5 Transformer+Classification：用于分类任务的Transformer(ICLR2021)

> **论文名称：An Image is Worth 16x16 Words:Transformers for Image Recognition at Scale**

论文地址：

https://arxiv.org/abs/2010.11929
arxiv.org/abs/2010.11929

- **5.1 ViT原理分析：**

这个工作本着尽可能少修改的原则，将原版的Transformer开箱即用地迁移到分类任务上面。并且作者认为没有必要总是依赖于CNN，只用Transformer也能够在分类任务中表现很好，尤其是在使用大规模训练集的时候。同时，在大规模数据集上预训练好的模型，在迁移到中等数据集或小数据集的分类任务上以后，也能取得比CNN更优的性能。**下面看具体的方法：**

图片预处理：分块和降维

这个工作首先把 $\mathbf{x} \in H \times W \times C$ 的图像，变成一个 $\mathbf{x}_p \in N \times \left(P^2 \cdot C\right)$ 的sequence of flattened 2D patches。它可以看做是一系列的展平的2D块的序列，这个序列中一共有 $N = HW/P^2$ 个展平的2D块，每个块的维度是 $\left(P^2 \cdot C\right)$ 。其中 P 是块大小， C 是channel数。

注意作者做这步变化的意图：根据我们**之前的讲解**，Transformer希望输入一个二维的矩阵 (N, D) ，其中 N 是sequence的长度， D 是sequence的每个向量的维度，常用256。

所以这里也要设法把 $H \times W \times C$ 的三维图片转化成 (N, D) 的二维输入。

所以有： $H \times W \times C \rightarrow N \times \left(P^2 \cdot C\right), \text{where } N = HW/P^2$ 。

其中， N 是Transformer输入的sequence的长度。

代码是：

```
x = rearrange(img, 'b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1=p, p2=p)
```

具体是采用了einops库实现，具体可以参考这篇博客。

科技猛兽：PyTorch 70.einops：优雅地操作张量维度
285 赞同 · 11 评论 文章

现在得到的向量维度是： $\mathbf{x}_p \in N \times \left(P^2 \cdot C\right)$ ，**要转化成 (N, D) 的二维输入，我们还需要做一步叫做Patch Embedding的步骤。**

Patch Embedding

方法是对每个向量都做**一个线性变换（即全连接层）**，压缩后的维度为 D ，这里我们称其为Patch Embedding。

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}}; \mathbf{x}_p^1\mathbf{E}; \mathbf{x}_p^2\mathbf{E}; \cdots; \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos} \tag{5.1}$$

这个全连接层就是上式(5.1)中的 $\mathbf{E}$ ，它的输入维度大小是 $(P^2 \cdot C)$ ，输出维度大小是 D。

```
# 将3072变成dim，假设是1024
self.patch_to_embedding = nn.Linear(patch_dim, dim)
x = self.patch_to_embedding(x)
```

注意这里的绿色字体 $\mathbf{x}_{\text{class}}$ ，假设切成9个块，但是最终到Transfomer输入是10个向量，这是人为增加的一个向量。

为什么要追加这个向量?

如果没有这个向量，假设 $N = 9$ 个向量输入Transformer Encoder，输出9个编码向量，然后呢？对于分类任务而言，我应该取哪个输出向量进行后续分类呢?

不知道。干脆就再来一个向量 $\mathbf{x}_{\text{class}}(\text{vector,dim} = D)$ ，这个向量是**可学习的嵌入向量**，它和那9个向量一并输入Transfomer Encoder，输出1+9个编码向量。然后就用第0个编码向量，即 $\mathbf{x}_{\text{class}}$ 的输出进行分类预测即可。

这么做的原因可以理解为：ViT其实只用到了Transformer的Encoder，而并没有用到Decoder，而 $\mathbf{x}_{\text{class}}$ 的作用有点类似于解码器中的 $\mathbf{Query}$ 的作用，相对应的 $\mathbf{Key,Value}$ 就是其他9个编码向量的输出。

$\mathbf{x}_{\text{class}}$ 是一个可学习的嵌入向量，它的意义说通俗一点为：寻找其他9个输入向量对应的 $\mathbf{image}$ 的类别。

