Transformer代码完全解读!

机器学习算法那些事 2022-03-22 11:38

以下文章来源于Datawhale ，作者安晟&闫永强

Datawhale
一个专注于AI领域的开源组织，汇聚了众多优秀学习者，愿景-for the learner，和学习者一起成长。

Datawhale干货

作者：安晟&闫永强，Datawhale成员

本篇正文部分约**10000字**，分模块解读并实践了Transformer，建议**收藏阅读**。

2017年谷歌在一篇名为《Attention Is All You Need》的论文中,提出了一个基于attention(自注意力机制)结构来处理序列相关的问题的模型，名为Transformer。

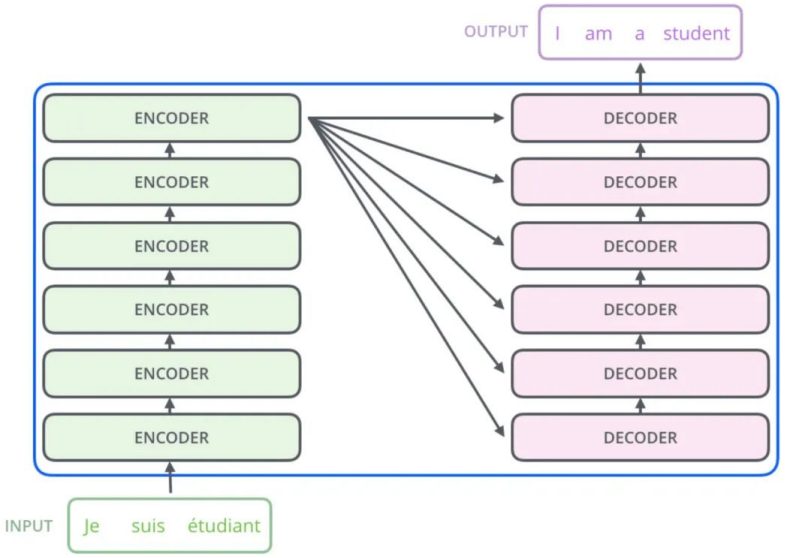
Transformer在很多不同nlp任务中获得了成功，例如：文本分类、机器翻译、阅读理解等。在解决这类问题时，Transformer模型摒弃了固有的定式，并没有用任何CNN或者RNN的结构，而是使用了Attention注意力机制，自动捕捉输入序列不同位置处的相对关联，善于处理较长文本，并且该模型可以高度并行地工作，训练速度很快。

本文将按照Transformer的模块进行讲解，每个模块配合 代码+注释+讲解 来介绍，最后会有一个玩具级别的序列预测任务进行实战。

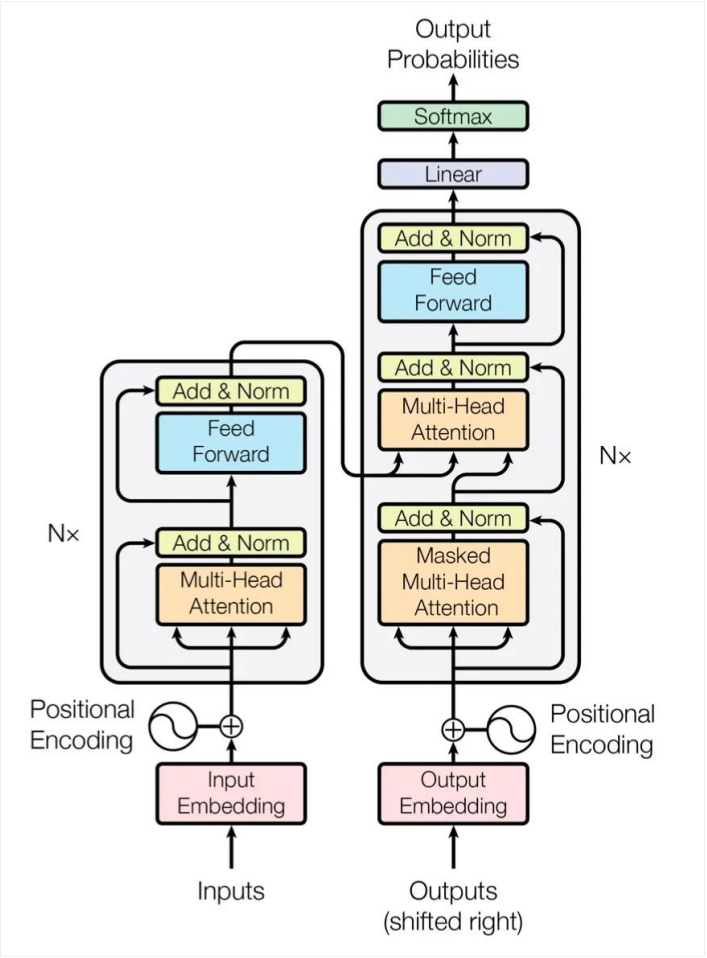
通过本文，希望可以帮助大家，初探Transformer的原理和用法，下面直接进入正式内容：

