Transformer相关——（10）Transformer代码分析

发表于 2021-08-19 | 更新于 2021-11-09 | 深度学习
| 字数总计: 6,956 | 阅读时长: 34分钟 | 阅读量: 2443

Transformer相关——（10）Transformer代码分析

引言

原理是原理，道理大概都懂了，代码也不能落下。这篇就把Transformer代码拿出来分析一下。代码来源：Pytorch编写完整的Transformer，我进一步做了一些修改和补充。

和之前一样，先把每个小模块分析一下，然后再把它们串起。来构建一整个model。

主要包括以下几个部分：

1. ”input“ embedding

2 position encoding

3 Multi-Head attention

4 Add&Norm

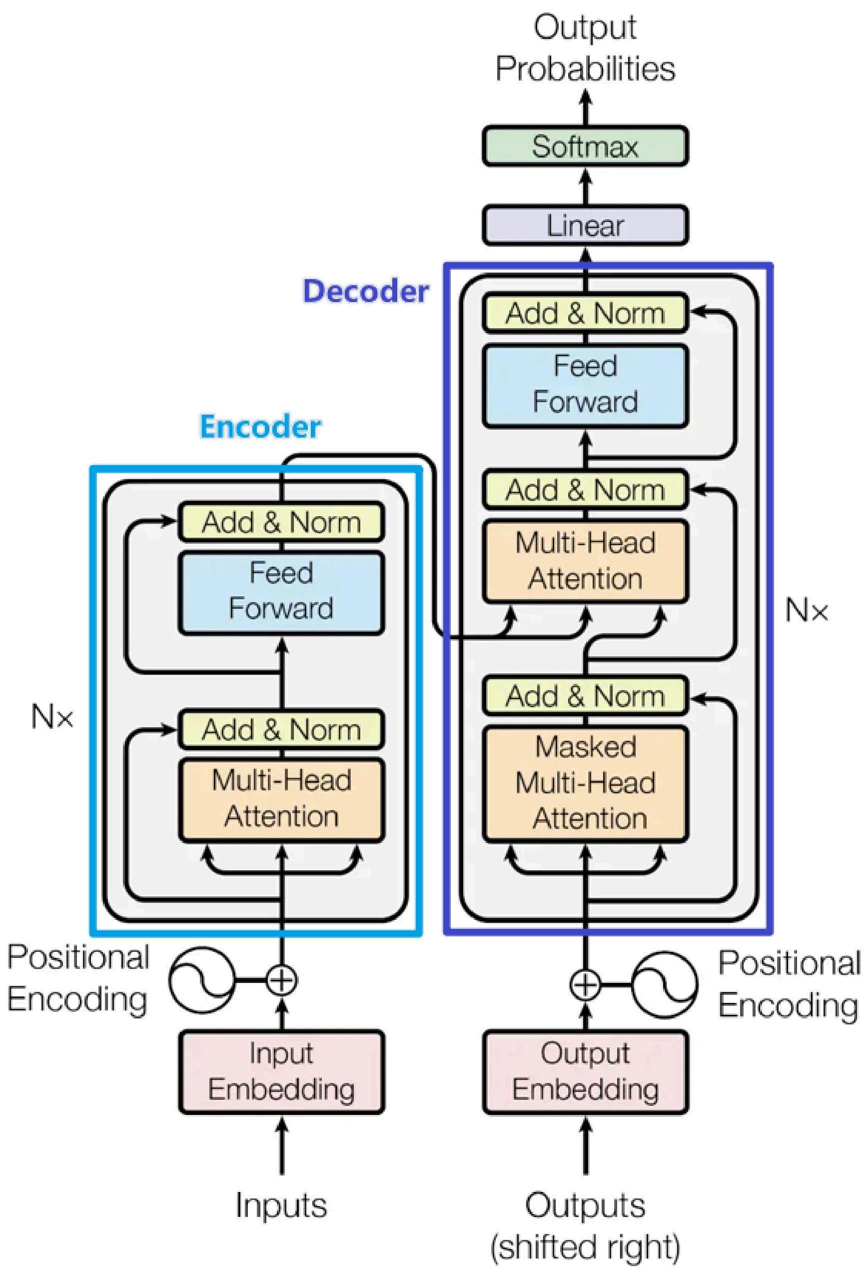


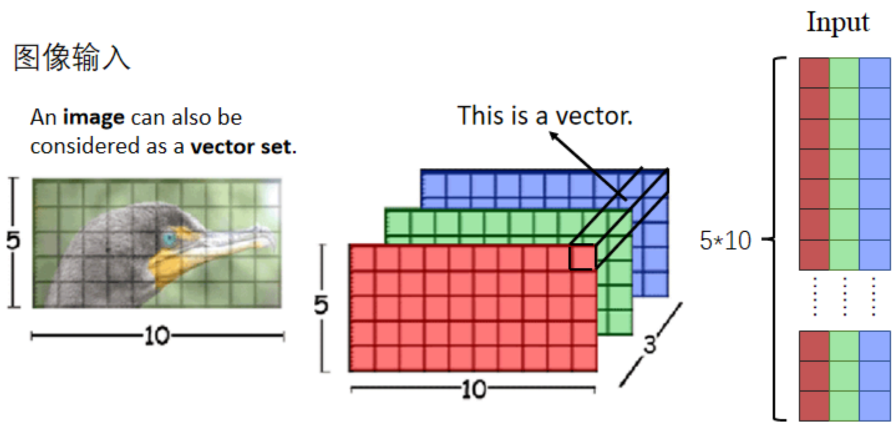
Figure 1: The Transformer - model architecture.

子模块

"input" embedding

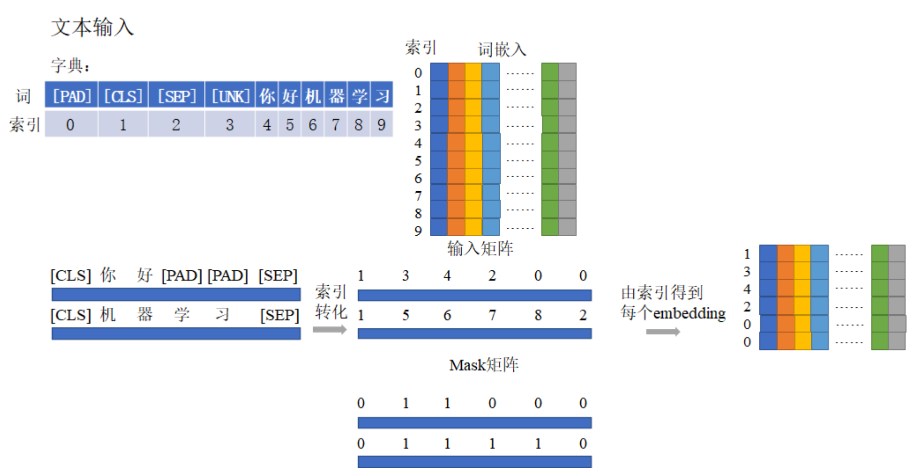
这一步其实就是之前说的"将真实问题转为数学问题"，比如输入的数据是图像、语音、文本、用户ID等等，把这些内容转成数值矩阵。

像图像这种自带数值输入数据，可以经过转换直接作为input embedding，如下图所示：



Source of image: https://www.researchgate.net/figure/Color-image-representation-and-RGB-matrix_fig15_282798184

