# Ch1.长文详解Transformer PyTorch实现与预训练

**[LeonYi](https://www.zhihu.com/people/lei-yi-43-4" \t "https://zhuanlan.zhihu.com/p/_blank)** 四川大学 计算机技术硕士 编辑于 2024-06-15 00:13・IP 属地广东

<https://zhuanlan.zhihu.com/p/686687252>

### 目录

一、Transformer概述

二、Transformer结构与实现

2.0 Tokenizer

2.1、嵌入表示层

2.1、多头自注意力(Multi-Head-self-Attention)

2.3、前馈网络

2.4、残差连接和层规一化

2.5、编码器

2.6、解码器

2.7、Transformer

三、Transformer训练

参考链接

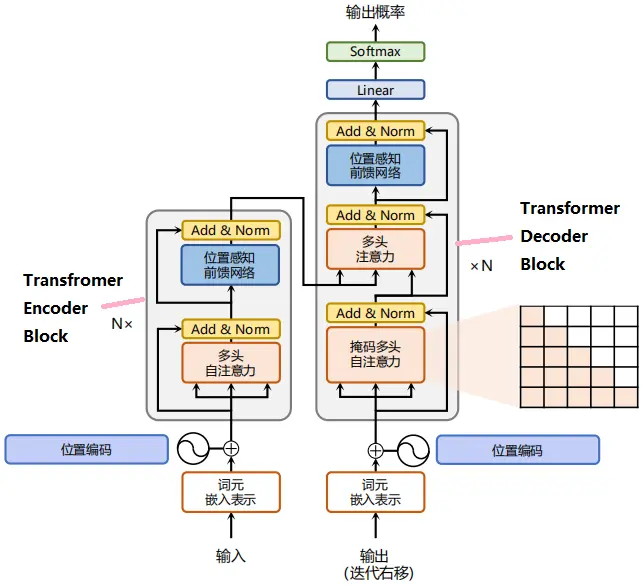
本文主要是基于动手学深度学习的相关章节的学习总结。

## **一、Transformer概述**

Transformer是由谷歌在17年提出并应用于神经机器翻译的seq2seq模型，其结构完全通过**自[注意力机制](https://zhida.zhihu.com/search?q=%E6%B3%A8%E6%84%8F%E5%8A%9B%E6%9C%BA%E5%88%B6&zhida_source=entity&is_preview=1" \t "https://zhuanlan.zhihu.com/p/_blank)**完成对源语言序列和目标语言序列的**全局依赖建模**。

Transformer由**编码器**和**解码器**构成。

下图展示了它的结构，其左侧和右侧分别对应着编码器（Encoder）和解码器（Decoder）结构，它们均由若干个基本的 **Transformer Encoder/Decoder Block**（N×表示N次堆叠）。



Transformer结构，图片源于《大规模语言模型：从理论到实践 -- 张奇、桂韬、郑锐、黄萱菁》

## **二、Transformer结构与实现**

### **2.0 Tokenizer**

对于翻译问题而言，输入文本序列和输出文本序列通常是两种不同的语言。

当然也可以推广到，text2code, text2sql以及其他奇奇怪怪的思路。输入文本序列和输出文本序列将通过tokenzier进行token化。

token化本质是一种分词，它不仅有压缩语料的目的，而且好的tokenzier应该可以极大程度避免OOV的情况。Transformer使用了BPE算法（一种很朴素思想的压缩算法）。

BPE算法的思想就进行最小程度的分词（字或字母，甚至byte级）后，初始化词表，迭代统计地最高频次的相邻token pair，例如<o,k>或<你，好>并进行合并再加入词表，从而让高频连续的序列片段尽量用越少的编码越好（原来"ok"要单独占2个token，合并后它只占1个token <ok>, 提升了语料的压缩率；有点类似于Huffman编码的贪心思想）。

### **2.1、嵌入表示层**

对于输入文本序列，首先通过**输入嵌入层（Input Embedding）**将每个token转换为其相对应的向量表示。通常直接对每个token创建一个向量表示。

**注意**：在翻译问题中，有两个token词库，分别对应源语言和目标语言。

由于Transfomer中没有任何信息能表示单词间的相对位置关系，故需在[词嵌入](https://zhida.zhihu.com/search?q=%E8%AF%8D%E5%B5%8C%E5%85%A5&zhida_source=entity&is_preview=1" \t "https://zhuanlan.zhihu.com/p/_blank)中加入**位置编码（Positional Encoding）**。

具体来说，序列中每一个token所在的位置都对应一个向量。这一向量会与单词表示对应相加并送入到后续模块中做进一步处理。**在训练的过程当中，模型会自动地学习到如何利用这部分位置信息。**

**2.1.1、词元嵌入层**

初始化词汇表（基于原始语料使用**BPE(Byte Pair Encoding)学习词表，**得到最终的token vocab）

self**.**embedding **=** nn**.**Embedding(vocab\_size, num\_hiddens)

**2.1.2、位置编码**

为了使用序列的顺序信息，通过在输入表示中添加**位置编码（positional encoding）**来注入绝对的或相对的位置信息。

位置编码可以通过学习得到也可以直接固定得到。接下将介绍基于正弦函数和余弦函数的固定位置编码。

假设输入X∈Rn×d表示包含一个序列中n个词元的d维嵌入表示。 位置编码使用相同形状的位置嵌入矩阵P∈Rn×d 输出 X+P， 矩阵第行pos、第列2i和列上2i+1的元素为：

p(pos,2i)=sin⁡(pos100002i/d),p(pos,2i+1)=cos⁡(pos100002i/d).

其中，pos表示单词所在的位置，2i和2i+1表示位置编码向量中的对应维度，d 则对应位置编码的总维度。

通过上面这种方式计算位置编码有这样几个好处：

* 首先，正余弦函数的范围是在 [-1,+1]，**导出的位置编码与原词嵌入相加不会使得结果偏离过远而破坏原有单词的语义信息**。
* 其次，依据三角函数的基本性质，可以得知第pos+k个位置的编码是第pos个位置的编码的线性组合，这就意味着**位置编码中蕴含着单词之间的距离信息**。

**class** **PositionalEncoding**(nn**.**Module):

"""位置编码"""

**def** \_\_init\_\_(self, num\_hiddens, dropout, max\_len**=**1000):

super(PositionalEncoding, self)**.**\_\_init\_\_()

self**.**dropout **=** nn**.**Dropout(dropout)

*# 创建一个足够长的P*

self**.**P **=** torch**.**zeros((1, max\_len, num\_hiddens))

X **=** torch**.**arange(max\_len, dtype**=**torch**.**float32)**.**reshape(

**-**1, 1) **/** torch**.**pow(10000, torch**.**arange(

0, num\_hiddens, 2, dtype**=**torch**.**float32) **/** num\_hiddens)

self**.**P[:, :, 0::2] **=** torch**.**sin(X)

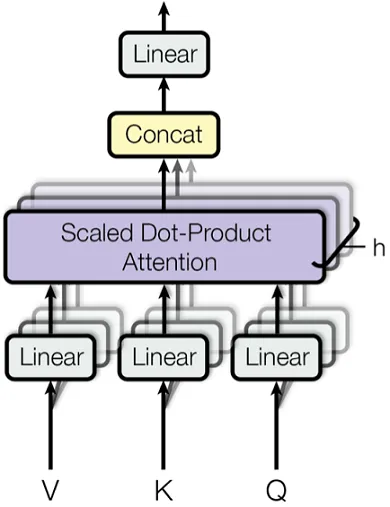
self**.**P[:, :, 1::2] **=** torch**.**cos(X)

**def** **forward**(self, X):

X **=** X **+** self**.**P[:, :X**.**shape[1], :]**.**to(X**.**device)

**return** self**.**dropout(X)

### **2.1、多头自注意力(Multi-Head-self-Attention)**



多头自注意力

**2.2.1、自注意力机制**

**1） 缩放点积注意力（scaled dot-product attention）**

假设有查询向量(query) q∈R1×d 和 键向量(key) k∈R1×d ，查询向量和键向量点积的结果即为注意力得分。

a(q,k)=qk⊤

将缩放点积注意力推广到批量矩阵形势，其公式为：

Z=softmax(QK⊤d)V∈Rn×d

其中，Q∈Rm×d、K∈Rn×d、V∈Rn×d。

考虑到在d过大时，点积值较大会使得后续Softmax操作溢出导致梯度爆炸，不利于模型优化。故将注意力得分除以d进行缩放。

注：当m=1时，就是传统的注意力机制（1个q, 多个k,v）。

**import** math**import** torch**from** torch **import** nn

**class** **DotProductAttention**(nn**.**Module):

"""缩放点积注意力"""

**def** \_\_init\_\_(self, dropout, **\*\***kwargs):

super(DotProductAttention, self)**.**\_\_init\_\_(**\*\***kwargs)

self**.**dropout **=** nn**.**Dropout(dropout)

*# queries的形状：(batch\_size，查询的个数，d)*

*# keys的形状：(batch\_size，“键－值”对的个数，d)*

*# values的形状：(batch\_size，“键－值”对的个数，值的维度)*

*# valid\_lens的形状:(batch\_size，)或者(batch\_size，查询的个数)*

**def** **forward**(self, queries, keys, values, valid\_lens**=**None):

d **=** queries**.**shape[**-**1]

*# 设置transpose\_b=True为了交换keys的最后两个维度*

scores **=** torch**.**bmm(queries, keys**.**transpose(1,2)) **/** math**.**sqrt(d)

self**.**attention\_weights **=** masked\_softmax(scores, valid\_lens)

**return** torch**.**bmm(self**.**dropout(self**.**attention\_weights), values)

为批量处理数据或在自回归处理时避免信息泄露等情况，**在Token序列中填充[mask]Token，从而使一些值不纳入注意力汇聚计算**。

这里可指定一个有效序列长度（即Token个数）， 以便在计算softmax时过滤掉超出指定范围的位置。

注：该缩放点积注意力的实现使用了dropout进行正则化。

**masked\_softmax函数**实现了掩码softmax操作（masked softmax operation）， 其中任何超出有效长度的位置都被掩蔽并置为0（将掩码位置的注意力系数变为无穷小−inf，Softmax后的值为一个接近0的值）

**def** **masked\_softmax**(X, valid\_lens):

"""通过在最后一个轴上掩蔽元素来执行softmax操作"""

*# X:3D张量，valid\_lens:1D或2D张量*

**if** valid\_lens **is** None:

**return** nn**.**functional**.**softmax(X, dim**=-**1)

**else**:

shape **=** X**.**shape

**if** valid\_lens**.**dim() **==** 1:

valid\_lens **=** torch**.**repeat\_interleave(valid\_lens, shape[1])

**else**:

valid\_lens **=** valid\_lens**.**reshape(**-**1)

*# 最后一轴上被掩蔽的元素使用一个非常大的负值替换，从而其softmax输出为0*

X **=** sequence\_mask(X**.**reshape(**-**1, shape[**-**1]), valid\_lens,

value**=-**1e9)

**return** nn**.**functional**.**softmax(X**.**reshape(shape), dim**=-**1)

**def** **sequence\_mask**(X, valid\_len, value**=**0):

"""在序列中屏蔽不相关的项"""

maxlen **=** X**.**size(1)

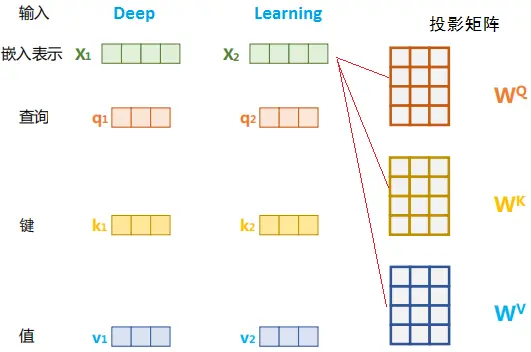
mask **=** torch**.**arange((maxlen), dtype**=**torch**.**float32,

device**=**X**.**device)[None, :] **<** valid\_len[:, None]

X[**~**mask] **=** value

**return** X

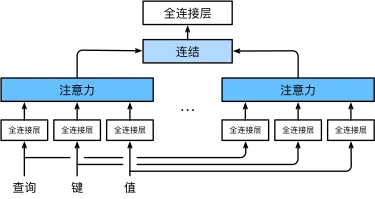
**2）自注意力**



当n=m时，且Q、K、V均源于输入X∈Rn×d经过不同的线性变换时，缩放点积注意力即推广为自注意力。

这时，每个查询都会关注所有的键值对并生成一个注意力输出。 由于查询、键和值来自同一组输，故称为**Self-Attention**。

**2.2.2、多头自注意力**



多头注意力：多个头连结然后线性变换

**class** **MultiHeadAttention**(nn**.**Module):

"""多头注意力"""

**def** \_\_init\_\_(self, key\_size, query\_size, value\_size, num\_hiddens,

num\_heads, dropout, bias**=**False, **\*\***kwargs):

super(MultiHeadAttention, self)**.**\_\_init\_\_(**\*\***kwargs)

self**.**num\_heads **=** num\_heads

self**.**attention **=** DotProductAttention(dropout)

self**.**W\_q **=** nn**.**Linear(query\_size, num\_hiddens, bias**=**bias)

self**.**W\_k **=** nn**.**Linear(key\_size, num\_hiddens, bias**=**bias)

self**.**W\_v **=** nn**.**Linear(value\_size, num\_hiddens, bias**=**bias)

self**.**W\_o **=** nn**.**Linear(num\_hiddens, num\_hiddens, bias**=**bias)

**def** **forward**(self, queries, keys, values, valid\_lens):

*# queries，keys，values的形状:*

*# (batch\_size，查询或者“键－值”对的个数，num\_hiddens)*

*# valid\_lens　的形状:*

*# (batch\_size，)或(batch\_size，查询的个数)*

*# 经过变换后，输出的queries，keys，values　的形状:*

*# (batch\_size\*num\_heads，查询或者“键－值”对的个数，*

*# num\_hiddens/num\_heads)*

queries **=** transpose\_qkv(self**.**W\_q(queries), self**.**num\_heads)

keys **=** transpose\_qkv(self**.**W\_k(keys), self**.**num\_heads)

values **=** transpose\_qkv(self**.**W\_v(values), self**.**num\_heads)

**if** valid\_lens **is** **not** None:

*# 在轴0，将第一项（标量或者矢量）复制num\_heads次，*

*# 然后如此复制第二项，然后诸如此类。*

valid\_lens **=** torch**.**repeat\_interleave(

valid\_lens, repeats**=**self**.**num\_heads, dim**=**0)

*# output的形状:(batch\_size\*num\_heads，查询的个数，*

*# num\_hiddens/num\_heads)*

output **=** self**.**attention(queries, keys, values, valid\_lens)

*# output\_concat的形状:(batch\_size，查询的个数，num\_hiddens)*

output\_concat **=** transpose\_output(output, self**.**num\_heads)

