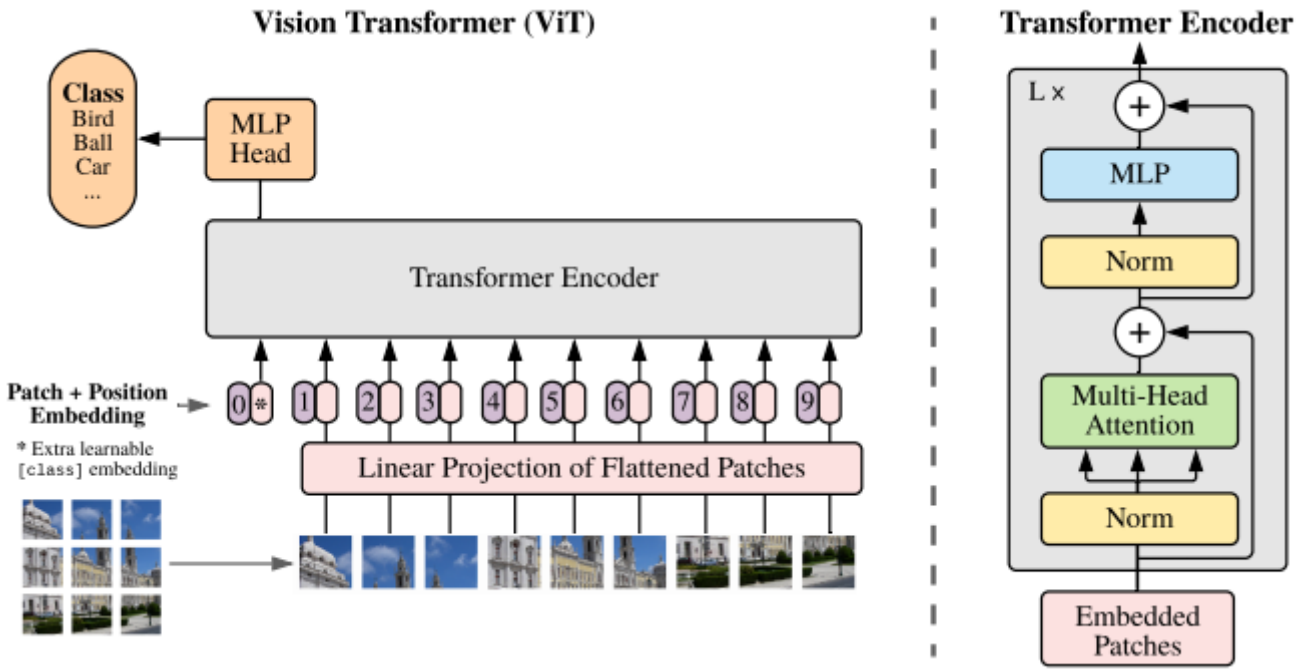
复习Transformer的同时顺便延申一下ViT。ViT原论文中最核心的结论是，**当拥有足够多的数据进行预训练的时候，ViT的表现就会超过CNN，突破transformer缺少归纳偏置的限制，可以在下游任务中获得较好的迁移效果**

把Transformer迁移到视觉工作的核心挑战：

- Transformer缺少CNNs的归纳偏差，比如平移不变性和局部受限感受野（CNN的归纳偏置是局部性和空间不变性平移等效性，即空间位置上的元素的联系/相关性近大远小，以及空间平移的不变性 (Kernel 权重共享) ）。
- CNNs是通过相似的卷积操作来提取特征，随着模型层数的加深，感受野也会逐步增加。但是由于Transformer的本质，其在计算量上会比CNNs更大。
- Transformer无法直接用于处理基于网格的数据，比如图像数据。

ViT的本质就是把把图像数据转换成序列数据，模型由三个模块组成：

- Linear Projection of Flattened Patches(Embedding层)
- Transformer Encoder
- MLP Head（最终用于分类的层结构）



1. 图像块嵌入 (Patch Embeddings)

