# 计算机视觉读书报告—视觉transformer及其延伸

18200100223 陶志威

**摘要** transformer结构是google在17年的《Attention Is All You Need》论文中提出，在NLP的多个任务上取得了非常好的效果，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。由于其出色性能以及对下游任务的友好性或者说下游任务仅仅微调即可得到不错效果，在计算机视觉领域不断有人尝试将transformer引入，近期也出现了一些效果不错的尝试，典型的如目标检测领域的detr和可变形detr，分类领域的vision transformer等等，因此本文将介绍transformer相关发展和理论。

**关键词** 图像分类 目标检测detr 编码器 分类vision transformer

**1 transformer简介**

NLP领域以机器翻译任务为例子，机器翻译任务是指将一种语言转换得到另一种语言，例如英语翻译为中文任务。从最上层来看，如下所示：

图片包含 文本

描述已自动生成

解码过程是顺序进行，每次仅解码出一个单词。对于CV领域来说，RNN模块构建的seq2seq算法，理解到这个程度就可以了，不需要深入探讨如何进行训练。但是上述结构其实有缺陷，具体来说是：

不论输入和输出的语句长度是什么，中间的上下文向量长度都是固定的，一旦长度过长，仅仅靠一个固定长度的上下文向量明显不合理。仅仅利用上下文向量解码，会有信息瓶颈，长度过长时候信息可能会丢失通俗理解是编码器与解码器的连接点仅仅是编码单元输出的隐含向量，其包含的信息有限，对于一些复杂任务可能信息不够，如要翻译的句子较长时，一个上下文向量可能存不下那么多信息，就会造成翻译精度的下降。

基于上述缺陷进而提出带有注意力机制Attention的seq2seq，同样可以应用于RNN、LSTM或者GRU模块中。注意力机制Attention对人类来说非常好理解，假设给定一张图片，我们会自动聚焦到一些关键信息位置，而不需要逐行扫描全图。此处的attention也是同一个意思，其本质是对输入的自适应加权，结合cv领域的senet中的se模块就能够理解了**。**

**2视觉领域应用**

在视觉领域分类vision transformer：

做法较简单，只含有编码器模块：

图形用户界面, 图示

描述已自动生成

**2.1图片分块与降维**

因为transformer的输入需要序列，所以最简单做法就是把图片切分为patch，然后拉成序列即可。假设输入图片大小是256x256，打算分成64个patch，每个patch是32x32像素

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

这个写法是采用了爱因斯坦表达式，具体是采用了einops库实现，内部集成了各种算子，rearrange就是其中一个，非常高效。不懂这种语法的请自行百度。p就是patch大小，假设输入是b,3,256,256，则rearrange操作是先变成(b,3,8x32,8x32)，最后变成(b,8x8,32x32x3)即(b,64,3072)，将每张图片切分成64个小块，每个小块长度是32x32x3=3072，也就是说输入长度为64的图像序列，每个元素采用3072长度进行编码。考虑到3072有点大，故我们先进行降维：

# 将3072变成dim，假设是1024

self.patch\_to\_embedding = nn.Linear(patch\_dim, dim)

x = self.patch\_to\_embedding(x)

仔细看论文上图，可以发现假设切成9个块，但是最终到transfomer输入是10个向量，额外追加了一个0和\_。为啥要追加？原因是我们现在没有解码器了，而是编码后直接就进行分类预测，那么该解码器就要负责一点点解码器功能，那就是：需要一个类似开启解码标志，非常类似于标准transformer解码器中输入的目标嵌入向量右移一位操作。试下如果没有额外输入，9个块输入9个编码向量输出，那么对于分类任务而言，我应该取哪个输出向量进行后续分类呢？选择任何一个都说不通，所以作者追加了一个可学习嵌入向量输入。图像领域没有这种真正意义上的离散token，有的只是一堆连续特征或者图像像素。自此现在就是变成10个向量输出，输出也是10个编码向量，然后取第0个编码输出进行分类预测即可。从这个角度看可以认为编码器多了一点点解码器功能。具体做法超级简单，0就是位置编码向量，\_是可学习的patch嵌入向量。

# dim=1024

self.cls\_token = nn.Parameter(torch.randn(1, 1, dim))

# 变成(b,64,1024)

cls\_tokens = repeat(self.cls\_token, '() n d -> b n d', b=b)

# 额外追加token，变成b,65,1024

x = torch.cat((cls\_tokens, x), dim=1)

**2.2 位置编码**

位置编码也是必不可少的，长度应该是1024，这里做的比较简单，没有采用sincos编码，而是直接设置为可学习，效果差不多

1. # num\_patches=64，dim=1024,+1是因为多了一个cls开启解码标志
2. self.pos\_embedding = nn.Parameter(torch.randn(1, num\_patches + 1, dim))