代码为：

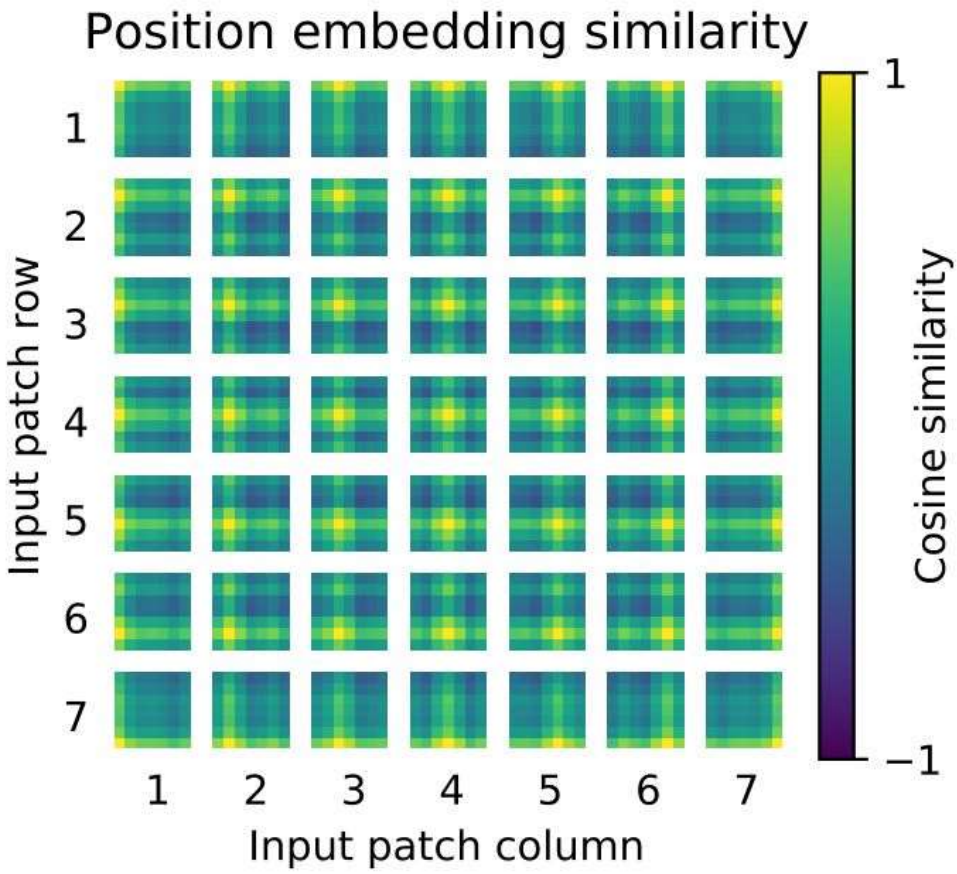
```
# dim=1024
self.cls_token = nn.Parameter(torch.randn(1, 1, dim))

# forward前向代码
# 变成(b,64,1024)
cls_tokens = repeat(self.cls_token, '() n d -> b n d', b=b)
# 跟前面的分块进行concat
# 额外追加token，变成b,65,1024
x = torch.cat((cls_tokens, x), dim=1)
```

Positional Encoding

按照Transformer的位置编码的习惯，这个工作也使用了位置编码。**引入了一个 Positional encoding $\mathbf{E}_{pos}$ 来加入序列的位置信息**，同样在这里也引入了pos_embedding，是**用一个可训练的变量**。

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}}; \mathbf{x}_p^1\mathbf{E}; \mathbf{x}_p^2\mathbf{E}; \cdots; \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos} \tag{5.2}$$

没有采用原版Transformer的 $\mathbf{sincos}$ 编码，而是直接设置为可学习的Positional Encoding，效果差不多。对训练好的pos_embedding进行可视化，如下图所示。我们发现，**位置越接近，往往具有更相似的位置编码。此外，出现了行列结构；同一行/列中的patch具有相似的位置编码。**



图：ViT的可学习的Positional Encoding

```
# num_patches=64，dim=1024,+1是因为多了一个cls开启解码标志
self.pos_embedding = nn.Parameter(torch.randn(1, num_patches + 1, dim))
```