ViT将输入图片分为多个patch（16x16），再将每个patch投影为固定长度的向量送入Transformer，后续encoder的操作和原始Transformer中完全相同。但是因为对图片分类，因此在输入序列中加入一个特殊的token，该token对应的输出即为最后的类别预测

```
class PatchEmbed(nn.Module):
    """ Image to Patch Embedding """

    def __init__(self, img_size=224, patch_size=16, in_chans=3, embed_dim=768):
        super().__init__()
        # (H, W)
        img_size = to_2tuple(img_size)
        # (P, P)
        patch_size = to_2tuple(patch_size)
        # N = (H // P) * (W // P)
        num_patches = (img_size[1] // patch_size[1]) * (img_size[0] // patch_size[0])

        self.img_size = img_size
        self.patch_size = patch_size
        self.num_patches = num_patches

        # 可训练的线性投影 - 获取输入嵌入
        self.proj = nn.Conv2d(in_chans, embed_dim, kernel_size=patch_size,
stride=patch_size)

    def forward(self, x):
        B, C, H, W = x.shape
        # FIXME look at relaxing size constraints

        assert H == self.img_size[0] and W == self.img_size[1], \
            f"Input image size ({H}*{W}) doesn't match model ({self.img_size[0]}*
{self.img_size[1]})."

        x = self.proj(x).flatten(2).transpose(1, 2)
        return x
```

例如输入图片大小为224x224，将图片分为固定大小的 patch，patch大小为16x16，则每张图像会生成 $(224 \times 224)/(16 \times 16) = 196$ 个 patch，即输入序列长度为**196**。

```
img_size = to_2tuple(img_size) # (P, P)
patch_size = to_2tuple(patch_size) # N = (H // P) * (W // P)
num_patches = (img_size[1] // patch_size[1]) * (img_size[0] // patch_size[0])
```

每个patch维度16x16x3=**768**，线性投射层的维度为$768 \times N$ (N=196)，因此输入通过线性投射层之后的维度依然为196x768，即一共有196个token，每个token的维度是768。

```
self.proj = nn.Conv2d(in_chans, embed_dim, kernel_size=patch_size, stride=patch_size)
```

用 16×16 大小的卷积核计算每个token的值，将每个patch的值进行编码。利用 `torch.flatten` 平展维度，展平起始维度和结束维度$(B, C, H, W) - (B, C, H \times W)$，之后

利用`transpose()`交换维度，因此最终的维度是197x768。到目前为止，已经通过patch embedding将一个视觉问题转化为了一个seq2seq问题

2. **可学习的嵌入** (Learnable Embedding)

类似于 BERT 的类别 token ，此处 为图像 patch 嵌入序列预设一个可学习的嵌入，该嵌入在 Vision Transformer 编码器输出的状态/特征。无论是预训练还是微调，都有一个 分类头 (Classification Head) 附加在之后，从而用于图像分类。在预训练时，分类头为一个单层 MLP；在微调时，分类头为单个线性层 (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 形式的 MLP)。

更明确地，假设将图像分为个图像块，输入到 Transformer 编码器中就有个向量，但该取哪一个向量用于分类预测呢？都不合适！一个合理的做法是手动添加一个可学习的嵌入向量作为用于分类的类别向量 ，同时与其他图像块嵌入向量一起输入到 Transformer 编码器中，最后取追加的首个可学习的嵌入向量作为类别预测结果。所以，追加的首个类别向量可理解为其他个图像块寻找的类别信息。从而，最终输入 Transformer 的嵌入向量总长度为。可学习嵌入 在训练时随机初始化，然后通过训练得到，其具体实现为：

```
### 随机初始化
self.cls_token = nn.Parameter(torch.zeros(1, 1, embed_dim))  # shape = (1, 1, D)

### 分类头 (Classifier head)
self.head = nn.Linear(self.num_features, num_classes) if num_classes > 0 else
nn.Identity()

### 前馈过程 (Forward)
B = x.shape[0]  # Batch Size

# 通过 可学习的线性投影 获取 Input Imgaes 的 Patch Embeddings (实现在 3.1 节)
x = self.patch_embed(x)  # x.shape = (B, N, D)

# 可学习嵌入 - 用于分类
cls_tokens = self.cls_token.expand(B, -1, -1)  # shape = (B, 1, D)

# 按元素相加 附带 Position Embeddings
x = x + self.pos_embed  # shape = (B, N, D) - Python 广播机制

# 按通道拼接 获取 N+1 维 Embeddings
x = torch.cat((cls_tokens, x), dim=1)  # shape = (B, N+1, D)
```

3. **位置嵌入** (Position Embeddings)