**return** self**.**W\_o(output\_concat)

为了使多个头的计算并行，上面的**MultiHeadAttention类**将使用下面定义的两个转置函数。

具体来说，transpose\_output函数反转了transpose\_qkv函数的操作。

```python**def** **transpose\_qkv**(X, num\_heads):

"""为了多注意力头的并行计算而变换形状"""

*# 输入X的形状:(batch\_size，查询或者“键－值”对的个数，num\_hiddens)*

*# 输出X的形状:(batch\_size，查询或者“键－值”对的个数，num\_heads，*

*# num\_hiddens/num\_heads)*

X **=** X**.**reshape(X**.**shape[0], X**.**shape[1], num\_heads, **-**1)

*# 输出X的形状:(batch\_size，num\_heads，查询或者“键－值”对的个数,*

*# num\_hiddens/num\_heads)*

X **=** X**.**transpose(0, 2, 1, 3)

*# 最终输出的形状:(batch\_size\*num\_heads,查询或者“键－值”对的个数,*

*# num\_hiddens/num\_heads)*

**return** X**.**reshape(**-**1, X**.**shape[2], X**.**shape[3])

**def** **transpose\_output**(X, num\_heads):

"""逆转transpose\_qkv函数的操作"""

X **=** X**.**reshape(**-**1, num\_heads, X**.**shape[1], X**.**shape[2])

X **=** X**.**transpose(0, 2, 1, 3)

**return** X**.**reshape(X**.**shape[0], X**.**shape[1], **-**1)

构建MultiHeadAttention

num\_hiddens, num\_heads **=** 100, 5attention **=** MultiHeadAttention(num\_hiddens, num\_hiddens, num\_hiddens, num\_hiddens, num\_heads, 0.5)**print**(attention)

batch\_size, num\_queries, valid\_lens **=** 2, 4, torch**.**tensor([3, 2])X **=** torch**.**ones((batch\_size, num\_queries, num\_hiddens))attention(X, X, X, valid\_lens)**.**shape

注: Transformer提出于2017年，在2024早已大放异彩，为了优化LLM下的效率问题，各种MulltiQueryAttention(多个Query共享一组KV)，GroupHeadAttention（将Query分成K组，每一组共享一组KV）被提出。

### **2.3、前馈网络**

**位置感知的前馈网络**对序列中的所有位置的表示进行变换时使用的是同一个2层全连接网络，故称其为**position-wise的前馈网络**。

FFN(x)=Relu(xW1+b1)W2+b2

在下面的实现中，输入X的形状（批量大小，时间步数或序列长度，隐单元数或特征维度）将被一个两层的感知机转换成形状为（批量大小，时间步数，ffn\_num\_outputs）的输出张量。

**class** **PositionWiseFFN**(nn**.**Module):

"""基于位置的前馈网络"""

**def** \_\_init\_\_(self, ffn\_num\_input, ffn\_num\_hiddens, ffn\_num\_outputs,

**\*\***kwargs):

super(PositionWiseFFN, self)**.**\_\_init\_\_(**\*\***kwargs)

self**.**dense1 **=** nn**.**Linear(ffn\_num\_input, ffn\_num\_hiddens)

self**.**relu **=** nn**.**ReLU()

self**.**dense2 **=** nn**.**Linear(ffn\_num\_hiddens, ffn\_num\_outputs)

**def** **forward**(self, X):

**return** self**.**dense2(self**.**relu(self**.**dense1(X)))

### **2.4、残差连接和层规一化**

**add&norm组件**是由残差连接和紧随其后的层规一化组成的，它被用来进一步提升训练的稳定性。

**1）残差连接**

残差连接引入输入直接到输出的通路，便于梯度回传从而缓解在优化过程中由于网络过深引起的[梯度消失问题](https://zhida.zhihu.com/search?q=%E6%A2%AF%E5%BA%A6%E6%B6%88%E5%A4%B1%E9%97%AE%E9%A2%98&zhida_source=entity&is_preview=1" \t "https://zhuanlan.zhihu.com/p/_blank)。

xl+1=f(xl)+xl

**2）层归一化**

层归一化（Layer Normalization）是基于特征维度进行规范化，将数据进行标准化（乘以缩放系数、加上平移系数，保留其非线性能力。

LN(x)=α(x−μσ)+β

层归一化可以有效地缓解优化过程中潜在的不稳定、收敛速度慢等问题。

以下代码对比不同维度的层规范化和批量规范化的效果。

ln **=** nn**.**LayerNorm(2)bn **=** nn**.**BatchNorm1d(2)X **=** torch**.**tensor([[1, 2], [2, 3]], dtype**=**torch**.**float32)*# 在训练模式下计算X的均值和方差***print**('layer norm:', ln(X), '\nbatch norm:', bn(X))

层归一化实现

**class** **NormLayer**(nn**.**Module):

**def** \_\_init\_\_(self, d\_model, eps **=** 1e-6):

super()**.**\_\_init\_\_()

self**.**size **=** d\_model

*# 层归一化包含两个可以学习的参数*

self**.**alpha **=** nn**.**Parameter(torch**.**ones(self**.**size))

self**.**bias **=** nn**.**Parameter(torch**.**zeros(self**.**size))

self**.**eps **=** eps

**def** **forward**(self, x):

norm **=** self**.**alpha **\*** (x **-** x**.**mean(dim**=-**1, keepdim**=**True)) \

**/** (x**.**std(dim**=-**1, keepdim**=**True) **+** self**.**eps) **+** self**.**bias

**return** norm

使用残差连接和层规一化来实现AddNorm类

**class** **AddNorm**(nn**.**Module):

"""残差连接后进行层规范化"""

**def** \_\_init\_\_(self, normalized\_shape, dropout, **\*\***kwargs):

super(AddNorm, self)**.**\_\_init\_\_(**\*\***kwargs)

self**.**dropout **=** nn**.**Dropout(dropout)

self**.**ln **=** nn**.**LayerNorm(normalized\_shape)

**def** **forward**(self, X, Y):

**return** self**.**ln(self**.**dropout(Y) **+** X)

### **2.5、编码器**

现在可以基于编码器的基础组件实现编码器的一个层。

下面的**EncoderBlock类**包含两个子层：**多头自注意力**和**基于位置的前馈网络**，这两个子层都使用了残差连接和紧随的层规一化。

**class** **EncoderBlock**(nn**.**Module):

"""Transformer编码器块"""

**def** \_\_init\_\_(self, key\_size, query\_size, value\_size, num\_hiddens,

norm\_shape, ffn\_num\_input, ffn\_num\_hiddens, num\_heads,

dropout, use\_bias**=**False, **\*\***kwargs):

super(EncoderBlock, self)**.**\_\_init\_\_(**\*\***kwargs)

self**.**attention **=** MultiHeadAttention(

key\_size, query\_size, value\_size, num\_hiddens, num\_heads, dropout,

use\_bias)

self**.**addnorm1 **=** AddNorm(norm\_shape, dropout)

self**.**ffn **=** PositionWiseFFN(

ffn\_num\_input, ffn\_num\_hiddens, num\_hiddens)

self**.**addnorm2 **=** AddNorm(norm\_shape, dropout)

**def** **forward**(self, X, valid\_lens):

Y **=** self**.**addnorm1(X, self**.**attention(X, X, X, valid\_lens))

**return** self**.**addnorm2(Y, self**.**ffn(Y))

下面的**Transformer编码器**中，堆叠了num\_layers个EncoderBlock类的实例。

由于这里使用的是值范围在-1和1之间的固定位置编码，因此通过学习得到的输入的嵌入表示的值需要先乘以嵌入维度的平方根进行重新缩放，然后再与位置编码相加。

**class** **TransformerEncoder**(Encoder):

"""Transformer编码器"""

**def** \_\_init\_\_(self, vocab\_size, key\_size, query\_size, value\_size,

num\_hiddens, norm\_shape, ffn\_num\_input, ffn\_num\_hiddens,

num\_heads, num\_layers, dropout, use\_bias**=**False, **\*\***kwargs):

super(TransformerEncoder, self)**.**\_\_init\_\_(**\*\***kwargs)

self**.**num\_hiddens **=** num\_hiddens

self**.**embedding **=** nn**.**Embedding(vocab\_size, num\_hiddens)

self**.**pos\_encoding **=** PositionalEncoding(num\_hiddens, dropout)

self**.**blks **=** nn**.**Sequential()

**for** i **in** range(num\_layers):

self**.**blks**.**add\_module("block"**+**str(i),

EncoderBlock(key\_size, query\_size, value\_size, num\_hiddens,

norm\_shape, ffn\_num\_input, ffn\_num\_hiddens,

num\_heads, dropout, use\_bias))

**def** **forward**(self, X, valid\_lens, **\***args):

*# 因为位置编码值在-1和1之间，*

*# 因此嵌入值乘以嵌入维度的平方根进行缩放，*

*# 然后再与位置编码相加。*

X **=** self**.**pos\_encoding(self**.**embedding(X) **\*** math**.**sqrt(self**.**num\_hiddens))

self**.**attention\_weights **=** [None] **\*** len(self**.**blks)

**for** i, blk **in** enumerate(self**.**blks):

X **=** blk(X, valid\_lens)

self**.**attention\_weights[

i] **=** blk**.**attention**.**attention**.**attention\_weights

**return** X

### **2.6、解码器**

**1） 掩码多头注意力**

解码器的每个Transformer块的第一个自注意力子层额外增加了**注意力掩码**，对应图中的**掩码多头注意力（Masked Multi-Head Attention）**部分。

因为在翻译的过程中，编码器用于编码已知的源语言序列的信息，因而它只需要考虑如何融合上下文语义信息即可。而解码端则负责生成目标语言序列，这一自回归的过程意味着，在生成每一个单词时，仅有当前单词之前的目标语言序列是可观测的。

增加的Mask是用来避免模型在训练阶段直接看到后续的文本序列（信息泄露）进而无法得到有效地训练。

**2） 交叉注意力**

解码器端还增加了一个**多头注意力（Multi-Head Attention）模块**，使用**交叉注意力（Cross-attention）方法**，同时接收来自编码器端的输出以及当前 Transformer 块的前一个掩码注意力层的输出。

Query是通过解码器前一层的输出进行投影的，而Key和Value是使用编码器的输出进行投影的。它的作用是在翻译的过程当中，为了生成合理的目标语言序列需要观测待翻译的源语言序列是什么。

基于上述的编码器和解码器结构，待翻译的源语言文本，首先经过编码器端的每个Transformer块对其上下文语义的层层抽象，最终输出每一个源语言单词上下文相关的表示。

解码器端以自回归的方式生成目标语言文本，即在每个时间步t，根据编码器端输出的源语言文本表示，以及前 t−1 个时刻生成的目标语言文本，生成当前时刻的目标语言单词。

**class** **DecoderBlock**(nn**.**Module):

"""解码器中第i个块"""

**def** \_\_init\_\_(self, key\_size, query\_size, value\_size, num\_hiddens,

norm\_shape, ffn\_num\_input, ffn\_num\_hiddens, num\_heads,

dropout, i, **\*\***kwargs):

super(DecoderBlock, self)**.**\_\_init\_\_(**\*\***kwargs)

self**.**i **=** i

self**.**attention1 **=** MultiHeadAttention(

key\_size, query\_size, value\_size, num\_hiddens, num\_heads, dropout)

self**.**addnorm1 **=** AddNorm(norm\_shape, dropout)

self**.**attention2 **=** MultiHeadAttention(

key\_size, query\_size, value\_size, num\_hiddens, num\_heads, dropout)

self**.**addnorm2 **=** AddNorm(norm\_shape, dropout)

self**.**ffn **=** PositionWiseFFN(ffn\_num\_input, ffn\_num\_hiddens,

num\_hiddens)

self**.**addnorm3 **=** AddNorm(norm\_shape, dropout)

**def** **forward**(self, X, state):

enc\_outputs, enc\_valid\_lens **=** state[0], state[1]

*# 训练阶段，输出序列的所有词元都在同一时间处理，*

*# 因此state[2][self.i]初始化为None。*

*# 预测阶段，输出序列是通过词元一个接着一个解码的，*

*# 因此state[2][self.i]包含着直到当前时间步第i个块解码的输出表示*

**if** state[2][self**.**i] **is** None:

key\_values **=** X

**else**:

key\_values **=** torch**.**cat((state[2][self**.**i], X), axis**=**1)

state[2][self**.**i] **=** key\_values

**if** self**.**training:

batch\_size, num\_steps, \_ **=** X**.**shape

*# dec\_valid\_lens的开头:(batch\_size,num\_steps),*

*# 其中每一行是[1,2,...,num\_steps]*

dec\_valid\_lens **=** torch**.**arange(

1, num\_steps **+** 1, device**=**X**.**device)**.**repeat(batch\_size, 1)

**else**:

dec\_valid\_lens **=** None

*# 自注意力*

X2 **=** self**.**attention1(X, key\_values, key\_values, dec\_valid\_lens)

Y **=** self**.**addnorm1(X, X2)

*# 编码器－解码器注意力。*

*# enc\_outputs的开头:(batch\_size,num\_steps,num\_hiddens)*

Y2 **=** self**.**attention2(Y, enc\_outputs, enc\_outputs, enc\_valid\_lens)

Z **=** self**.**addnorm2(Y, Y2)

**return** self**.**addnorm3(Z, self**.**ffn(Z)), state

### **2.7、Transformer**

**class** **EncoderDecoder**(nn**.**Module):

"""编码器-解码器架构的基类"""

**def** \_\_init\_\_(self, encoder, decoder, **\*\***kwargs):

super(EncoderDecoder, self)**.**\_\_init\_\_(**\*\***kwargs)

self**.**encoder **=** encoder

self**.**decoder **=** decoder

**def** **forward**(self, enc\_X, dec\_X, **\***args):

enc\_outputs **=** self**.**encoder(enc\_X, **\***args)

dec\_state **=** self**.**decoder**.**init\_state(enc\_outputs, **\***args)

**return** self**.**decoder(dec\_X, dec\_state)

## **三、Transformer训练**

损失

**class** **MaskedSoftmaxCELoss**(nn**.**CrossEntropyLoss):

"""带遮蔽的softmax交叉熵损失函数"""

*# pred的形状：(batch\_size,num\_steps,vocab\_size)*

*# label的形状：(batch\_size,num\_steps)*

*# valid\_len的形状：(batch\_size,)*

**def** **forward**(self, pred, label, valid\_len):

weights **=** torch**.**ones\_like(label)

weights **=** sequence\_mask(weights, valid\_len)

self**.**reduction**=**'none'

unweighted\_loss **=** super(MaskedSoftmaxCELoss, self)**.**forward(

pred**.**permute(0, 2, 1), label)

weighted\_loss **=** (unweighted\_loss **\*** weights)**.**mean(dim**=**1)

**return** weighted\_loss

**def** **train\_seq2seq**(net, data\_iter, lr, num\_epochs, tgt\_vocab, device):

"""训练序列到序列模型"""

**def** **xavier\_init\_weights**(m):

**if** type(m) **==** nn**.**Linear:

nn**.**init**.**xavier\_uniform\_(m**.**weight)

**if** type(m) **==** nn**.**GRU:

**for** param **in** m**.**\_flat\_weights\_names:

**if** "weight" **in** param:

nn**.**init**.**xavier\_uniform\_(m**.**\_parameters[param])

net**.**apply(xavier\_init\_weights)

net**.**to(device)

optimizer **=** torch**.**optim**.**Adam(net**.**parameters(), lr**=**lr)

loss **=** MaskedSoftmaxCELoss()

net**.**train()

animator **=** d2l**.**Animator(xlabel**=**'epoch', ylabel**=**'loss',

xlim**=**[10, num\_epochs])

**for** epoch **in** range(num\_epochs):

timer **=** d2l**.**Timer()

metric **=** d2l**.**Accumulator(2) *# 训练损失总和，词元数量*

**for** batch **in** data\_iter:

optimizer**.**zero\_grad()

X, X\_valid\_len, Y, Y\_valid\_len **=** [x**.**to(device) **for** x **in** batch]

bos **=** torch**.**tensor([tgt\_vocab['<bos>']] **\*** Y**.**shape[0],

device**=**device)**.**reshape(**-**1, 1)

dec\_input **=** torch**.**cat([bos, Y[:, :**-**1]], 1) *# 强制教学*

Y\_hat, \_ **=** net(X, dec\_input, X\_valid\_len)

l **=** loss(Y\_hat, Y, Y\_valid\_len)

l**.**sum()**.**backward() *# 损失函数的标量进行“反向传播”*

d2l**.**grad\_clipping(net, 1)

num\_tokens **=** Y\_valid\_len**.**sum()

optimizer**.**step()

**with** torch**.**no\_grad():

metric**.**add(l**.**sum(), num\_tokens)

**if** (epoch **+** 1) **%** 10 **==** 0:

animator**.**add(epoch **+** 1, (metric[0] **/** metric[1],))