Transformer Encoder的前向过程

$$
\begin{aligned}
\mathbf{z}_0 &= [\mathbf{x}_{\text{class}}; \mathbf{x}_p^1\mathbf{E}; \mathbf{x}_p^2\mathbf{E}; \cdots; \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos}, && \mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D},\ \mathbf{E}_{pos} \in \mathbb{R}^{(N+1) \times D} \\
\mathbf{z}'_\ell &= \mathrm{MSA}(\mathrm{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, && \ell = 1 \ldots L \\
\mathbf{z}_\ell &= \mathrm{MLP}(\mathrm{LN}(\mathbf{z}'_\ell)) + \mathbf{z}'_\ell, && \ell = 1 \ldots L \\
\mathbf{y} &= \mathrm{LN}(\mathbf{z}_L^0)
\end{aligned} \tag{5.3}
$$

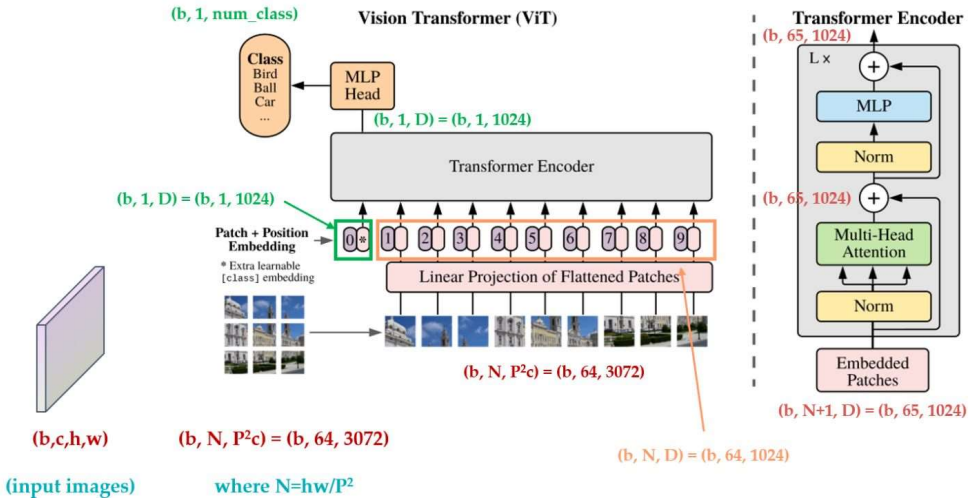
其中，第1个式子为上面讲到的Patch Embedding和Positional Encoding的过程。

第2个式子为Transformer Encoder的 **Multi-head Self-attention, Add and Norm** 的过程，重复 L 次。

第3个式子为Transformer Encoder的 **Feed Forward Network, Add and Norm** 的过程，重复 L 次。

作者采用的是没有任何改动的transformer。

最后是一个 **MLP** 的 **Classification Head** ，整个的结构只有这些，如下图所示，为了方便读者的理解，我把变量的维度变化过程标注在了图中。



图：ViT整体结构

训练方法：

先在大数据集上预训练，再迁移到小数据集上面。做法是把ViT的 **prediction head** 去掉，换成一个 $D \times K$ 的 **Feed Forward Layer** 。其中 K 为对应数据集的类别数。

当输入的图片是更大的shape时，patch size P 保持不变，则 $N = HW/P^2$ 会增大。

ViT可以处理任意 N 的输入，但是Positional Encoding是按照预训练的输入图片的尺寸设计的，所以输入图片变大之后，Positional Encoding需要根据它们在原始图像中的位置做2D插值。

最后，展示下ViT的动态过程：

Experiments：

预训练模型使用到的数据集有：

- ILSVRC-2012 ImageNet dataset：1000 classes
- ImageNet-21k：21k classes
- JFT：18k High Resolution Images

将预训练迁移到的数据集有：

- CIFAR-10/100

- Oxford-IIIT Pets
- Oxford Flowers-102
- VTAB

作者设计了3种不同答小的ViT模型，它们分别是：

DModel	Layers	Hidden size	MLP size	Heads	Params
ViT-Base	12	768	3072	12	86M
ViT-Large	24	1024	4096	16	307M
ViT-Huge	32	1280	5120	16	632M