1 模型结构概览

如下是Transformer的两个结构示意图：



上图是从一篇英文博客中截取的Transformer的结构简图，下图是原论文中给出的结构简图，更细粒度一些，可以结合着来看。



模型大致分为 `Encoder` (编码器)和 `Decoder` (解码器)两个部分，分别对应上图中的左右两部分。

其中编码器由N个相同的层堆叠在一起(我们后面的实验取N=6)，每一层又有两个子层。

第一个子层是一个 `Multi-Head Attention` (多头的自注意机制)，第二个子层是一个简单的 `Feed Forward` (全连接前馈网络)。两个子层都添加了一个残差连接+layer normalization的操作。

模型的解码器同样是堆叠了N个相同的层，不过和编码器中每层的结构稍有不同。对于解码器的每一层，除了编码器中的两个子层 `Multi-Head Attention` 和 `Feed Forward` ，解码器还包含一个子层 `Masked Multi-Head Attention` ，如图中所示每个子层同样也用了residual以及layer normalization。

模型的输入由 `Input Embedding` 和 `Positional Encoding` (位置编码)两部分组合而成，模型的输出由Decoder的输出简单的经过softmax得到。

结合上图，我们对Transformer模型的结构做了个大致的梳理，只需要先有个初步的了解，下面对提及的每个模块进行详细介绍。

2 模型输入

首先我们来看模型的输入是什么样的，先明确模型输入，后面的模块理解才会更直观。

输入部分包含两个模块， `Embedding` 和 `Positional Encoding` 。

2.1 Embedding层

Embedding层的作用是将某种格式的输入数据，例如文本，转变为模型可以处理的向量表示，来描述原始数据所包含的信息。

`Embedding` 层输出的可以理解为当前时间步的特征，如果是文本任务，这里就可以是 `Word Embedding` ，如果是其他任务，就可以是任何合理方法所提取的特征。

构建Embedding层的代码很简单，核心是借助torch提供的 `nn.Embedding` ，如下：

```
class Embeddings(nn.Module):
    def __init__(self, d_model, vocab):
        """
        类的初始化函数
        d_model: 指词嵌入的维度
        vocab:指词表的大小
        """
        super(Embeddings, self).__init__()
        #之后就是调用nn中的预定义层Embedding，获得一个词嵌入对象self.lut
        self.lut = nn.Embedding(vocab, d_model)
        #最后就是将d_model传入类中
        self.d_model =d_model
    def forward(self, x):
        """
        Embedding层的前向传播逻辑
        参数x: 这里代表输入给模型的单词文本通过词表映射后的one-hot向量
        将x传给self.lut并与根号下self.d_model相乘作为结果返回
        """
        embedds = self.lut(x)
        return embedds * math.sqrt(self.d_model)
```

2.2 位置编码：

`Positional Encodding` 位置编码的作用是为模型提供当前时间步的前后出现顺序的信息。因为Transformer不像RNN那样的循环结构有前后不同时间步输入间天然的先后顺序，所有的时间步是同时输入，并行推理的，因此在时间步的特征中融合进位置编码的信息是合理的。

位置编码可以有很多选择，可以是固定的，也可以设置成可学习的参数。

这里，我们使用固定的位置编码。具体地，使用不同频率的sin和cos函数来进行位置编码，如下所示：

其中pos代表时间步的下标索引，向量 PE_{pos} 也就是第pos个时间步的位置编码，编码长度同 `Embedding` 层，这里我们设置的是512。上面有两个公式，代表着位置编码向量中的元素，奇数位置和偶数位置使用两个不同的公式。

思考：**为什么上面的公式可以作为位置编码？**

我的理解：在上面公式的定义下，时间步p和时间步p+k的位置编码的内积，即 $PE_p \cdot PE_{p+k}$ 是与p无关，只与k有关的定值（不妨自行证明下试试）。也就是说，任意两个相距k个时间步的位置编码向量的内积都是相同的，这就相当于蕴含了两个时间步之间相对位置关系的信息。此外，每个时间步的位置编码又是唯一的，这两个很好的性质使得上面的公式作为位置编码是有理论保障的。

下面是位置编码模块的代码实现：

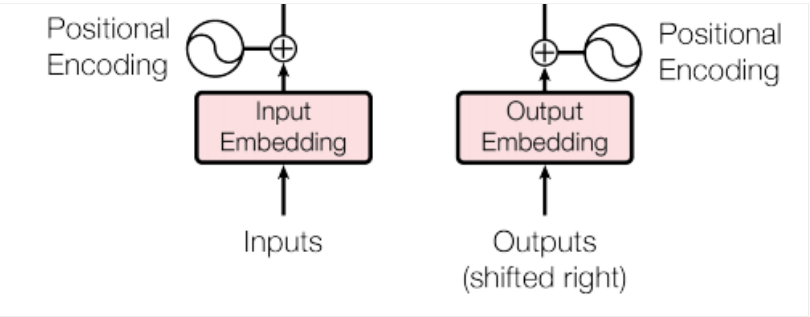
```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, dropout, max_len=5000):
        """
```

```
        位置编码器类的初始化函数

        共有三个参数，分别是
        d_model：词嵌入维度
        dropout：dropout触发比率
        max_len：每个句子的最大长度
        """
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)

        # Compute the positional encodings
        # 注意下面代码的计算方式与公式中给出的是不同的，但是是等价的，你可以尝试简单推导证明一下。
        # 这样计算是为了避免中间的数值计算结果超出float的范围，
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) *
                             -(math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        x = x + Variable(self.pe[:, :x.size(1)], requires_grad=False)
        return self.dropout(x)
```