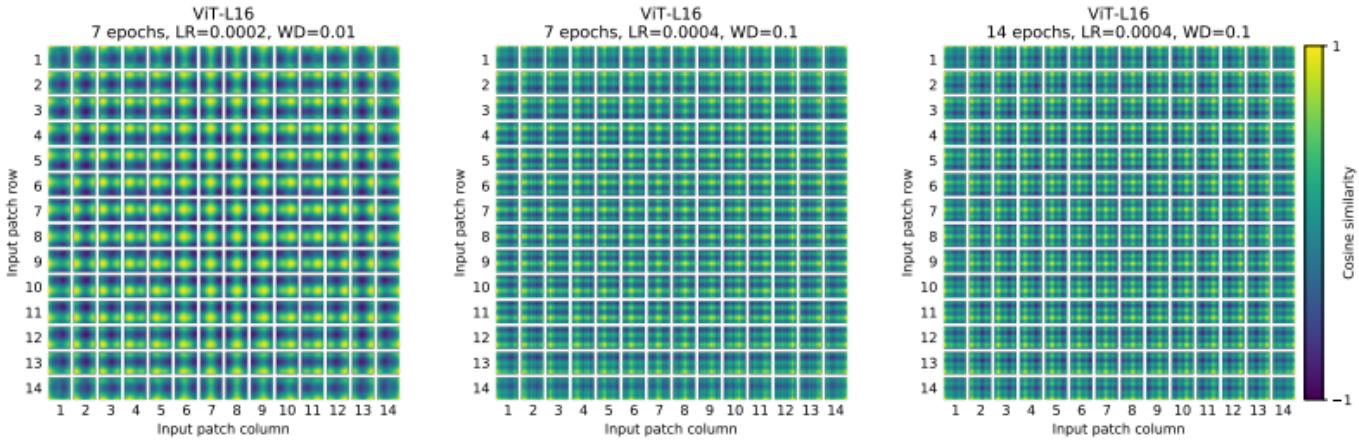
位置嵌入也被加入图像块嵌入，以保留输入图像块之间的空间位置信息。与CNNs不同，此时模型并不知道序列数据中的patches的位置信息。所以这些patches必须先追加一个位置信息，

也就是图中的带数字的向量。Transformer 需要位置嵌入来编码 patch tokens 的位置信息，这主要是由于自注意力的扰动不变性 (Permutation-invariant)，即打乱 Sequence 中 tokens 的顺序并不会改变结果。

```
# 多 +1 是为了加入 class token
# embed_dim 即 patch embed_dim
self.pos_embed = nn.Parameter(torch.zeros(1, num_patches + 1, embed_dim))

# patch emded + pos_embed : 图像块嵌入 + 位置嵌入
x = x + self.pos_embed
```

实验表明，不同的位置编码embedding对最终的结果影响不大，在Transformer原论文中使用的是固定位置编码，在ViT中使用的可学习的位置embedding 向量，将它们加到对应的输出patch embeddings上。论文中也对学习到的位置编码进行了可视化，发现相近的图像块的位置编码较相似，且同行或列的位置编码也相近

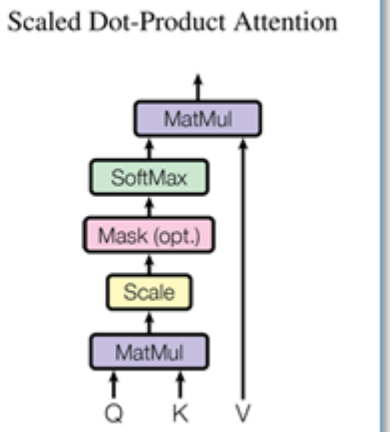


4. Transformer 编码器

将patch embedding和class token拼接起来输入标准的Transformer Encoder中。 Transformer Encoder其实就是重复堆叠Encoder Block次，主要由Layer Norm、Multi-Head Attention、Dropout和MLP Block几部分组成。

Scaled Dot-Product Attention

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

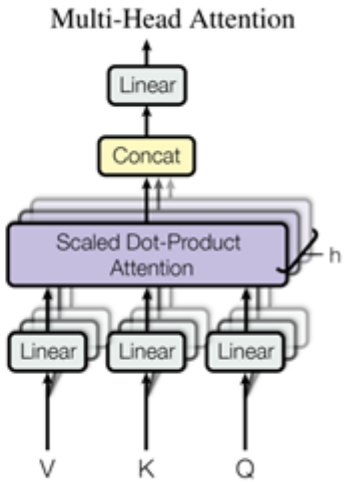


Multi-Head Attention

$$\text{head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$

- Multi-head attention allows the model to jointly attend to **information from different representation subspaces** at different positions.