ViT-L/16代表ViT-Large + 16 patch size P

评价指标 Metrics ：

结果都是下游数据集上经过finetune之后的Accuracy，记录的是在各自数据集上finetune后的性能。

实验1：性能对比

实验结果如下图所示，整体模型还是挺大的，而经过大数据集的预训练后，性能也超过了当前CNN的一些SOTA结果。对比的**CNN模型**主要是：

2020年ECCV的Big Transfer (BiT)模型，它使用大的ResNet进行有监督转移学习。

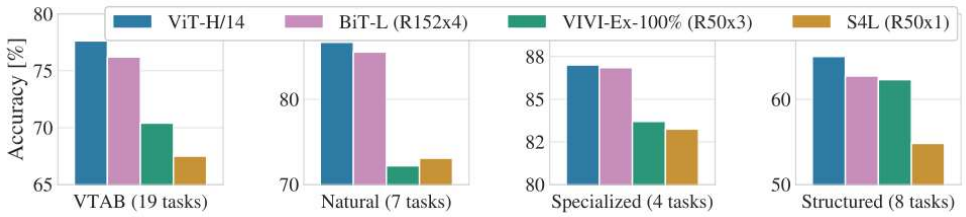
2020年CVPR的Noisy Student模型，这是一个在ImageNet和JFT300M上使用半监督学习进行训练的大型高效网络，去掉了标签。

All models were trained on TPUv3 hardware。

	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

在JFT-300M上预先训练的较小的ViT-L/16模型在所有任务上都优于BiT-L(在同一数据集上预先训练的)，同时训练所需的计算资源要少得多。 更大的模型ViT-H/14进一步提高了性能，特别是在更具挑战性的数据集上——ImageNet, CIFAR-100和VTAB数据集。 与现有技术相比，该模型预训练所需的计算量仍然要少得多。

下图为VTAB数据集在Natural, Specialized, 和Structured子任务与CNN模型相比的性能，ViT模型仍然可以取得最优。



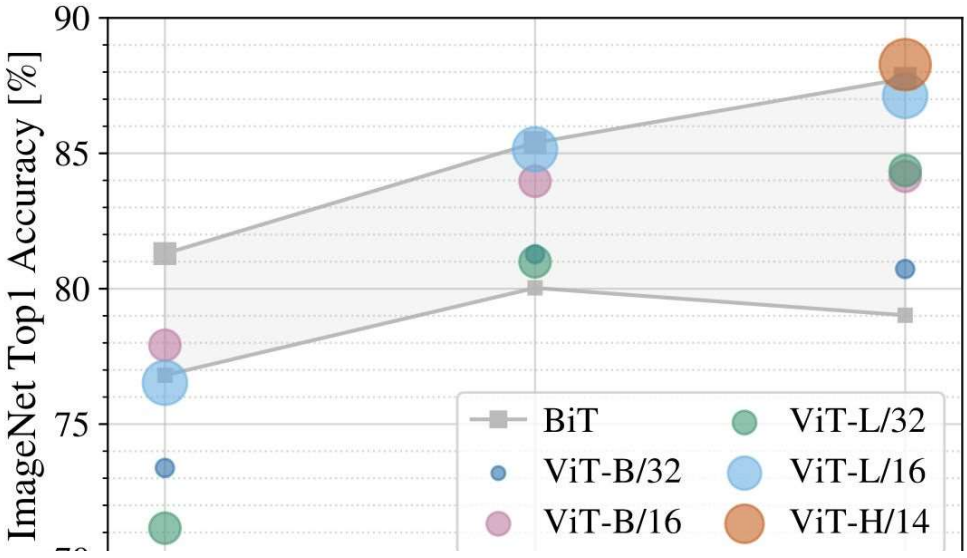
图：VTAB数据集在Natural, Specialized, 和Structured子任务与CNN模型相比的性能

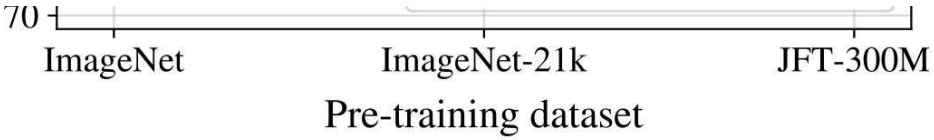
实验2：ViT对预训练数据的要求

ViT对于预训练数据的规模要求到底有多苛刻?