因此，可以认为，最终模型的输入是若干个时间步对应的embedding，每一个时间步对应一个embedding，可以理解为是当前时间步的一个综合的特征信息，即包含了本身的语义信息，又包含了当前时间步在整个句子中的位置信息。

2.3 Encoder和Decoder都包含输入模块

此外有一个点刚刚接触Transformer的同学可能不太理解，编码器和解码器两个部分都包含输入，且两部分的输入的结构是相同的，只是推理时的用法不同，编码器只推理一次，而解码器是类似RNN那样循环推理，不断生成预测结果的。



怎么理解？假设我们现在做的是一个法语-英语的机器翻译任务，想把 `Je suis étudiant` 翻译为 `I am a student` 。

那么我们输入给编码器的就是时间步数为3的embedding数组，编码器只进行一次并行推理，即获得了对于输入的法语句子所提取的若干特征信息。

而对于解码器，是循环推理，逐个单词生成结果的。最开始，由于什么都还没预测，我们会将编码器提取的特征，以及一个句子起始符传给解码器，解码器预期会输出一个单词 `I` 。然后有了预测的第一个单词，我们就将 `I` 输入给解码器，会再预测出下一个单词 `am` ，再然后我们将 `I`

am 作为输入喂给解码器，以此类推直到预测出句子终止符完成预测。

3 Encoder

这一小节介绍编码器部分的实现。

3.1 编码器

编码器作用是用于对输入进行特征提取，为解码环节提供有效的语义信息

整体来看编码器由N个编码器层简单堆叠而成，因此实现非常简单，代码如下：

```
# 定义一个clones函数，来更方便的将某个结构复制若干份
def clones(module, N):
    "Produce N identical layers."
    return nn.ModuleList([copy.deepcopy(module) for _ in range(N)])



class Encoder(nn.Module):
    """
    Encoder
    The encoder is composed of a stack of N=6 identical layers.
    """
    def __init__(self, layer, N):
        super(Encoder, self).__init__()
        # 调用时会将编码器层传进来，我们简单克隆N分，叠加在一起，组成完整的Encoder
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)

    def forward(self, x, mask):
        "Pass the input (and mask) through each layer in turn."
        for layer in self.layers:
            x = layer(x, mask)
        return self.norm(x)
```

上面的代码中有一个小细节，就是编码器的输入除了x，也就是embedding以外，还有一个 mask ，为了介绍连续性，这里先忽略，后面会讲解。

下面我们来看看单个的编码器层都包含什么，如何实现。

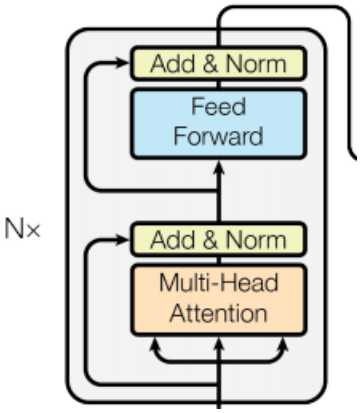
3.2 编码器层

每个编码器层由两个子层连接结构组成：

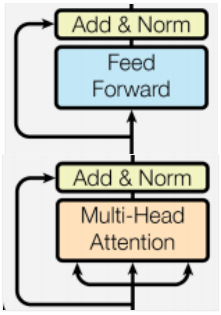
第一个子层包括一个多头自注意力层和规范化层以及一个残差连接；

第二个子层包括一个前馈全连接层和规范化层以及一个残差连接；

如下图所示：



可以看到，两个子层的结构其实是一致的，只是中间核心层的实现不同



我们先定义一个SubLayerConnection类来描述这种结构关系

```
class SublayerConnection(nn.Module):
    """
    实现子层连接结构的类
    """
    def __init__(self, size, dropout):
        super(SublayerConnection, self).__init__()
        self.norm = LayerNorm(size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, sublayer):

        # 原paper的方案
        #sublayer_out = sublayer(x)
        #x_norm = self.norm(x + self.dropout(sublayer_out))

        # 稍加调整的版本
        sublayer_out = sublayer(x)
        sublayer_out = self.dropout(sublayer_out)
        x_norm = x + self.norm(sublayer_out)
        return x_norm
```

注：上面的实现中，我对残差的链接方案进行了小小的调整，和原论文有所不同。把x从norm中拿出来，保证永远有一条“高速公路”，这样理论上会收敛的快一些，但我无法确保这样做一定是对的，请一定注意。