对于语音、文本这一类数据，就需要用特殊的技巧转成数字信号输入。比如在文本中，就是使用**词嵌入**。示意图如下所示。



那么词嵌入（word embedding）怎么获得呢？

我觉得总的来说可以有三种方式：

1. **根据字典构建one-hot编码**，假设字典大小为N的话，这就是一个$N \times N$的矩阵，词对应索引位置的向量值为1，其他位置为0，每个词的表示向量相互垂直，没有语义信息，字典太大这个矩阵也会很大很稀疏。

2. **随机一个$N \times M$矩阵**，M为自定义的每个词的特征维度大小。在网络中不断学习或者用一些方法求每个词在训练语料中的语义特

征，最后输出每个词的embedding；经典的方法比如DeepWalk、Word2Vec等，深度学系的方法比如图神经网络等等都可以用于学习词嵌入。

3. **从预训练好的词嵌入中取。**像现在NLP还非常流行的范式是pre-trained+fine tuning，就是从一些已经训练好的词嵌入中取对应词的embedding，然后进行微调作不同场景下的下游任务。

词嵌入在Pytorch里基于 `torch.nn.Embedding` 实现，实例化时需要设置的参数为词表的大小和被映射的向量的维度， `embed = nn.Embedding(vocab_size,embed_dim)` 。 `vocab_size` 是词表的大小，词表中包括了一些特殊作用的字符（比如用于padding的[PAD]；表示句子开头的[CLS]；表示句子结束的[SEP]；表示字典中没有出现过的未知字符[UNK]等等）。

代码

```
import torch
import torch.nn as nn
X = torch.zeros((2,4),dtype=torch.long)
embed = nn.Embedding(10,8)
print(embed(X).shape)
```

position encoding位置编码

位置编码用以表达元素在序列中的位置特征，比如名词经常出现在句子开头。

位置编码直接与元素的embedding相加。

代码中需要注意：X_只是初始化的矩阵；**完成位置编码之后会加一个dropout**。另外，位置编码是最后加上去的，因此输入输出形状不变。

这里使用的是GPT-3中的相对位置编码：

$$\overrightarrow{p_t}^{(i)} = f(t)^{(i)} := \begin{cases} \sin(\omega_k . t), & \text{if } i = 2k \\ \cos(\omega_k . t), & \text{if } i = 2k+1 \end{cases}$$

其中，t为绝对位置，w_i为：

$$w_i = \frac{1}{10000^{2i/d_{model}}}$$

代码

```
Tensor = torch.Tensor
def positional_encoding(X, num_features, dropout_p=0.1, max_len=512) -> Tensor:
    r'''
        给输入加入位置编码
    参数：
        - embed_dim: 词嵌入维度
        - dropout_p: dropout的概率，当其为非零时执行dropout
        - max_len: 句子的最大长度，默认512

    形状：
        - 输入： [batch_size, seq_length, embed_dim]
        - 输出： [batch_size, seq_length, embed_dim]
        - seq_length表示这个batch中句子的长度（每个batch都做了截断或补齐操作）

    例子：
        >>> X = torch.randn((2,4,10))
        >>> X = positional_encoding(X, 10)
        >>> print(X.shape)
        >>> torch.Size([2, 4, 10])
    '''
    dropout = nn.Dropout(dropout_p)
    P = torch.zeros((1,max_len,num_features))
    p_t = torch.arange(max_len,dtype=torch.float32).reshape(-1,
/ torch.pow(
        10000,
        torch.arange(0,num_features,2,dtype=torch.float32) /num_features) #每个位置的t*w_k
    P[:,:,0::2] = torch.sin(p_t) #0::2偶数位
    P[:,:,1::2] = torch.cos(p_t) #0::1奇数位
    X = X + P[:,:X.shape[1],:].to(X.device) #位置
```

```
码和初始embedding直接相加
    return dropout(X)
```

```
# 位置编码例子
X = torch.randn((2,4,10)) # (batch_size,seq_length,embed_dim)
X = positional_encoding(X, 10)
print(X.shape)
```

Multi-Head attention

我们先来分析attention机制的代码再转到Multi-Head attention。

attention机制

Transformer用缩放点积相关性计算attention score：

$$\alpha_{i,j} = \frac{(q^i \cdot k^j)}{\sqrt{d}}$$

当矩阵特征向量以行向量形式表示时，attention的输出矩阵可以按照下述公式计算（以缩放点积相关性+softmax为例）：

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

代码

```
from typing import Optional, Tuple, Any
Tensor = torch.Tensor
def _scaled_dot_product_attention(
    q: Tensor,
    k: Tensor,
    v: Tensor,
    attn_mask: Optional[Tensor] = None,
    dropout_p: float = 0.0,
) -> Tuple[Tensor, Tensor]:
    r'''
    在query, key, value上计算点积注意力，若有注意力遮盖则使用，并且应用一个概率为dropout_p的dropout
    参数：
```

```
            - q: shape:`(B, Nt, E)` B代表batch size， Nt是目标语言序列长度，E是嵌入后的特征维度
            - key: shape:`(B, Ns, E)` Ns是源语言序列长度
            - value: shape:`(B, Ns, E)`与key形状一样
            - attn_mask: 要么是3D的tensor，形状为:`(B, Nt, Ns)`或者2D的tensor，形状如:`(Nt, Ns)`
            - Output: attention values: shape:`(B, Nt, E)`，与q的形状一致;attention weights: shape:`(B, Nt, Ns)`

        例子：
            >>> q = torch.randn((2,3,6))
            >>> k = torch.randn((2,4,6))
            >>> v = torch.randn((2,4,6))
            >>> out = scaled_dot_product_attention(q, k, v)
            >>> out[0].shape, out[1].shape
            >>> torch.Size([2, 3, 6]) torch.Size([2, 3, 4])
        '''
        B, Nt, E = q.shape #每个词的特征向量是行向量
        q = q / math.sqrt(E)
        # (B, Nt, E) x (B, E, Ns) -> (B, Nt, Ns)
        attn = torch.bmm(q, k.transpose(-2,-1)) #Q·K^T
        if attn_mask is not None:
            attn = attn + attn_mask
        # attn意味着目标序列的每个词对源语言序列做注意力
        attn = F.softmax(attn, dim=-1) #softmax(Q·K^T)
        if dropout_p:
            attn = F.dropout(attn, p=dropout_p)
        # (B, Nt, Ns) x (B, Ns, E) -> (B, Nt, E)
        output = torch.bmm(attn, v) #softmax(Q·K^T)·V
        return output, attn
```

获得Q、K、V

之前提到序列向量输入attention之前还需要经过三个线性变换分别生成Q、K、V。

$$Q = W^q X + b^q \tag{1}$$
$$K = W^k X + b^k \tag{2}$$
$$V = W^v X + b^v \tag{3}$$

利用 `nn.functional.linear` 函数实现线性变换，与 `nn.Linear` 不同的是，前者可以提供权重矩阵和偏置，执行$y = xW^T + b$，而后者是可以自由决定输出的维度（比如下游分类任务是回归任务时，可设定输出维度为1）。