**print**(f'loss {metric[0] / metric[1]:.3f}, {metric[1] / timer.stop():.1f} '

f'tokens/sec on {str(device)}')

训练语料为句子对

**import** torch**from** torch **import** nn

num\_hiddens, num\_layers, dropout, batch\_size, num\_steps **=** 32, 2, 0.1, 64, 10lr, num\_epochs, device **=** 0.005, 200, d2l**.**try\_gpu()ffn\_num\_hiddens, num\_heads **=** 64, 4

train\_iter, src\_vocab, tgt\_vocab **=** d2l**.**load\_data\_nmt(batch\_size, num\_steps)

encoder **=** TransformerEncoder(

len(src\_vocab), num\_hiddens, ffn\_num\_hiddens, num\_heads, num\_layers,

dropout)decoder **=** TransformerDecoder(

len(tgt\_vocab), num\_hiddens, ffn\_num\_hiddens, num\_heads, num\_layers,

dropout)net **=** EncoderDecoder(encoder, decoder)train\_seq2seq(net, train\_iter, lr, num\_epochs, tgt\_vocab, device)

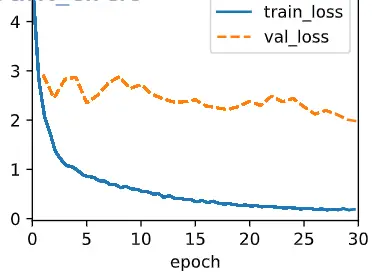
*# Test*engs **=** ['go .', "i lost .", 'he\'s calm .', 'i\'m home .']fras **=** ['va !', 'j\'ai perdu .', 'il est calme .', 'je suis chez moi .']**for** eng, fra **in** zip(engs, fras):

translation, dec\_attention\_weight\_seq **=** d2l**.**predict\_seq2seq(

net, eng, src\_vocab, tgt\_vocab, num\_steps, device, True)

**print**(f'{eng} => {translation}, ',

f'bleu {d2l.bleu(translation, fra, k=2):.3f}')

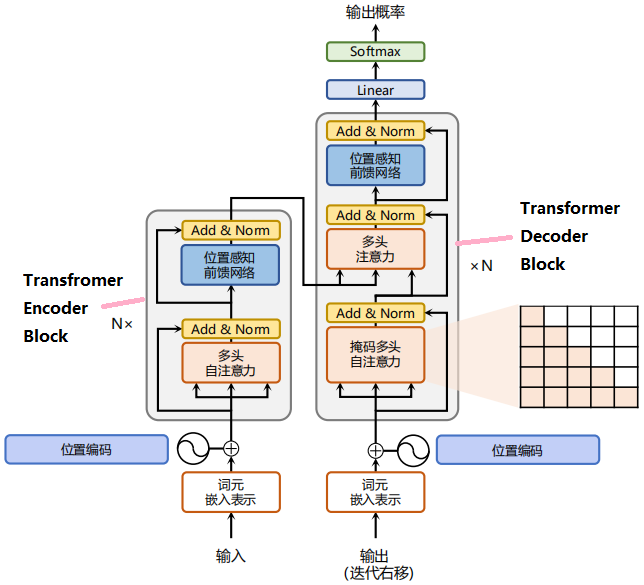


训练曲线

## **参考链接**

【1】大规模语言模型：从理论到实践， 第二章  
【2】动手学深度学习 [https://zh.d2l.ai/chapter\_attention-mechanisms/transformer.html#id7](https://link.zhihu.com/?target=https://zh.d2l.ai/chapter_attention-mechanisms/transformer.html%23id7" \t "https://zhuanlan.zhihu.com/p/_blank)  
【3】NoteBook: [https://colab.research.google.com/github/d2l-ai/d2l-pytorch-colab/blob/master/chapter\_attention-mechanisms-and-transformers/transformer.ipynb#scrollTo=74f2da96](https://link.zhihu.com/?target=https://colab.research.google.com/github/d2l-ai/d2l-pytorch-colab/blob/master/chapter_attention-mechanisms-and-transformers/transformer.ipynb%23scrollTo=74f2da96" \t "https://zhuanlan.zhihu.com/p/_blank)

博客与23年10月首发于我的博客园（更好的排版见此）：

[【大语言模型基础】Transformer模型Torch代码详解和训练实战 - LeonYi - 博客园​www.cnblogs.com/justLittleStar/p/17786071.html](https://link.zhihu.com/?target=https://www.cnblogs.com/justLittleStar/p/17786071.html" \t "https://zhuanlan.zhihu.com/p/_blank)

本文使用 [Zhihu On VSCode](https://zhuanlan.zhihu.com/p/106057556) 创作并发布

# Ch2.pytorch中的transformer

**[明年复明年](https://www.zhihu.com/people/yinxianrui" \t "https://zhuanlan.zhihu.com/p/_blank)** 发布于 2020-02-18 14:04

https://zhuanlan.zhihu.com/p/107586681

pytorch 文档中有五个相关class：

* [Transformer](https://link.zhihu.com/?target=https://pytorch.org/docs/stable/nn.html%23transformer" \t "https://zhuanlan.zhihu.com/p/_blank)
* [TransformerEncoder](https://link.zhihu.com/?target=https://pytorch.org/docs/stable/nn.html%23transformerencoder" \t "https://zhuanlan.zhihu.com/p/_blank)
* [TransformerDecoder](https://link.zhihu.com/?target=https://pytorch.org/docs/stable/nn.html%23transformerdecoder" \t "https://zhuanlan.zhihu.com/p/_blank)
* [TransformerEncoderLayer](https://link.zhihu.com/?target=https://pytorch.org/docs/stable/nn.html%23transformerencoderlayer" \t "https://zhuanlan.zhihu.com/p/_blank)
* [TransformerDecoderLayer](https://link.zhihu.com/?target=https://pytorch.org/docs/stable/nn.html%23transformerdecoderlayer" \t "https://zhuanlan.zhihu.com/p/_blank)

## 1、Transformer

**init:**

torch**.**nn**.**Transformer(d\_model**=**512, nhead**=**8, num\_encoder\_layers**=**6, num\_decoder\_layers**=**6, dim\_feedforward**=**2048, dropout**=**0.1, activation**=**'relu', custom\_encoder**=**None, custom\_decoder**=**None)

transformer模型，该结构基于论文 Attention Is All You Need

用户可以使用相应的参数构建BERT [https://arxiv.org/abs/1810.04805](https://link.zhihu.com/?target=https://arxiv.org/abs/1810.04805" \t "https://zhuanlan.zhihu.com/p/_blank)

参数：

* d\_model –编码器/解码器输入大小（默认 512）。
* nhead –多头注意力模型的头数（默认为8）。
* num\_encoder\_layers –编码器中子编码器层的数量（默认为6）。
* num\_decoder\_layers –解码器中子解码器层的数量（默认为6）。
* dim\_feedforward –前馈网络模型的中间层维度（默认= 2048）。
* dropout –默认值= 0.1。
* activation–编码器/解码器中间层的[激活函数](https://zhida.zhihu.com/search?q=%E6%BF%80%E6%B4%BB%E5%87%BD%E6%95%B0&zhida_source=entity&is_preview=1" \t "https://zhuanlan.zhihu.com/p/_blank)，relu或gelu（默认值= relu）。
* custom\_encoder –自定义编码器（默认=None）。
* custom\_decoder –自定义解码器（默认=None）。

例子：

>>> transformer\_model = nn.Transformer(nhead=16, num\_encoder\_layers=12)

>>> src = torch.rand((10, 32, 512))

>>> tgt = torch.rand((20, 32, 512))

>>> out = transformer\_model(src, tgt)

[https://github.com/pytorch/examples/tree/master/word\_language\_model](https://link.zhihu.com/?target=https://github.com/pytorch/examples/tree/master/word_language_model" \t "https://zhuanlan.zhihu.com/p/_blank)

链接是一个transformer构建语言模型的完整示例

**forward**

forward(src, tgt, src\_mask**=**None, tgt\_mask**=**None, memory\_mask**=**None, src\_key\_padding\_mask**=**None, tgt\_key\_padding\_mask**=**None, memory\_key\_padding\_mask**=**None)

* src – the sequence to the encoder (required).
* tgt – the sequence to the decoder (required).
* src\_mask – the additive mask for the src sequence (optional).
* tgt\_mask – the additive mask for the tgt sequence (optional).
* memory\_mask – the additive mask for the encoder output (optional).
* src\_key\_padding\_mask – the ByteTensor mask for src keys per batch (optional).
* tgt\_key\_padding\_mask – the ByteTensor mask for tgt keys per batch (optional).
* memory\_key\_padding\_mask – the ByteTensor mask for memory keys per batch (optional).

Shape:

* src: (S, N, E)(S,N,E ) .
* tgt: (T, N, E)(T,N,E ) .
* src\_mask: (S, S)(S,S ) .
* tgt\_mask: (T, T)(T,T ) .
* memory\_mask: (T, S)(T,S ) .
* src\_key\_padding\_mask: (N, S)(N,S ) .
* tgt\_key\_padding\_mask: (N, T)(N,T ) .
* memory\_key\_padding\_mask: (N, S)(N,S ) .

注意：[src / tgt / memory] ​​\_mask 应该用float（'-inf'）表示被遮盖的位置，而float（0.0）表示其他。 这些掩码可确保对位置 i 的预测仅取决于未掩码的位置 j，并且对批次中的每个序列均应用相同的预测。

[src / tgt / memory] ​​\_key\_padding\_mask应该是ByteTensor，其中True值是应该用float（'-inf'）掩盖的位置，而False值将保持不变。此掩码可确保在屏蔽后不会从位置i获取任何信息，并且对于批次中的每个序列都有单独的掩码。

* output:(T, N, E)(T,N,E ).

注意：由于transformer中的多头注意力机构，transformer的输出序列长度与解码的输入序列（即目标）长度相同。

其中S是源序列长度，T是目标序列长度，N是批处理大小，E是特征编号

* >>> output = transformer\_model(src, tgt, src\_mask=src\_mask, tgt\_mask=tgt\_mask)

**mask**

generate\_square\_subsequent\_mask(sz)

为该序列生成一个square mask。屏蔽的位置填充有float（'-inf'）未屏蔽的位置填充有float（0.0）

## 2、TransformerEncoder

**init**

torch**.**nn**.**TransformerEncoder(encoder\_layer, num\_layers, norm**=**None)

TransformerEncoder是N个编码器层的堆叠

参数：

* coder\_layer – TransformerEncoderLayer（）类的实例（必需）。
* num\_layers –编码器中的子编码器层数（必填）。
* norm –层归一化组件（可选）。

**>>>** encoder\_layer **=** nn**.**TransformerEncoderLayer(d\_model**=**512, nhead**=**8)**>>>** transformer\_encoder **=** nn**.**TransformerEncoder(encoder\_layer, num\_layers**=**6)**>>>** src **=** torch**.**rand(10, 32, 512)**>>>** out **=** transformer\_encoder(src)

**forward**

forward(src, mask**=**None, src\_key\_padding\_mask**=**None)

* src – the sequnce to the encoder (required).
* mask – the mask for the src sequence (optional).
* src\_key\_padding\_mask – the mask for the src keys per batch (optional).

## 3、TransformerDecoder

**init**

torch**.**nn**.**TransformerDecoder(decoder\_layer, num\_layers, norm**=**None)

transformerDecoder是N个解码器层的堆叠

* decoder\_layer – TransformerDecoderLayer（）类的实例（必需）。
* num\_layers –解码器中子解码器层的数量（必需）。
* norm –层归一化组件（可选）。

**>>>** decoder\_layer **=** nn**.**TransformerDecoderLayer(d\_model**=**512, nhead**=**8)**>>>** transformer\_decoder **=** nn**.**TransformerDecoder(decoder\_layer, num\_layers**=**6)**>>>** memory **=** torch**.**rand(10, 32, 512)**>>>** tgt **=** torch**.**rand(20, 32, 512)**>>>** out **=** transformer\_decoder(tgt, memory)

forward:

forward(tgt, memory, tgt\_mask**=**None, memory\_mask**=**None, tgt\_key\_padding\_mask**=**None, memory\_key\_padding\_mask**=**None)

* tgt – the sequence to the decoder (required).
* memory – the sequnce from the last layer of the encoder (required).
* tgt\_mask – the mask for the tgt sequence (optional).
* memory\_mask – the mask for the memory sequence (optional).
* tgt\_key\_padding\_mask – the mask for the tgt keys per batch (optional).
* memory\_key\_padding\_mask – the mask for the memory keys per batch (optional).

## 4、TransformerEncoderLayer

**init**

torch**.**nn**.**TransformerEncoderLayer(d\_model, nhead, dim\_feedforward**=**2048, dropout**=**0.1, activation**=**'relu')

TransformerEncoderLayer 由self-attn和feedforward组成，此标准编码器层基于“Attention Is All You Need”一文。

* d\_model – the number of expected features in the input (required).
* nhead – the number of heads in the multiheadattention models (required).
* dim\_feedforward – the dimension of the feedforward network model (default=2048).
* dropout – the dropout value (default=0.1).
* activation – the activation function of intermediate layer, relu or gelu (default=relu).

**>>>** encoder\_layer **=** nn**.**TransformerEncoderLayer(d\_model**=**512, nhead**=**8)**>>>** src **=** torch**.**rand(10, 32, 512)**>>>** out **=** encoder\_layer(src)

**forward**

forward(src, src\_mask**=**None, src\_key\_padding\_mask**=**None)

* src – the sequnce to the encoder layer (required).
* src\_mask – the mask for the src sequence (optional).
* src\_key\_padding\_mask – the mask for the src keys per batch (optional).

## 5、TransformerDecoderLayer

**init**

torch**.**nn**.**TransformerDecoderLayer(d\_model, nhead, dim\_feedforward**=**2048, dropout**=**0.1, activation**=**'relu')

TransformerEncoderLayer 由self-attn和feedforward组成，此标准编码器层基于“Attention Is All You Need”一文。

* d\_model – the number of expected features in the input (required).
* nhead – the number of heads in the multiheadattention models (required).
* dim\_feedforward – the dimension of the feedforward network model (default=2048).
* dropout – the dropout value (default=0.1).
* activation – the activation function of intermediate layer, relu or gelu (default=relu).

>>> decoder\_layer = nn.TransformerDecoderLayer(d\_model=512, nhead=8)

>>> memory = torch.rand(10, 32, 512)

>>> tgt = torch.rand(20, 32, 512)

>>> out = decoder\_layer(tgt, memory)

**forward**

forward(tgt, memory, tgt\_mask**=**None, memory\_mask**=**None, tgt\_key\_padding\_mask**=**None, memory\_key\_padding\_mask**=**None)

* tgt – the sequence to the decoder layer (required).
* memory – the sequnce from the last layer of the encoder (required).
* tgt\_mask – the mask for the tgt sequence (optional).
* memory\_mask – the mask for the memory sequence (optional).
* tgt\_key\_padding\_mask – the mask for the tgt keys per batch (optional).
* memory\_key\_padding\_mask – the mask for the memory keys per batch (optional).

## 6、MultiheadAttention

**init**

torch**.**nn**.**MultiheadAttention(embed\_dim, num\_heads, dropout**=**0.0, bias**=**True, add\_bias\_kv**=**False, add\_zero\_attn**=**False, kdim**=**None, vdim**=**None)

* embed\_dim – total dimension of the model.
* num\_heads – parallel attention heads.
* dropout – a Dropout layer on attn\_output\_weights. Default: 0.0.
* bias – add bias as module parameter. Default: True.
* add\_bias\_kv – add bias to the key and value sequences at dim=0.
* add\_zero\_attn – add a new batch of zeros to the key and value sequences at dim=1.
* kdim – total number of features in key. Default: None.
* vdim – total number of features in key. Default: None.
* Note – if kdim and vdim are None, they will be set to embed\_dim such that