定义好了SubLayerConnection，我们就可以实现EncoderLayer的结构了

```
class EncoderLayer(nn.Module):
    "EncoderLayer is made up of two sublayer: self-attn and feed forward"
    def __init__(self, size, self_attn, feed_forward, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = self_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 2)
        self.size = size   # embedding's dimention of model, 默认512

    def forward(self, x, mask):
        # attention sub layer
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, mask))
        # feed forward sub layer
        z = self.sublayer[1](x, self.feed_forward)
        return z
```

继续往下拆解，我们需要了解 attention层 和 feed_forward层的结构以及如何实现。

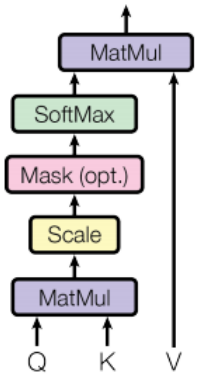
3.3 注意力机制

人类在观察事物时，无法同时仔细观察眼前的一切，只能聚焦到某一个局部。通常我们大脑在简单了解眼前的场景后，能够很快把注意力聚焦到最有价值的局部来仔细观察，从而作出有效判断。或许是基于这样的启发，大家想到了在算法中利用注意力机制。

注意力计算：它需要三个指定的输入Q（query），K（key），V（value），然后通过下面公式得到注意力的计算结果。

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

计算流程图如下：



可以这么简单的理解，当前时间步的注意力计算结果，是一个组系数 * 每个时间步的特征向量value的累加，而这个系数，通过当前时间步的query和其他时间步对应的key做内积得到，这个过程相当于用自己的query对别的时间步的key做查询，判断相似度，决定以多大的比例将对应时间步的信息继承过来。

> 注意力机制的原理和思考十分值得深究，鉴于本文篇幅已经很长，这里只着眼于代码实现，如果你在阅读前对Transformer的原理完全不了解，获取更多的原理讲解，这里推荐两个学习资料：
> 李宏毅老师的B站视频：https://www.bilibili.com/video/BV1J441137V6?from=search&seid=3530913447603589730

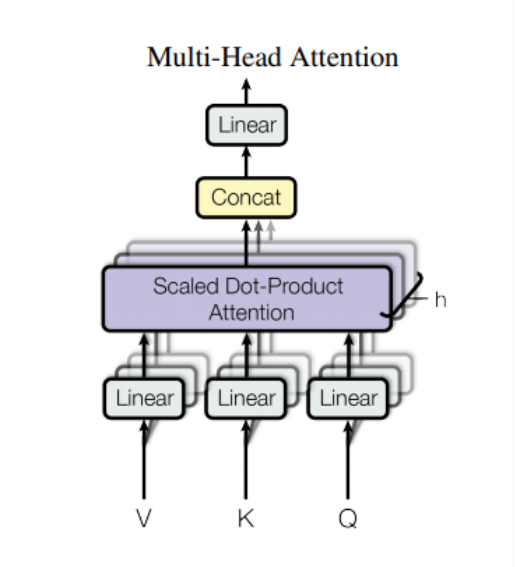
DataWhale开源项目：https://github.com/datawhalechina/learn-nlp-with-transformer

下面是注意力模块的实现代码：

```
def attention(query, key, value, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"

    #首先取query的最后一维的大小，对应词嵌入维度
    d_k = query.size(-1)
    #按照注意力公式，将query与key的转置相乘，这里面key是将最后两个维度进行转置，再除以缩放系数得到注意力得分张量scores
    scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)

    #接着判断是否使用掩码张量
    if mask is not None:
        #使用tensor的masked_fill方法，将掩码张量和scores张量每个位置一一比较，如果掩码张量则对应的scores张量用-1e9这个置来替换
        scores = scores.masked_fill(mask == 0, -1e9)

    #对scores的最后一维进行softmax操作，使用F.softmax方法，这样获得最终的注意力张量
    p_attn = F.softmax(scores, dim = -1)

    #之后判断是否使用dropout进行随机置0
    if dropout is not None:
        p_attn = dropout(p_attn)

    #最后，根据公式将p_attn与value张量相乘获得最终的query注意力表示，同时返回注意力张量
    return torch.matmul(p_attn, value), p_attn
```

3.4 多头注意力机制

刚刚介绍了attention机制，在搭建EncoderLayer时候所使用的Attention模块，实际使用的是多头注意力，可以简单理解为多个注意力模块组合在一起。



多头注意力机制的作用：这种结构设计能让每个注意力机制去优化每个词汇的不同特征部分，从而均衡同一种注意力机制可能产生的偏差，让词义拥有来自更多元表达，实验表明可以从而提升模型效果。

举个更形象的例子，bank是银行的意思，如果只有一个注意力模块，那么它大概率会学习去关注类似money、loan贷款这样的词。如果我们使用多个多头机制，那么不同的头就会去关注不同的语义，比如bank还有一种含义是河岸，那么可能有一个头就会去关注类似river这样的词汇，这时多头注意力的价值就体现出来了。