代码

把三个Q、K、V的权重拼在一起，用一个大的权重参数矩阵进行线性变换（也就是W=[W1;W2;W3]），生成Q、K、V使用的大权重参数矩阵不同的分块。

```
def _in_projection_packed(
        q: Tensor,
        k: Tensor,
        v: Tensor,
        w: Tensor,
        b: Optional[Tensor] = None,
) -> List[Tensor]:
        r"""
        用一个大的权重参数矩阵进行线性变换
        参数:
            q, k, v: 对自注意来说，三者都是src；对于seq2seq模型，k和v是一致的tensor。
                     但它们的最后一维(num_features或者叫做embed_dim)都必须保持一致。
            w: 用以线性变换的大矩阵，按照q,k,v的顺序压在一个tensor里面。
            b: 用以线性变换的偏置，按照q,k,v的顺序压在一个tensor里面。

        形状:
            输入:
            - q: shape:`(..., E)`，E是词嵌入的维度（下面出现的E均为此意）。
            - k: shape:`(..., E)`
            - v: shape:`(..., E)`
            - w: shape:`(E * 3, E)`
            - b: shape:`E * 3`
```

```
        
        输出:
        - 输出列表 :`[q', k', v']`，q,k,v经过线性变换前后的形状都一致。
    """
    E = q.size(-1)
    # 若为自注意，则q = k = v = src，因此它们的引用变量都是src
    # 即k is v和q is k结果均为True
    # 若为seq2seq，k = v，因而k is v的结果是True，但q!=k，也就是cross attention
    if k is v:
        if q is k: # 自注意力
            return F.linear(q, w, b).chunk(3, dim=-1)
        else:
            # seq2seq模型
            w_q, w_kv = w.split([E, E * 2])
            if b is None:
                b_q = b_kv = None
            else:
                b_q, b_kv = b.split([E, E * 2])
            return (F.linear(q, w_q, b_q),) + F.linear(k, w_kv, b_kv).chunk(2, dim=-1)
    else: #q!=k!=v
        w_q, w_k, w_v = w.chunk(3)
        if b is None:
            b_q = b_k = b_v = None
        else:
            b_q, b_k, b_v = b.chunk(3)
        return F.linear(q, w_q, b_q), F.linear(k, w_k, b_k), F.linear(v, w_v, b_v)
```

```
q, k, v = _in_projection_packed(query, key, value, in_proj_weight, in_proj_bias)
```

获得W^q, W^k, W^v（参数初始化）

由获得Q、K、V自然而然引出，如何获得$W^q, W^k, W^v, b^q, b^k, b^v$，也就是如何进行参数初始化。

代码

在PyTorch的源码里，projection代表是一种线性变换的意思，`in_proj_bias` 是指一开始的线性变换的偏置b^q，这里用 `q_proj_weight` 表示W^q，其他同理。

用 `torch.empty` 按照所给的形状形成对应的tensor，**相当于预留了一个位置，其填充的值还未初始化**，类比torch.randn（标准正态分布），这是一种初始化的方式。

在PyTorch中，tensor变量类型是无法修改值的，而Parameter()函数可以看作为一种类型转变函数，将不可改值的tensor转换为可训练可修改的模型参数（也就是可以随着模型一起学习训练），即与model.parameters绑定在一起，register_parameter的意思是是否将这个参数放到model.parameters，None表示没有这个参数。

if判断句用于判断q,k,v的最后一维是否一致，若一致，则将这个大的权重矩阵全部乘然后分割出来，若不是，则各初始化各的，初始化不会改变原来的形状（如$q = qW_q + b_q$，见注释）。

```
if self._qkv_same_embed_dim is False:
    # 初始化前后形状维持不变
    # (seq_length x embed_dim) x (embed_dim x embed_dim) ==> (seq_length x embed_dim)
    self.q_proj_weight = Parameter(torch.empty((embed_dim, embed_dim)))
    self.k_proj_weight = Parameter(torch.empty((embed_dim, self.kdim)))
    self.v_proj_weight = Parameter(torch.empty((embed_dim, self.vdim)))
    self.register_parameter('in_proj_weight', None)
else:
    self.in_proj_weight = Parameter(torch.empty((3 * embed_dim, embed_dim)))
    self.register_parameter('q_proj_weight', None)
    self.register_parameter('k_proj_weight', None)
    self.register_parameter('v_proj_weight', None)
```

```

if bias:
    self.in_proj_bias = Parameter(torch.empty(3 * embed_dim))
else:
    self.register_parameter('in_proj_bias', None)
# 后期会将所有头的注意力拼接在一起然后乘上权重矩阵输出
# out_proj是为了后期准备的
self.out_proj = nn.Linear(embed_dim, embed_dim, bias=bias)
self._reset_parameters()
```

然后用 `_reset_parameters()` 函数初始化参数数值。xavier_uniform 是从连续型均匀分布里面随机取样出值来作为初始化的值，xavier_normal_取样的分布是正态分布。正因为初始化值在训练神经网络的时候很重要，所以才需要这两个函数。

constant_意思是用所给值来填充输入的向量。

```
def _reset_parameters(self):
    if self._qkv_same_embed_dim:
        xavier_uniform_(self.in_proj_weight)
    else:
        xavier_uniform_(self.q_proj_weight)
        xavier_uniform_(self.k_proj_weight)
        xavier_uniform_(self.v_proj_weight)
    if self.in_proj_bias is not None:
        constant_(self.in_proj_bias, 0.)
        constant_(self.out_proj.bias, 0.)
```

给注意力机制补充Mask机制

Encoder和Decoder多头注意力机制不同有两处：

1. decoder的第一个多头注意力机制需要masked；
2. decoder的第二个多头注意力机制是cross attention。

这两点在 `_scaled_dot_product_attention` 中分别由 `attn_mask` 和 判断q与k是否相等 时进行了考虑，我们分析一下 `attn_mask` 这一输

入要怎么写。

这里再回忆一下**为什么要进行mask**：

1. sequence mask：因为在decoder解码的时候，只能看该位置和它之前的，如果看后面就相当于提前知道答案了，所以需要attn_mask遮挡住；
2. padding mask：而key_padding_mask用于不计算较短句子用"[PAD]"字符补齐的部分。

代码

接下来会使用到两个函数：

1. `logical_or`，输入两个tensor，并对这两个tensor里的值做 逻辑或 运算，**只有当两个值均为0的时候才为 `False`**，其他时候均为 `True`；

```
a = torch.tensor([0,1,10,0],dtype=torch.int8)
b = torch.tensor([4,0,1,0],dtype=torch.int8)
print(torch.logical_or(a,b))
# tensor([ True,  True,  True, False])
```

2. `masked_fill`，输入是一个mask，和用以填充的值。mask由1，0组成，0的位置值维持不变，1的位置用新值填充。

```
r = torch.tensor([[0,0,0,0],[0,0,0,0]])
mask = torch.tensor([[1,1,1,1],[0,0,0,0]])
print(r.masked_fill(mask,1))
# tensor([[1, 1, 1, 1],
#         [0, 0, 0, 0]])
```

sequence mask

> 对于attn_mask来说，若为2D，形状如（L，S），L和S分别代表着目标语言和源语言序列长度，若为3D,形状如（N ∗ num_heads，L，S），N代表着batch_size，num_heads代表注意力头的数目。若为attn_mask的dtype为ByteTensor，非0的位置会被忽略不做注意力；若为BoolTensor，True对应的位置会被忽略；若为数值，则会直接加到attn_weights。

```
def generate_square_subsequent_mask(self, sz: int) -> Tensor:
    r'''产生关于序列的mask，被遮住的区域赋值`-inf`，未被遮住的区域赋值为`0`'''
    mask = (torch.triu(torch.ones(sz, sz)) == 1).transpose(0, 1) #返回(sz, sz)大小方阵的上三角部分,其余部分定义为0，转置后为下三角，这个就是sequence mask。
    mask = mask.float().masked_fill(mask == 0, float('-inf')).masked_fill(mask == 1, float(0.0)) # 这里将有效的部分mask矩阵设置为0，无效的部分设置为-inf
    return mask
```

注意，**为什么这里我们把有效的部分mask为0呢**，注意前面attention机制中有这样的代码：

```
if attn_mask is not None:
    attn = attn + attn_mask
```