对训练好的pos\_embedding进行可视化，相邻位置有相近的位置编码向量，整体呈现2d空间位置排布一样。将patch嵌入向量和位置编码向量相加即可作为编码器输入。

x += self.pos\_embedding[:, :(n + 1)]

x = self.dropout(x)

图示

描述已自动生成

**2.3分类器head**

在编码器后接fc分类器head

self.mlp\_head = nn.Sequential(

nn.LayerNorm(dim),

nn.Linear(dim, mlp\_dim),

nn.GELU(),

nn.Dropout(dropout),

nn.Linear(mlp\_dim, num\_classes) )

# 65个输出里面只需要第0个输出进行后续分类即可

self.mlp\_head(x[:, 0])

**2.5 vision实验结果分析**

表格

描述已自动生成

图形用户界面

描述已自动生成

结论分析：cv领域应用transformer需要大量数据进行预训练，在同等数据量的情况下性能不然cnn。一旦数据量上来了，对应的训练时间也会加长很多，那么就可以轻松超越cnn。同时应用transformer，一个突出优点是可解释性比较强。

**3 目标检测detr算法**

对于目标检测任务，其要求输出给定图片中所有前景物体的类别和bbox坐标，该任务实际上是无序集合预测问题。针对该问题，detr做法非常简单：给定一张图片，经过CNN进行特征提取，然后变成特征序列输入到transformer的编解码器中，直接输出指定长度为N的无序集合，集合中每个元素包含物体类别和坐标。其中N表示整个数据集中图片上最多物体的数目，因为整个训练和测试都Batch进行，如果不设置最大输出集合数，无法进行batch训练，如果图片中物体不够N个，那么就采用no object填充，表示该元素是背景。

整个思想看起来非常简单，相比faster rcnn或者yolo算法那就简单太多了，因为其不需要设置先验anchor，超参几乎没有，也不需要nms(因为输出的无序集合没有重复情况)，并且在代码程度相比faster rcnn那就不知道简单多少倍了，通过简单修改就可以应用于全景分割任务。

**3.1 detr核心思想分析**

相比faster rcnn等做法，detr最大特点是将目标检测问题转化为无序集合预测问题。论文中特意指出faster rcnn这种设置一大堆anchor，然后基于anchor进行分类和回归其实属于代理做法即不是最直接做法，目标检测任务就是输出无序集合，而faster rcnn等算法通过各种操作，并结合复杂后处理最终才得到无序集合属于绕路了，而detr就比较纯粹了。

尽管将transformer引入目标检测领域可以避免上述各种问题，但是其依然存在两个核心操作：

无序集合输出的loss计算

针对目标检测的transformer改进

**3.2 detr算法实现细节**

下面结合代码和原理对其核心环节进行深入分析。

**3.2.1 无序集合输出的loss计算**

detr输出是包括batchx100个无序集合，每个集合包括类别和坐标信息。对于coco数据而言，作者设置类别为91(coco类别标注索引是1-91,但是实际就标注了80个类别)，加上背景一共92个类别，对于坐标分支采用4个归一化值表征即cxcywh中心点、wh坐标，然后除以图片宽高进行归一化(没有采用复杂变换策略)，故每个集合是 ，c是长度为92的分类向量，b是长度为4的bbox坐标向量。总之detr输出集合包括两个分支：分类分支shape=(b,100,92)，bbox坐标分支shape=(b,100,4)，对应的target也是包括分类target和bbox坐标target，如果不够100，则采用背景填充，计算loss时候bbox分支仅仅计算有物体位置，背景集合忽略。

匈牙利算法是一个标准优化算法，具体是组合优化算法，在scipy.optimize.linear\_sum\_assignmen函数中有实现，一行代码就可以得到最优匹配，网上解读也非常多，这里就不写细节了，该函数核心是需要输入A集合和B集合两两元素之间的连接权重，基于该重要性进行内部最优匹配，连接权重大的优先匹配。

匈牙利算法核心是需要提供输入A集合和B集合两两元素之间的连接权重，这里就是要输入N个输出集合和M个gt bbox之间的关联程度，如下所示

图片包含 文本

描述已自动生成

而Lbox具体是



需要计算M个gt bbox和N个输出集合两两之间的广义距离，距离越近表示越可能是最优匹配关系，也就是两者最密切。

**3.3针对目标检测的transformer改进**

图示

描述已自动生成

(1) cnn骨架特征提取

骨架网络可以是任何一种，作者选择resnet50，将最后一个stage即stride=32的特征图作为编码器输入。由于resnet仅仅作为一个小部分且已经经过了imagenet预训练，故和常规操作一样，会进行如下操作：

resnet中所有BN都固定，即采用全局均值和方差

resnet的stem和第一个stage不进行参数更新，即parameter.requires\_grad\_(False)

backbone的学习率小于transformer,lr\_backbone=1e-05,其余为0.0001

(2) 编码器设计和输入

编码器结构设计没有任何改变，但是输入改变了。

a) 位置编码需要考虑2d空间

由于图像特征是2d特征，故位置嵌入向量也需要考虑xy方向。前面说过编码方式可以采用sincos，也可以设置为可学习，本文采用的依然是sincos模式，和前面说的一样，但是需要考虑xy两个方向(前面说的序列只有x方向)。