下面是多头注意力机制的实现代码：

```
class MultiHeadedAttention(nn.Module):
    def __init__(self, h, d_model, dropout=0.1):
        #在类的初始化时，会传入三个参数，h代表头数，d_model代表词嵌入的维度，dropout代表进行dropout操作时置0比率，默认是0.1
        super(MultiHeadedAttention, self).__init__()
        #在函数中，首先使用了一个测试中常用的assert语句，判断h是否能被d_model整除，这是因为我们之后要给每个头分配等量的词特征，也就是embedding_dim/head个
        assert d_model % h == 0
        #得到每个头获得的分割词向量维度d_k
        self.d_k = d_model // h
        #传入头数h
        self.h = h

        #创建linear层，通过nn的Linear实例化，它的内部变换矩阵是embedding_dim x embedding_dim，然后使用，为什么是四个呢，这是因为在多头注意力中，Q,K,V各需要
        self.linears = clones(nn.Linear(d_model, d_model), 4)
        #self.attn为None，它代表最后得到的注意力张量，现在还没有结果所以为None
        self.attn = None
        self.dropout = nn.Dropout(p=dropout)

    def forward(self, query, key, value, mask=None):
        #前向逻辑函数，它输入参数有四个，前三个就是注意力机制需要的Q,K,V，最后一个是注意力机制中可能需要的mask掩码张量，默认是None
        if mask is not None:
            # Same mask applied to all h heads.
            #使用unsqueeze扩展维度，代表多头中的第n头
            mask = mask.unsqueeze(1)
        #接着，我们获得一个batch_size的变量，他是query尺寸的第1个数字，代表有多少条样本
        nbatches = query.size(0)

        # 1) Do all the linear projections in batch from d_model => h x d_k
        # 首先利用zip将输入QKV与三个线性层组到一起，然后利用for循环，将输入QKV分别传到线性层中，做完线性变换后，开始为每个头分割输入，这里使用view方法对线性变
        query, key, value = \
            [l(x).view(nbatches, -1, self.h, self.d_k).transpose(1, 2)
             for l, x in zip(self.linears, (query, key, value))]

        # 2) Apply attention on all the projected vectors in batch.
        # 得到每个头的输入后，接下来就是将他们传入到attention中，这里直接调用我们之前实现的attention函数，同时也将mask和dropout传入其中
        x, self.attn = attention(query, key, value, mask=mask,
                                 dropout=self.dropout)

        # 3) "Concat" using a view and apply a final linear.
        # 通过多头注意力计算后，我们就得到了每个头计算结果组成的4维张量，我们需要将其转换为输入的形状以方便后续的计算，因此这里开始进行第一步处理环节的逆操作，
        x = x.transpose(1, 2).contiguous() \
```

```
            .view(nbatches, -1, self.h * self.d_k)
        #最后使用线性层列表中的最后一个线性变换得到最终的多头注意力结构的输出
        return self.linears[-1](x)
```

3.5 前馈全连接层

EncoderLayer中另一个核心的子层是 Feed Forward Layer，我们这就介绍一下。

在进行了Attention操作之后，encoder和decoder中的每一层都包含了一个全连接前向网络，对每个position的向量分别进行相同的操作，包括两个线性变换和一个ReLU激活输出：

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

Feed Forward Layer 其实就是简单的由两个前向全连接层组成，核心在于，Attention模块每个时间步的输出都整合了所有时间步的信息，而Feed Forward Layer每个时间步只是对自己的特征的一个进一步整合，与其他时间步无关。

实现代码如下：

```
class PositionwiseFeedForward(nn.Module):
    def __init__(self, d_model, d_ff, dropout=0.1):
        #初始化函数有三个输入参数分别是d_model，d_ff，和dropout=0.1，第一个是线性层的输入维度也是第二个线性层的输出维度，因为我们希望输入通过前馈全连接层后输
        super(PositionwiseFeedForward, self).__init__()
        self.w_1 = nn.Linear(d_model, d_ff)
        self.w_2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        #输入参数为x，代表来自上一层的输出，首先经过第一个线性层，然后使用F中的relu函数进行激活，之后再使用dropout进行随机置0，最后通过第二个线性层w2，返回最
        return self.w_2(self.dropout(F.relu(self.w_1(x))))
```

到这里Encoder中包含的主要结构就都介绍了，上面的代码中涉及了两个小细节还没有介绍，layer normalization 和 mask，下面来简单讲解一下。

3.6. 规范化层

规范化层的作用：它是所有深层网络模型都需要的标准网络层，因为随着网络层数的增加，通过多层的计算后输出可能开始出现过大或过小的情况，这样可能会导致学习过程出现异常，模型可能收敛非常慢。因此都会在一定层后接规范化层进行数值的规范化，使其特征数值在合理范围内。

