Vision Transformer 源码解析

Dexing Huang
dxhuang@bupt.edu.cn
Beijing University of Posts and Telecommunications

日期：2021 年 12 月 7 日

1 写在前面

本代码是前人实现的 ViT 源码[1], 本人只是对其进行了简单的学习和总结. 本代码复现的是发表在 ICLR 2021 上的论文[2]. 在理解完论文之后再来看这个代码还是比较轻松的, B 站上也有许多讲解 ViT 源码的视频, 其中我参考了这位[3]的思路, 感觉他讲的确实很清晰, 在这里推荐一下. 最后, 本文中的所有代码都放在[4]上, 欢迎 star.

2 总览

ViT 源码主要包括上面这几个文件, 下面按模型训练到测试的流程介绍一下每个文件的含义, 本文重点关注模型的构建, 训练测试以及数据集建立等细节可以参见上面提到的 up 主视频.

- ViT_model.py & ViT_model_test.py
  这个文件应该是代码最重要的部分, 这两个文件没什么区别, 只是 test 文件我做了一些修改读起来稍微简单, 它的作用是构建一个 ViT 模型
- train_model.py
  对模型进行训练, utils.py 和 MyDataset.py 为训练脚本提供一些训练以及数据集处理划分的接口
- inference.py
  训练完模型后使用模型对新的数据进行推理

[1]https://github.com/WZMIAOMIAO/deep-learning-for-image-processing/tree/master/tensorflow_classification/vision_transformer

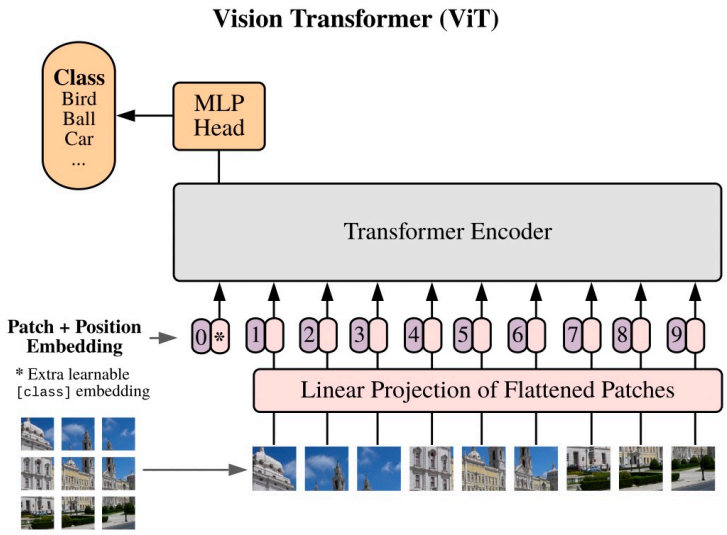
[2]Dosovitskiy A, Beyer L, Kolesnikov A, et al. An image is worth 16x16 words: Transformers for image recognition at scale[J]. arXiv preprint arXiv:2010.11929, 2020.

[3]https://space.bilibili.com/18161609

[4]https://github.com/Dxhuang-CASIA/model

3 ViT_model.py

首先来回顾一下 Vision Transformer 的总体流程, ViT 的结构图如下所示. 下面的说明以 ViT-Base 的分类问题为例, 其他情况同理. 输入的图片格式为 $C \times H \times W(3 \times 224 \times 224)$. 首先将输入的图片进行PatchEmbed, 将其转化为 $N \times dim(196 \times 768)$ 的序列. 然后加入Position Embedding以及Class Token, 将序列转化为 197×768 的序列输入 Transformer Encoder. 然后进入一个个解码器的 Block, Transformer Encoder 是由数个 Block 堆叠而成的, 输出仍然是 197×768. 接下来会输出一个 $197 \times features$ 的序列, 取其中的第一个向量, 送入全连接层进行分类处理.



3.1 PatchEmbed

这一步需要将输入图像 $3 \times 224 \times 224$ 转化为 196×768 的序列, 使用的 patchsize 大小是 16, 首先使用 $kernelsize = 16, stride = 16, channel = 768$ 的卷积核对图片进行采样, 先得到 $768 \times 14 \times 14$ 的张量.

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

然后将得到的张量的最后两个维度合并, 然后最后两个维度转置得到 196×768 的目标序列

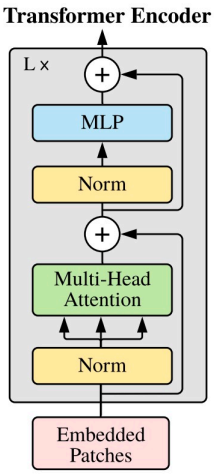
```
input = self.proj(input).flatten(2).transpose(1, 2) # 196 * 768
```

下面是添加 Position Embedding 以及 Class Token, 因为这两个东西不足以构成一个模块, 因此放在后面 $Vision\_Transformer$ 中一并实现.