- 多头注意力具体代码

```python
# MHA
class Attention(nn.Module):
    def __init__(self, dim, num_heads=8, qkv_bias=False, qk_scale=None, attn_drop=0.,
proj_drop=0.):
        super().__init__()

        self.num_heads = num_heads
        head_dim = dim // num_heads

        self.scale = qk_scale or 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)

        # 附带 dropout
        self.proj_drop = nn.Dropout(proj_drop)

    def forward(self, x):
        B, N, C = x.shape
        qkv = self.qkv(x).reshape(B, N, 3, self.num_heads, C //
self.num_heads).permute(2, 0, 3, 1, 4)
        q, k, v = qkv[0], qkv[1], qkv[2]  # make torchscript happy (cannot use tensor
as tuple)

        attn = (q @ k.transpose(-2, -1)) * self.scale
        attn = attn.softmax(dim=-1)
        attn = self.attn_drop(attn)

        x = (attn @ v).transpose(1, 2).reshape(B, N, C)
        x = self.proj(x)
        x = self.proj_drop(x)

        return x
```

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 变换成 ，第二个 FC 将特征从维度 恢复成 ，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
```

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
```

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。

5. 总结

- 归纳偏置 (Inductive bias)：

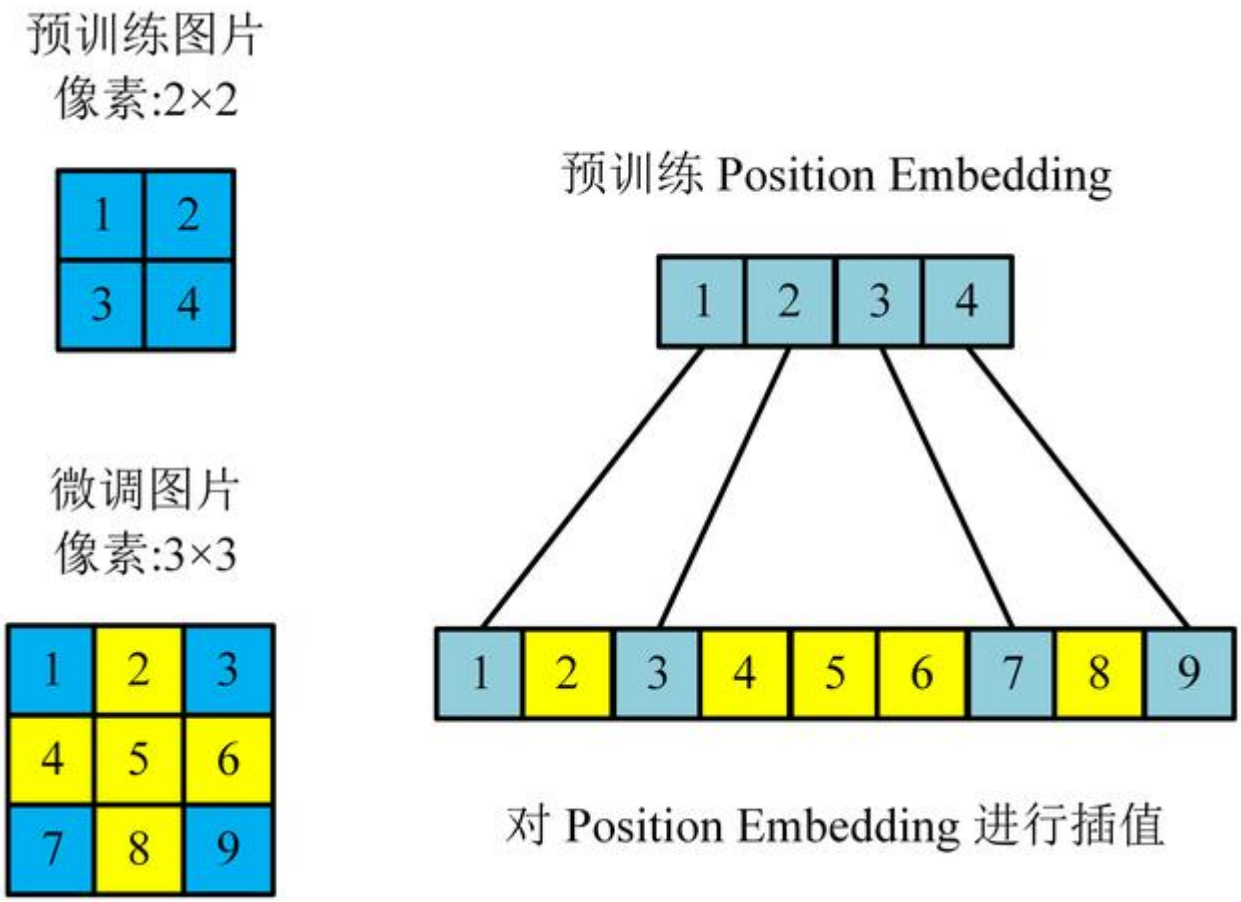
注意到，Vision Transformer 的图像特定归纳偏置比 CNN 少得多。在 CNN 中，局部性、二维邻域结构和平移等效性存在于整个模型的每一层中。而在 ViT 中，只有 MLP 层是局部和平移等变的，因为自注意力层都是全局的。二维邻域结构的使用非常谨慎：在模型开始时通过将图像切分成块，并在微调时调整不同分辨率图像的位置嵌入 (如下所述)。此外，初始化时的位置嵌入不携带有关图像块的 2D 位置的信息，图像块之间的所有空间关系都必须从头开始学习。