Transformer中使用的normalization手段是layer norm，实现代码很简单，如下：

```
class LayerNorm(nn.Module):
    "Construct a layernorm module (See citation for details)."
    def __init__(self, feature_size, eps=1e-6):
        #初始化函数有两个参数，一个是features，表示词嵌入的维度，另一个是eps它是一个足够小的数，在规范化公式的分母中出现，防止分母为0，默认是1e-6。
        super(LayerNorm, self).__init__()
```

```
        #根据features的形状初始化两个参数张量a2，和b2，第一初始化为1张量，也就是里面的元素都是1，第二个初始化为0张量，也就是里面的元素都是0，这两个张量就是规
        self.a_2 = nn.Parameter(torch.ones(feature_size))
        self.b_2 = nn.Parameter(torch.zeros(feature_size))
        #把eps传到类中
        self.eps = eps

    def forward(self, x):
    #输入参数x代表来自上一层的输出，在函数中，首先对输入变量x求其最后一个维度的均值，并保持输出维度与输入维度一致，接着再求最后一个维度的标准差，然后就是根据规
    #最后对结果乘以我们的缩放参数，即a2,*号代表同型点乘，即对应位置进行乘法操作，加上位移参b2，返回即可
        mean = x.mean(-1, keepdim=True)
        std = x.std(-1, keepdim=True)
        return self.a_2 * (x - mean) / (std + self.eps) + self.b_2
```

3.7 掩码及其作用

掩码：掩代表遮掩，码就是我们张量中的数值，它的尺寸不定，里面一般只有0和1；代表位置被遮掩或者不被遮掩。

掩码的作用：在transformer中，掩码主要的作用有两个，一个是屏蔽掉无效的padding区域，一个是屏蔽掉来自“未来”的信息。Encoder中的掩码主要是起到第一个作用，Decoder中的掩码则同时发挥着两种作用。

屏蔽掉无效的padding区域：我们训练需要组batch进行，就以机器翻译任务为例，一个batch中不同样本的输入长度很可能是不一样的，此时我们要设置一个最大句子长度，然后对空白区域进行padding填充，而填充的区域无论在Encoder还是Decoder的计算中都是没有意义的，因此需要用mask进行标识，屏蔽掉对应区域的响应。

屏蔽掉来自未来的信息：我们已经学习了attention的计算流程，它是会综合所有时间步的计算的，那么在解码的时候，就有可能获取到未来的信息，这是不行的。因此，这种情况也需要我们使用mask进行屏蔽。现在还没介绍到Decoder，如果没完全理解，可以之后再回过头来思考下。

mask的构造代码如下：

```
def subsequent_mask(size):
    #生成向后遮掩的掩码张量，参数size是掩码张量最后两个维度的大小，它最后两维形成一个方阵

    "Mask out subsequent positions."
    attn_shape = (1, size, size)

    #然后使用np.ones方法向这个形状中添加1元素，形成上三角阵
    subsequent_mask = np.triu(np.ones(attn_shape), k=1).astype('uint8')

    #最后将numpy类型转化为torch中的tensor，内部做一个1- 的操作。这个其实是做了一个三角阵的反转，subsequent_mask中的每个元素都会被1减。
    #如果是0，subsequent_mask中的该位置由0变成1
    #如果是1，subsequect_mask中的该位置由1变成0
    return torch.from_numpy(subsequent_mask) == 0
```

以上便是编码器部分的全部内容，有了这部分内容的铺垫，解码器的介绍就会轻松一些。

4 Decoder

本小节介绍解码器部分的实现

4.1 解码器整体结构

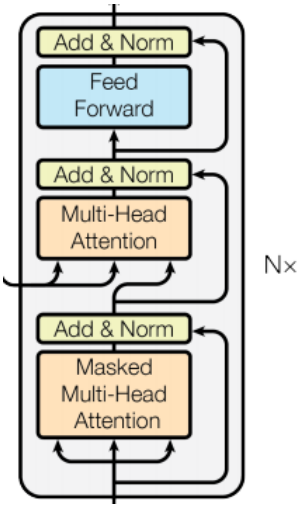
解码器的作用：根据编码器的结果以及上一次预测的结果，输出序列的下一个结果。

整体结构上，解码器也是由N个相同层堆叠而成。构造代码如下：

```
#使用类Decoder来实现解码器
class Decoder(nn.Module):
    "Generic N layer decoder with masking."
    def __init__(self, layer, N):
        #初始化函数的参数有两个，第一个就是解码器层layer，第二个是解码器层的个数N
        super(Decoder, self).__init__()
        #首先使用clones方法克隆了N个layer，然后实例化一个规范化层，因为数据走过了所有的解码器层后最后要做规范化处理。
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)

    def forward(self, x, memory, src_mask, tgt_mask):
        #forward函数中的参数有4个，x代表目标数据的嵌入表示，memory是编码器层的输出，source_mask，target_mask代表源数据和目标数据的掩码张量，然后就是对每个
        for layer in self.layers:
            x = layer(x, memory, src_mask, tgt_mask)
        return self.norm(x)
```

