer中，MSA 后跟一个 FFN (Feed-forward network)，其包含 两个 FC 层，第一个 FC 将特征从维度D变换成4D，第二个 FC 将特征从维度4D 恢复成D ，中间的非线性激活函数均采用 GeLU (Gaussian Error Linear Unit，高斯误差线性单元) —— 这实质是一个 MLP (多层感知机与线性模型类似，区别在于 MLP 相对于 FC 层数增加且引入了非线性激活函数，例如 FC + GeLU + FC)，实现如下：

```
class Mlp(nn.Module):
    def __init__(self, in_features, hidden_features=None, out_features=None, act_layer=nn.GELU, drop=0.):
        super().__init__()

        out_features = out_features or in_features
        hidden_features = hidden_features or in_features
        self.fc1 = nn.Linear(in_features, hidden_features)
        self.act = act_layer()
        self.fc2 = nn.Linear(hidden_features, out_features)
        self.drop = nn.Dropout(drop)

    def forward(self, x):
        x = self.fc1(x)
        x = self.act(x)
        x = self.drop(x)
        x = self.fc2(x)
        x = self.drop(x)

        return x
```

Transformer Encoder 就是将 Encoder Block 重复堆叠 L 次。

首先输入一个 Norm 层，这里的 Norm 指的是 Layer Normalization 层（有论文比较了 BN 在 transformer 中为什么不好，不如 LN ｜这里先 Norm 再 Multihead Attention 也是有论文研究的，原始的 Transformer 先 Attention 再 Norm，此外这个先 Norm 再操作和 DenseNet 的先 BN 再 Conv 异曲同工）。经过 LN 后经过 Multi-Head Attention，然后源码经过 Dropout 层，有些其他人复现使用的是 DropPath 方法，根据以往的经验可能使用后者会更好一点。然后残差之后经过 LN，MLP Block，Dropout/DropPath 之后残差即可。

MLP Block 其实也很简单，就是一个全连接，GELU 激活函数，Dropout，全连接，Dropout。

一个 **Transformer Encoder Block** 就包含一个 **MSA** 和一个 **FFN**，二者都有 **跳跃连接** 和 **层归一化** 操作构成 **MSA Block** 和 **MLP Block**，实现如下：

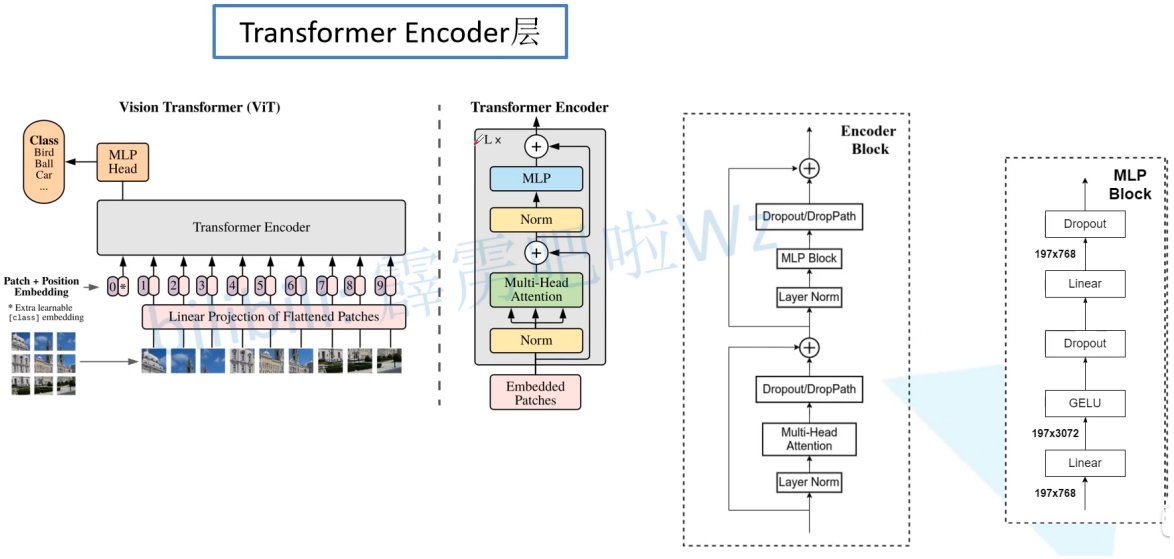
```
# Transformer Encoder Block
class Block(nn.Module):
    def __init__(self, dim, num_heads, mlp_ratio=4., qkv_bias=False, qk_scale=None, drop=0., attn_drop=0.,
                 drop_path=0., act_layer=nn.GELU, norm_layer=nn.LayerNorm):
        super().__init__()

        # 后接于 MHA 的 Layer Norm
        self.norm1 = norm_layer(dim)
        # MHA
        self.attn = Attention(dim, num_heads=num_heads, qkv_bias=qkv_bias,
                              qk_scale=qk_scale, 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()

        # 后接于 MLP 的 Layer Norm
        self.norm2 = norm_layer(dim)
        # 隐藏层维度
        mlp_hidden_dim = int(dim * mlp_ratio)
        # MLP
        self.mlp = Mlp(in_features=dim, hidden_features=mlp_hidden_dim, act_layer=act_layer, drop=drop)

    def forward(self, x):
        # MHA + Add & Layer Norm
        x = x + self.drop_path(self.attn(self.norm1(x)))
        # MLP + Add & Layer Norm
        x = x + self.drop_path(self.mlp(self.norm2(x)))
        return x
```