作者分别在下面这几个数据集上进行预训练：ImageNet, ImageNet-21k, 和JFT-300M。

结果如下图所示：





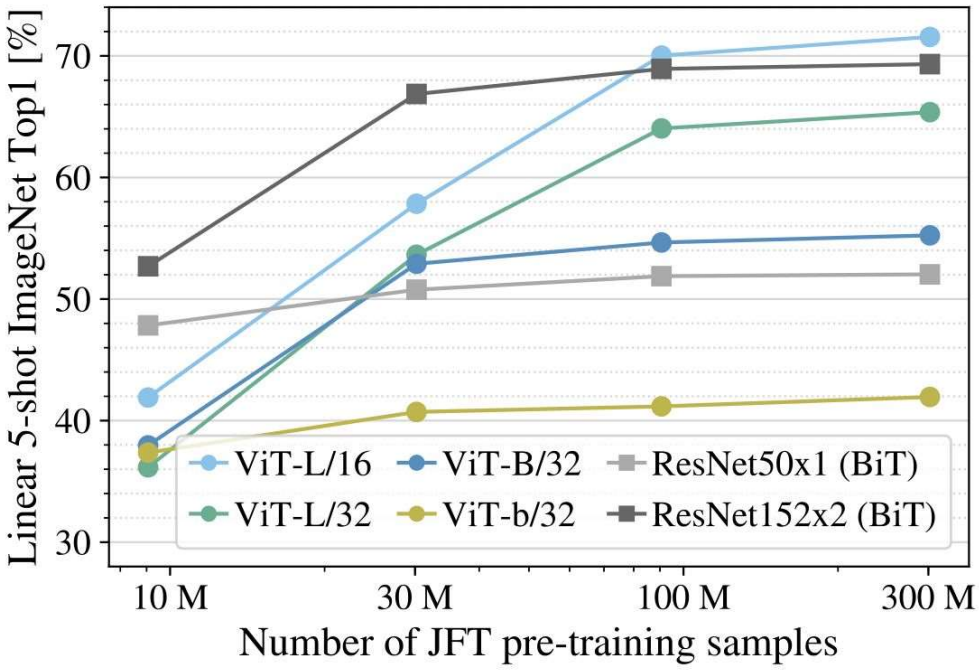
我们发现： 当在**最小数据集ImageNet**上进行预训练时，尽管进行了大量的正则化等操作，但**ViT-大模型的性能不如ViT-Base模型。**

但是有了**稍微大一点的ImageNet-21k预训练，它们的表现也差不多。**

只有到了**JFT 300M，我们才能看到更大的ViT模型全部优势。**图3还显示了不同大小的BiT模型跨越的性能区域。BiT CNNs在ImageNet上的表现优于ViT(尽管进行了正则化优化)，但在更大的数据集上，ViT超过了所有的模型，取得了SOTA。

作者还进行了一个实验： **在9M、30M和90M的随机子集以及完整的JFT300M数据集上训练模型**，结果如下图所示。 ViT在较小数据集上的计算成本比ResNet高， ViT-B/32比ResNet50稍快；它在9M子集上表现更差， 但在90M+子集上表现更好。ResNet152x2和ViT-L/16也是如此。这个结果强化了一种直觉，即：

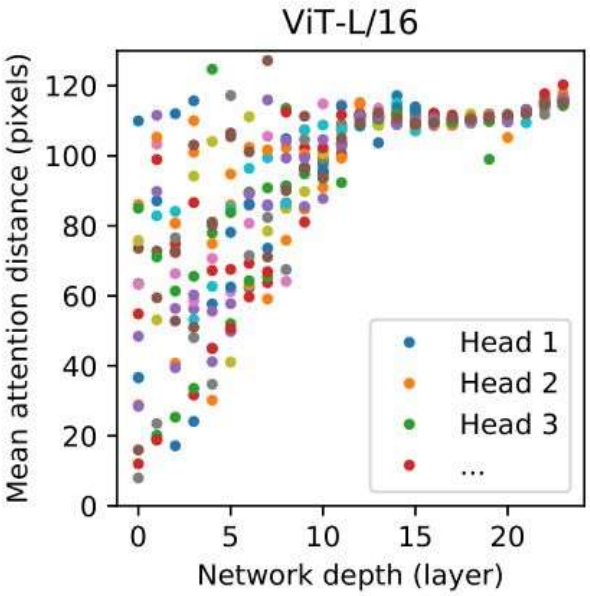
残差对于较小的数据集是有用的，但是对于较大的数据集，像attention一样学习相关性就足够了，甚至是更好的选择。



实验3：ViT的注意力机制Attention

作者还给了注意力观察得到的图片块， Self-attention使得ViT能够整合整个图像中的信息，甚至是最底层的信息。作者欲探究网络在多大程度上利用了这种能力。