4.2 解码器层

每个解码器层由三个子层连接结构组成，第一个子层连接结构包括一个多头自注意力子层和规范化层以及一个残差连接，第二个子层连接结构包括一个多头注意力子层和规范化层以及一个残差连接，第三个子层连接结构包括一个前馈全连接子层和规范化层以及一个残差连接。



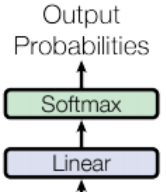
解码器层中的各个子模块，如，多头注意力机制，规范化层，前馈全连接都与编码器中的实现相同。

有一个细节需要注意，第一个子层的多头注意力和编码器中完全一致，第二个子层，它的多头注意力模块中，query来自上一个子层，key 和 value 来自编码器的输出。可以这样理解，就是第二层负责，利用解码器已经预测出的信息作为query，去编码器提取的各种特征中，查找相关信息并融合到当前特征中，来完成预测。

```
#使用DecoderLayer的类实现解码器层
class DecoderLayer(nn.Module):
    "Decoder is made of self-attn, src-attn, and feed forward (defined below)"
    def __init__(self, size, self_attn, src_attn, feed_forward, dropout):
        #初始化函数的参数有5个，分别是size，代表词嵌入的维度大小，同时也代表解码器的尺寸，第二个是self_attn，多头自注意力对象，也就是说这个注意力机制需要Q=K=
        super(DecoderLayer, self).__init__()
        self.size = size
        self.self_attn = self_attn
        self.src_attn = src_attn
        self.feed_forward = feed_forward
        #按照结构图使用clones函数克隆三个子层连接对象
        self.sublayer = clones(SublayerConnection(size, dropout), 3)

    def forward(self, x, memory, src_mask, tgt_mask):
        #forward函数中的参数有4个，分别是来自上一层的输入x，来自编码器层的语义存储变量memory，以及源数据掩码张量和目标数据掩码张量，将memory表示成m之后方便使
        "Follow Figure 1 (right) for connections."
        m = memory
        #将x传入第一个子层结构，第一个子层结构的输入分别是x和self-attn函数，因为是自注意力机制，所以Q,K,V都是x，最后一个参数时目标数据掩码张量，这时要对目标数
        #比如在解码器准备生成第一个字符或词汇时，我们其实已经传入了第一个字符以便计算损失，但是我们不希望在生成第一个字符时模型能利用这个信息，因此我们会将其遮
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, tgt_mask))
        #接着进入第二个子层，这个子层中常规的注意力机制，q是输入x;k,v是编码层输出memory，同样也传入source_mask，但是进行源数据遮掩的原因并非是抑制信息泄露，
        x = self.sublayer[1](x, lambda x: self.src_attn(x, m, m, src_mask))

        #最后一个子层就是前馈全连接子层，经过它的处理后就可以返回结果，这就是我们的解码器结构
        return self.sublayer[2](x, self.feed_forward)
```