* key, and value have the same number of features. (query,)

multihead\_attn **=** nn**.**MultiheadAttention(embed\_dim, num\_heads)attn\_output, attn\_output\_weights **=** multihead\_attn(query, key, value)

**forward**

forward(query, key, value, key\_padding\_mask**=**None, need\_weights**=**True, attn\_mask**=**None)

* key, value (query,) – map a query and a set of key-value pairs to an output. See “Attention Is All You Need” for more details.
* key\_padding\_mask – if provided, specified padding elements in the key will be ignored by the attention. This is an binary mask. When the value is True, the corresponding value on the attention layer will be filled with -inf.
* need\_weights – output attn\_output\_weights.
* attn\_mask – mask that prevents attention to certain positions. This is an additive mask (i.e. the values will be added to the attention layer).

**Shape:**

Inputs:

* query: (L, N, E)(L,N,E) where L is the target sequence length, N is the batch size, E is the embedding dimension.
* key: (S, N, E)(S,N,E) , where S is the source sequence length, N is the batch size, E is the embedding dimension.
* value: (S, N, E)(S,N,E) where S is the source sequence length, N is the batch size, E is the embedding dimension.
* key\_padding\_mask: (N, S)(N,S) , ByteTensor, where N is the batch size, S is the source sequence length.
* attn\_mask: (L, S)(L,S) where L is the target sequence length, S is the source sequence length.

Outputs:

* attn\_output: (L, N, E)(L,N,E) where L is the target sequence length, N is the batch size, E is the embedding dimension.
* attn\_output\_weights: (N, L, S)(N,L,S) where N is the batch size, L is the target sequence length, S is the source sequence length.

# **Ch3【项目实训10】基于PyTorch的Transformer**

[SophoraeT\_t](https://blog.csdn.net/SophoraeT_t" \o "SophoraeT_t" \t "https://blog.csdn.net/SophoraeT_t/article/details/_blank) 已于 2024-06-24 11:20:31

原文链接：https://blog.csdn.net/SophoraeT\_t/article/details/139874015

Transformer模型自从被引入以来，已经成为自然语言处理（NLP）领域中的重要工具。理解其底层实现对于研究和应用Transformer模型至关重要。本文将基于PyTorch的源码，深入解析Transformer模型的核心组件和实现细节。

## 1.Transformer模型概述

Transformer模型依赖于自注意力机制和前馈神经网络，摆脱了传统RNN和LSTM的顺序处理限制，实现了并行计算和更好的长距离依赖建模。我们将从PyTorch的源码出发，逐步解析其实现。

## 2.主要组件分析

Transformer模型主要由以下几个组件构成：

多头自注意力机制（Multi-Head Self-Attention）

前馈神经网络（Feed-Forward Neural Network）

编码器和解码器（Encoder and Decoder）

位置编码（Positional Encoding）

### 2.1多头自注意力机制

自注意力机制的核心在于计算查询（Query）、键（Key）和值（Value）之间的相似性，并利用这些相似性加权求和值。PyTorch中的实现如下：

class MultiheadAttention(nn.Module):

    def \_\_init\_\_(self, embed\_dim, num\_heads):

        super(MultiheadAttention, self).\_\_init\_\_()

        self.embed\_dim = embed\_dim

        self.num\_heads = num\_heads

        self.head\_dim = embed\_dim // num\_heads

        assert self.head\_dim \* num\_heads == self.embed\_dim, "embed\_dim must be divisible by num\_heads"

        self.qkv\_proj = nn.Linear(embed\_dim, 3 \* embed\_dim)

        self.o\_proj = nn.Linear(embed\_dim, embed\_dim)

        self.attn\_dropout = nn.Dropout(p=0.1)

        self.softmax = nn.Softmax(dim=-1)

    def forward(self, x):

        batch\_size, seq\_length, embed\_dim = x.size()

        qkv = self.qkv\_proj(x)

        qkv = qkv.reshape(batch\_size, seq\_length, 3, self.num\_heads, self.head\_dim)

        qkv = qkv.permute(2, 0, 3, 1, 4)  # (3, batch\_size, num\_heads, seq\_length, head\_dim)

        q, k, v = qkv[0], qkv[1], qkv[2]

        scores = torch.matmul(q, k.transpose(-2, -1)) / math.sqrt(self.head\_dim)

        attn\_weights = self.softmax(scores)

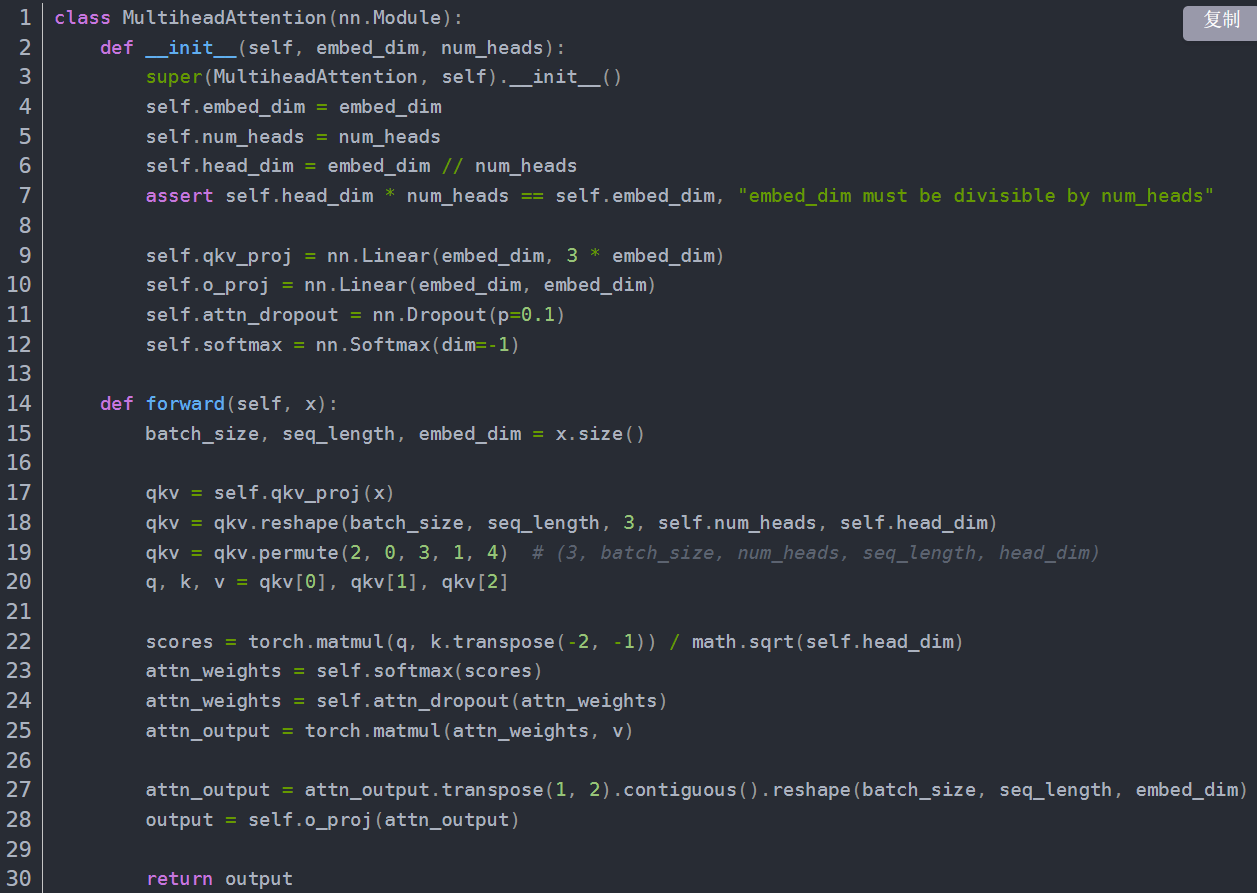
        attn\_weights = self.attn\_dropout(attn\_weights)

        attn\_output = torch.matmul(attn\_weights, v)

        attn\_output = attn\_output.transpose(1, 2).contiguous().reshape(batch\_size, seq\_length, embed\_dim)

        output = self.o\_proj(attn\_output)

        return output



在这个实现中：

* qkv\_proj：一个线性层，将输入投影到查询、键和值的空间。
* forward方法：计算注意力权重，并对值进行加权求和，最终通过输出线性层生成结果。

### 2.2前馈神经网络

前馈神经网络在每个Transformer层中用于进一步处理自注意力机制的输出。其实现如下：

class FeedForward(nn.Module):

    def \_\_init\_\_(self, embed\_dim, ff\_dim):

        super(FeedForward, self).\_\_init\_\_()

        self.linear1 = nn.Linear(embed\_dim, ff\_dim)

        self.dropout = nn.Dropout(p=0.1)

        self.linear2 = nn.Linear(ff\_dim, embed\_dim)

    def forward(self, x):

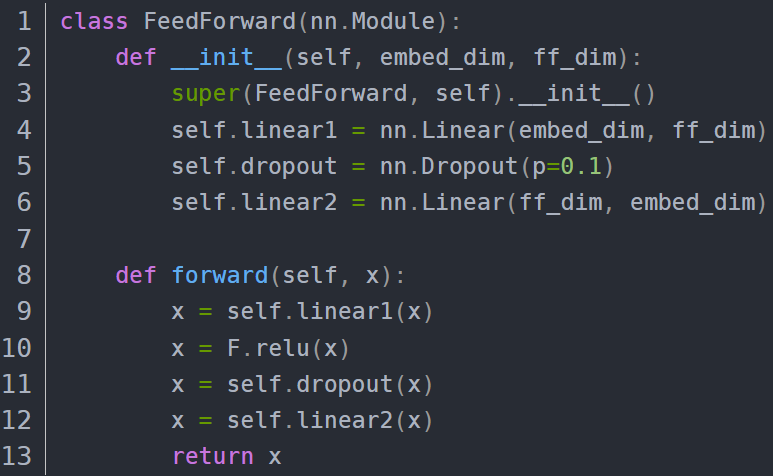
        x = self.linear1(x)

        x = F.relu(x)

        x = self.dropout(x)

        x = self.linear2(x)

        return x



在这个实现中：

* linear1：第一个线性层，将输入维度扩大。
* linear2：第二个线性层，将维度还原。
* forward方法：通过ReLU激活函数和dropout层，实现非线性变换和正则化。

### 2.3编码器和解码器

Transformer的编码器和解码器由多个相同的层堆叠而成，每一层包括一个多头自注意力机制和一个前馈神经网络。以下是编码器的实现：

class EncoderLayer(nn.Module):

    def \_\_init\_\_(self, embed\_dim, num\_heads, ff\_dim):

        super(EncoderLayer, self).\_\_init\_\_()

        self.self\_attn = MultiheadAttention(embed\_dim, num\_heads)

        self.ff = FeedForward(embed\_dim, ff\_dim)

        self.layernorm1 = nn.LayerNorm(embed\_dim)

        self.layernorm2 = nn.LayerNorm(embed\_dim)

        self.dropout = nn.Dropout(p=0.1)

    def forward(self, x):

        attn\_output = self.self\_attn(x)

        x = x + self.dropout(attn\_output)

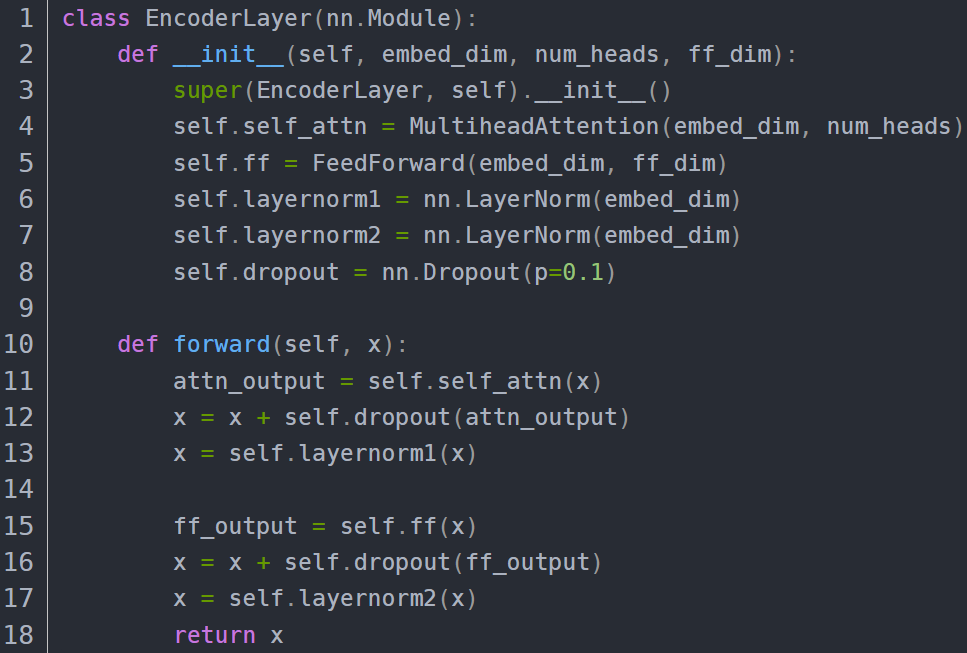
        x = self.layernorm1(x)

        ff\_output = self.ff(x)

        x = x + self.dropout(ff\_output)

        x = self.layernorm2(x)

        return x



在这个实现中：

* self\_attn：多头自注意力机制。
* ff：前馈神经网络。
* layernorm1和layernorm2：层归一化，用于稳定和加速训练。
* forward方法：通过残差连接和层归一化，处理自注意力和前馈网络的输出。

### 2.4位置编码

由于Transformer模型没有内置的序列顺序信息，需要通过位置编码显式地注入位置信息：

class PositionalEncoding(nn.Module):

    def \_\_init\_\_(self, embed\_dim, max\_len=5000):

        super(PositionalEncoding, self).\_\_init\_\_()

        position = torch.arange(0, max\_len).unsqueeze(1)

        div\_term = torch.exp(torch.arange(0, embed\_dim, 2) \* -(math.log(10000.0) / embed\_dim))

        pe = torch.zeros(max\_len, embed\_dim)

        pe[:, 0::2] = torch.sin(position \* div\_term)

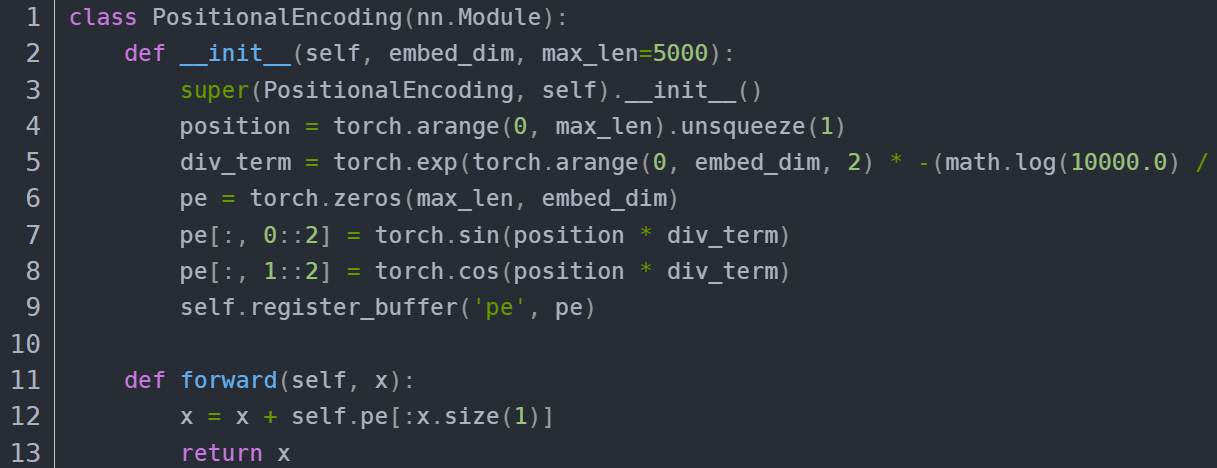
        pe[:, 1::2] = torch.cos(position \* div\_term)

        self.register\_buffer('pe', pe)

    def forward(self, x):

        x = x + self.pe[:x.size(1)]

        return x



在这个实现中：

* position：生成位置索引。
* div\_term：计算位置编码的分母。
* pe：根据正弦和余弦函数生成位置编码矩阵。
* forward方法：将位置编码添加到输入表示中。

## 3.结论

通过对PyTorch源码的分析，我们可以看到Transformer模型的实现涉及到多头自注意力机制、前馈神经网络、编码器和解码器、以及位置编码等多个关键组件。这些组件的巧妙组合，使得Transformer能够高效地处理序列数据，并在各种NLP任务中表现出色。

理解这些核心组件的实现，不仅有助于我们更深入地理解Transformer模型的工作原理，还为我们进一步优化和改进模型提供了坚实的基础。随着NLP技术的不断发展，Transformer模型及其变种将继续在更多领域中发挥重要作用。

# Ch4.pytorch 2d 3d图形识别模型 pytorch vision transformer

文章标签[pytorch 2d 3d图形识别模型](https://blog.51cto.com/topic/c9d4dc1079f4084.html" \t "https://blog.51cto.com/u_16099325/_blank)[pytorch](https://blog.51cto.com/topic/pytorch.html" \t "https://blog.51cto.com/u_16099325/_blank)[深度学习](https://blog.51cto.com/topic/shenduxuexi.html" \t "https://blog.51cto.com/u_16099325/_blank)[神经网络](https://blog.51cto.com/topic/shenjingwangluo.html" \t "https://blog.51cto.com/u_16099325/_blank)[transformer](https://blog.51cto.com/topic/transformer.html" \t "https://blog.51cto.com/u_16099325/_blank)文章分类[PyTorch](https://blog.51cto.com/nav/pytorch" \t "https://blog.51cto.com/u_16099325/_blank)[人工智能](https://blog.51cto.com/nav/ai" \t "https://blog.51cto.com/u_16099325/_blank)阅读数231

我整理的一些关于【架构设计】的项目学习资料+视频（附讲解～～）和大家一起分享、学习一下：

**[https://d.51cto.com/bLN8S1](https://d.51cto.com/bLN8S1" \t "https://blog.51cto.com/u_16099325/_blank)**

大家好，今天和各位分享一下如何使用 Pytorch 构建 Vision Transformer 网络模型，并使用 权重迁移学习方法 训练模型并预测。

Vision Transformer 的原理和 TensorFlow2 的实现方法可以看我下面这篇博文：

## 1. 引言

经典的 Transformer 由 Encoder 和 Decoder 组成，其中，最重要的就是多头注意力机制（Multi-head attention）。**在 Vision Transformer 中**，**作者通过 Transformer 的 Encoder 部分直接进行分类任务**，与 NLP 中的处理方式类似，**在图片序列中加入分类 token**，**图片序列则由原始图像切割成若干个图像块（Patch）得到**，如下图所示。

主要通过以下步骤进行转换：

**（1）**假设一张输入图片的形状为（B,C,H,W），其中 B 代表 Batch 大小，C 表示输入图片通道个数，H 和 W 表示输入图片的高和宽。

**（2）**那么，通过矩阵变换，可以将其转化为

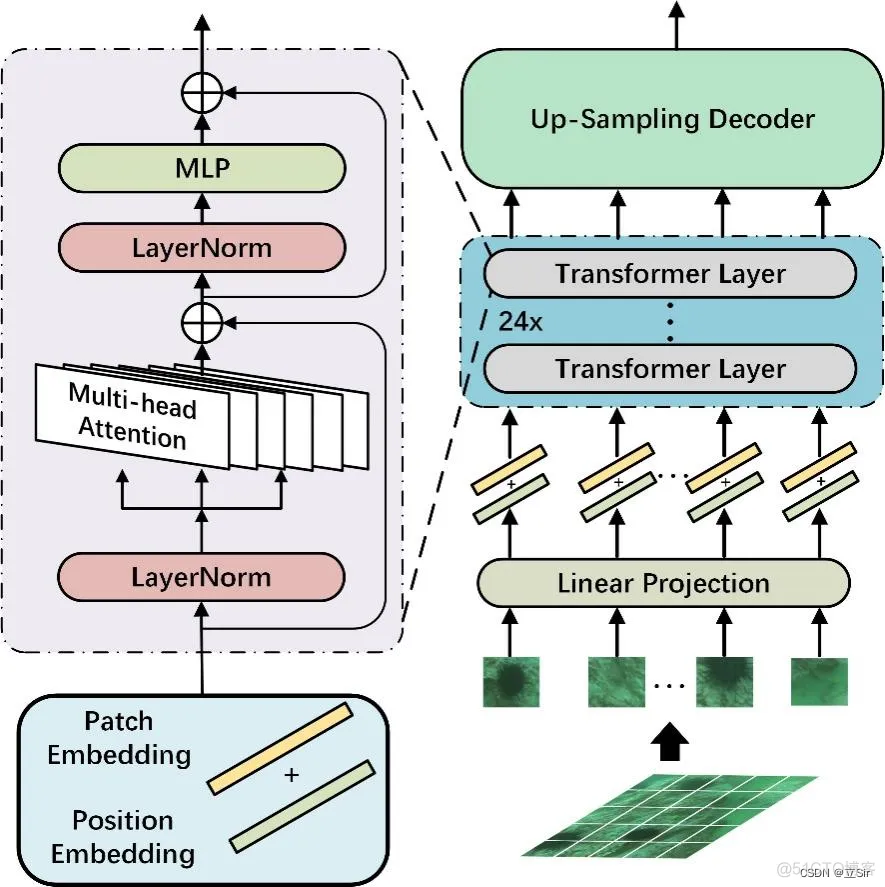
(B,N,P^{2}C)

，其中，P 代表 Patch 的大小，N 的值为

H*W/P^{2}

。

**（3）**将切分为若干个 Patch 的图片分别送到 TransFormer  Layer 中处理，在此过程中通过注意力机制进行输入特征的提取。



## 2. 模型构建

接下来构建 Vision Transformer 的主干模型，**本小节的代码都写在 VisionTransformer\_model.py 文件中**。先导入模型构建过程中需要用到的工具包。

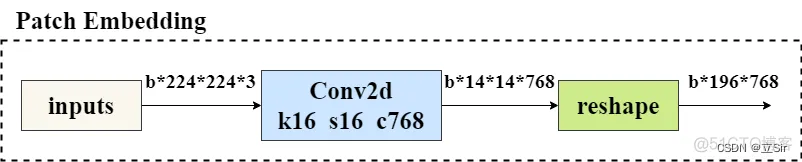
import torch

from torch import nn

from functools import partial

### 2.1 Patch Embedding

首先对**输入图像 [b,3,224,224]**做Patch Embedding 操作。**首先进行图像分块，将图片切分成14\*14个图像块（Patch），每个 Patch 的尺寸为 16\*16**。通过提取输入图片中的平坦像素向量，将每个输入 Patch 送入线性投影层，得到 Patch Embeddings。



在代码中其流程如上图，**先经过一个 kernel=(16,16)，strides=16 的卷积层划分图像块**，再**将 h和w 维度整合为 num\_patches 维度**，代表**一共有 196 个 patch，每个 patch 为 16\*16**

代码如下：

# --------------------------------------- #

#（1）patch embedding

'''

img\_size=224 : 输入图像的宽高

patch\_size=16 ： 每个patch的宽高，也是卷积核的尺寸和步长

in\_c=3 ： 输入图像的通道数

embed\_dim=768 ： 卷积输出通道数

'''

# --------------------------------------- #

class patchembed(nn.Module):

    # 初始化

    def \_\_init\_\_(self, img\_size=224, patch\_size=16, in\_c=3, embed\_dim=768):

        super(patchembed, self).\_\_init\_\_()

        # 输入图像的尺寸224\*224

        self.img\_size = (img\_size, img\_size)

        # 每个patch的大小16\*16

        self.patch\_size = (patch\_size, patch\_size)

        # 将输入图像划分成14\*14个patch

        self.grid\_size = (img\_size//patch\_size, img\_size//patch\_size)

        # 一共有14\*14个patch

        self.num\_patches = self.grid\_size[0] \* self.grid\_size[1]

        # 使用16\*16的卷积切分图像，将图像分成14\*14个

        self.proj = nn.Conv2d(in\_channels=in\_c, out\_channels=embed\_dim,

                              kernel\_size=patch\_size, stride=patch\_size)

        # 定义标准化方法，给LN传入默认参数eps

        norm\_layer = partial(nn.LayerNorm, eps=1e-6)

        self.norm = norm\_layer(embed\_dim)

    # 前向传播

    def forward(self, inputs):

        # 获得输入图像的shape

        B, C, H, W = inputs.shape

        # 如果输入图像的宽高不等于224\*224就报错

        assert H==self.img\_size[0] and W==self.img\_size[1], 'input shape does not match 224\*224'

        # 卷积层切分patch [b,3,224,224]==>[b,768,14,14]

        x = self.proj(inputs)

        # 展平 [b,768,14,14]==>[b,768,14\*14]

        x = x.flatten(start\_dim=2, end\_dim=-1)  # 将索引为 start\_dim 和 end\_dim 之间（包括该位置）的数量相乘

        # 维度调整 [b,768,14\*14]==>[b,14\*14,768]

        x = x.transpose(1, 2)  # 实现一个张量的两个轴之间的维度转换

        # 标准化

        x = self.norm(x)

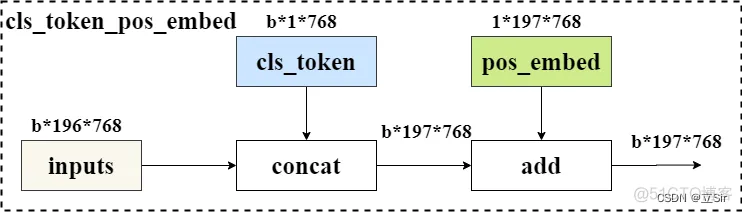
        return x

### 2.2 类别标签和位置编码

为了输出融合了全局语义信息的向量表示，**在第一个输入张量前添加可学习分类变量**。经过编码器编码后，**在最后一层输出中，该位置对应的输出张量就可以用于分类任务**。**与其他位置对应的输出向量相比，该向量可以更好的融合图像中各个图像块之间的依赖关系**。

在 Transformer 更新的过程中，**输入序列的顺序信息会丢失**。Transformer 本身并没有办法学习这个信息，所以**需要一种方法将位置表示聚合到模型的输入嵌入中**。我们**对每个 Patch 进行位置编码**，**该位置编码采用随机初始化**，之后参与模型训练。与传统三角函数的位置编码方法不同，**该方法是可学习的。**

最后，将 Patch-Embeddings 和 class-token 进行堆叠，和 Position-Embeddings 进行叠加，得到最终嵌入向量，该向量输入给 Transformer 层进行后续处理。



在代码中，**要注意 cls\_token 和 inputs 做堆叠 torch.cat() 时**，**需要将类别标签 cls\_token 放在最前面**。

代码如下：

# --------------------------------------- #

#（2）类别标签和位置标签

'''

embed\_dim : 代表patchembed层输出的通道数

'''

# --------------------------------------- #

class class\_token\_pos\_embed(nn.Module):

    # 初始化

    def \_\_init\_\_(self, embed\_dim):

        super(class\_token\_pos\_embed, self).\_\_init\_\_()

        # patchembed层将图像划分的patch个数==14\*14

        num\_patches = patchembed().num\_patches