集合了**类别向量、图像块嵌入和 位置编码**三者到一体的**输入嵌入向量**后，即可馈入**Transformer Encoder**。ViT 类似于 CNN，不断前向通过 由 **Transformer Encoder Blocks** 串行堆叠构成的 **Transformer Encoder**，最后 **提取可学习的类别嵌入向量 —— class token** 对应的特征用于 **图像分类**。整体前向计算过程如下：

$$
\begin{aligned}
z_0 &= [\mathbf{x}_{\text{class}}; \mathbf{x}_p^1\mathbf{E}; \mathbf{x}_p^2\mathbf{E}; \cdots; \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos}, && \mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D}, \mathbf{E}_{pos} \in \mathbb{R}^{(N+1) \times D} \\
z' &= \text{MSA}(\text{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, && \ell = 1 \ldots L \\
\text{z} &= \text{MLP}(\text{LN}(\mathbf{z}'_{\ell})) + \mathbf{z}'_{\ell}, && \ell = 1 \ldots L \\
\text{y} &= \text{LN}(\mathbf{z}_I^0)
\end{aligned}
$$

- **等式 1：** 由 图像块嵌入$x_p^i E, i \in 1, 2, \ldots, N$、类别向量$x_{class}$和位置编码 E_{pos}构成 嵌入输入向量z_0
- **等式 2：** 由 多头注意力机制、层归一化 和 跳跃连接 (Laer Norm & Add) 构成的 MSA Block，可重复L个，其中第l个输出为 z_l'
- **等式 3：** 由 前馈网络 (FFN)、层归一化 和 跳跃连接 (Layer Norm & Add) 构成的 MLP Block，可重复L个，其中第l个输出为 z_l
- **等式 4：** 由 层归一化 (Layer Norm) 和 分类头 (MLP or FC) 输出 图像表示y