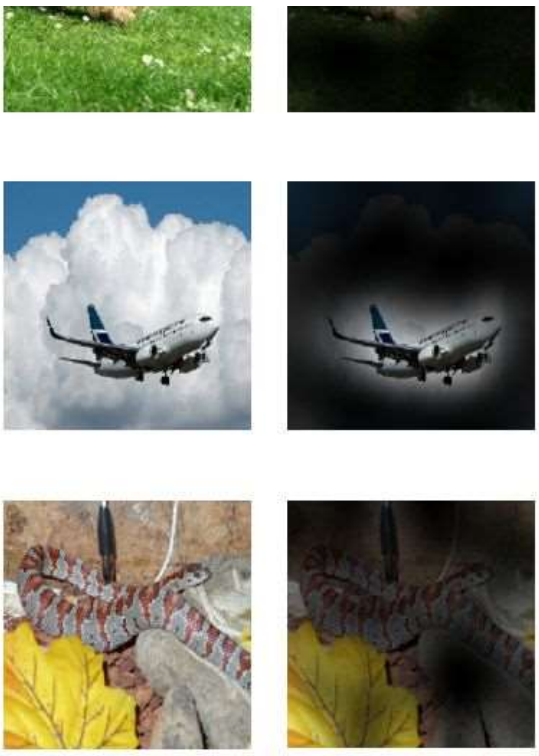
具体来说，我们根据**注意力权重**计算图像**空间中整合信息的平均距离**，如下图所示。



注意这里我们只使用了attention，而没有使用CNN，所以这里的attention distance相当于CNN的receptive field的大小。作者发现：**在最底层，**有些head也已经注意到了图像的大部分，说明模型已经可以globally地整合信息了，**说明它们负责global信息的整合。其他的head**只注意到图像的一小部分，**说明它们负责local信息的整合。Attention Distance随深度的增加而增加。**

整合局部信息的attention head在混合模型(有CNN存在)时，效果并不好，说明它可能与CNN的底层卷积有着类似的功能。

作者给出了attention的可视化，注意到了适合分类的位置：



图：attention的可视化

- **5.2 ViT代码解读：**

代码来自：

https://github.com/google-research/vision_transformer
github.com/google-research/vision_transformer

首先是介绍使用方法：

安装：

```
pip install vit-pytorch
```

使用：

```
import torch
from vit_pytorch import ViT

v = ViT(
    image_size = 256,
    patch_size = 32,
    num_classes = 1000,
    dim = 1024,
    depth = 6,
    heads = 16,
    mlp_dim = 2048,
    dropout = 0.1,
    emb_dropout = 0.1
)

img = torch.randn(1, 3, 256, 256)
mask = torch.ones(1, 8, 8).bool() # optional mask, designating which patch to attend t
```

```
preds = v(img, mask = mask) # (1, 1000)
```

> **传入参数的意义：**
> **image_size**：输入图片大小。
> **patch_size**：论文中 patch size： P 的大小。
> **num_classes**：数据集类别数。
> **dim**：Transformer的隐变量的维度。
> **depth**：Transformer的Encoder，Decoder的Layer数。
> **heads**：Multi-head Attention layer的head数。
> **mlp_dim**：MLP层的hidden dim。
> **dropout**：Dropout rate。
> **emb_dropout**：Embedding dropout rate。

> **定义残差，Feed Forward Layer 等：**

```
class Residual(nn.Module):
    def __init__(self, fn):
        super().__init__()
        self.fn = fn
    def forward(self, x, **kwargs):
        return self.fn(x, **kwargs) + x

class PreNorm(nn.Module):
    def __init__(self, dim, fn):
        super().__init__()
        self.norm = nn.LayerNorm(dim)
        self.fn = fn
    def forward(self, x, **kwargs):
        return self.fn(self.norm(x), **kwargs)

class FeedForward(nn.Module):
    def __init__(self, dim, hidden_dim, dropout = 0.):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(dim, hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, dim),
            nn.Dropout(dropout)
        )
    def forward(self, x):
        return self.net(x)
```