        self.num\_tokens = 1  # 类别标签

        # 创建可学习的类别标签 [1,1,768]

        self.cls\_token = nn.Parameter(torch.zeros(1, 1, embed\_dim))

        # 创建可学习的位置编码 [1,196+1,768]

        self.pos\_embed = nn.Parameter(torch.zeros(1, num\_patches+self.num\_tokens, embed\_dim))

        # 权重以正态分布初始化

        nn.init.trunc\_normal\_(self.pos\_embed, std=0.02)

        nn.init.trunc\_normal\_(self.cls\_token, std=0.02)

    # 前向传播

    def forward(self, x):  # 输入特征图的shape=[b,196,768]

        # 类别标签扩充维度 [1,1,768]==>[b,1,768]

        cls\_token = self.cls\_token.expand(x.shape[0], -1, -1)

        # 将类别标签添加到特征图中 [b,1,768]+[b,196,768]==>[b,197,768]

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

        # 添加位置编码 [b,197,768]+[1,197,768]==>[b,197,768]

        x = x + self.pos\_embed

        return x

### 2.3 多头自注意力模块

Transformer 层中，主要包含多头注意力机制和多层感知机模块，下面先介绍多头自注意力模块。

单个的注意力机制，其每个输入包含三个不同的向量，分别为**Query向量（Q），Key向量（K），Value向量（V）**。**他们的结果分别由输入特征图和三个权重做矩阵乘法得到**。

IMG_261

接着为每一个输入计算一个得分

IMG_262

为了使梯度稳定，**对 Score 的值进行归一化处理，并将结果通过 softmax 函数进行映射**。**之后再和 v 做矩阵相乘**，得到加权后每个输入向量的得分 v。计算完后再乘以一个权重张量 W 提取特征。

计算公式如下，其中

\sqrt{d_{k}}

**代表 K 向量维度的平方根**

IMG_264

代码如下：

# --------------------------------------- #

#（3）多头注意力模块

'''

dim : 代表输入特征图的通道数

num\_heads : 多头注意力中heads的个数

qkv\_bias ： 生成qkv时是否使用偏置

atten\_drop\_ratio ：qk计算完之后的dropout层

proj\_drop\_ratio ： qkv计算完成之后的dropout层

'''

# --------------------------------------- #

class attention(nn.Module):

    # 初始化

    def \_\_init\_\_(self, dim, num\_heads=12, qkv\_bias=False, atten\_drop\_ratio=0., proj\_drop\_ratio=0.):

        super(attention, self).\_\_init\_\_()

        # 多头注意力的数量

        self.num\_heads = num\_heads

        # 将生成的qkv均分成num\_heads个。得到每个head的qkv对应的通道数。

        head\_dim = dim // num\_heads

        # 公式中的分母

        self.scale = head\_dim \*\* -0.5

        # 通过一个全连接层计算qkv

        self.qkv = nn.Linear(in\_features=dim, out\_features=dim\*3, bias=qkv\_bias)

        # dropout层

        self.atten\_drop = nn.Dropout(atten\_drop\_ratio)

        # 再qkv计算完之后通过一个全连接提取特征

        self.proj = nn.Linear(in\_features=dim, out\_features=dim)

        # dropout层

        self.proj\_drop = nn.Dropout(proj\_drop\_ratio)

    # 前向传播

    def forward(self, inputs):

        # 获取输入图像的shape=[b,197,768]

        B, N, C = inputs.shape

        # 将输入特征图经过全连接层生成qkv [b,197,768]==>[b,197,768\*3]

        qkv = self.qkv(inputs)

        # 维度调整 [b,197,768\*3]==>[b, 197, 3, 12, 768//12]

        qkv = qkv.reshape(B, N, 3, self.num\_heads, C//self.num\_heads)

        # 维度重排==> [3, B, 12, 197, 768//12]

        qkv = qkv.permute(2,0,3,1,4)

        # 切片提取q、k、v的值，单个的shape=[B, 12, 197, 768//12]

        q, k, v = qkv[0], qkv[1], qkv[2]

        # 针对每个head计算 ==> [B, 12, 197, 197]

        atten = (q @ k.transpose(-2,-1)) \* self.scale  # @ 代表在多维tensor的最后两个维度矩阵相乘

        # 对计算结果的每一行经过softmax

        atten = atten.softmax(dim=-1)

        # dropout层

        atten = self.atten\_drop(atten)

        # softmax后的结果和v加权 ==> [B, 12, 197, 768//12]

        x = atten @ v

        # 通道重排 ==> [B, 197, 12, 768//12]

        x = x.transpose(1,2)

        # 维度调整 ==> [B, 197, 768]

        x = x.reshape(B,N,C)

        # 通过全连接层融合特征 ==> [B, 197, 768]

        x = self.proj(x)

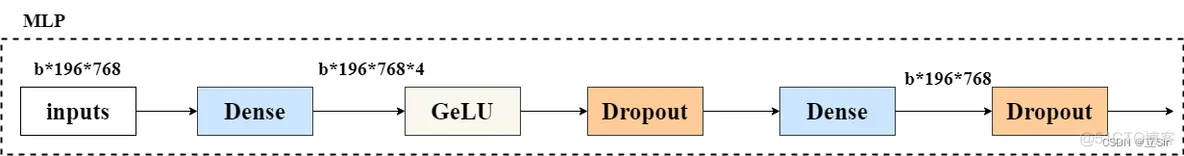
        # dropout层

        x = self.proj\_drop(x)

        return x

### 2.4 MLP 多层感知器

这个部分简单来看就是两个全连接层提取特征，流程图如下。第一个全连接层通道上升4倍，第二个全连接层通道下降为原来。



代码如下：

# --------------------------------------- #

#（4）MLP多层感知器

'''

in\_features : 输入特征图的通道数

hidden\_features : 第一个全连接层上升通道数

out\_features : 第二个全连接层的下降的通道数

drop : 全连接层后面的dropout层的杀死神经元的概率

'''

# --------------------------------------- #

class MLP(nn.Module):

    # 初始化

    def \_\_init\_\_(self, in\_features, hidden\_features, out\_features=None, drop=0.):

        super(MLP, self).\_\_init\_\_()

        # MLP的输出通道数默认等于输入通道数

        out\_features = out\_features or in\_features

        # 第一个全连接层上升通道数

        self.fc1 = nn.Linear(in\_features=in\_features, out\_features=hidden\_features)

        # GeLU激活函数

        self.act = nn.GELU()

        # 第二个全连接下降通道数

        self.fc2 = nn.Linear(in\_features=hidden\_features, out\_features=out\_features)

        # dropout层

        self.drop = nn.Dropout(drop)

    # 前向传播

    def forward(self, inputs):

        # [b,197,768]==>[b,197,3072]

        x = self.fc1(inputs)

        x = self.act(x)

        x = self.drop(x)

        # [b,197,3072]==>[b,197,768]

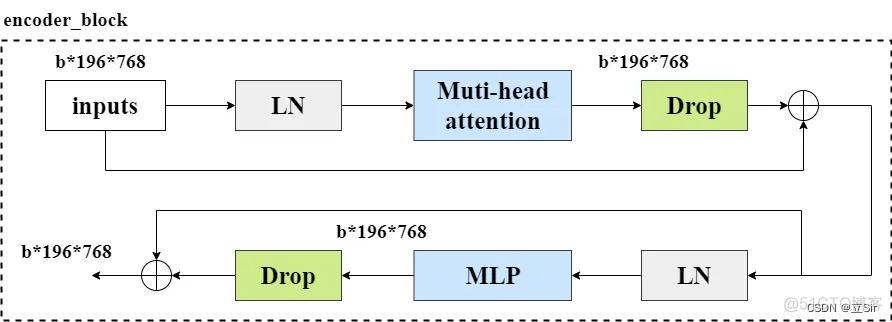
        x = self.fc2(x)

        x = self.drop(x)

        return x

### 2.5 特征提取模块

Transformer 的**单个特征提取模块**是由**多头注意力机制** 和 **多层感知机模块** 组合而成，encoder\_block 模块的流程图如下。



输入图像像经过**LayerNormalization 标准化后**，**再经过我们上面定义的多头注意力模块，将输出结果和输入特征图残差连接**，**图像在特征提取过程中shape保持不变**。

**将输出结果再经过标准化，然后送入多层感知器提取特征**，再使用残差连接输入和输出。

而 transformer 的特征提取模块是由多个 encoder\_block 叠加而成，这里连续使用12个 encoder\_block 模块来提取特征。

代码如下：

# --------------------------------------- #

#（5）Encoder Block

'''

dim : 该模块的输入特征图个数

mlp\_ratio ： MLP中第一个全连接层上升的通道数

drop\_ratio : 该模块的dropout层的杀死神经元的概率

'''

# --------------------------------------- #

class encoder\_block(nn.Module):

    # 初始化

    def \_\_init\_\_(self, dim, mlp\_ratio=4., drop\_ratio=0.):

        super(encoder\_block, self).\_\_init\_\_()

        # LayerNormalization层

        self.norm1 = nn.LayerNorm(dim)

        # 实例化多头注意力

        self.atten = attention(dim)

        # dropout

        self.drop = nn.Dropout()

        # LayerNormalization层

        self.norm2 = nn.LayerNorm(dim)

        # MLP中第一个全连接层上升的通道数

        hidden\_features = int(dim \* mlp\_ratio)

        # MLP多层感知器

        self.mlp = MLP(in\_features=dim, hidden\_features=hidden\_features)

    # 前向传播

    def forward(self, inputs):

        # [b,197,768]==>[b,197,768]

        x = self.norm1(inputs)

        x = self.atten(x)

        x = self.drop(x)

        feat1 = x + inputs  # 残差连接

        # [b,197,768]==>[b,197,768]

        x = self.norm2(feat1)

        x = self.mlp(x)

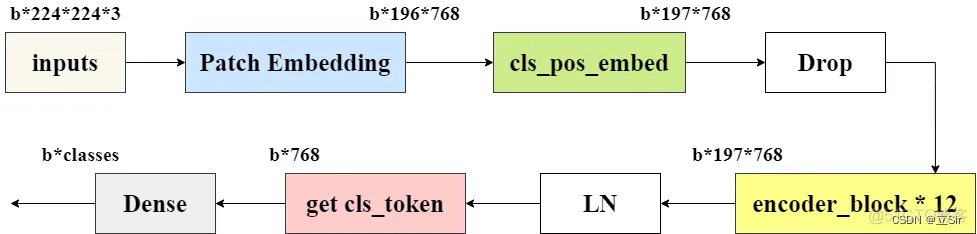
        x = self.drop(x)

        feat2 = x + feat1  # 残差连接

        return feat2

### 2.6 主干网络

接下来就搭建网络了，将上面所有的模块组合到一起，如下图所示。



**在下面代码中要注意的是 x= x[:,0] 取出所有的类别标签**。 因为在 cls\_pos\_embed 模块中，我们**将 cls\_token 和输入图像在 patch 维度上堆叠，用于学习每张特征图的类别信息**。最后经过一个全连接层得出每张图片属于每个类别的得分。

代码如下：

# --------------------------------------- #

#（6）主干网络

'''

num\_class: 分类数

depth : 重复堆叠encoder\_block的次数

drop\_ratio : 位置编码后的dropout层

embed\_dim : patchembed层输出通道数

'''

# --------------------------------------- #

class VIT(nn.Module):

    # 初始化

    def \_\_init\_\_(self, num\_classes=1000, depth=12, drop\_ratio=0., embed\_dim=768):

        super(VIT, self).\_\_init\_\_()

        self.num\_classes = num\_classes  # 分类类别数

        # 实例化patchembed层

        self.patchembed = patchembed()

        # 实例化类别标签和位置编码

        self.cls\_pos\_embed = class\_token\_pos\_embed(embed\_dim=embed\_dim)

        # 位置编码后做dropout

        self.pos\_drop = nn.Dropout(drop\_ratio)

        # 在列表中添加12个encoder\_block

        self.blocks = nn.Sequential(\*[encoder\_block(dim=embed\_dim) for \_ in range(depth)])

        # 定义LayerNormalization标准化方法

        norm\_layer = partial(nn.LayerNorm, eps=1e-6)

        # 经过12个encoder之后的标准化层

        self.norm = norm\_layer(embed\_dim)

        # 分类层

        self.head = nn.Linear(in\_features=embed\_dim, out\_features=num\_classes)

        # 权值初始化

        for m in self.modules():

            # 对卷积层使用kaiming初始化

            if isinstance(m, nn.Conv2d):

                nn.init.kaiming\_normal\_(m.weight, mode='fan\_out')

                # 对偏置初始化

                if m.bias is not None:

                    nn.init.zeros\_(m.bias)

            # 对标准化层初始化

            elif isinstance(m, nn.LayerNorm):

                nn.init.ones\_(m.weight)

                nn.init.zeros\_(m.bias)

            # 对全连接层初始化

            elif isinstance(m, nn.Linear):

                nn.init.normal\_(m.weight, 0, 0.01)

                if m.bias is not None:

                    nn.init.zeros\_(m.bias)

    # 前向传播

    def forward(self, inputs):

        # 先将输入传递给patchembed [b,3,224,224]==>[b,196,768]

        x = self.patchembed(inputs)

        # 对特征图添加类别标签和位置编码

        x = self.cls\_pos\_embed(x)

        # dropout层

        x = self.pos\_drop(x)

        # 经过12个encoder层==>[b,197,768]

        x = self.blocks(x)

        # LN标准化层

        x = self.norm(x)

        # 提取类别标签的输出,因为在cat时将类别标签放在最前面

        x = x[:, 0]  # [b,197,768]==>[b,768]

        # 全连接层分类 [b,768]==>[b,1000]

        x = self.head(x)

        return x

## 3. 训练阶段

接下来对使用权重迁移学习的方法训练模型，**这里用的网络是 VIT B-16 模型，patch的尺寸为16\*16，patchembedding的输出通道数为768**。首先导入所有的工具包，定义好所有需要的参数，找到文件路径，方便后期使用管理。

import  torch

from torch import nn, optim

from torchvision import transforms, datasets

from torch.utils.data import DataLoader

from VisionTransformer\_model import VIT  # 导入我们之前定义的 VIT B-16 模型

import numpy as np

import matplotlib.pyplot as plt

plt.rcParams['font.sans-serif'] = ['SimHei']  # 绘图显示中文

# --------------------------------------------- #

#（0）参数设置

# --------------------------------------------- #

batch\_size = 16  # 每个step处理16张图片

epochs = 10  # 训练10轮

best\_loss = 1.0  # 当验证集损失小于1时才保存权重

# 数据集目录位置

filepath = 'D:/deeplearning/test/数据集/animal/'

# 预训练权重位置

weightpath = 'D:/deeplearning/imgnet/pytorchimgnet/pretrained\_weights/vit\_base\_patch16\_224.pth'

# 训练时保存权重文件的位置

savepath = 'D:/deeplearning/imgnet/pytorchimgnet/save\_weights/'

# 获取GPU设备，检测到了就用GPU，检测不到就用CPU

if torch.cuda.is\_available():

    device = torch.device('cuda:0')

else:

    device = torch.device('cpu')

### 3.1 构造数据集

首先定义训练集和验证集的**数据预处理方法 data\_transform**。通过**transforms.Resize()** 将输入图像的**尺寸缩放**到模型要求的 224\*224 大小，然后再通过**transforms.ToTensor()**将像素值类型从 numpy 变成 tensor 类型，并归一化处理，像素值大小从 [0,255] 变换到 [0,1]，再调整输入图像的维度，从 [h,w,c] 变成 [c,h,w]；接着 **transforms.Normalize()**对图像的每个颜色通道做标准化处理，使像素值满足正态分布。

预处理之后就**构造训练集和验证集 dataloader**，指定 batch\_size=16，代表训练时每个 step 训练16张图片。

接着查看数据集信息，查看分类类别及其对应的索引信息，其中 datasets['train'].class\_to\_idx 的结果是 {'cats':0, 'dogs':1, 'panda':2}

代码如下：

# --------------------------------------------- #

#（1）数据集处理

# --------------------------------------------- #

# 定义预处理方法

data\_transform = {

    # 训练集预处理方法

    'train' : transforms.Compose([

        transforms.Resize((224,224)),  # 将原始图片缩放至224\*224大小

        transforms.RandomHorizontalFlip(),  # 随机水平翻转

        transforms.ToTensor(),  # numpy类型变tensor，维度调整，数据归一化

        transforms.Normalize(mean=[0.485,0.456,0.406], std=[0.229,0.224,0.225])   # 对图像的三个通道分别做标准化

        ]),

    # 验证集预处理方法

    'val' : transforms.Compose([

        transforms.Resize((224,224)),  # 将输入图像缩放至224\*224大小

        transforms.ToTensor(),

        transforms.Normalize(mean=[0.485,0.456,0.406], std=[0.229,0.224,0.225])

        ])

    }

# 加载数据集

datasets = {

    'train' : datasets.ImageFolder(filepath+'train', transform=data\_transform['train']),  # 读取训练集

    'val'   : datasets.ImageFolder(filepath+'val', transform=data\_transform['val'])  # 读取验证集

    }

# 构造数据集

dataloader = {

    'train' : DataLoader(datasets['train'], batch\_size=batch\_size, shuffle=True),  # 构造训练集

    'val'   : DataLoader(datasets['val'], batch\_size=batch\_size, shuffle=True)   # 构造验证集

    }

# --------------------------------------------- #

#（2）查看数据集信息

# --------------------------------------------- #

train\_num = len(datasets['train'])  # 查看训练集数量