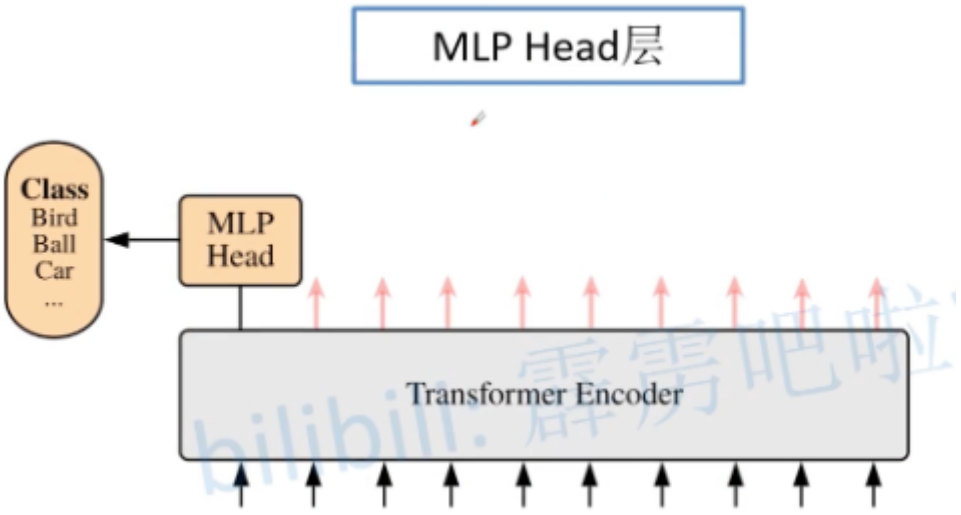
需要注意上面提到的，第一个全连接的节点个数是输入向量长度的 4 倍，第二个全连接层会还原会原来的大小。



有一个地方要注意，在 Transformer Encoder 前有个 Dropout 层，在之后有一个 Layer Norm 层，这些在图中还没有画出来的。**在 Transformer Encoder 前有个 Dropout 层，目的是在原图上随机加 Mask 遮挡，然后依然要进行分类。**

20.2.3 MLP Head 层

在训练 ImageNet21K 时候是由 Linear + tanh 激活函数 + Linear 构成的。但是迁移到 ImageNet1k 之后或者做迁移学习时，其实只需要一个 Linear 就足够了。（获得类别概率需要一个 softmax）