这样我们在相加的时候，可以不改变attn有效部分的值，而把需要被遮盖的地方设置为了-inf，从而实现了掩膜。

```
if attn_mask is not None:
    if attn_mask.dtype == torch.uint8:
        warnings.warn("Byte tensor for attn_mask in nn.MultiheadAttention is deprecated. Use bool tensor instead.")
        attn_mask = attn_mask.to(torch.bool)
    else:
        assert attn_mask.is_floating_point() or attn_mask.dtype == torch.bool, \
            f"Only float, byte, and bool types are supported for attn_mask, not {attn_mask.dtype}"
```

```
    # 对不同维度的形状判定
    if attn_mask.dim() == 2:
        correct_2d_size = (tgt_len, src_len)
        if attn_mask.shape != correct_2d_size:
            raise RuntimeError(f"The shape of the 2D attn_mask is {attn_mask.shape}, but should be {correct_2d_size}.")
            attn_mask = attn_mask.unsqueeze(0)
    elif attn_mask.dim() == 3:
        correct_3d_size = (bsz * num_heads, tgt_len, src_len)
        if attn_mask.shape != correct_3d_size:
            raise RuntimeError(f"The shape of the 3D attn_mask is {attn_mask.shape}, but should be {correct_3d_size}.")
    else:
        raise RuntimeError(f"attn_mask's dimension {attn_mask.dim()} is not supported")
```

padding mask

与 `attn_mask` 不同的是， **`key_padding_mask` 是用来遮挡住key里面的值，详细来说应该是 `[PAD]`**，被忽略的情况与 `attn_mask` 一致。

```
# 将key_padding_mask值改为布尔值
if key_padding_mask is not None and key_padding_mask.dtype == torch.uint8:
    warnings.warn("Byte tensor for key_padding_mask in nn.MultiheadAttention is deprecated. Use bool tensor instead.")
    key_padding_mask = key_padding_mask.to(torch.bool)
```

其实 `attn_mask` 和 `key_padding_mask` 有些时候对象是一致的，所以有时候可以合起来看。`-inf` 做softmax之后值为0，即被忽略。

```
if key_padding_mask is not None:
    assert key_padding_mask.shape == (bsz, src_len), \
        f"expecting key_padding_mask shape of {(bsz, src_len)}, but got {key_padding_mask.shape}"
    key_padding_mask =
```

```
key_padding_mask.view(bsz, 1, 1, src_len).expand(-1, num_heads, -1, -1).reshape(bsz * num_heads, 1, src_len)
        # 若attn_mask为空，直接用key_padding_mask
        if attn_mask is None:
            attn_mask = key_padding_mask
        elif attn_mask.dtype == torch.bool:
            attn_mask = attn_mask.logical_or(key_padding_mask)
        else:
            attn_mask = attn_mask.masked_fill(key_padding_mask, float("-inf"))

    # 若attn_mask值是布尔值，则将mask转换为float
    if attn_mask is not None and attn_mask.dtype == torch.bool:
        new_attn_mask = torch.zeros_like(attn_mask, dtype=torch.float)
        new_attn_mask.masked_fill_(attn_mask, float("-inf"))
        attn_mask = new_attn_mask
```

Multi-Head attention

接下来分析一下Multi-Head attention的代码。

代码

之前在Attention机制中提到：

> 多头注意力机制往往满足：self-attention的隐藏层维度（$hidden\_emb\_dim$）*n_heads=输入特征（经过线性变换）的维度(emb_dim)，然后W^O的维度为emb_dim。

所以令每个head的Q、K、Vd的$head\dim = embed_dim = emb_dim//n_heads$。Multi-Head attention核心代码如下。

```
import torch
Tensor = torch.Tensor
```

```
def multi_head_attention_forward(
    query: Tensor,
    key: Tensor,
    value: Tensor,
    num_heads: int,
    in_proj_weight: Tensor,
    in_proj_bias: Optional[Tensor],
    dropout_p: float,
    out_proj_weight: Tensor,
    out_proj_bias: Optional[Tensor],
    training: bool = True,
    key_padding_mask: Optional[Tensor] = None,
    need_weights: bool = True,
    attn_mask: Optional[Tensor] = None,
    use_seperate_proj_weight = None,
    q_proj_weight: Optional[Tensor] = None,
    k_proj_weight: Optional[Tensor] = None,
    v_proj_weight: Optional[Tensor] = None,
) -> Tuple[Tensor, Optional[Tensor]]:
    r'''
    形状:
        输入:
        - query: `(L, N, E)`
        - key: `(S, N, E)`
        - value: `(S, N, E)`
        - key_padding_mask: `(N, S)`
        - attn_mask: `(L, S)` or `(N * num_heads, L, S)`
        输出:
        - attn_output:`(L, N, E)`
        - attn_output_weights:`(N, L, S)`
    '''
    tgt_len, bsz, embed_dim = query.shape
    src_len, _, _ = key.shape
    head_dim = embed_dim // num_heads
    q, k, v = _in_projection_packed(query, key, value, in_proj_weight, in_proj_bias)

    if attn_mask is not None:
        if attn_mask.dtype == torch.uint8:
            warnings.warn("Byte tensor for attn_mask in nn.MultiheadAttention is deprecated. Use bool tensor instead.")
```

```
            attn_mask = attn_mask.to(torch.bool)
        else:
            assert attn_mask.is_floating_point() or attn_mask.dtype == torch.bool, \
                f"Only float, byte, and bool types are supported for attn_mask, not {attn_mask.dtype}"

        if attn_mask.dim() == 2:
            correct_2d_size = (tgt_len, src_len)
            if attn_mask.shape != correct_2d_size:
                raise RuntimeError(f"The shape of the 2D attn_mask is {attn_mask.shape}, but should be {correct_2d_size}.")
            attn_mask = attn_mask.unsqueeze(0)
        elif attn_mask.dim() == 3:
            correct_3d_size = (bsz * num_heads, tgt_len, src_len)
            if attn_mask.shape != correct_3d_size:
                raise RuntimeError(f"The shape of the 3D attn_mask is {attn_mask.shape}, but should be {correct_3d_size}.")
        else:
            raise RuntimeError(f"attn_mask's dimension {attn_mask.dim()} is not supported")

    if key_padding_mask is not None and key_padding_mask.dtype == torch.uint8:
        warnings.warn("Byte tensor for key_padding_mask in nn.MultiheadAttention is deprecated. Use bool tensor instead.")
        key_padding_mask = key_padding_mask.to(torch.bool)

    if key_padding_mask is not None:
        assert key_padding_mask.shape == (bsz, src_len), \
            f"expecting key_padding_mask shape
```

```
of {(bsz, src_len)}, but got {key_padding_mask.shape}"
        key_padding_mask = key_padding_mask.view(bsz, 1, 1, src_len).expand(-1, num_heads, -1, -1).reshape(bsz * num_heads, 1, src_len)
        if attn_mask is None:
            attn_mask = key_padding_mask
        elif attn_mask.dtype == torch.bool:
            attn_mask = attn_mask.logical_or(key_padding_mask)
        else:
            attn_mask = attn_mask.masked_fill(key_padding_mask, float("-inf"))
    # 若attn_mask值是布尔值，则将mask转换为float
    if attn_mask is not None and attn_mask.dtype == torch.bool:
        new_attn_mask = torch.zeros_like(attn_mask, dtype=torch.float)
        new_attn_mask.masked_fill_(attn_mask, float("-inf"))
        attn_mask = new_attn_mask

    # reshape q,k,v将Batch放在第一维以适合点积注意力
    # 同时为多头机制，将不同的头拼在一起组成一层
    q = q.contiguous().view(tgt_len, bsz * num_heads, head_dim).transpose(0, 1)
    k = k.contiguous().view(-1, bsz * num_heads, head_dim).transpose(0, 1)
    v = v.contiguous().view(-1, bsz * num_heads, head_dim).transpose(0, 1)

    # 若training为True时才应用dropout
    if not training:
        dropout_p = 0.0
    # Attention计算
    attn_output, attn_output_weights = _scaled_dot_product_attention(q, k, v, attn_mask, dropout_p)
    attn_output = attn_output.transpose(0, 1).contiguous().view(tgt_len, bsz, embed_dim)
```

```
    attn_output = nn.functional.linear(attn_output, out_proj_weight, out_proj_bias)
    if need_weights:
        # average attention weights over heads
        attn_output_weights = attn_output_weights.view(bsz, num_heads, tgt_len, src_len)
        return attn_output, attn_output_weights.sum(dim=1) / num_heads
    else:
        return attn_output, None
```