val\_num = len(datasets['val'])  # 查看验证集数量

# 查看分类类别及其索引  {0: 'cats', 1: 'dogs', 2: 'panda'}

class\_names = dict((v,k) for k,v in datasets['train'].class\_to\_idx.items())

print(class\_names)

# 从训练集中取出一个batch，接收图片及其标签

train\_imgs, train\_labels = next(iter(dataloader['train']))

# 查看图像的标签及其shape  [32, 3, 224, 224]  [32]

print('img:', train\_imgs.shape, 'labels:', train\_labels.shape)

### 3.2 数据可视化

**可视化训练集中的前12张图像**。由于构造数据集时使用了一系列预处理方法，因此这里要**将像素类型从 tensor 变成 numpy**，**调整图像的维度 [b,c,h,w]==>[b,h,w,c]**，**对图像的每个通道执行反标准化操作**，恢复到0-1之间的随机分布。

标准化：

img = \frac{img-mean}{std}

     反标准化：

img = img * std + mean

代码如下：

# --------------------------------------------- #

#（3）数据可视化

# --------------------------------------------- #

# 从数据集中取出12张图及其标签

frames = train\_imgs[:12]

frames\_labels = train\_labels[:12]

# 将数据类型从tensor变回numpy

frames = frames.numpy()

# 维度调整 [b,c,h,w]==>[b,h,w,c]

frames = np.transpose(frames, [0,2,3,1])

# 对图像做反标准化处理

mean = [0.485, 0.456, 0.406]  # 均值

std = [0.229, 0.224, 0.225]   # 标准化

# 图像的每个通道的特征图乘标准化加均值

frames = frames \* std + mean

# 将像素值限制在0-1之间

frames = np.clip(frames, 0, 1)

# 绘制12张图像及其标签

plt.figure()  # 创建画板

for i in range(12):

    plt.subplot(3,4,i+1)

    plt.imshow(frames[i])

    plt.axis('off')  # 不显示坐标刻度

    plt.title(class\_names[frames\_labels[i].item()])  # 显示每张图片的标签

plt.tight\_layout()  # 轻量化布局

plt.show()

查看训练集中的图像



### 3.3 模型加载，迁移学习

首先**加载预训练权重 torch.load()**到内存中。由于**预训练模型的分类数有1000个，即最后一个全连接层有 1000 个神经元**，因此我们**只用预训练权重的特征提取部分，不需要分类层部分**。

遍历预训练权重文件，**删除分类层 'head.weight', 'head.bias' 的权重**。

这里**不冻结预训练权重**，所有权重参数都能通过反向传播更新。

代码如下：

# --------------------------------------------- #

#（4）模型加载，迁移学习

# --------------------------------------------- #

# 接收VIT模型，三分类

model = VIT(num\_classes=3)

# 加载预训练权重文件，文件中的分类层神经元个数是1k

pre\_weights = torch.load(weightpath, map\_location=device)

# 删除权重文件中不需要的层，保留除了分类层以外的所有层的权重

del\_keys = ['head.weight', 'head.bias']

# 删除字典中的对应key

for k in del\_keys:

    del pre\_weights[k]

# 将修改后的权重加载到模型上

# 当strict=True,要求预训练权重层数的键值与新构建的模型中的权重层数名称完全吻合

missing\_keys, unexpected\_keys = model.load\_state\_dict(pre\_weights, strict=False)

print('miss:', len(missing\_keys), 'unexpected:', len(unexpected\_keys))

# model.parameters() 代表网络的所有参数

for params in model.parameters():

    params.requires\_grad = True  # 所有权重参与训练可以更新

### 3.4 模型训练

接下来进行网络训练，将所有需要计算的部分都搬运到 GPU 上，加快训练速度。

我这里**使用每个epoch的验证集损失作为网络监控指标**，**如果损失小于规定值且一直在下降**就保存当前 epoch 的权重。

还要注意的就是**网络训练和测试的模式不一样**，**训练时** Dropout 层随机杀死神经元，BN 层取一个batch的均值和方差；**验证时** Dropout 层不起作用，BN 层取整个训练集计算得到的均值和方差。**通过 net.train() 和 net.eval（) 来切换训练和验证模式**。

代码如下：

# --------------------------------------------- #

#（5）网络编译

# --------------------------------------------- #

# 将模型搬运至GPU上

model.to(device)

# 定义交叉熵损失

loss\_function = nn.CrossEntropyLoss()

# 获取所有需要梯度更新的权重参数

params\_optim = []

# 遍历网络的所有权重

for p in model.parameters():

    if p.requires\_grad is True:  # 查看权重是否需要更新

        params\_optim.append(p)   # 保存所有需要更新的权重

print('训练参数：', len(params\_optim))

# 定义优化器，定义学习率，动量，正则化系数

optimizer = optim.SGD(params\_optim, lr=0.001, momentum=0.9, weight\_decay=3e-4)

# --------------------------------------------- #

#（6）训练阶段

# --------------------------------------------- #

for epoch in range(epochs):

    print('='\*30)  # 显示当前是第几个epoch

    # 将模型设置为训练模式

    model.train()

    # 记录一个epoch的训练集总损失

    total\_loss = 0.0

    # 每个step训练一个batch，每次取出一个数据集及其标签

    for step, (images, labels) in enumerate(dataloader['train']):

        # 将数据集搬运到GPU上

        images, labels = images.to(device), labels.to(device)

        # 梯度清零，因为梯度是累加的

        optimizer.zero\_grad()

        # 前向传播==>[b,3]

        logits = model(images)  # 得到每张图属于3个类别的分数

        #（1）损失计算

        # 计算每个step的预测值和真实值的交叉熵损失

        loss = loss\_function(logits, labels)

        # 累加每个step的损失

        total\_loss += loss

        #（2）反向传播

        # 梯度计算

        loss.backward()

        # 梯度更新

        optimizer.step()

        # 每50个epoch打印一次损失值

        if step % 50 == 0:

            print(f'step:{step}, train\_loss:{loss}')

    # 计算一个epoch的训练集平均损失

    train\_loss = total\_loss / len(dataloader['train'])

# --------------------------------------------- #

#（7）验证训练

# --------------------------------------------- #

    model.eval（)  # 切换到验证模式

    total\_val\_loss = 0.0  # 记录一个epoch的验证集总损失

    total\_val\_correct = 0  # 记录一个epoch中验证集一共预测对了几个

    with torch.no\_grad():  # 接下来不计算梯度

        # 每个step验证一个batch

        for (images, labels) in dataloader['val']:

            # 将数据集搬运到GPU上

            images, labels = images.to(device), labels.to(device)

            # 前向传播[b,c,h,w]==>[b,3]

            logits = model(images)

            #（1）计算损失

            # 计算每个batch的预测值和真实值的交叉熵损失

            loss = loss\_function(logits, labels)

            # 累加每个batch的损失，得到一个epoch的总损失

            total\_val\_loss += loss

            #（2）计算准确率

            # 找到预测值对应的最大索引，即该图片对应的类别

            pred = logits.argmax(dim=1)  # [b,3]==>[b]

            # 比较预测值和标签值，计算每个batch有多少预测对了

            val\_correct = torch.eq(pred, labels).float().sum()

            # 累加每个batch的正确个数，计算整个epoch的正确个数

            total\_val\_correct += val\_correct

        # 计算一个epoch的验证集的平均损失和平均准确率

        val\_loss = total\_val\_loss / len(dataloader['val'])

        val\_acc = total\_val\_correct / val\_num

        # 打印每个epoch的训练集平均损失，验证集平均损失和平均准确率

        print('-'\*30)

        print(f'train\_loss:{train\_loss}, val\_loss:{val\_loss}, val\_acc:{val\_acc}')

# --------------------------------------------- #

#（8）保存权重

# --------------------------------------------- #

        # 保存最小损失值对应的权重文件

        if val\_loss < best\_loss:

            # 权重文件名称

            savename = savepath + f'epoch{epoch}\_valacc{round(val\_acc.item()\*100)}%\_' + 'VIT.pth'

            # 保存该轮次的权重

            torch.save(model.state\_dict(), savename)

            # 切换最小损失值

            best\_loss = val\_loss

            # 打印结果

            print(f'weights has been saved, best\_loss has changed to {val\_loss}')

训练过程如下：

==============================

step:0, train\_loss:0.9088920950889587

step:50, train\_loss:2.3867087364196777

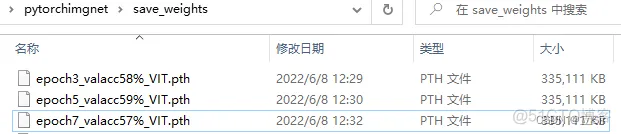
step:100, train\_loss:2.1412224769592285

------------------------------

train\_loss:1.7520136833190918, val\_loss:2.2571213245391846, val\_acc:0.5276381969451904

==============================

训练过程中保存权重：



## 4. 预测阶段

接下来我们用训练好了的权重文件来预测图像的类别。同样先导入所有需要用到的工具包。

代码如下：

import torch

from torchvision import transforms, datasets

from torch.utils.data import DataLoader

from PIL import Image

from VisionTransformer\_model import VIT

import numpy as np

import matplotlib.pyplot as plt

# -------------------------------------------------- #

#（0）参数设置

# -------------------------------------------------- #

batch\_size = 32  # 每次测试32张图

# 测试集文件夹所在位置

file\_path = 'D:/deeplearning/test/数据集/animal/test'

# 权重参数路径

weights\_path = 'D:/deeplearning/imgnet/pytorchimgnet/save\_weights/epoch5\_valacc59%\_VIT.pth'

# 获取GPU设备

if torch.cuda.is\_available():  # 如果有GPU就用，没有就用CPU

    device = torch.device('cuda:0')

else:

device = torch.device('cpu')

## 4.1 构造数据集

这里测试集的预处理采用和验证集相同的预处理方法。这部分和上面相同，就不多做介绍。

# -------------------------------------------------- #

#（1）构造测试集

# -------------------------------------------------- #

# 定义测试集的数据预处理方法

data\_transforms = transforms.Compose([

    transforms.Resize((224,224)),  # 将输入图像的size缩放至224\*224

    transforms.ToTensor(),  # numpy边tensor，像素归一化，维度调整

    transforms.Normalize(mean=[0.485,0.456,0.406], std=[0.229,0.224,0.225])  # 对每个通道标准化

    ])

# 加载测试集，并预处理

datasets = datasets.ImageFolder(file\_path, transform=data\_transforms)

# 构造测试集

dataloader = DataLoader(datasets, batch\_size=batch\_size, shuffle=True)

# 查看测试集一共有多少张图

test\_num = len(datasets)

# 获取测试集的分类类别及其索引  {0: 'cats', 1: 'dogs', 2: 'panda'}

class\_names = dict((v,k) for k,v in datasets.class\_to\_idx.items())

## 4.2 多图像预测

取测试集的每个 Batch 的前12张图片，查看其预测结果。这里也需要做一次反标准化操作，和上面第三小节中相同。

在网络前向传播之前**将模型设置为验证模式 model.eval（)**，**只做前向传播的操作，不进行梯度更新操作 with torch.no\_grad()**不计算梯度。

经过前向传播后，**图像的shape变成 [b,3]**，即图片预测属于3种类别的分数，然后**经过softmax()求出图片属于每个类别的概率**，**通过torch.max()找出最大概率及其索引**，得到图片属于哪个类别。

代码如下：

# -------------------------------------------------- #

#（2）绘图展示预测结果

# imgs:代表输入图像[b,c,h,w]，labels代表图像的真实标签[b]

# cls:代表每张图属的类别索引[b]，scores:代表每张图的类别概率[b]

# -------------------------------------------------- #

def im\_show(imgs, labels, cls, scores):

    # 从数据集中取出12张图及其标签索引、概率

    frames = imgs[:12]

    true\_labels = labels[:12]

    pred\_labels = cls[:12]

    pred\_scores = scores[:12]

    # 将数据类型从tensor变回numpy

    frames = frames.numpy()

    # 维度调整 [b,c,h,w]==>[b,h,w,c]

    frames = np.transpose(frames, [0,2,3,1])

    # 对图像做反标准化处理

    mean = [0.485, 0.456, 0.406]  # 均值

    std = [0.229, 0.224, 0.225]   # 标准化

    # 图像的每个通道的特征图乘标准化加均值

    frames = frames \* std + mean

    # 将像素值限制在0-1之间

    frames = np.clip(frames, 0, 1)

    # 绘制12张图像及其标签

    plt.figure()  # 创建画板

    for i in range(12):

        plt.subplot(3,4,i+1)

        plt.imshow(frames[i])

        plt.axis('off')  # 不显示坐标刻度

        # 显示每张图片的真实标签、预测标签、预测概率

        plt.title('true:'+class\_names[true\_labels[i].item()] + '\n' +

                  'pred:'+class\_names[pred\_labels[i].item()] + '\n' +

                  'scores:'+str(round(pred\_scores[i].item(), 3))

                )

    plt.tight\_layout()  # 轻量化布局

    plt.show()

# -------------------------------------------------- #

#（3）图像预测

# -------------------------------------------------- #

# 模型构建

model = VIT(num\_classes=3)

# 加载权重文件

model.load\_state\_dict(torch.load(weights\_path, map\_location=device))

# 将模型搬运到GPU上

model.to(device)

# 模型切换成测试模式，切换LN标准化和dropout的工作方式

model.eval（)

# 测试阶段不计算梯度

with torch.no\_grad():

    # 每次测试一个batch

    for step, (imgs, labels) in enumerate(dataloader):

        # 将数据集搬运到GPU上

        images, labels = imgs.to(device), labels.to(device)

        # 前向传播==>[b,3]

        logits = model(images)

        # 求出图像属于哪个类别索引[b,3]==>[b]

        pred\_cls = logits.argmax(dim=1)

        # 计算图像属于每个类别的概率[b,3]==>[b,3]

        predicts = torch.softmax(logits, dim=1)

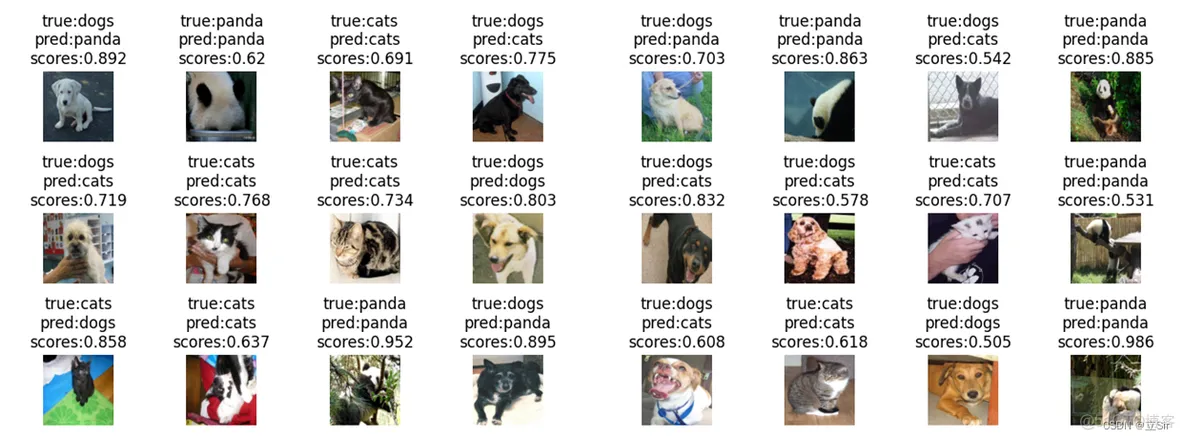
        # 获取最大预测类别的概率[b,3]==>[b]

        predicts\_score, \_ = predicts.max(dim=1)

        # 绘制预测结果

        im\_show(imgs, labels, pred\_cls, predicts\_score)

查看预测结果：图像标题是真实类别、预测类别、预测概率值



整理的一些关于【架构设计】的项目学习资料+视频（附讲解～～），需要自取：

**[https://d.51cto.com/bLN8S1](https://d.51cto.com/bLN8S1" \t "https://blog.51cto.com/u_16099325/_blank)**

# Ch5.基于深度学习的车牌检测识别(Pytorch)(ResNet +Transformer）

[小白学视觉](javascript:void(0);) 2024年08月27日 10:05 中国香港

**车牌识别概述**

基于深度学习的车牌识别，其中，车辆检测网络直接使用YOLO侦测。而后，才是使用网络侦测车牌与识别车牌号。

车牌的侦测网络，采用的是resnet18，网络输出检测边框的仿射变换矩阵，可检测任意形状的四边形。

车牌号序列模型，采用Resnet18+transformer模型，直接输出车牌号序列。

数据集上，车牌检测使用CCPD 2019数据集，在训练检测模型的时候，会使用程序生成虚假的车牌，覆盖于数据集图片上，来加强检测的能力。

车牌号的序列识别，直接使用程序生成的车牌图片训练，并佐以适当的图像增强手段。模型的训练直接采用端到端的训练方式，输入图片，直接输出车牌号序列，损失采用CTCLoss。

## 一、网络模型

### 1、车牌的侦测网络模型：

网络代码定义如下：

网络代码定义如下：

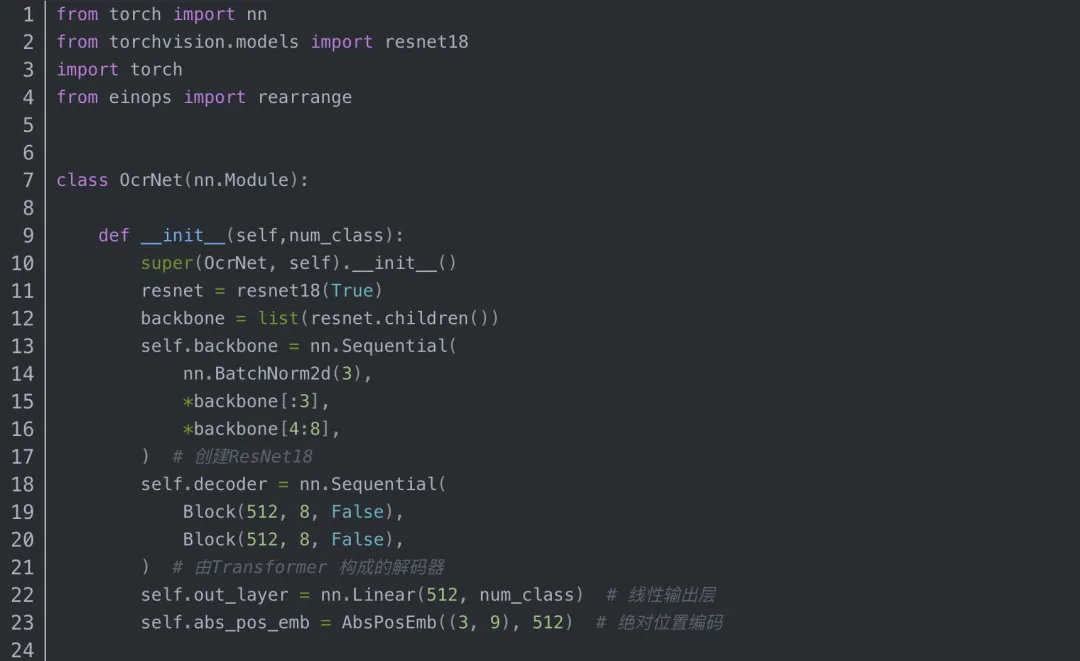


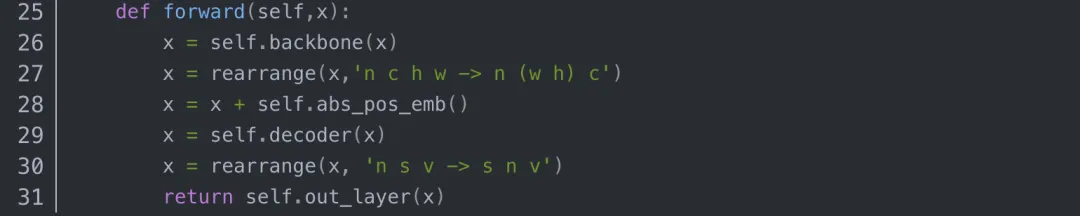
该网络，相当于直接对图片划分cell，即在16X16的格子中，侦测车牌，输出的为该车牌边框的反射变换矩阵。

### 2、车牌号的序列识别网络：

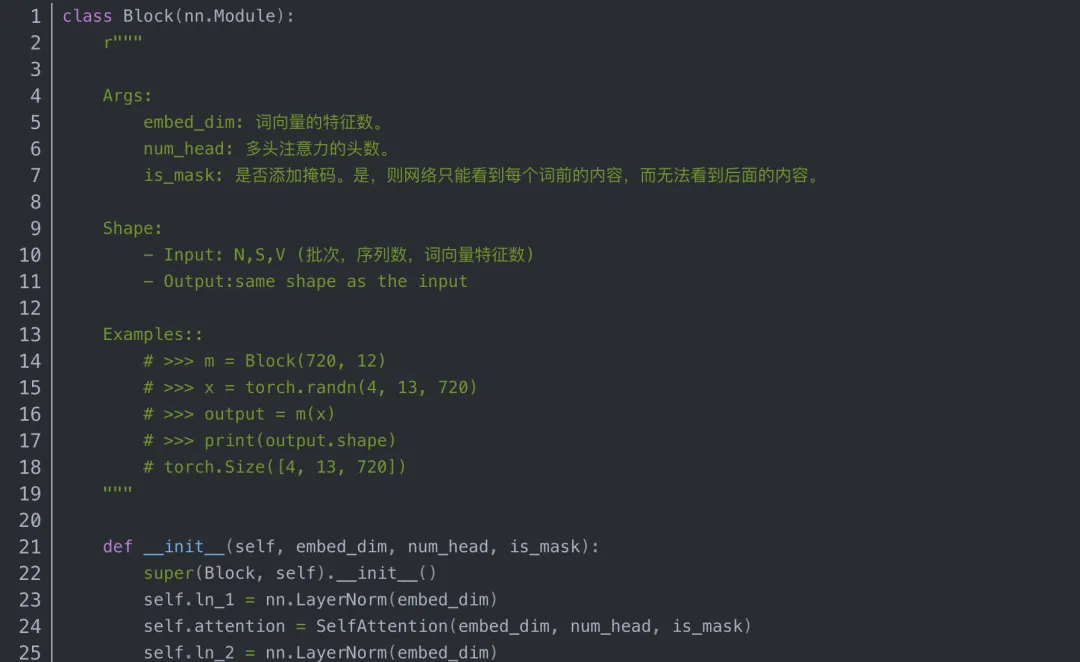
车牌号序列识别的主干网络：采用的是ResNet18+transformer，其中有ResNet18完成对图片的编码工作，再由transformer解码为对应的字符。

网络代码定义如下：





其中的Block类的代码如下：

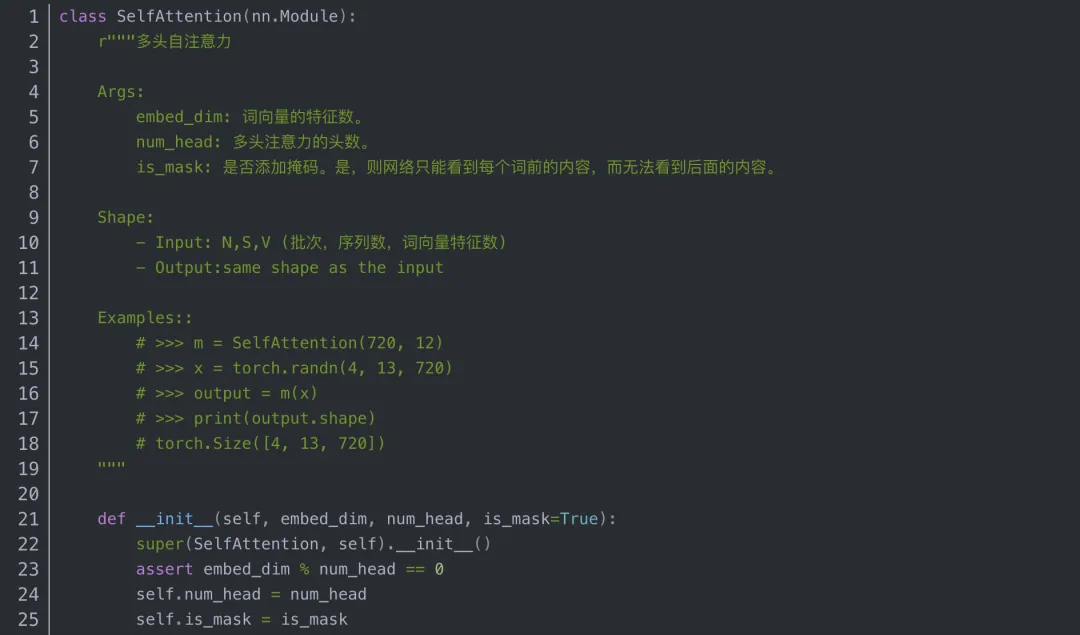