- 混合架构 (Hybrid Architecture)：

作为原始图像块的替代方案，输入序列可由 CNN 的特征图构成。在这种混合模型中，图像块嵌入投影 被用在 经 CNN 特征提取的块 而非原始输入图像块。作为一种特殊情况，块的空间尺寸可以为 ，这意味着输入序列是通过 简单地将特征图的空间维度展平并投影到 Transformer 维度获得的。然后，如上所述添加了分类输入嵌入和位置嵌入，再将三者组成的整体馈入 Transformer 编码器。简单来说，就是先用 CNN 提取图像特征，然后由 CNN 提取的特征图构成图像块嵌入。由于 CNN 已经将图像降采样了，所以块尺寸可为 。

6. 微调及更高分辨率（只是了解了一下）

当提供更高分辨率的图像时，需保持图像 patchs 大小相同，此时有效图像 patchs 数变多，从而有效序列长度会变长。Vision Transformer 可处理任意序列长度 (取决于内存限制)，但 预训练的位置嵌入 (pos_embed) 可能不再有意义，因为当前的位置嵌入无法与之一一对应了。因此，根据它们在原图中的位置，对预训练的位置嵌入执行 2D 插值，以扩展到微调尺寸。注意，此分辨率调整和图像 patches 提取是将有关图像 2D 结构的归纳偏置手动注入 Vision Transformer 的唯一点。



```python
def resize_pos_embed(posemb, posemb_new):
    # Rescale the grid of position embeddings when loading from state_dict. Adapted
from
    # https://github.com/google-
research/vision_transformer/blob/00883dd691c63a6830751563748663526e811cee/vit_jax/chec
kpoint.py#L224
    _logger.info('Resized position embedding: %s to %s', posemb.shape,
posemb_new.shape)
    ntok_new = posemb_new.shape[1]

    # 除去 class token 的 pos_embed
    posemb_tok, posemb_grid = posemb[:, :1], posemb[0, 1:]
    ntok_new -= 1
    gs_old = int(math.sqrt(len(posemb_grid)))
    gs_new = int(math.sqrt(ntok_new))
    _logger.info('Position embedding grid-size from %s to %s', gs_old, gs_new)

    # 把 pos_embed 变换到 2-D 维度再进行插值
    posemb_grid = posemb_grid.reshape(1, gs_old, gs_old, -1).permute(0, 3, 1, 2)
    posemb_grid = F.interpolate(posemb_grid, size=(gs_new, gs_new), mode='bilinear')
    posemb_grid = posemb_grid.permute(0, 2, 3, 1).reshape(1, gs_new * gs_new, -1)
    posemb = torch.cat([posemb_tok, posemb_grid], dim=1)

    return posemb
```