再结合参数初始化函数，完整代码如下：

```
class MultiheadAttention(nn.Module):
    r'''
    参数：
        embed_dim: 词嵌入的维度
        num_heads: 平行头的数量
        batch_first: 若`True`，则为(batch, seq, feture)，若为`False`，则为(seq, batch, feature)

    例子：
        >>> multihead_attn = MultiheadAttention(embed_dim, num_heads)
        >>> attn_output, attn_output_weights = multihead_attn(query, key, value)
    '''
    def __init__(self, embed_dim, num_heads, dropout=0., bias=True,
                 kdim=None, vdim=None, batch_first=False) -> None:
        # factory_kwargs = {'device': device, 'dtype': dtype}
        super(MultiheadAttention, self).__init__()
        self.embed_dim = embed_dim
        self.kdim = kdim if kdim is not None else embed_dim
        self.vdim = vdim if vdim is not None else embed_dim
        self._qkv_same_embed_dim = self.kdim
```

```
== embed_dim and self.vdim == embed_dim

        self.num_heads = num_heads
        self.dropout = dropout
        self.batch_first = batch_first
        self.head_dim = embed_dim // num_heads
        assert self.head_dim * num_heads == self.embed_dim, "embed_dim must be divisible by num_heads"

        if self._qkv_same_embed_dim is False:
            self.q_proj_weight = Parameter(torch.empty((embed_dim, embed_dim)))
            self.k_proj_weight = Parameter(torch.empty((embed_dim, self.kdim)))
            self.v_proj_weight = Parameter(torch.empty((embed_dim, self.vdim)))
            self.register_parameter('in_proj_weight', None)
        else:
            self.in_proj_weight = Parameter(torch.empty((3 * embed_dim, embed_dim)))
            self.register_parameter('q_proj_weight', None)
            self.register_parameter('k_proj_weight', None)
            self.register_parameter('v_proj_weight', None)

        if bias:
            self.in_proj_bias = Parameter(torch.empty(3 * embed_dim))
        else:
            self.register_parameter('in_proj_bias', None)
        self.out_proj = nn.Linear(embed_dim, embed_dim, bias=bias)

        self._reset_parameters()

    def _reset_parameters(self):
        if self._qkv_same_embed_dim:

```

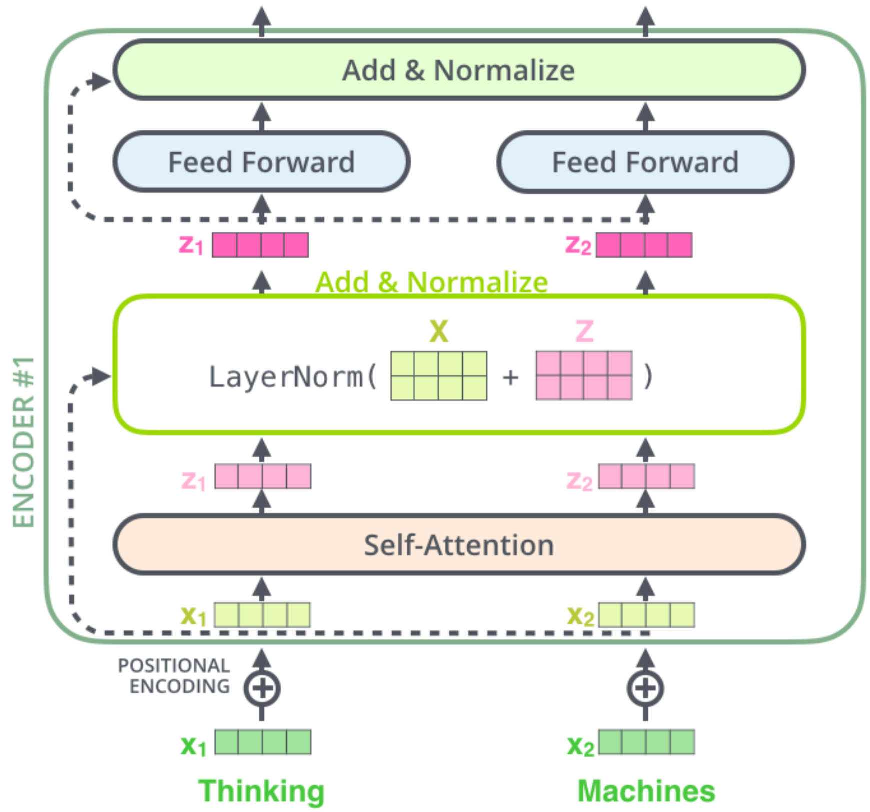
```
xavier_uniform_(self.in_proj_weight)
        else:

xavier_uniform_(self.q_proj_weight)

xavier_uniform_(self.k_proj_weight)

xavier_uniform_(self.v_proj_weight)

        if self.in_proj_bias is not None:
            constant_(self.in_proj_bias, 0.)
            constant_(self.out_proj.bias, 0.)

    def forward(self, query: Tensor, key:
Tensor, value: Tensor, key_padding_mask:
Optional[Tensor] = None,
                need_weights: bool = True,
attn_mask: Optional[Tensor] = None) ->
Tuple[Tensor, Optional[Tensor]]:
        if self.batch_first:
            query, key, value =
[x.transpose(1, 0) for x in (query, key, value)]

        if not self._qkv_same_embed_dim:
            attn_output, attn_output_weights =
multi_head_attention_forward(
                query, key, value,
self.num_heads,
                self.in_proj_weight,
self.in_proj_bias,
                self.dropout,
self.out_proj.weight, self.out_proj.bias,
                training=self.training,

key_padding_mask=key_padding_mask,
need_weights=need_weights,
                attn_mask=attn_mask,
use_separate_proj_weight=True,

q_proj_weight=self.q_proj_weight,
k_proj_weight=self.k_proj_weight,

v_proj_weight=self.v_proj_weight)
```

```
        else:
            attn_output, attn_output_weights = multi_head_attention_forward(
                query, key, value, self.num_heads,
                self.in_proj_weight, self.in_proj_bias,
                self.dropout, self.out_proj.weight, self.out_proj.bias,
                training=self.training,
                key_padding_mask=key_padding_mask, need_weights=need_weights,
                attn_mask=attn_mask)
        if self.batch_first:
            return attn_output.transpose(1, 0), attn_output_weights
        else:
            return attn_output, attn_output_weights
```