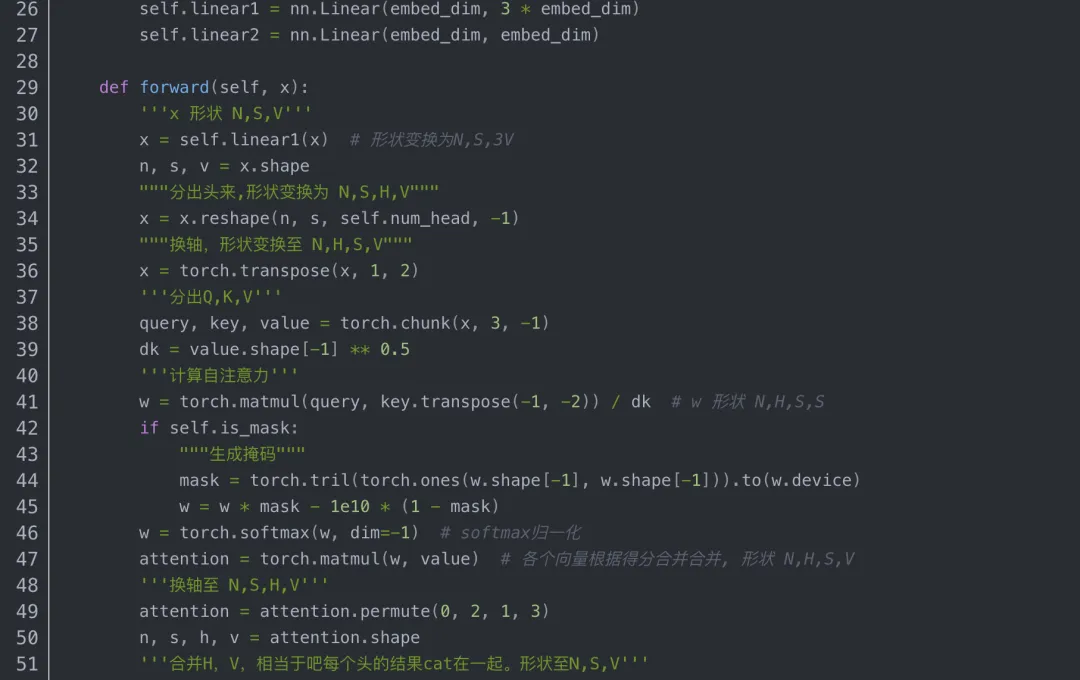


位置编码的代码如下：



Block类使用的自注意力代码如下：





IMG_264

## 二、数据加载

### 1、车牌号的数据加载

同过程序生成一组车牌号:



再通过数据增强，

主要包括：



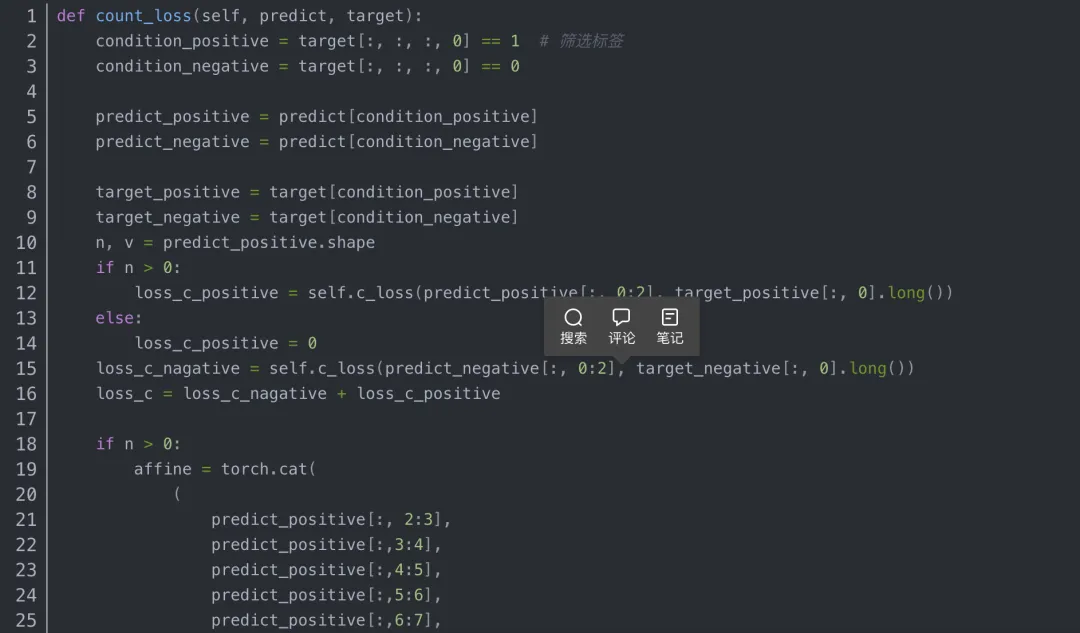


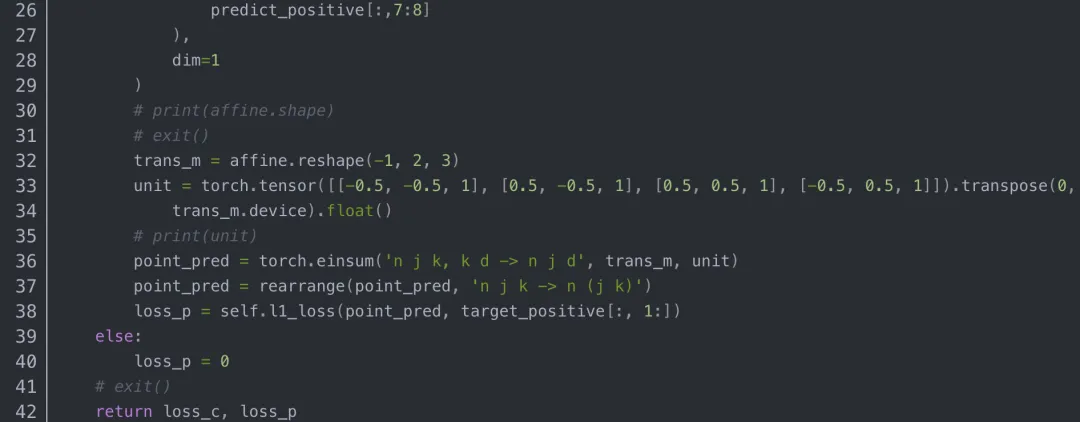


## 三、训练

分别训练即可

其中，侦测网络的损失计算，如下：





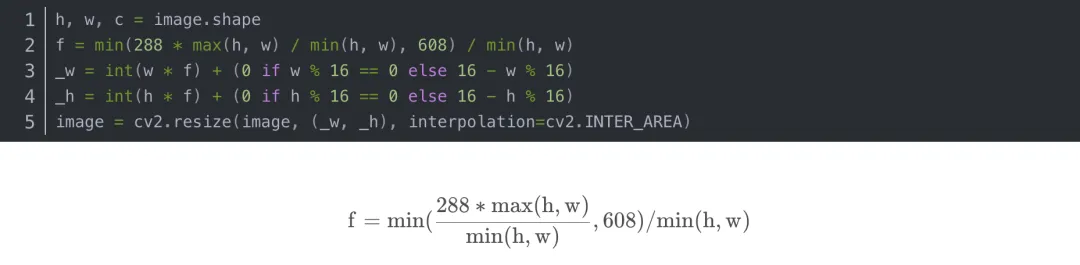
侦测网络输出的反射变换矩阵，但对车牌位置的标签给的是四个角点的位置，所以需要响应转换后，做损失。其中，该cell是否有目标，使用CrossEntropyLoss，而对车牌位置损失，采用的则是L1Loss。

## ****四、推理****

### ****1、侦测网络的推理****

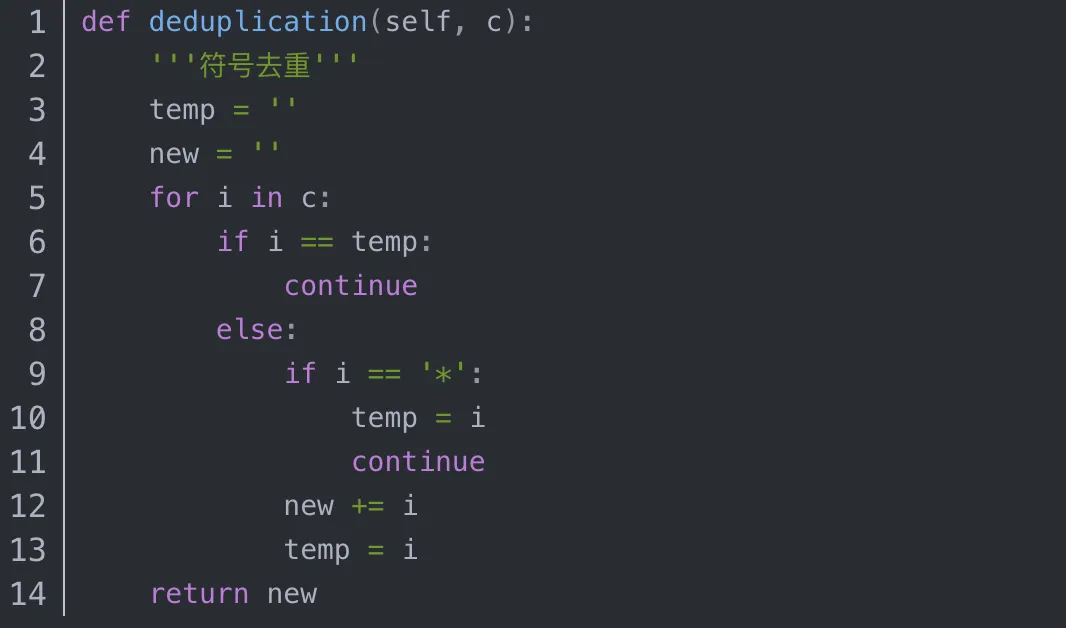
按照一般侦测网络，推理即可。只是，多了一步将反射变换矩阵转换为边框位置的计算。

另外，在YOLO侦测到得测量图片传入该级进行车牌检测的时候，会做一步操作。代码见下，将车辆检测框的图片扣出，然后resize到长宽均为16的整数倍。



### 2、序列检测网络的推理

对网络输出的序列，进行去重操作即可，如间隔标识符为“\*”时：



完整代码

https://github.com/HibikiJie/LicensePlate

原文地址

https://blog.csdn.net/weixin\_48866452/article/details/120319588

# Ch6.transformer在图像分类上的应用以及pytorch代码实现\_transformer 图片分类

python\_知世 于 2024-06-30 14:45:56 发布

原文链接：<https://blog.csdn.net/zhishi0000/article/details/140080859>

本文简单介绍transformers的原理，主要介绍transformers如何应用在图像分类上的任务。

## 1.对transformers的简单介绍

transformers在自然语言处理领域有着天然的优势，transformers改进了RNN(循环神经网络)训练慢，不能够建立序列之间的长期依赖，记忆消失的缺点。transformers的核心在是self-attention,输入一组序列，它能够平行的计算并平行的输出另一组经过编码的序列。

### 1.1序列数据的介绍（seq2seq）

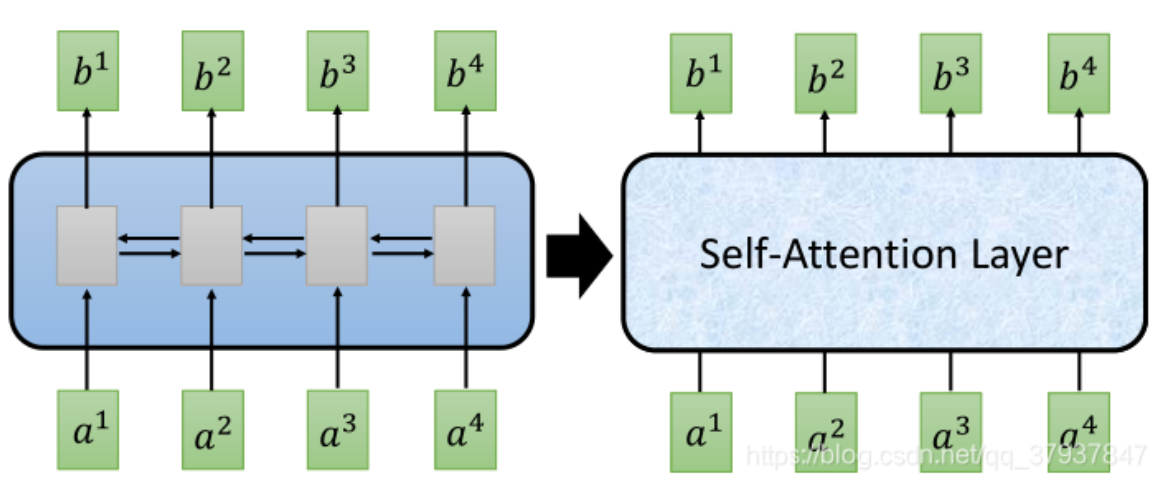
比如说在机器翻译领域：

输入的数据是一整串（单词）的句子，那么首先就需要对单词进行编码，将每个单词用一个固定维度的向量（比如512维度）来表示，网络的输入的是一个句子，那么就是一个向量列表，而列表的长度应该是训练集中最长的句子的长度。

在图像分类的领域：类比机器翻译领域，一张图片就是一个句子，然后将图片进行切块（可以用卷积来完成），切成n\_patch快，那么每一个图像块就是一个单词。

比如（bs,3,224,224)经过卷积–>(bs,768,14,14)—>(bs,768,196)

### 1.2self-Attention

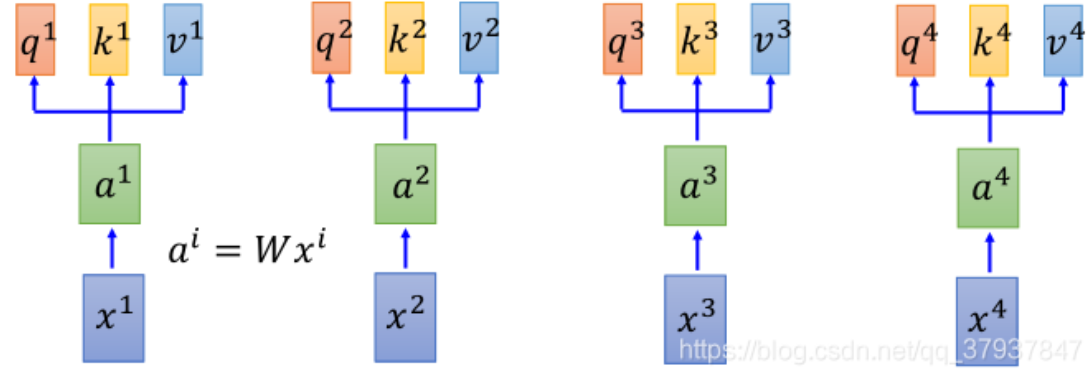


左边的网络结构，是最开始的RNN循环神经网络，b的输出不是平行的，需要等待前面的b生成结束才能继续后面的生成。

右边的是transformers中的Self-Attention结构，输入一组序列a1,a2,…,可以平行的输出另一组序列b1,b2,b3,…。

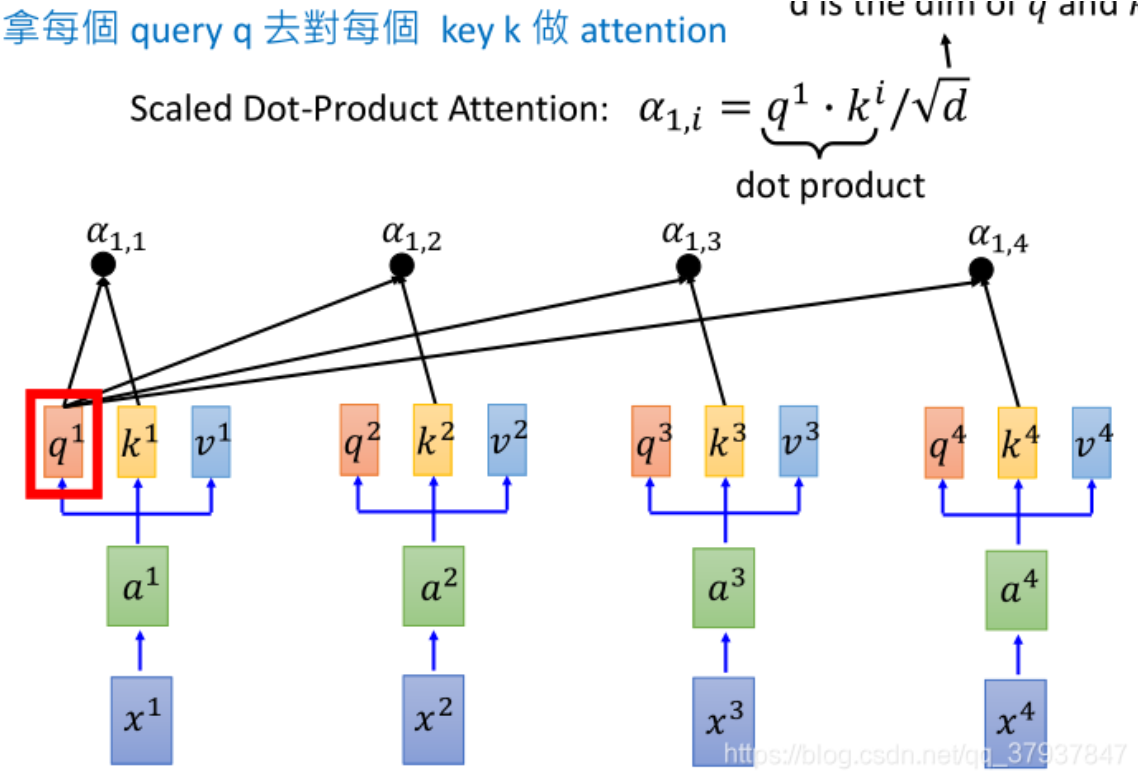
那么在self-attention具体做了哪些操作呢？

上图的a1,a2,…都是一个固定维度的向量，表示一个词向量。



首先将a通过3个不同的矩阵（Q,K,V）映射成三个不同的向量q(查询向量),k（匹配向量）,v（该单词的信息向量），

然后拿每一个q与其他的k进行点乘（做attention）:

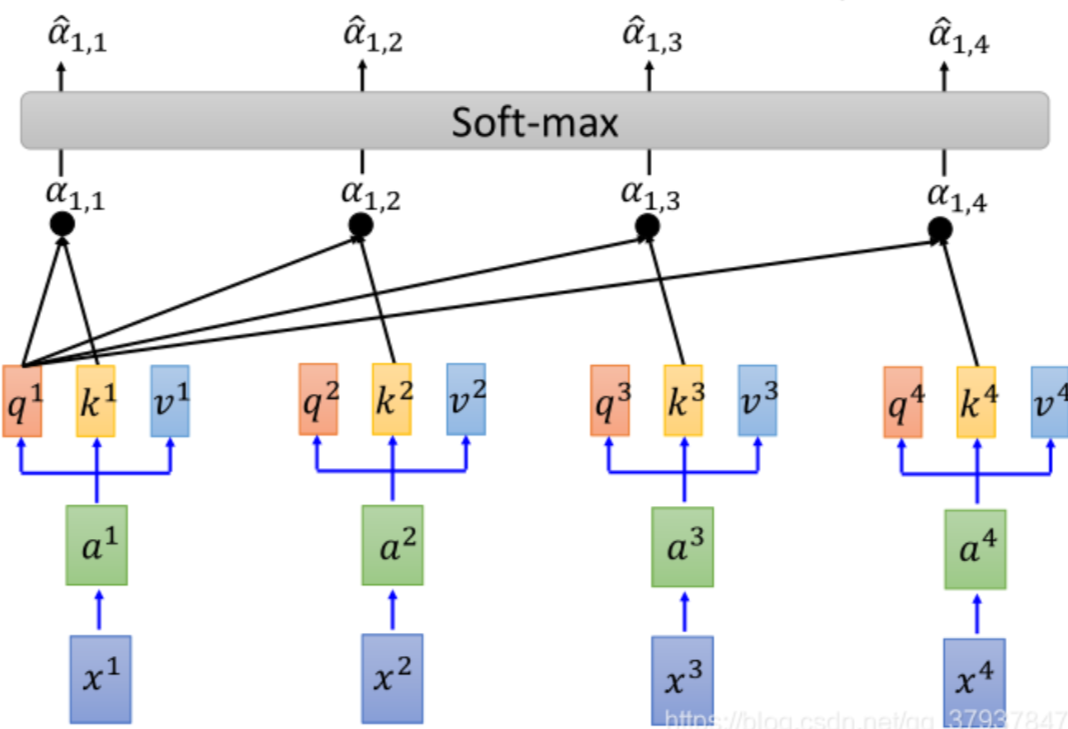


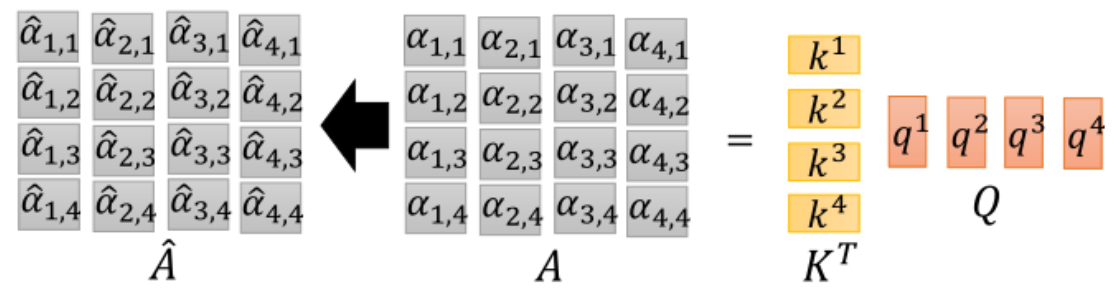
点乘的结果：

a1.1,a1.2,a1.3,…,都是标量，代表着当前a1这个单词与其他单词之间的某种attenttion关系（相关程度的打分值），然后再将q2与每一个k做点乘，得到a2.1,a2.2,a2.3,…,代表着a2这个单词与其他单词的关系。这样对一句话中所有的单词进行同样的操作，那么就可以建立所有单词之间的某种attention关系。

实际举例：如果一句话的长度为32个单词构成，那么进行上述的操作结果后，应该得到一个（32，32）的矩阵，每一行代表着，其他32个单词对当前行（单词）的贡献程度或者某种的attention关系。

然后在对点乘的结果做softmax()函数操作，将打分情况转换为0-1之间的概率。实际上就是表示，一句话中，其他单词对当前这个单词的贡献程度。



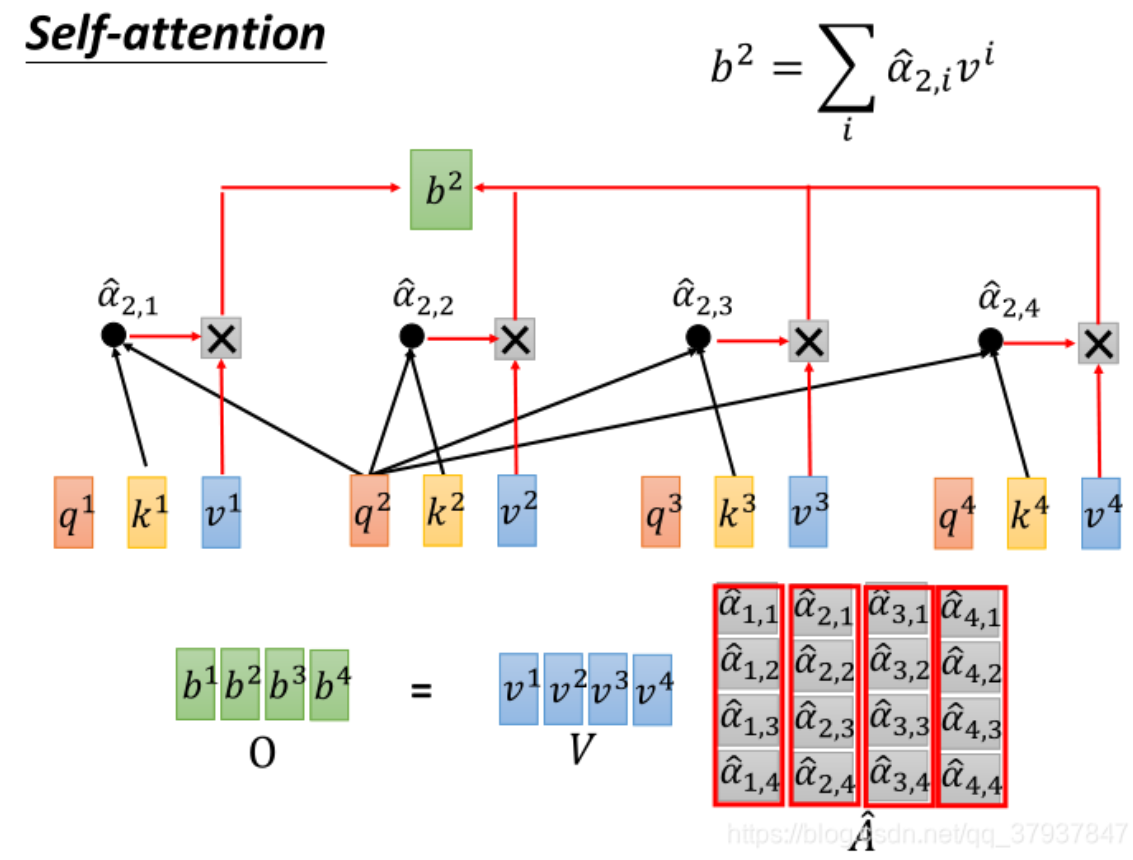
经过上述的操作，可获得：  


这里A’为（32,32）,每一列的值的和为1，每一个值代表第几个单词对当前列的单词的贡献程度

假设单词的信息向量为64维度(这个维度可以自己调整，不需要和前面的Q,K的维度一致)，一句话的单词长度为32，那么句子的信息矩阵Value为（32,64）

接下来将每个信息向量乘以softmax分数(这是为了准备之后将它们求和)。这里的直觉是希望关注语义上相关的单词，并弱化不相关的单词(例如，让它们乘以0.001这样的小数)。

最后是对加权值向量求和（译注：自注意力的另一种解释就是在编码某个单词时，就是将所有单词的表示（单词的信息向量）进行加权求和，而权重是通过该词的表示（键向量）与被编码词表示（查询向量）的点积并通过softmax得到。），然后即得到自注意力层在该位置的输出(在我们的例子中是对于第一个单词)。



#这里的v为向量，具体操作时代码，可以写成下列形式

query\_layer#（bs,32,64)

key\_layer#(bs,32,64)

value\_layer#(bs,32,64)

#1.首先q和k进行匹配相乘，得到每个单词之前的关系

attention\_scores = torch.matmul(query\_layer, key\_layer.transpose(-1, -2))# 将q向量和k向量进行相乘（bs,32,32)

#2.将得分情况进行softmax()操作

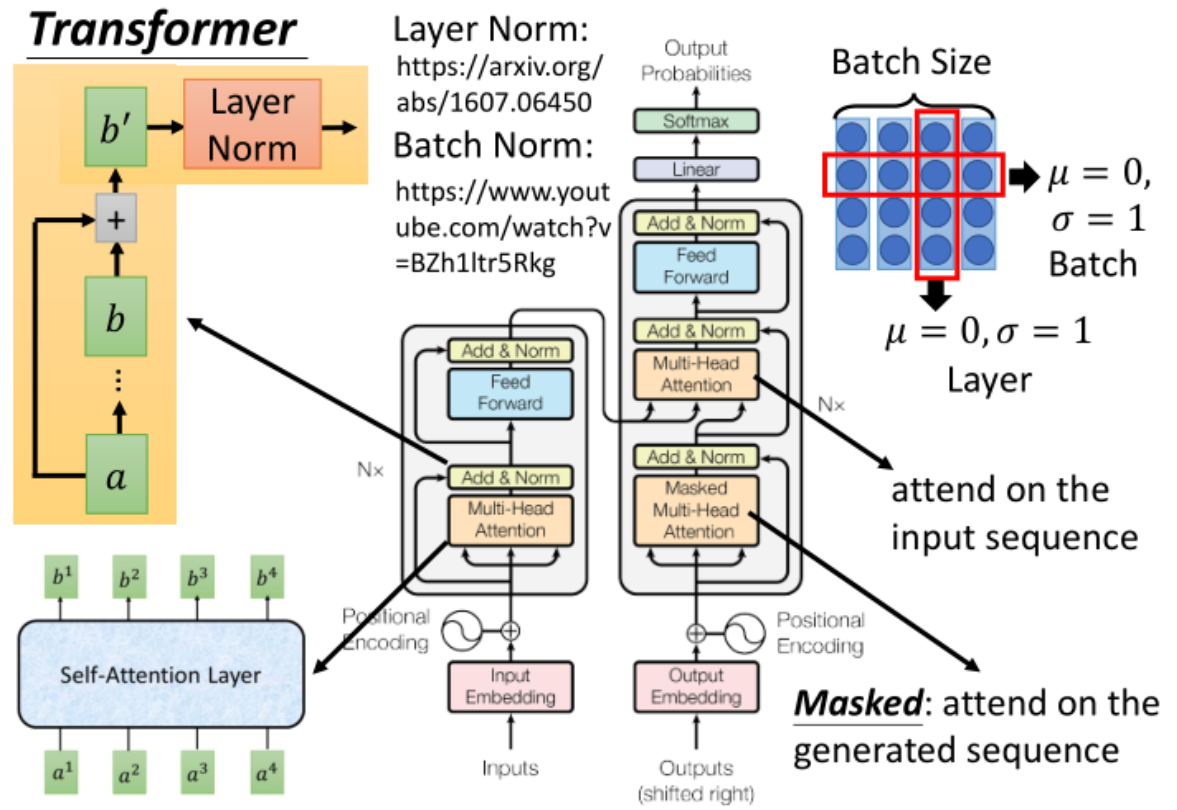
attention\_scores = attention\_scores / math.sqrt(attention\_head\_size)  #将结果除以向量维数的开方

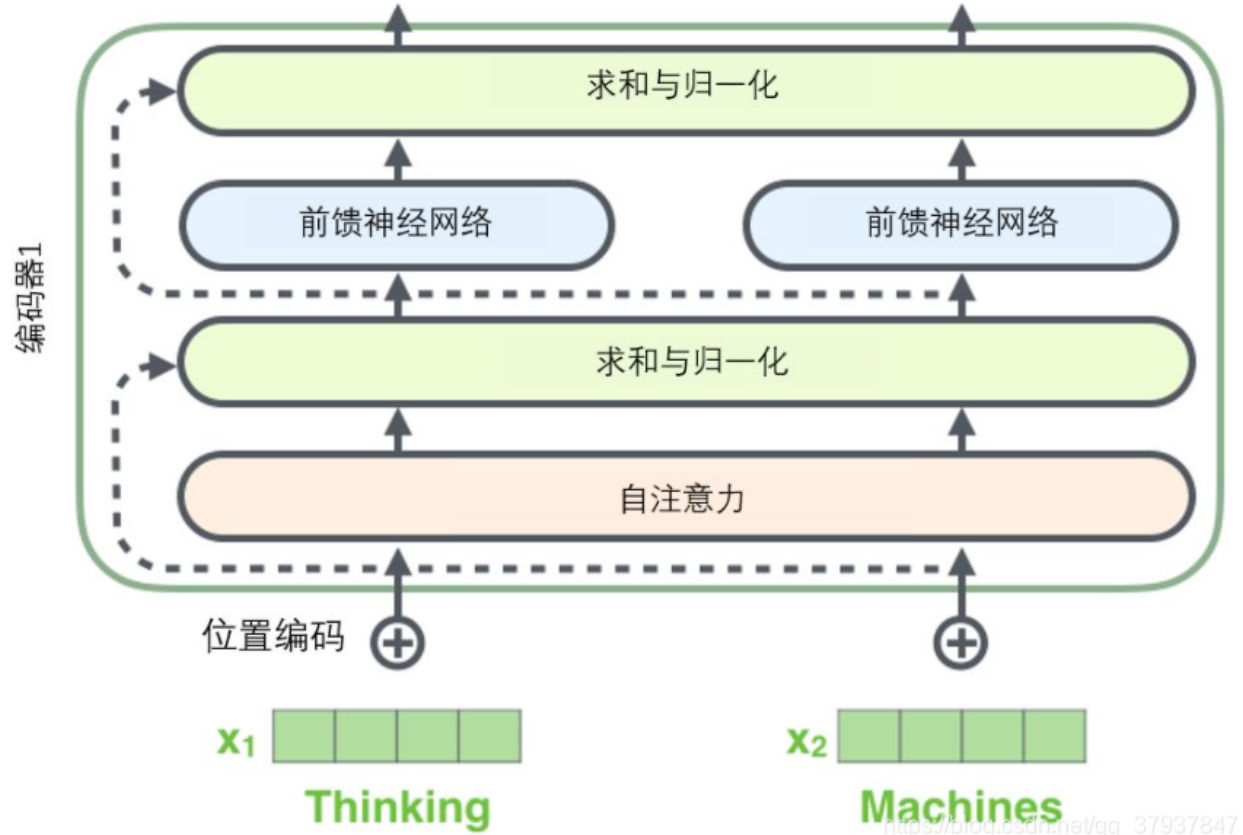
attention\_probs = softmax(attention\_scores)  # 将得到的分数进行softmax,得到概率

#3.将概率与value详细进行相乘

context\_layer = torch.matmul(attention\_probs, value\_layer) #(bs,32,64)

### 1.3 transformer的完整结构





## **2.transformers在图像分类上的pytorch代码**

### **2.1加载cifar10数据集**

#1.加载cifar10数据集，返回的是train\_loader,test\_loader

def get\_loader(args):

    #设置数据加载时的变换形式，包括撞转成tensor,裁剪，归一化

    transform\_train=transforms.Compose([

        transforms.RandomResizedCrop((args.img\_size,args.img\_size),scale=(0.05,1.0)),

        transforms.ToTensor(),

        transforms.Normalize(mean=[0.5,0.5,0.5],std=[0.5,0.5,0.5])

    ])

    transform\_test = transforms.Compose([

        transforms.Resize((args.img\_size, args.img\_size)),

        transforms.ToTensor(),

        transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])

    ])

    #默认使用cifar10数据集

    if args.dataset=="cifar10":

        trainset=datasets.CIFAR10(root=r'../data',

                                  train=True,

                                  download=False,

                                  transform=transform\_train)

        testset=datasets.CIFAR10(root=r'../data',

                                 train=False,

                                 download=False,

                                 transform=transform\_train)

    else:

        trainset = datasets.CIFAR100(root='./data',

                                     train=True,

                                     download=True,

                                     transform=transform\_train)

        testset = datasets.CIFAR100(root='./data',

                                    train=False, download=True,

                                    transform=transform\_train)

    print("train number:",len(trainset))

    print("test number:",len(testset))

    train\_loader=DataLoader(trainset,batch\_size=args.train\_batch\_size,shuffle=True)

    test\_loader=DataLoader(testset,batch\_size=args.eval\_batch\_size,shuffle=False)

    print("train\_loader:",len(train\_loader))

    print("test\_loader:",len(test\_loader))

    return train\_loader,test\_loader

进行测试：

#定义一个实例配置文件

parser = argparse.ArgumentParser()

parser.add\_argument("--dataset", choices=["cifar10", "cifar100"], default="cifar10")

parser.add\_argument("--img\_size", type=int, default=224,)

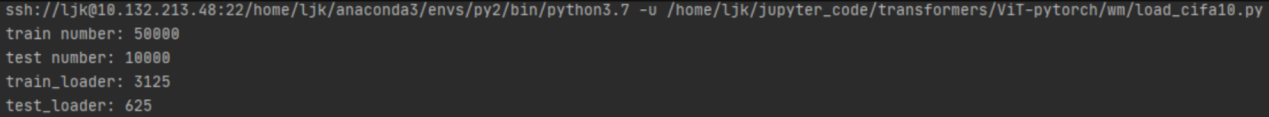
parser.add\_argument("--train\_batch-size", default=16, type=int,)

parser.add\_argument("--eval\_batch-size", default=16, type=int,)

args = parser.parse\_args()

get\_loader(args)

结果为：



### 2.2构建transformers模型

#### 2.2.1构建图像编码模块 Embeddings

class Embeddings(nn.Module):

    '''

    对图像进行编码，把图片当做一个句子，把图片分割成块，每一块表示一个单词

    '''

    def \_\_init\_\_(self,config,img\_size,in\_channels=3):

        super(Embeddings,self).\_\_init\_\_()

        img\_size=img\_size#224

        patch\_size=config.patches["size"]#16

        ##将图片分割成多少块（224/16）\*（224/16）=196

        n\_patches=(img\_size//patch\_size)\*(img\_size//patch\_size)

        #对图片进行卷积获取图片的块，并且将每一块映射成config.hidden\_size维（768）

        self.patch\_embeddings=Conv2d(in\_channels=in\_channels,

                                     out\_channels=config.hidden\_size,

                                     kernel\_size=patch\_size,

                                     stride=patch\_size)

        #设置可学习的位置编码信息，（1,196+1,786）

        self.position\_embeddings=nn.Parameter(torch.zeros(1,

                                                          n\_patches+1,

                                                          config.hidden\_size))

        #设置可学习的分类信息的维度

        self.classifer\_token=nn.Parameter(torch.zeros(1,1,config.hidden\_size))

        self.dropout=Dropout((config.transformer["dropout\_rate"]))

    def forward(self,x):

        bs=x.shape[0]

        cls\_tokens=self.classifer\_token.expand(bs,-1,-1)(bs,1,768)

        x=self.patch\_embeddings(x)#（bs,768,14,14）

        x=x.flatten(2)#(bs,768,196)

        x=x.transpose(-1,-2)#(bs,196,768)

        x=torch.cat((cls\_tokens,x),dim=1)#将分类信息与图片块进行拼接（bs,197,768）

        embeddings=x+self.position\_embeddings#将图片块信息和对其位置信息进行相加(bs,197,768)

        embeddings=self.dropout(embeddings)

        return  embeddings

进行测试：

def get\_config():

    """Returns the ViT-B/16 configuration."""

    config = ml\_collections.ConfigDict()

    config.patches = ml\_collections.ConfigDict({'size':16})

    config.hidden\_size = 768

    config.transformer = ml\_collections.ConfigDict()

    config.transformer.mlp\_dim = 3072

    config.transformer.num\_heads = 12

    config.transformer.num\_layers = 12

    config.transformer.attention\_dropout\_rate = 0.0

    config.transformer.dropout\_rate = 0.1

    config.classifier = 'token'

    config.representation\_size = None

    return config

config=get\_config()

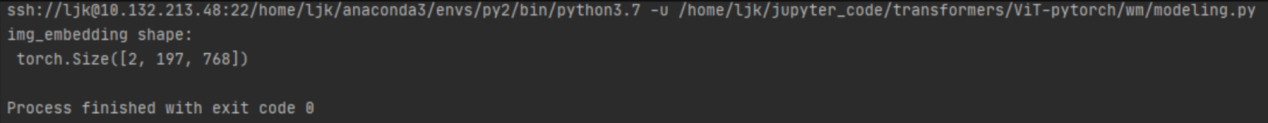
embedding=Embeddings(config,img\_size=224)

#模拟图片信息

img=torch.rand(2,3,224,224)

out\_embedding=embedding(img)

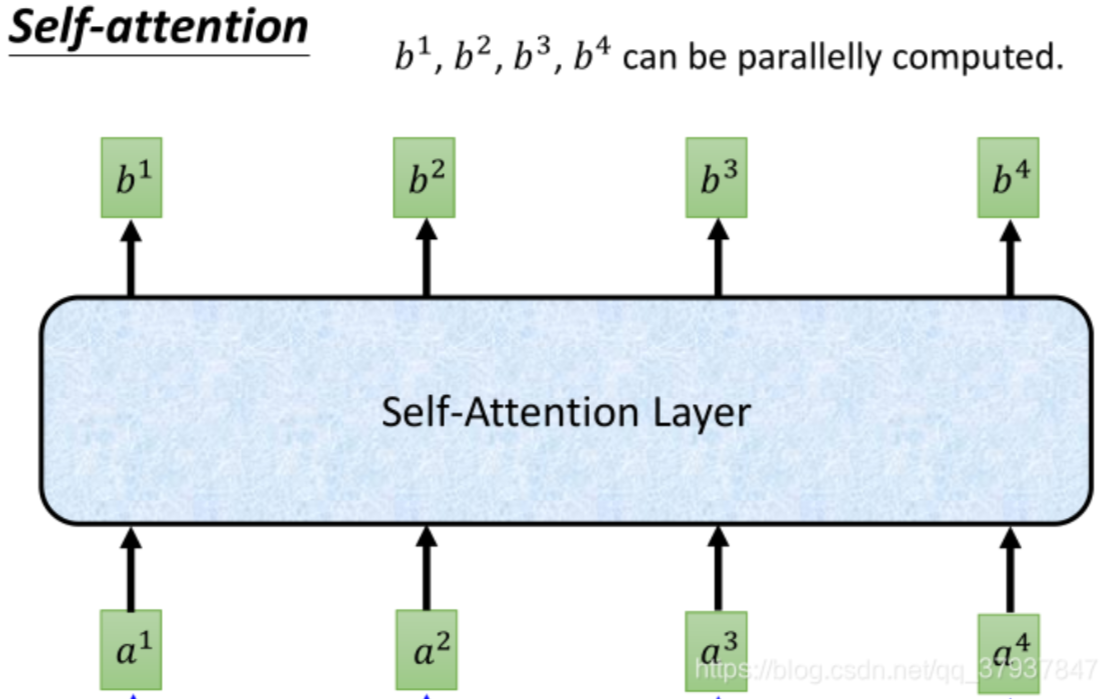
print("img\_embedding shape:\n",out\_embedding.shape)

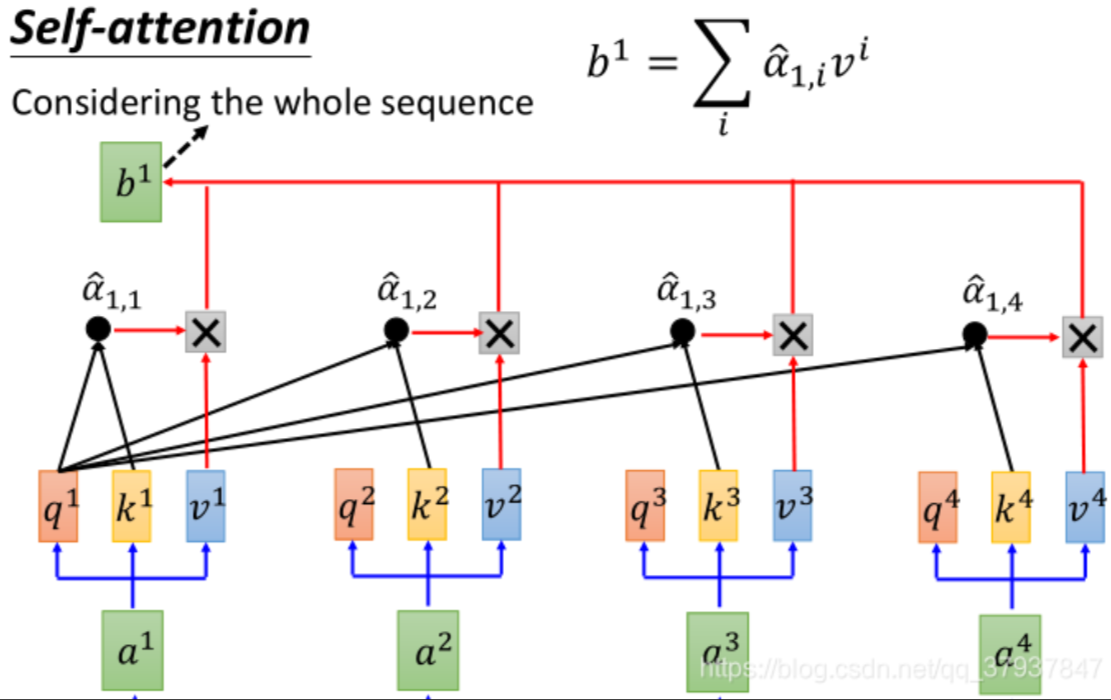


注：这个模块实际就已经完成将图片分割成块，每一块类比nlp领域就是一个词向量，在本代码中，这个词向量的维度

是config.hidden\_size=768.而图片相当于一个句子，句子的长度就是图片分割成块的数目，这里是14x14=196块，这里每一个块的位置编码信息，采用可学习的编码方式加到了每个块的向量上。由于本文是将transformers应用在图像分类上的，所有这里添加了一个cls\_token(bs，1,768)这样一个维度，主要是用了完成后期分类任务所添加的

###2.2构建self-Attention模块





#2.构建self-Attention模块

class Attention(nn.Module):

    def \_\_init\_\_(self,config,vis):

        super(Attention,self).\_\_init\_\_()

        self.vis=vis

        self.num\_attention\_heads=config.transformer["num\_heads"]#12

        self.attention\_head\_size = int(config.hidden\_size / self.num\_attention\_heads)  # 768/12=64

        self.all\_head\_size = self.num\_attention\_heads \* self.attention\_head\_size  # 12\*64=768

        self.query = Linear(config.hidden\_size, self.all\_head\_size)#wm,768->768，Wq矩阵为（768,768）

        self.key = Linear(config.hidden\_size, self.all\_head\_size)#wm,768->768,Wk矩阵为（768,768）

        self.value = Linear(config.hidden\_size, self.all\_head\_size)#wm,768->768,Wv矩阵为（768,768）

        self.out = Linear(config.hidden\_size, config.hidden\_size)  # wm,768->768

        self.attn\_dropout = Dropout(config.transformer["attention\_dropout\_rate"])

        self.proj\_dropout = Dropout(config.transformer["attention\_dropout\_rate"])

        self.softmax = Softmax(dim=-1)

    def transpose\_for\_scores(self, x):

        new\_x\_shape = x.size()[:-1] + (

        self.num\_attention\_heads, self.attention\_head\_size)  # wm,(bs,197)+(12,64)=(bs,197,12,64)

        x = x.view(\*new\_x\_shape)

        return x.permute(0, 2, 1, 3)  # wm,(bs,12,197,64)

    def forward(self, hidden\_states):

        # hidden\_states为：(bs,197,768)

        mixed\_query\_layer = self.query(hidden\_states)#wm,768->768

        mixed\_key\_layer = self.key(hidden\_states)#wm,768->768

        mixed\_value\_layer = self.value(hidden\_states)#wm,768->768

        query\_layer = self.transpose\_for\_scores(mixed\_query\_layer)#wm，(bs,12,197,64)

        key\_layer = self.transpose\_for\_scores(mixed\_key\_layer)

        value\_layer = self.transpose\_for\_scores(mixed\_value\_layer)

        attention\_scores = torch.matmul(query\_layer, key\_layer.transpose(-1, -2))#将q向量和k向量进行相乘（bs,12,197,197)

        attention\_scores = attention\_scores / math.sqrt(self.attention\_head\_size)#将结果除以向量维数的开方

        attention\_probs = self.softmax(attention\_scores)#将得到的分数进行softmax,得到概率

        weights = attention\_probs if self.vis else None#wm,实际上就是权重

        attention\_probs = self.attn\_dropout(attention\_probs)

        context\_layer = torch.matmul(attention\_probs, value\_layer)#将概率与内容向量相乘

        context\_layer = context\_layer.permute(0, 2, 1, 3).contiguous()

        new\_context\_layer\_shape = context\_layer.size()[:-2] + (self.all\_head\_size,)#wm,(bs,197)+(768,)=(bs,197,768)

        context\_layer = context\_layer.view(\*new\_context\_layer\_shape)

        attention\_output = self.out(context\_layer)

        attention\_output = self.proj\_dropout(attention\_output)

        return attention\_output, weights#wm,(bs,197,768),(bs,197,197)

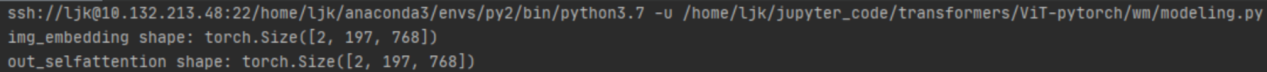
进行测试self-attention

attention=Attention(config,vis=True)