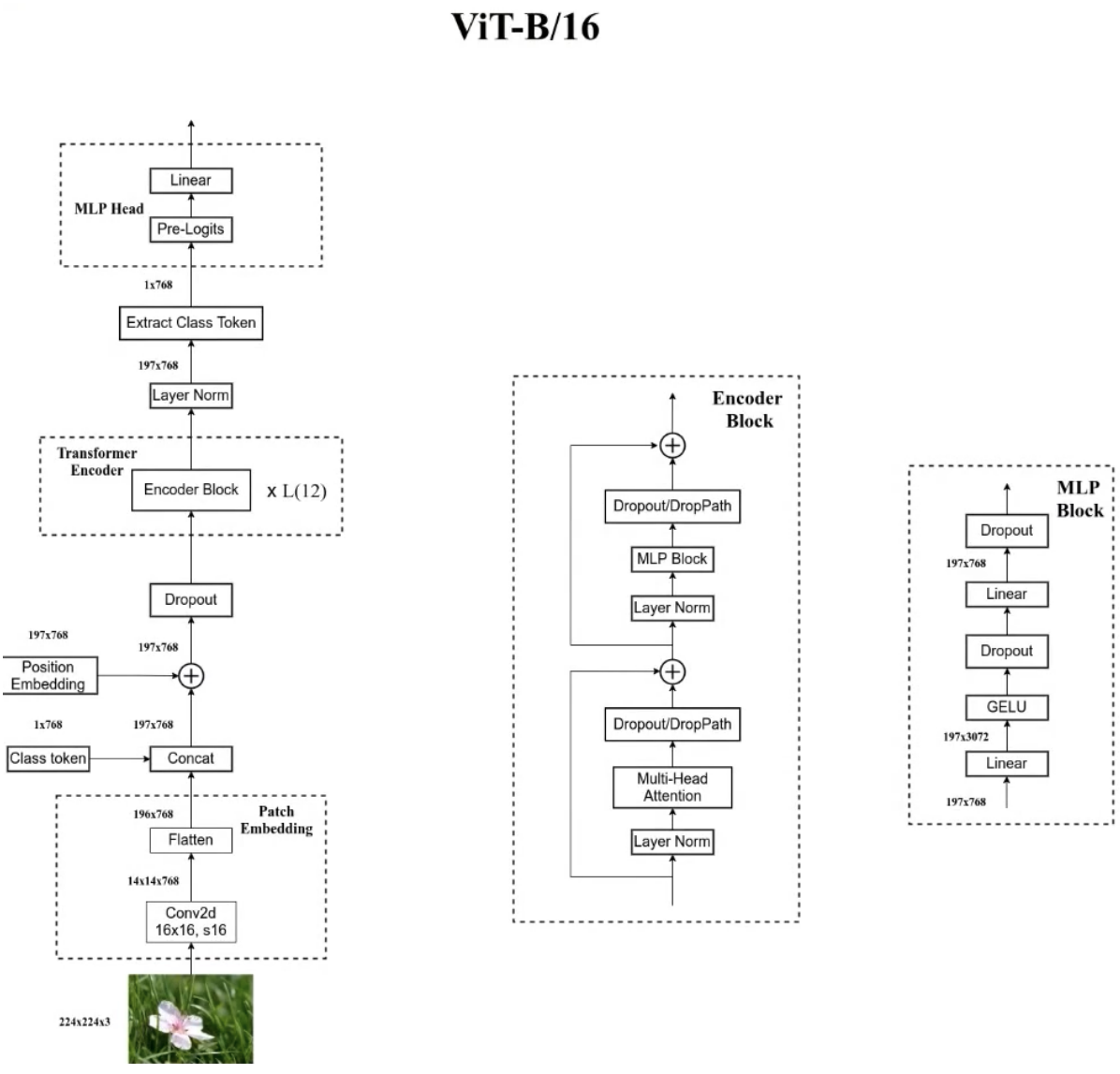


20.2.4 ViT B/16

我们来从头梳理一次 ViT B/16 的结构，假设输入图为 224×224×3.

- 首先经过一个卷积层，然后进行高度和宽度方向的展平处理。
- 紧接着 concat 一个 class token，再加上 Position Embedding 的相加操作，这里的 Position Embedding 也是可训练的参数。
- 经过 Dropout 之后输入 12 个堆叠的 Encoder Block。Encoder 输出经过 LN 得到的输出为 197×768，即是不变的。
- 然后我们提取第一个 class token 对应的输出，切片之后即变成了1×768，将其输入 MLP Head 中。

如果在 ImageNet21K 预训练的时候，Pre-Logits 就是一个全连接层，tanh 激活函数。如果是在 ImageNet1k 或者自己的数据集上的时候训练的时候，可以不要这个 Pre-Logits。



20.2.5 ViT 模型参数

我ViT B 对应的就是 ViT-Base，ViT L 对应的是 ViT-Large，ViT H 对应的是 ViT-Huge。

- patch size 是图片切片大小（源码中还有32×32 的）；
- layers 则是 encoder block 堆叠的次数；
- Hidden size 是 token 向量的长度；

- MLP size 是 Hidden size 的四倍，即 Encoder block 中 MLP block 第一个全连接层节点个数；
- Heads 则是 Multi-head Attention 中 heads 的个数。

Model	Patch Size	Layers	Hidden Size D	MLP size	Heads	Params
ViT-Base	16x16	12	768	3072	12	86M
ViT-Large	16x16	24	1024	4096	16	307M
ViT-Huge	14x14	32	1280	5120	16	632M

20.3 Hybrid 混合模型

对于CNN 和 Transformer 的混合模型。首先用传统的神经网络 backbone 来提取特征，然后再通过 ViT 模型进一步得到最终的结果。

这里的特征提取部分采用的是 ResNet59 网络，但是和原来的有所不同，第一点是采用 stdConv2d，第二点则是使用GN而非BN，第三点是将 stage4 中的 3 个 block 移动到 stage3 中。R50 backbone 的输出为 14×14×1024，然后通过 1×1卷积变为 14×14×768，然后进行展平处理就得到 token 了。之后就是和 ViT 一摸一样的了。