> Attention和Transformer，注释已标注在代码中：

```
class Attention(nn.Module):
    def __init__(self, dim, heads = 8, dim_head = 64, dropout = 0.):
        super().__init__()
        inner_dim = dim_head *  heads
        self.heads = heads
        self.scale = dim ** -0.5

        self.to_qkv = nn.Linear(dim, inner_dim * 3, bias = False)
        self.to_out = nn.Sequential(
            nn.Linear(inner_dim, dim),
            nn.Dropout(dropout)
```

```
        )

    def forward(self, x, mask = None):
# b, 65, 1024, heads = 8
        b, n, _, h = *x.shape, self.heads

# self.to_qkv(x): b, 65, 64*8*3
# qkv: b, 65, 64*8
        qkv = self.to_qkv(x).chunk(3, dim = -1)

# b, 65, 64, 8
        q, k, v = map(lambda t: rearrange(t, 'b n (h d) -> b h n d', h = h), qkv)

# dots:b, 65, 64, 64
        dots = torch.einsum('bhid,bhjd->bhij', q, k) * self.scale
        mask_value = -torch.finfo(dots.dtype).max

        if mask is not None:
            mask = F.pad(mask.flatten(1), (1, 0), value = True)
            assert mask.shape[-1] == dots.shape[-1], 'mask has incorrect dimensions'
            mask = mask[:, None, :] * mask[:, :, None]
            dots.masked_fill_(~mask, mask_value)
            del mask

# attn:b, 65, 64, 64
        attn = dots.softmax(dim=-1)

# 使用einsum表示矩阵乘法:
# out:b, 65, 64, 8
        out = torch.einsum('bhij,bhjd->bhid', attn, v)

# out:b, 64, 65*8
        out = rearrange(out, 'b h n d -> b n (h d)')

# out:b, 64, 1024
        out =  self.to_out(out)
        return out

class Transformer(nn.Module):
    def __init__(self, dim, depth, heads, dim_head, mlp_dim, dropout):
        super().__init__()
        self.layers = nn.ModuleList([])
        for _ in range(depth):
            self.layers.append(nn.ModuleList([
                Residual(PreNorm(dim, Attention(dim, heads = heads, dim_head = dim_hea
                Residual(PreNorm(dim, FeedForward(dim, mlp_dim, dropout = dropout)))
            ]))

    def forward(self, x, mask = None):
        for attn, ff in self.layers:
            x = attn(x, mask = mask)
            x = ff(x)
        return x
```

> ViT整体:

```
class ViT(nn.Module):
    def __init__(self, *, image_size, patch_size, num_classes, dim, depth, heads, mlp_
        super().__init__()
        assert image_size % patch_size == 0, 'Image dimensions must be divisible by th
        num_patches = (image_size // patch_size) ** 2
```

```
        patch_dim = channels * patch_size ** 2
        assert num_patches > MIN_NUM_PATCHES, f'your number of patches ({num_patches})
        assert pool in {'cls', 'mean'}, 'pool type must be either cls (cls token) or m

        self.patch_size = patch_size

        self.pos_embedding = nn.Parameter(torch.randn(1, num_patches + 1, dim))
        self.patch_to_embedding = nn.Linear(patch_dim, dim)
        self.cls_token = nn.Parameter(torch.randn(1, 1, dim))
        self.dropout = nn.Dropout(emb_dropout)

        self.transformer = Transformer(dim, depth, heads, dim_head, mlp_dim, dropout)

        self.pool = pool
        self.to_latent = nn.Identity()

        self.mlp_head = nn.Sequential(
            nn.LayerNorm(dim),
            nn.Linear(dim, num_classes)
        )

    def forward(self, img, mask = None):
        p = self.patch_size

# 图片分块
        x = rearrange(img, 'b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1 = p, p2 = p)

# 降维(b,N,d)
        x = self.patch_to_embedding(x)

        b, n, _ = x.shape

# 多一个可学习的x_class，与输入concat在一起，一起输入Transformer的Encoder。(b,1,d)
        cls_tokens = repeat(self.cls_token, '() n d -> b n d', b = b)
        x = torch.cat((cls_tokens, x), dim=1)

# Positional Encoding：(b,N+1,d)
        x += self.pos_embedding[:, :(n + 1)]
        x = self.dropout(x)

# Transformer的输入维度x的shape是：(b,N+1,d)
        x = self.transformer(x, mask)

# (b,1,d)
        x = x.mean(dim = 1) if self.pool == 'mean' else x[:, 0]

        x = self.to_latent(x)
        return self.mlp_head(x)
# (b,1,num_class)
```