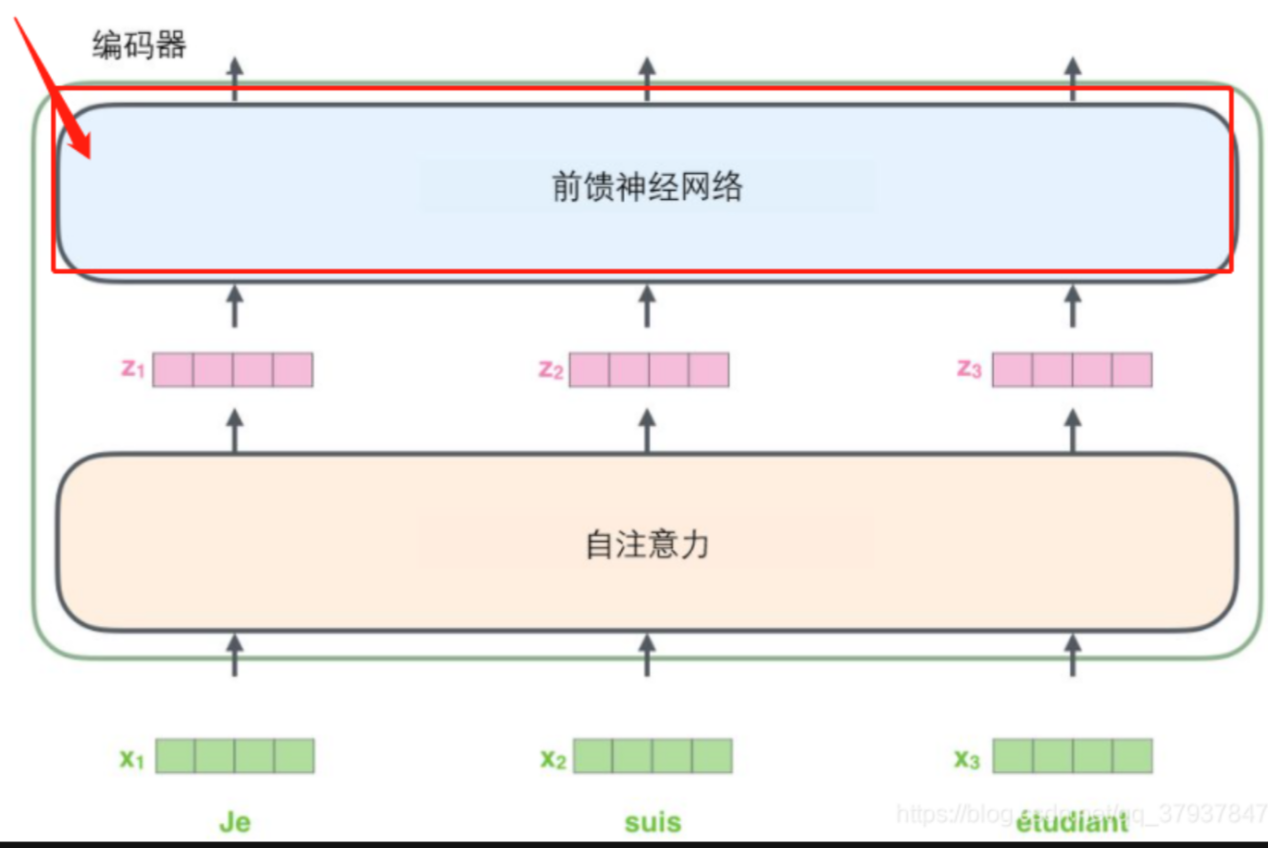
out\_selfattention,\_=attention(out\_embedding)

print("out\_selfattention shape:",out\_selfattention.shape)

结果：



#### 2.2.3构建前向传播神经网络模块



#3.构建前向传播神经网络

#两个全连接神经网络，中间加了激活函数

class Mlp(nn.Module):

    def \_\_init\_\_(self, config):

        super(Mlp, self).\_\_init\_\_()

        self.fc1 = Linear(config.hidden\_size, config.transformer["mlp\_dim"])#wm,786->3072

        self.fc2 = Linear(config.transformer["mlp\_dim"], config.hidden\_size)#wm,3072->786

        self.act\_fn = torch.nn.functional.gelu#wm,激活函数

        self.dropout = Dropout(config.transformer["dropout\_rate"])

        self.\_init\_weights()

    def \_init\_weights(self):

        nn.init.xavier\_uniform\_(self.fc1.weight)

        nn.init.xavier\_uniform\_(self.fc2.weight)

        nn.init.normal\_(self.fc1.bias, std=1e-6)

        nn.init.normal\_(self.fc2.bias, std=1e-6)

    def forward(self, x):

        x = self.fc1(x)#wm,786->3072

        x = self.act\_fn(x)#激活函数

        x = self.dropout(x)#wm,丢弃

        x = self.fc2(x)#wm3072->786

        x = self.dropout(x)

        return x

测试mlp模块：

##################测试3.MLP模块###########################################################

mlp=Mlp(config)

out\_mlp=mlp(out\_selfattention)

print("out\_mlp shape:",out\_mlp.shape)

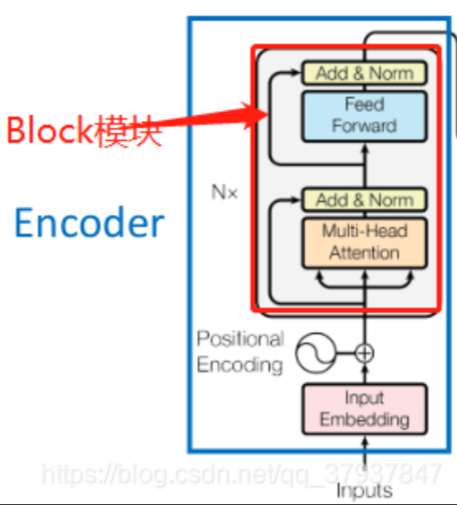
结果：



##### **2.2.4构建编码器的可重复利用Block模块**

该模块是可以重复利用的，原始论文中，堆叠了6个该模块，构成一个transformers的编码器

该模块包含了self-attention模块和mlp模块,还有残差结构，层归一化



#4.构建编码器的可重复利用的Block()模块：每一个block包含了self-attention模块和MLP模块

class Block(nn.Module):

    def \_\_init\_\_(self, config, vis):

        super(Block, self).\_\_init\_\_()

        self.hidden\_size = config.hidden\_size#wm,768

        self.attention\_norm = LayerNorm(config.hidden\_size, eps=1e-6)#wm，层归一化

        self.ffn\_norm = LayerNorm(config.hidden\_size, eps=1e-6)

        self.ffn = Mlp(config)

        self.attn = Attention(config, vis)

    def forward(self, x):

        h = x

        x = self.attention\_norm(x)

        x, weights = self.attn(x)

        x = x + h#残差结构

        h = x

        x = self.ffn\_norm(x)

        x = self.ffn(x)

        x = x + hh#残差结构

        return x, weights

测试Block模块：

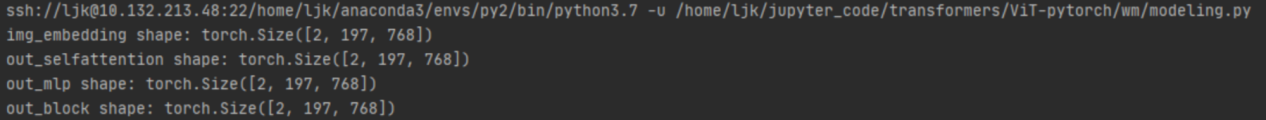
#############################4.测试Block模块###########################

block=Block(config,vis=True)

out\_block,\_=block(out\_embedding)

print("out\_block shape:",out\_block.shape)

结果：



##### **2.2.5构建Encoder模块**

该模块实际上就是堆叠N个Block模块,结构图如上图所示：

#5.构建Encoder模块，该模块实际上就是堆叠N个Block模块

class Encoder(nn.Module):

    def \_\_init\_\_(self, config, vis):

        super(Encoder, self).\_\_init\_\_()

        self.vis = vis

        self.layer = nn.ModuleList()

        self.encoder\_norm = LayerNorm(config.hidden\_size, eps=1e-6)

        for \_ in range(config.transformer["num\_layers"]):

            layer = Block(config, vis)

            self.layer.append(copy.deepcopy(layer))

    def forward(self, hidden\_states):

        attn\_weights = []

        for layer\_block in self.layer:

            hidden\_states, weights = layer\_block(hidden\_states)

            if self.vis:

                attn\_weights.append(weights)

        encoded = self.encoder\_norm(hidden\_states)

        return encoded, attn\_weights

测试Encoder模块：

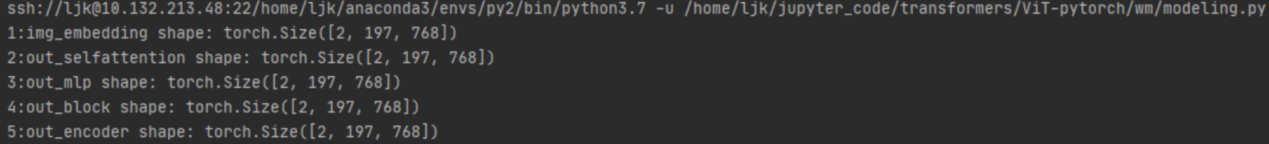
##################5.测试Encoder模块############################

encoder=Encoder(config,vis=True)

out\_encoder,\_=encoder(out\_embedding)

print("5:out\_encoder shape:",out\_encoder.shape)

结果：



##### **2.2.6 构建完整的transformers**

#6构建transformers完整结构，首先图片被embedding模块编码成序列数据，然后送入Encoder中进行编码

class Transformer(nn.Module):

    def \_\_init\_\_(self, config, img\_size, vis):

        super(Transformer, self).\_\_init\_\_()

        self.embeddings = Embeddings(config, img\_size=img\_size)#wm,对一幅图片进行切块编码，得到的是（bs,n\_patch+1（196）,每一块的维度（768））

        self.encoder = Encoder(config, vis)

    def forward(self, input\_ids):

        embedding\_output = self.embeddings(input\_ids)#wm,输出的是（bs,196,768)

        encoded, attn\_weights = self.encoder(embedding\_output)#wm,输入的是（bs,196,768)

        return encoded, attn\_weights#输出的是（bs,197,768）

测试transformers

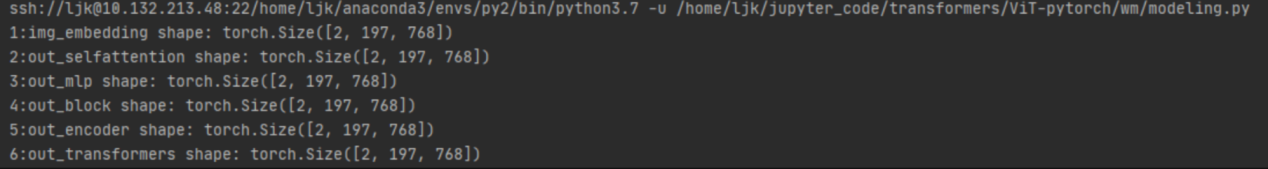
###################6.测试transformers的完整结构#########################

#transformers的输入数据是图像数据,输出的数据（bs,197,768）中197的第一个的向量（768）可以用于图像分类

transformers=Transformer(config,img\_size=224,vis=True)

out\_transformers,\_=transformers(img)

print("6:out\_transformers shape:",out\_transformers.shape)



##### **2.2.7构建VisionTransformers,用于图像分类**

#7构建VisionTransformer，用于图像分类

class VisionTransformer(nn.Module):

    def \_\_init\_\_(self, config, img\_size=224, num\_classes=21843, zero\_head=False, vis=False):

        super(VisionTransformer, self).\_\_init\_\_()

        self.num\_classes = num\_classes

        self.zero\_head = zero\_head

        self.classifier = config.classifier

        self.transformer = Transformer(config, img\_size, vis)

        self.head = Linear(config.hidden\_size, num\_classes)#wm,768-->10

    def forward(self, x, labels=None):

        x, attn\_weights = self.transformer(x)

        logits = self.head(x[:, 0])

        #如果传入真实标签，就直接计算损失值

        if labels is not None:

            loss\_fct = CrossEntropyLoss()

            loss = loss\_fct(logits.view(-1, self.num\_classes), labels.view(-1))

            return loss

        else:

            return logits, attn\_weights

测试VisionTransformer模块：

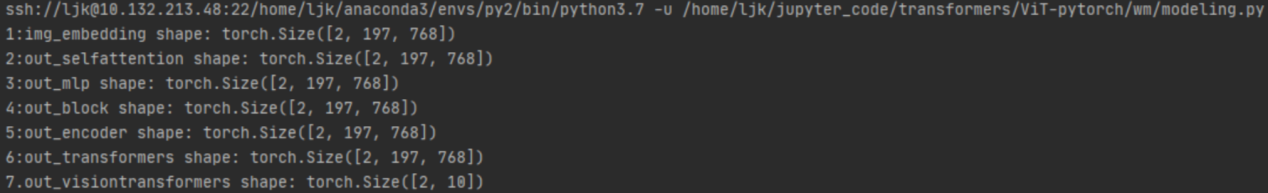
#################7.验证visionTransformers###########################

visiontransformer=VisionTransformer(config,num\_classes=10,vis=True)

out\_visiontransformer,\_=visiontransformer(img)

print("7.out\_visiontransformers shape:",out\_visiontransformer.shape)

结果：



##2.3.利用VisionTransformer模块，训练[图像分类模型](https://so.csdn.net/so/search?q=%E5%9B%BE%E5%83%8F%E5%88%86%E7%B1%BB%E6%A8%A1%E5%9E%8B&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/zhishi0000/article/details/_blank)

新建一个train.py文件

import ml\_collections

import argparse

from wm.modeling import VisionTransformer

from wm.load\_cifa10 import get\_loader

import torch

import os

import numpy as np

def get\_config():

    '''

    配置transformer的模型的参数

    '''

    config = ml\_collections.ConfigDict()

    config.patches = ml\_collections.ConfigDict({'size':16})

    config.hidden\_size = 768

    config.transformer = ml\_collections.ConfigDict()

    config.transformer.mlp\_dim = 3072

    config.transformer.num\_heads = 12

    config.transformer.num\_layers = 12

    config.transformer.attention\_dropout\_rate = 0.0

    config.transformer.dropout\_rate = 0.1

    config.classifier = 'token'

    config.representation\_size = None

    return config

def save\_model(args, model,epoch\_index):

    '''

    保存每个epoch训练的模型

    '''

    model\_to\_save = model.module if hasattr(model, 'module') else model

    model\_checkpoint = os.path.join(args.output\_dir, "epoch%s\_checkpoint.bin" % epoch\_index)

    torch.save(model\_to\_save.state\_dict(), model\_checkpoint)

#实例化模型

def getVisionTransformers\_model(args):

    config=get\_config()#获取模型的配置文件

    num\_classes = 10 if args.dataset == "cifar10" else 100

    model = VisionTransformer(config, args.img\_size, zero\_head=True, num\_classes=num\_classes)

    model.to(args.device)

    return args,model

#用测试集评估模型的训练好坏

def eval(args,model,test\_loader):

    eval\_loss=0.0

    total\_acc=0.0

    model.eval()

    loss\_function = torch.nn.CrossEntropyLoss()

    for i,batch in enumerate(test\_loader):

        batch = tuple(t.to(args.device) for t in batch)

        x, y = batch

        with torch.no\_grad():

            logits,\_= model(x)#model返回的是（bs,num\_classes）和weight

            batch\_loss=loss\_function(logits,y)

            #记录误差

            eval\_loss+=batch\_loss.item()

            #记录准确率

            \_,preds= logits.max(1)

            num\_correct=(preds==y).sum().item()

            total\_acc+=num\_correct

    loss=eval\_loss/len(test\_loader)

    acc=total\_acc/(len(test\_loader)\*args.eval\_batch\_size)

    return loss,acc

def train(args,model):

    print("load dataset.........................")

    #加载数据

    train\_loader, test\_loader = get\_loader(args)

    # Prepare optimizer and scheduler

    optimizer = torch.optim.SGD(model.parameters(),

                                lr=args.learning\_rate,

                                momentum=0.9,

                                weight\_decay=args.weight\_decay)

    print("training.........................")

    #设置测试损失list,和测试acc 列表

    val\_loss\_list=[]

    val\_acc\_list=[]

    #设置训练损失list

    train\_loss\_list=[]

    for i in range(args.total\_epoch):

        model.train()

        train\_loss=0

        for step, batch in enumerate(train\_loader):

            batch = tuple(t.to(args.device) for t in batch)

            x, y = batch

            loss = model(x, y)

            train\_loss +=loss.item()

            loss.backward()

            optimizer.step()

            optimizer.zero\_grad()

        #每训练一个epoch,记录一次训练损失

        train\_loss=train\_loss/len(train\_loader)

        train\_loss\_list.append(train\_loss)

        np.savetxt("train\_loss\_list.txt", train\_loss\_list)

        print("train Epoch:{},loss:{}".format(i,train\_loss))

        # 每个epcoh保存一次模型参数

        save\_model(args, model,i)

        # 每训练一个epoch,用当前训练的模型对验证集进行测试

        eval\_loss, eval\_acc = eval(args, model, test\_loader)

        #将每一个测试集验证的结果加入列表

        val\_loss\_list.append(eval\_loss)

        val\_acc\_list.append(eval\_acc)

        np.savetxt("val\_loss\_list.txt",val\_loss\_list)

        np.savetxt("val\_acc\_list.txt",val\_acc\_list)

        print("val Epoch:{},eval\_loss:{},eval\_acc:{}".format(i, eval\_loss, eval\_acc))

def main():

    parser = argparse.ArgumentParser()

    # Required parameters

    parser.add\_argument("--dataset", choices=["cifar10", "cifar100"], default="cifar10",

                        help="Which downstream task.")

    parser.add\_argument("--output\_dir", default="../output", type=str,

                        help="The output directory where checkpoints will be written.")

    parser.add\_argument("--img\_size", default=224, type=int,help="Resolution size")

    parser.add\_argument("--train\_batch\_size", default=32, type=int,

                        help="Total batch size for training.")

    parser.add\_argument("--eval\_batch\_size", default=32, type=int,

                        help="Total batch size for eval.")

    parser.add\_argument("--learning\_rate", default=3e-2, type=float,

                        help="The initial learning rate for SGD.")

    parser.add\_argument("--weight\_decay", default=0, type=float,

                        help="Weight deay if we apply some.")

    parser.add\_argument("--total\_epoch", default=1000, type=int,

                        help="Total number of training epochs to perform.")

    args = parser.parse\_args()

    device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

    args.device = device

    args,modle=getVisionTransformers\_model(args)

    train(args,modle)

if \_\_name\_\_ == "\_\_main\_\_":

    main()

后续待更新

## 3.参考文献

1.A N I MAGE IS W ORTH 16 X 16 W ORDS :T RANSFORMERS FOR I MAGE R ECOGNITION AT S CALE

2.https://blog.csdn.net/longxinchen\_ml/article/details/86533005

3.李宏毅transformers视频讲解