结果可见，混合模型比纯 transformer 模型的效果会好一些，这也是迁移学习之后的结果。在少量微调中混合模型占有，但是随着迭代次数的上升，纯 transformer 也能达到混合模型的效果，例如 14 个 epoches 时 ViT-L/16 和 Res50x1+ViT-L/16 就基本一样了。

name	Epochs	ImageNet	ImageNet ReaL	CIFAR-10	CIFAR-100	Pets	Flowers	exaFLOPs
ViT-B/32	7	80.73	86.27	98.61	90.49	93.40	99.27	55
ViT-B/16	7	84.15	88.85	99.00	91.87	95.80	99.56	224
ViT-L/32	7	84.37	88.28	99.19	92.52	95.83	99.45	196
ViT-L/16	7	86.30	89.43	99.38	93.46	96.81	99.66	783
ViT-L/16	14	87.12	89.99	99.38	94.04	97.11	99.56	1567
ViT-H/14	14	88.08	90.36	99.50	94.71	97.11	99.71	4262
ResNet50x1	7	77.54	84.56	97.67	86.07	91.11	94.26	50
ResNet50x2	7	82.12	87.94	98.29	89.20	93.43	97.02	199
ResNet101x1	7	80.67	87.07	98.48	89.17	94.08	95.95	96
ResNet152x1	7	81.88	87.96	98.82	90.22	94.17	96.94	141
ResNet152x2	7	84.97	89.69	99.06	92.05	95.37	98.62	563
ResNet152x2	14	85.56	89.89	99.24	91.92	95.75	98.75	1126
ResNet200x3	14	87.22	90.15	99.34	93.53	96.32	99.04	3306
R50x1+ViT-B/32	7	84.90	89.15	99.01	92.24	95.75	99.46	106
R50x1+ViT-B/16	7	85.58	89.65	99.14	92.63	96.65	99.40	274
R50x1+ViT-L/32	7	85.68	89.04	99.24	92.93	96.97	99.43	246
R50x1+ViT-L/16	7	86.60	89.72	99.18	93.64	97.03	99.40	859
R50x1+ViT-L/16	14	87.12	89.76	99.31	93.89	97.36	99.11	1668

Table 6: Detailed results of model scaling experiments. These correspond to Figure 5 in the main paper. We show transfer accuracy on several datasets, as well as the pre-training compute (in exaFLOPs).

20.4 补充：GELU激活函数

GELU激活函数介绍

GELU (Gaussian Error Linear Units) 是一种基于高斯误差函数的激活函数，相较于 ReLU 等激活函数，GELU 更加平滑，有助于提高训练过程的收敛速度和性能。下面是 GELU 激活层的数学表达式：

$$\mathrm{GELU}(x) = x * P(X \leqslant x) = x * \Phi(x)$$

其中$\Phi(x)$表示正态分布的累积分布函数，即：

$$\Phi(x) = \frac{1}{2} \cdot \left(1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$$

其中$erf(x)$表示高斯误差函数。该函数可进一步表示为

$$x * P(X \leqslant x) = x \int_{-\infty}^{x} \frac{e^{-\frac{(X-\mu)^2}{2\sigma^2}}}{\sqrt{2\pi}\sigma} dX$$