3.2 Block

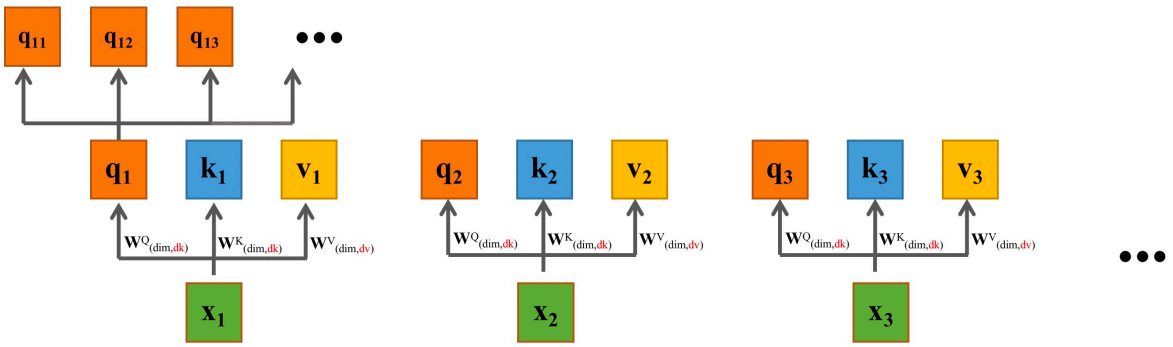
Transformer Encoder Block 的结构如下所示, 这部分输入与输出形状是相同的, 因此理论上可以堆叠任意个 Block(不过没必要). 由两大部分组成, 分别是 MSA 以及 MLP. 中间加了一些 LayerNorm 以及残差连接. 值得注意的是, 在每个输出与残差块相加之前都会经过一个 Dropout

操作, 下图中并没有画出. 但是本代码使用的是 Dropath 操作而非 Dropout, 这是因为在后面的实践中发现这个方法行性能更好, 下面对两个主要模块进行介绍.



3.2.1 MSA

先来简单过一下 MSA 的过程, 首先是输入形状为 $N \times dim(197 \times 784)$ 的序列, 然后分别通过 W^Q, W^K, W^V 三个转化矩阵变成 Q, K, V 三个矩阵, 接着进行分头操作. 如下所示.



数据先是被转化为 $N \times d_k, N \times d_k, N \times d_v$ 的矩阵, 然后在最后一个维度上根据设置的 head 的大小进行均分. 那么就变成 $N \times d_k/head, N \times d_k/head, N \times d_v/head$, 然后进行

$$Attention(Q_i, K_i, V_i) = Softmax(\frac{Q_i K_i^T}{\sqrt{d_k/head}})V_i \tag{1}$$

输出的是 head 个 $N \times d_v/head$ 维的矩阵, 然后将其按列堆叠, 得到 $N \times d_v$ 的矩阵, 再将其乘上一个转化矩阵 W^O 得到输出, 显然要得到 $N \times dim$ 的输出, $W^O \in \mathbb{R}^{d_v \times dim}$.

在本模型中, $d_v = d_k = dim$. 得到输出后再经过一个 Dropout 层得到本 Block 的输出, 这样重复堆叠数次. 在代码实现中, 并不是将 Q, K, V 分别做转换, 而是一次性进行转化, 因此这个转换的形状 $dim \times dim \Rightarrow dim \times 3 * dim$, 部分代码如下

```
self.qkv = nn.Linear(dim, dim * 3, bias = qkv_bias)
qkv = self.qkv(input).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]

attn = (q @ k.transpose(-2, -1)) * self.scale
attn = attn.softmax(dim = -1) # 每一行

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

3.2.2 MLP

MLP 结构十分简单, 只含一个隐藏层, 先将 $N \times dim \Rightarrow N \times hidden\_features$, 再将 $N \times hidden\_features \Rightarrow N \times dim$, 中间夹了一个 Dropout.

```
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)
```

3.2.3 Blcok 的搭建

有了 MSA 以及 MLP 这两块积木后, 就可以搭建一个 Block 了. 根据 Blcok 的图示即可, 不过要记得加上 Droppath. 为了简介起见, MSA 中的参数用 param_MSA 替代.

```
# 声明积木
self.norm1 = norm_layer(dim)
self.attn = MSA(param_MSA)
self.drop_path = DropPath(drop_path_ratio) if drop_path_ratio > 0. else nn.
    Identity()
self.norm2 = norm_layer(dim)
mlp_hidden_dim = int(dim * mlp_ratio)
self.mlp = MLP(dim, mlp_hidden_dim, act_layer = act_layer, drop = drop_ratio)

# 前向传播
input = input + self.drop_path(self.attn(self.norm1(input)))
input = input + self.drop_path(self.mlp(self.norm2(input)))
```

3.3 Vision_Transformer

得到了 Block 和 PatchEmbed 之后, 可以看到 ViT 的几块大积木就都有了, 下面开始构建 ViT 的完整模型.

数据先是经过 PatchEmbed 层得到 196×768 的序列.