考虑了xy方向同时编码，每个方向各编码128维向量，这种编码方式更符合图像特定。

还有一个细节需要注意：原始transformer的n个编码器输入中，只有第一个编码器需要输入位置编码向量，但是detr里面对每个编码器都输入了同一个位置编码向量，论文中没有写为啥要如此修改。

b) QKV处理逻辑不同

作者设置编码器一共6个，并且位置编码向量仅仅加到QK中，V中没有加入位置信息，这个和原始做法不一样，原始做法是QKV都加上了位置编码，论文中也没有写为啥要如此修改。

**总结下和原始transformer编码器不同的地方：**

输入编码器的位置编码需要考虑2d空间位置

位置编码向量需要加入到每个编码器中

在编码器内部位置编码仅仅和QK相加，V不做任何处理

经过6个编码器forward后，输出shape为(h//32xw//32,b,256)。

(3) 解码器设计和输入

解码器结构设计没有任何改变，但是输入也改变了。

a) 新引入Object queries

object queries(shape是(100,256))可以简单认为是输出位置编码,其作用主要是在学习过程中提供目标对象和全局图像之间的关系,相当于全局注意力，必不可少非常关键。代码形式上是可学习位置编码矩阵。和编码器一样，该可学习位置编码向量也会输入到每一个解码器中。我们可以尝试通俗理解：object queries矩阵内部通过学习建模了100个物体之间的全局关系，例如房间里面的桌子旁边(A类)一般是放椅子(B类)，而不会是放一头大象(C类)，那么在推理时候就可以利用该全局注意力更好的进行解码预测输出。

# num\_queries=100,hidden\_dim=256

self.query\_embed = nn.Embedding(num\_queries, hidden\_dim)

论文中指出object queries作用非常类似faster rcnn中的anchor，只不过这里是可学习的，不是提前设置好的。

b) 位置编码也需要

编码器环节采用的sincos位置编码向量也可以考虑引入，且该位置编码向量输入到每个解码器的第二个Multi-Head Attention中，后面有是否需要该位置编码的对比实验。

c) QKV处理逻辑不同

解码器一共包括6个，和编码器中QKV一样，V不会加入位置编码。上述说的三个操作，只要看下网络结构图就一目了然了。

d) 一次解码输出全部无序集合

和原始transformer顺序解码操作不同的是，detr一次就把N个无序框并行输出了(因为任务是无序集合，做成顺序推理有序输出没有很大必要)。为了说明如何实现该功能，我们需要先回忆下原始transformer的顺序解码过程：输入BOS\_WORD，解码器输出i；输入前面已经解码的BOS\_WORD和i，解码器输出am...，输入已经解码的BOS\_WORD、i、am、a和student，解码器输出解码结束标志位EOS\_WORD,每次解码都会利用前面已经解码输出的所有单词嵌入信息。现在就是一次解码，故只需要初始化时候输入一个全0的查询向量A，类似于BOS\_WORD作用，然后第一个解码器接受该输入A，解码输出向量作为下一个解码器输入，不断推理即可，最后一层解码输出即为我们需要的输出，不需要在第二个解码器输入时候考虑BOS\_WORD和第一个解码器输出。

**总结下和原始transformer解码器不同的地方：**

额外引入可学习的Object queries，相当于可学习anchor，提供全局注意力

编码器采用的sincos位置编码向量也需要输入解码器中，并且每个解码器都输入。QKV处理逻辑不同

不需要顺序解码，一次即可输出N个无序集合

**4 总结**

detr整体做法非常简单，基本上没有改动原始transformer结构，其显著优点是：不需要设置啥先验，超参也比较少，训练和部署代码相比faster rcnn算法简单很多，理解上也比较简单。但是其缺点是：改了编解码器的输入，在论文中也没有解释为啥要如此设计，而且很多操作都是实验对比才确定的，比较迷。算法层面训练epoch次数远远大于faster rcnn(300epoch)，在同等epoch下明显性能不如faster rcnn，而且训练占用内存也大于faster rcnn。

本文从transformer发展历程入手，并且深入介绍了transformer思想和实现细节；最后结合计算机视觉领域的几篇有典型代表文章进行分析。

**5 参考文献**

[1] End-to-End Object Detection with Transformers arXiv:2005.12872

[2] An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale

arXiv:2010.11929

[3] https://github.com/lucidrains/vit-pytorch

[4]https://github.com/jadore801120/attention-is-all-you-need-pytorch

[5]https://github.com/facebookresearch/detr

[6]A Survey on Visual Transformer [arXiv:2012.12556](https://arxiv.org/abs/2012.12556)