其中μ μμ和σσσ分别代表正太分布的均值和标准差.由于上面这个函数是无法直接计算的，研究者在研究过程中发现 GELU 函数可以被近似地表示为

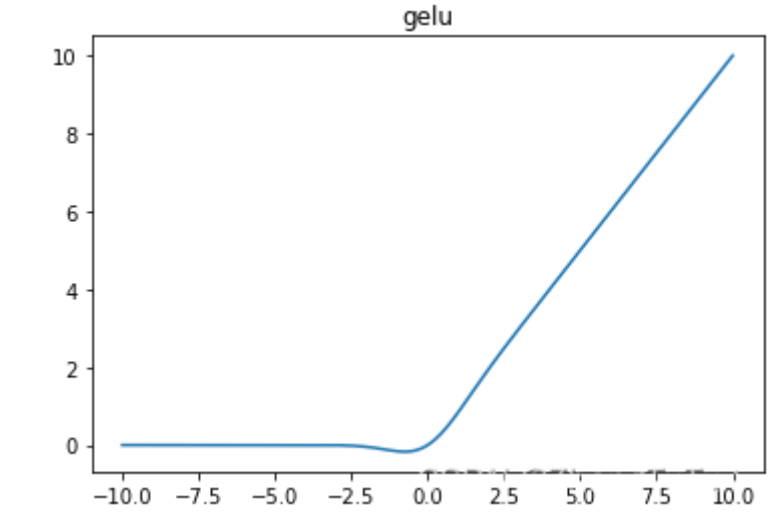
$$\mathrm{GELU}(x) = 0.5x \left[1 + \tanh\left(\sqrt{\frac{2}{\pi}}\left(x + 0.047715x^3\right)\right)\right]$$

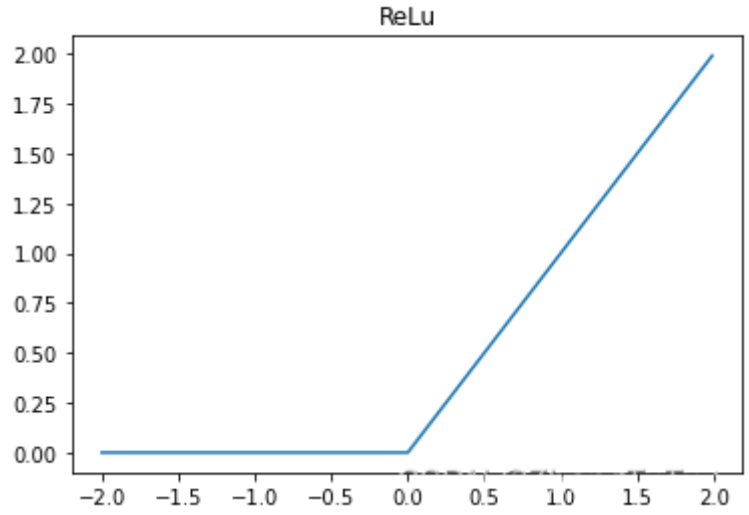
或者

$$\mathrm{GELU}(x) = x * \sigma(1.702x)$$

相较于 ReLU 函数，GELU 函数在负值区域又一个非零的梯度，从而避免了死亡神经元的问题。另外，GELU 在 0 附近比 ReLU 更加平滑，因此在训练过程中更容易收敛。值得注意的是，GELU 的计算比较复杂，因此需要消耗更多的计算资源。

GeLu和ReLu函数图像对比





各自的优势和缺点
相对于 Sigmoid 和 Tanh 激活函数，ReLU 和 GeLU 更为准确和高效，因为它们在神经网络中的梯度消失问题上表现更好。梯度消失通常发生在深层神经网络中，意味着梯度的值在反向传播过程中逐渐变小，导致网络梯度无法更新，从而影响网络的训练效果。而 ReLU 和 GeLU 几乎没有梯度消失的现象，可以更好地支持深层神经网络的训练和优化。

而 ReLU 和 GeLU 的区别在于形状和计算效率。ReLU 是一个非常简单的函数，仅仅是输入为负数时返回0，而输入为正数时返回自身，从而仅包含了一次分段线性变换。但是，ReLU 函数存在一个问题，就是在输入为负数时，输出恒为0，这个问题可能会导致神经元死亡，从而降低模型的表达能力。GeLU 函数则是一个连续的 S 形曲线，介于 Sigmoid 和 ReLU 之间，形状比 ReLU 更为平滑，可以在一定程度上缓解神经元死亡的问题。不过，由于 GeLU 函数中包含了指数运算等复杂计算，所以在实际应用中通常比 ReLU 慢。

20.5 代码

代码出处见 此处。