```
# 声明
self.patch_embed = embed_layer(patch_embed_param)
# 前向传播
input = self.patch_embed(input)
```

紧接着在数据的首部添加一个 class token 形状是 1×768, 序列变为 197×768, 然后再对这个序列进行位置编码, 然后连接一个 Dropout 层输入 Encoder.

```
self.cls_token = nn.Parameter(torch.zeros(1, 1, embed_dim))
self.pos_embed = nn.Parameter(torch.zeros((1, num_patches + self.num_tokens,
    embed_dim)))

cls_token = self.cls_token.expand(input.shape[0], -1, -1)
input = torch.cat((cls_token, input), dim = 1)
input = self.pos_drop(input + self.pos_embed)
```

Encoder 是由 depth 个 Block 堆叠而成, 并且每个 Blcok 的 Droppath 概率各有不同.

```
self.blocks = nn.Sequential(*[
        Block(Blcok_param)
        for i in range(depth)
    ])
input = self.blocks(input)
```

随后 Block 的输出经过一个 LayerNorm, 然后取第一个输出, 在这里可以看一下, 输出的形状是 $1 \times dim$, 在这里可以直接使用全连接层将 $dim \Rightarrow class\_num$, 或者加入一个 representation layer, 先 $dim \Rightarrow num\_features$, 再 $num\_features \Rightarrow class\_num$.

```
# 声明
if representation_size and not distilled:
    self.has_logits = True
    self.num_features = representation_size
    self.pre_logits = nn.Sequential(OrderedDict([
        ("fc", nn.Linear(embed_dim, representation_size)),
        ("act", nn.Tanh())
    ]))
else:
    self.has_logits = False
    self.pre_logits = nn.Identity()

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

# 前向传播
return self.pre_logits(input[:, 0])
input = self.head(input)
```

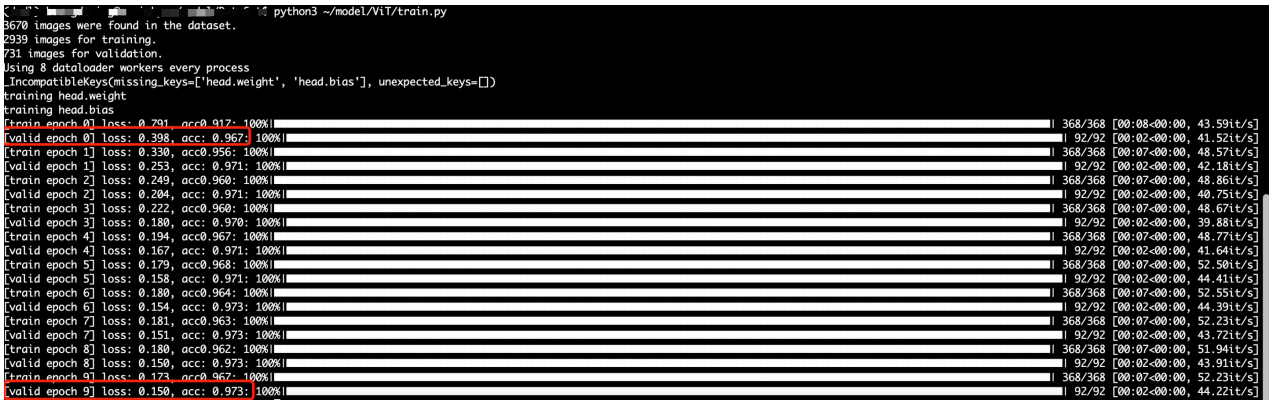
这样就完成了 ViT 的整个过程, 通过下面函数就可以得到 ViT-B/16 模型.

```
def vit_base_patch16_224_in21k(num_classes: int = 21843, has_logits: bool =
    True):
    model = Vision_Transformer(image_size = 224,
                               patch_size = 16,
                               embed_dim = 768,
                               depth = 12,
```

```
                                num_heads = 12,
                                representation_size = 768 if has_logits else
                                    None,
                                num_classes = num_classes)
    return model
```

4 训练 & 测试

ViT 模型属实是太大了, 如果从头开始训练十分费时, 而且论文也说了需要大数据训练在进行迁移学习才能得到不错的效果, 因此在训练时先下载了 ViT-B/16 在 ImageNet-21K 上预训练的模型, 再进行迁移学习, 固定前面所有层再微调最后的分类全连接层. 其实训练一个 epoch 后在验证集的正确率就达到了 97%, 具体训练的参数见代码, 下面是训练的结果, 总共训练了 10 个 epoch.



在网上选取一张图片测试, 结果如下

```
class: daisy          prob: 0.00481
class: dandelion      prob: 0.00467
class: roses          prob: 0.00506
class: sunflowers     prob: 0.98
class: tulips         prob: 0.00559
```