Add&Norm

残差模块就是多加了self-attention的输入，如下图所示。

```
x=x+z
```



LayerNorm用Pytorch的 `nn.LayerNorm` 实现：

```
norm = nn.LayerNorm(embed_dim,
eps=layer_norm_eps)
```

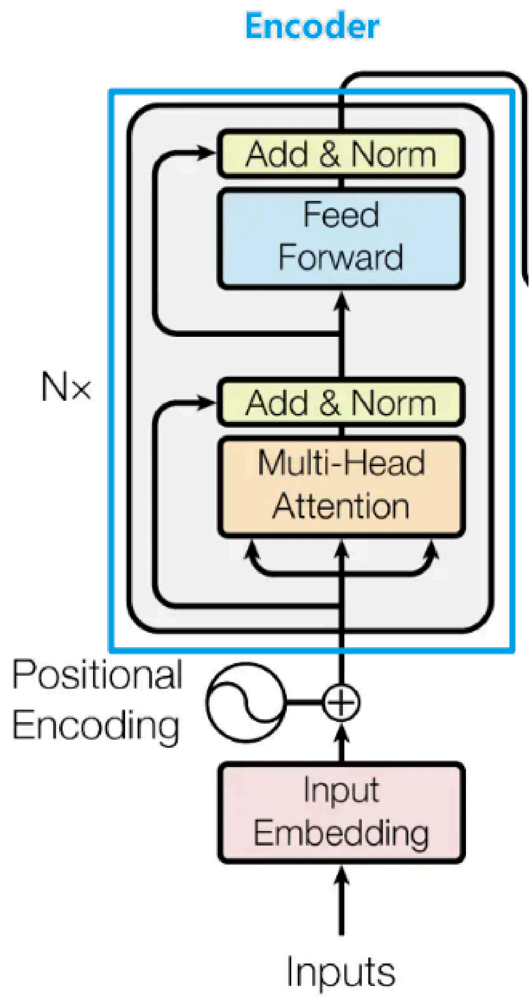
Feed forward

用Pytorch的 `nn.Linear` 实现：

```
linear1 = nn.Linear(input_dim, output_dim1)
linear2 = nn.Linear(output_dim1, output_dim2)
dropout=nn.Dropout(0.01)
activation=F.relu
res=linear2(dropout(activation(linear1(src))))
```

Encoder

Encoder结构如下图所示：



EncoderLayer

先写EncoderLayer根据示意图把之前将各个模块连起来。

代码

```
class TransformerEncoderLayer(nn.Module):
    r'''
    参数：
        embed_dim: 词嵌入的维度（必备）
        nhead: 多头注意力中平行头的数目（必备）
        dim_feedforward: 全连接层的神经元的数目，又称
入的维度（Default = 2048）
        dropout: dropout的概率（Default = 0.1）
        activation: 两个线性层中间的激活函数，默认relu
        lay_norm_eps: layer normalization中的微小量
母为0（Default = 1e-5）
        batch_first: 若`True`，则为(batch, seq, fe
为`False`，则为(seq, batch, feature)（Default: False）

```

```
    例子:
        >>> encoder_layer =
TransformerEncoderLayer(embed_dim=512, nhead=8)
        >>> src = torch.randn((32, 10, 512))
        >>> out = encoder_layer(src)
    '''

    def __init__(self, embed_dim, nhead,
dim_feedforward=2048, dropout=0.1, activation=F.relu,
                 layer_norm_eps=1e-5, batch_firs
-> None:
        super(TransformerEncoderLayer, self).__i
        self.self_attn = MultiheadAttention(embe
nhead, dropout=dropout, batch_first=batch_first)
        self.linear1 = nn.Linear(embed_dim,
dim_feedforward)
        self.dropout = nn.Dropout(dropout)
        self.linear2 = nn.Linear(dim_feedforward
embed_dim)

        self.norm1 = nn.LayerNorm(embed_dim,
eps=layer_norm_eps)
        self.norm2 = nn.LayerNorm(embed_dim,
eps=layer_norm_eps)
        self.dropout1 = nn.Dropout(dropout)
        self.dropout2 = nn.Dropout(dropout)
        self.activation = activation


    def forward(self, src: Tensor, src_mask:
Optional[Tensor] = None, src_key_padding_mask:
Optional[Tensor] = None) -> Tensor:
        src = positional_encoding(src, src.shape
置编码
        src2 = self.self_attn(src, src, src,
attn_mask=src_mask,
        key_padding_mask=src_key_padding_mask)[0
力，(attn_output, attn_output_weights)
        src = src + self.dropout1(src2) #Add
        src = self.norm1(src) #Norm
        src2 =
self.linear2(self.dropout(self.activation(self.linear
#Feed forward
```

```
        src = src + self.dropout(src2)#Add
        src = self.norm2(src) #Norm
        return src

```

```
# 用小例子看一下
encoder_layer = TransformerEncoderLayer(embed_dim=512, nhead=8)
src = torch.randn((32, 10, 512))
out = encoder_layer(src)
print(out.shape)
# torch.Size([32, 10, 512])
```

EncoderLayer组成Encoder

Encoder层由多个EncoderLayer串联。

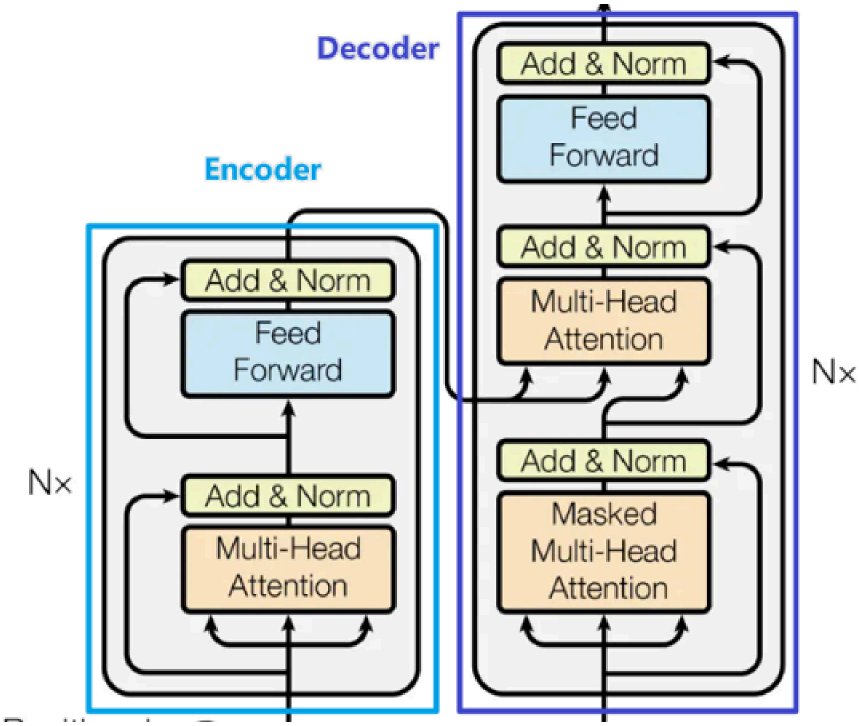
```
class TransformerEncoder(nn.Module):
    r'''
    参数:
        encoder_layer (必备)
        num_layers:  encoder_layer的层数 (必备)
        norm: 归一化的选择 (可选)

    例子:
        >>> encoder_layer = TransformerEncoderLayer(embed_dim=512, nhead=8)
        >>> transformer_encoder = TransformerEncoder(encoder_layer, num_layers=6)
        >>> src = torch.randn((10, 32, 512))
        >>> out = transformer_encoder(src)
    '''

    def __init__(self, encoder_layer, num_layers, norm=None):
        super(TransformerEncoder, self).__init__()
        self.layer = encoder_layer
        self.num_layers = num_layers
        self.norm = norm

```

```
    def forward(self, src: Tensor, mask: Optional[Tensor] = None, src_key_padding_mask: Optional[Tensor] = None) -> Tensor:
        output = positional_encoding(src, src.shape[-1])
        for _ in range(self.num_layers):
            output = self.layer(output, src_mask=mask, src_key_padding_mask=src_key_padding_mask)

        if self.norm is not None:
            output = self.norm(output)

        return output
```

```
# 例子
encoder_layer = TransformerEncoderLayer(embed_dim=512, nhead=8)
transformer_encoder = TransformerEncoder(encoder_layer, num_layers=6)
src = torch.randn((10, 32, 512))
out = transformer_encoder(src)
print(out.shape)
# torch.Size([10, 32, 512])
```

Decoder

Decoder结构如下图紫色框所示，因为涉及到和Encoder的交互，这里把Encoder的图示也放上。



DecoderLayer

同理把各个模块串成DecoderLayer。

注意：

1. **第一个self-attention需要输入掩膜(tgt_mask)。**
2. **第二个multihead_attn输入的Q为上一层输出tgt，K=V为Encoder最后一层输出memory。**

```
class TransformerDecoderLayer(nn.Module):
    r'''
    参数：
        embed_dim: 词嵌入的维度（必备）
        nhead: 多头注意力中平行头的数目（必备）
        dim_feedforward: 全连接层的神经元的数目，又称
的维度（Default = 2048）
        dropout: dropout的概率（Default = 0.1）
        activation: 两个线性层中间的激活函数，默认relu
        lay_norm_eps: layer normalization中的微小量
为0（Default = 1e-5）
        batch_first: 若`True`，则为(batch, seq, fe
为`False`，则为(seq, batch, feature)（Default：False）

    例子：
```

```
        >>> decoder_layer = TransformerDecoderLayer(embed_dim=512, nhead=8)
        >>> memory = torch.randn((10, 32, 512))
        >>> tgt = torch.randn((20, 32, 512))
        >>> out = decoder_layer(tgt, memory)
    '''
    def __init__(self, embed_dim, nhead, dim_feedforward=2048, dropout=0.1, activation=F.relu,
                 layer_norm_eps=1e-5, batch_firs
> None:
        super(TransformerDecoderLayer, self).__i
        self.self_attn = MultiheadAttention(embe
nhead, dropout=dropout, batch_first=batch_first)
        self.multihead_attn = MultiheadAttention(embed_dim, nhead, dropout=dropout, batch_first=batch_first)

        self.linear1 = nn.Linear(embed_dim, dim_feedforward)
        self.dropout = nn.Dropout(dropout)
        self.linear2 = nn.Linear(dim_feedforward
embed_dim)

        self.norm1 = nn.LayerNorm(embed_dim, eps=layer_norm_eps)
        self.norm2 = nn.LayerNorm(embed_dim, eps=layer_norm_eps)
        self.norm3 = nn.LayerNorm(embed_dim, eps=layer_norm_eps)
        self.dropout1 = nn.Dropout(dropout)
        self.dropout2 = nn.Dropout(dropout)
        self.dropout3 = nn.Dropout(dropout)

        self.activation = activation

    def forward(self, tgt: Tensor, memory: Tenso
tgt_mask: Optional[Tensor] = None,
                memory_mask: Optional[Tensor] = None,tgt_key_padding_mask: Optional[Tensor] = None, memory_key_padding_mask: Optional[Tensor] = None) ->
        r'''
        参数:
            tgt: 目标语言序列（必备）
```

```
                memory: 从最后一个encoder_layer跑出的句
                tgt_mask: 目标语言序列的mask（可选）
                memory_mask（可选）
                tgt_key_padding_mask（可选）
                memory_key_padding_mask（可选）
            '''
            tgt_mask = self.self_attn.
            tgt2 = self.self_attn(tgt, tgt, tgt, attn_mask=tgt_mask,
key_padding_mask=tgt_key_padding_mask)[0] #Masked自注
            tgt = tgt + self.dropout1(tgt2) #Add
            tgt = self.norm1(tgt)       #Norm
            tgt2 = self.multihead_attn(tgt, memory, 
attn_mask=memory_mask,key_padding_mask=memory_key_pac
[0] #cross attention,Q为上一层输出tgt，K=V为Encoder最后一
            tgt = tgt + self.dropout2(tgt2)#Add
            tgt = self.norm2(tgt)#Norm
            tgt2 = 
self.linear2(self.dropout(self.activation(self.linear
#Feed Forward
            tgt = tgt + self.dropout3(tgt2)#Add
            tgt = self.norm3(tgt)#Norm
            return tgt
```

```
# 例子
decoder_layer = nn.TransformerDecoderLayer(embed_dim=512, nhead=8)
memory = torch.randn((10, 32, 512))
tgt = torch.randn((20, 32, 512))
out = decoder_layer(tgt, memory)
print(out.shape)
# torch.Size([20, 32, 512])
```

DecoderLayer组成Decoder

```
class TransformerDecoder(nn.Module):
    r'''
    参数:
        decoder_layer（必备）
        num_layers: decoder_layer的层数（必备）
```

```
        norm: 归一化选择

    例子:
        >>> decoder_layer =TransformerDecoderLayer(embed_dim=512, nhead=8)
        >>> transformer_decoder = TransformerDecoder(decoder_layer, num_layers=6)
        >>> memory = torch.rand(10, 32, 512)
        >>> tgt = torch.rand(20, 32, 512)
        >>> out = transformer_decoder(tgt, memory)
    '''
    def __init__(self, decoder_layer, num_layers, norm=None):
        super(TransformerDecoder, self).__init__()
        self.layer = decoder_layer
        self.num_layers = num_layers
        self.norm = norm

    def forward(self, tgt: Tensor, memory: Tensor, tgt_mask: Optional[Tensor] = None,
                memory_mask: Optional[Tensor] = None, tgt_key_padding_mask: Optional[Tensor] = None,
                memory_key_padding_mask: Optional[Tensor] = None) -> Tensor:
        output = tgt
        for _ in range(self.num_layers):
            output = self.layer(output, memory, tgt_mask=tgt_mask,
memory_mask=memory_mask,
tgt_key_padding_mask=tgt_key_padding_mask,
memory_key_padding_mask=memory_key_padding_mask)
        if self.norm is not None:
            output = self.norm(output)

        return output
```

```python
# 例子
decoder_layer =TransformerDecoderLayer(embed_dim=512, nhead=8)
transformer_decoder = TransformerDecoder(decoder_layer, num_layers=6)
memory = torch.rand(10, 32, 512)
tgt = torch.rand(20, 32, 512)
out = transformer_decoder(tgt, memory)
print(out.shape)
# torch.Size([20, 32, 512])
```

Transformer

完整的Transformer模型，结合input embedding、Encoder、Decoder、最后输出的线性层和Softmax层。

代码

```python
class Transformer(nn.Module):
    r'''
    参数:
        embed_dim: 词嵌入的维度（必备）（Default=512）
        nhead: 多头注意力中平行头的数目（必备）（Default=8）
        num_encoder_layers:编码层层数（Default=8）
        num_decoder_layers:解码层层数（Default=8）
        dim_feedforward: 全连接层的神经元的数目，又称经过此层输入的维度（Default = 2048）
        dropout: dropout的概率（Default = 0.1）
        activation: 两个线性层中间的激活函数，默认relu或gelu
        custom_encoder: 自定义encoder（Default=None）
        custom_decoder: 自定义decoder（Default=None）
        lay_norm_eps: layer normalization中的微小量，防止分母为0（Default = 1e-5）
        batch_first: 若`True`，则为(batch, seq,
```

```
feture)，若为`False`，则为(seq, batch, feature) （Default：False）

    例子：
        >>> transformer_model = Transformer(nhead=16, num_encoder_layers=12)
        >>> src = torch.rand((10, 32, 512))
        >>> tgt = torch.rand((20, 32, 512))
        >>> out = transformer_model(src, tgt)
    '''
    def __init__(self, embed_dim: int = 512, nhead: int = 8, num_encoder_layers: int = 6,
                 num_decoder_layers: int = 6, dim_feedforward: int = 2048, dropout: float = 0.1,
                 activation = F.relu, custom_encoder: Optional[Any] = None, custom_decoder: Optional[Any] = None,
                 layer_norm_eps: float = 1e-5, batch_first: bool = False) -> None:
        super(Transformer, self).__init__()
        if custom_encoder is not None:
            self.encoder = custom_encoder
        else:
            encoder_layer = TransformerEncoderLayer(embed_dim, nhead, dim_feedforward, dropout,
activation, layer_norm_eps, batch_first)
            encoder_norm = nn.LayerNorm(embed_dim, eps=layer_norm_eps)
            self.encoder = TransformerEncoder(encoder_layer, num_encoder_layers)

        if custom_decoder is not None:
            self.decoder = custom_decoder
        else:
            decoder_layer = TransformerDecoderLayer(embed_dim, nhead, dim_feedforward, dropout,
activation, layer_norm_eps, batch_first)
            decoder_norm =
```

```
nn.LayerNorm(embed_dim, eps=layer_norm_eps)
            self.decoder = TransformerDecoder(decoder_layer, num_decoder_layers, decoder_norm)

        self._reset_parameters()

        self.embed_dim = embed_dim
        self.nhead = nhead

        self.batch_first = batch_first

    def forward(self, src: Tensor, tgt: Tensor, src_mask: Optional[Tensor] = None, tgt_mask: Optional[Tensor] = None,
                memory_mask: Optional[Tensor] = None, src_key_padding_mask: Optional[Tensor] = None,
                tgt_key_padding_mask: Optional[Tensor] = None, memory_key_padding_mask: Optional[Tensor] = None) -> Tensor:
        r'''
        参数:
            src: 源语言序列（送入Encoder）（必备）
            tgt: 目标语言序列（送入Decoder）（必备）
            src_mask: （可选）
            tgt_mask: （可选）
            memory_mask: （可选）
            src_key_padding_mask: （可选）
            tgt_key_padding_mask: （可选）
            memory_key_padding_mask: （可选）

        形状:
            - src: shape:`(S, N, E)`, `(N, S, E)` if batch_first.
            - tgt: shape:`(T, N, E)`, `(N, T, E)` if batch_first.
            - src_mask: shape:`(S, S)`.
            - tgt_mask: shape:`(T, T)`.
            - memory_mask: shape:`(T, S)`.
            - src_key_padding_mask: shape:`(N, S)`.
            - tgt_key_padding_mask: shape:`(N,
```

```
T)`.
            - memory_key_padding_mask: shape:`(N, S)`.

            [src/tgt/memory]_mask确保有些位置不被看到，如做decode的时候，只能看该位置及其以前的，而不能看后面的。
            若为ByteTensor，非0的位置会被忽略不做注意力；若为BoolTensor，True对应的位置会被忽略；
            若为数值，则会直接加到attn_weights

            [src/tgt/memory]_key_padding_mask 使得key里面的某些元素不参与attention计算，三种情况同上

            - output: shape:`(T, N, E)`, `(N, T, E)` if batch_first.

        注意:
            src和tgt的最后一维需要等于embed_dim，batch的那一维需要相等

        例子:
            >>> output = transformer_model(src, tgt, src_mask=src_mask, tgt_mask=tgt_mask)
        '''
        memory = self.encoder(src, mask=src_mask, src_key_padding_mask=src_key_padding_mask)
        output = self.decoder(tgt, memory, tgt_mask=tgt_mask, memory_mask=memory_mask,
                              tgt_key_padding_mask=tgt_key_padding_mask,
                              memory_key_padding_mask=memory_key_padding_mask)

        return output

    def generate_square_subsequent_mask(self, sz: int) -> Tensor:
        r'''产生关于序列的mask，被遮住的区域赋值`-inf`，未被遮住的区域赋值为`0`'''
        mask = (torch.triu(torch.ones(sz, sz))
```

```
== 1).transpose(0, 1)
        mask = mask.float().masked_fill(mask == 0, float('-inf')).masked_fill(mask == 1, float(0.0))
        return mask

    def _reset_parameters(self):
        r'''用正态分布初始化参数'''
        for p in self.parameters():
            if p.dim() > 1:
                xavier_uniform_(p)
```

```
# 例子
transformer_model = Transformer(nhead=16, num_en
src = torch.rand((10, 32, 512)) # (seq_len,batch_
tgt = torch.rand((20, 32, 512)) # (seq_len,batch_
(seq_len,batch_size,embed_dim)=(20, 32, 512)
# 这里假设所有句子都不需要padding，src_key_padding_mask=None,tgt_key_padding_mask=None
# 生成tgt_mask
tgt_mask=transformer_model.generate_square_subsequent
out = transformer_model(src, tgt, tgt_mask=tgt_m
print(out.shape)
# torch.Size([20, 32, 512])
```

基于Transformer的模型

上面的代码实际上只写到了Decoder输出结束，还未根据下游任务继续设计网络，以文本翻译任务为例的模型其实就和上面的图一样，在Transformer的Decoder输出后接上一个线性层和一个softmax层。

```
class MyModel(nn.Module):
    def __init__(self, transformer_layer,output_
        super(MyModel, self).__init__()
        self.transformer_layer = transformer_lay

        #这里根据下游任务继续设计层，以文本翻译为例，这里
            self.linear = nn.Linear(self.transfor

```

```
    def forward(self, src: Tensor, tgt: Tensor,
                memory_mask: Optional[Tensor] = I
                tgt_key_padding_mask: Optional[T
            output=self.transformer_layer(src=src
src_mask=src_mask,tgt_mask=tgt_mask,memory_mask=memor

        output=F.softmax(self.linear(output))
        return output

```

```
vocab_size = 154 # 这里是随便举的字典大小的例子
transformer_layer = Transformer(embed_dim, nhead
                                                activati
model=MyModel(transformer_layer,output_dim = voc
src = torch.rand((10, 32, 512)) # (seq_len,batch
tgt = torch.rand((20, 32, 512)) # (seq_len,batch
(seq_len,batch_size,embed_dim)=(20, 32, 512)
# 这里假设所有句子都不需要padding，src_key_padding_ma
# 生成tgt_mask
tgt_mask=MyModel.transformer_layer.generate_square_su
out = model(src, tgt, tgt_mask=tgt_mask)
print(out.shape)
# torch.Size([20, 32, 154])
# 然后取概率分布最大的值对应索引作为输出
res=torch.max(out,2)[1]
print(res.shape)
# torch.Size([20, 32])
```

参考文献

Pytorch编写完整的Transformer

Transformer相关——（8）Transformer模型

Transformer相关——（7）Mask机制

Transformer相关——（6）Normalization方式

Transformer相关——（5）残差模块

Transformer相关——（4）Poisition encoding

Transformer相关——（3）Attention机制