5 模型输出

输出部分就很简单了，每个时间步都过一个 线性层 + softmax层



线性层的作用：通过对上一步的线性变化得到指定维度的输出，也就是转换维度的作用。转换后的维度对应着输出类别的个数，如果是翻译任务，那就对应的是文字字典的大小。

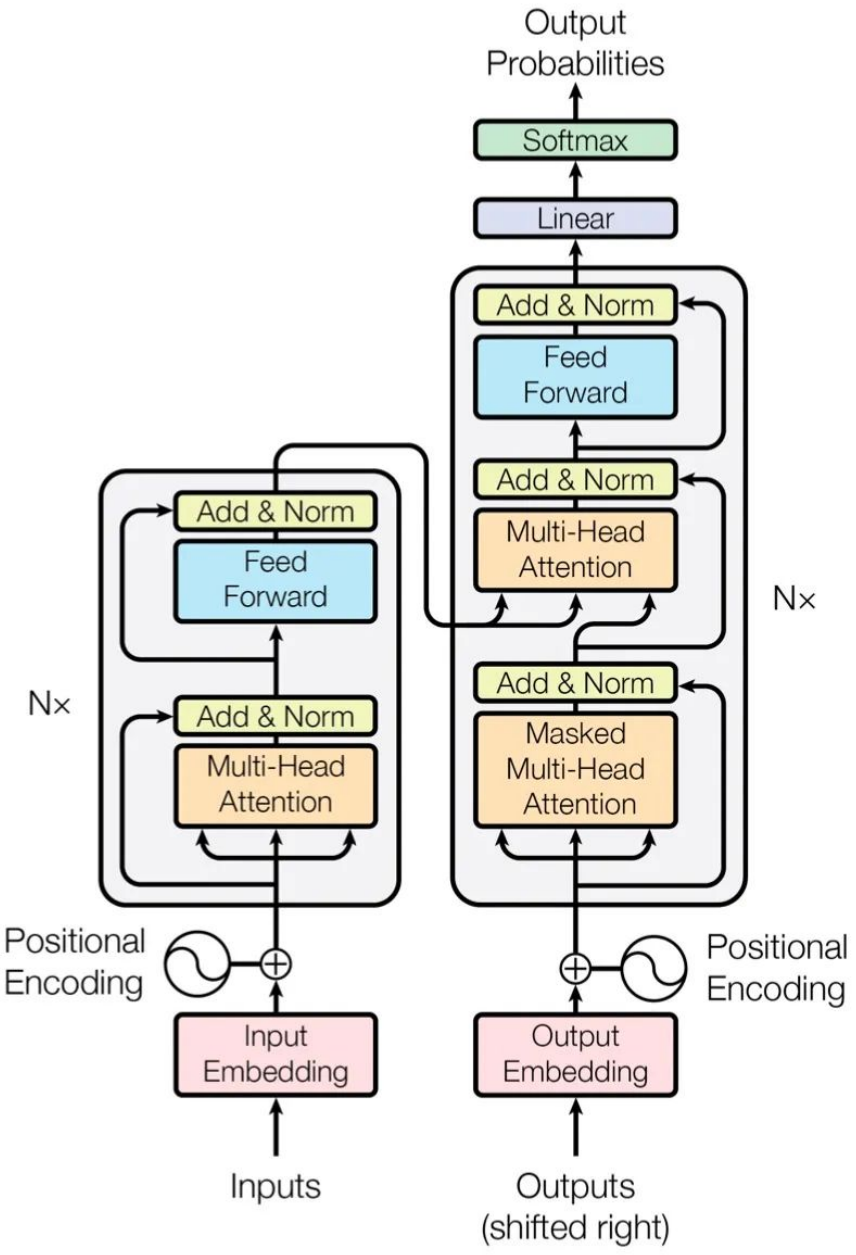
代码如下：

```
#将线性层和softmax计算层一起实现，因为二者的共同目标是生成最后的结构
#因此把类的名字叫做Generator，生成器类
class Generator(nn.Module):
    "Define standard linear + softmax generation step."
    def __init__(self, d_model, vocab):
        #初始化函数的输入参数有两个，d_model代表词嵌入维度，vocab.size代表词表大小
        super(Generator, self).__init__()
```

```
        #首先就是使用nn中的预定义线性层进行实例化，得到一个对象self.proj等待使用
        #这个线性层的参数有两个，就是初始化函数传进来的两个参数：d_model，vocab_size
        self.proj = nn.Linear(d_model, vocab)

    def forward(self, x):
        #前向逻辑函数中输入是上一层的输出张量x,在函数中，首先使用上一步得到的self.proj对x进行线性变化,然后使用F中已经实现的log_softmax进行softmax处理。
        return F.log_softmax(self.proj(x), dim=-1)
```

6 模型构建

下面是Transformer总体架构图，回顾一下，再看这张图，是不是每个模块的作用都有了基本的认知。



下面我们就可以搭建出整个网络的结构

```
# Model Architecture
#使用EncoderDecoder类来实现编码器-解码器结构
class EncoderDecoder(nn.Module):
```

```
    """
    A standard Encoder-Decoder architecture.
    Base for this and many other models.
    """
    def __init__(self, encoder, decoder, src_embed, tgt_embed, generator):
        #初始化函数中有5个参数，分别是编码器对象，解码器对象,源数据嵌入函数，目标数据嵌入函数，以及输出部分的类别生成器对象.
        super(EncoderDecoder, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.src_embed = src_embed    # input embedding module(input embedding + positional encode)
        self.tgt_embed = tgt_embed    # ouput embedding module
        self.generator = generator    # output generation module

    def forward(self, src, tgt, src_mask, tgt_mask):
        "Take in and process masked src and target sequences."
        #在forward函数中，有四个参数，source代表源数据，target代表目标数据,source_mask和target_mask代表对应的掩码张量,在函数中，将source source_mask传入
        memory = self.encode(src, src_mask)
        res = self.decode(memory, src_mask, tgt, tgt_mask)
        return res

    def encode(self, src, src_mask):
        #编码函数，以source和source_mask为参数,使用src_embed对source做处理，然后和source_mask一起传给self.encoder
        src_embedds = self.src_embed(src)
        return self.encoder(src_embedds, src_mask)

    def decode(self, memory, src_mask, tgt, tgt_mask):
        #解码函数，以memory即编码器的输出，source_mask target target_mask为参数,使用tgt_embed对target做处理，然后和source_mask,target_mask,memory一起传
        target_embedds = self.tgt_embed(tgt)
        return self.decoder(target_embedds, memory, src_mask, tgt_mask)


# Full Model
def make_model(src_vocab, tgt_vocab, N=6, d_model=512, d_ff=2048, h=8, dropout=0.1):
    """
    构建模型
    params:
        src_vocab:
        tgt_vocab:
        N: 编码器和解码器堆叠基础模块的个数
        d_model: 模型中embedding的size，默认512
        d_ff: FeedForward Layer层中embedding的size，默认2048
        h: MultiHeadAttention中多头的个数，必须被d_model整除
        dropout:
    """
    c = copy.deepcopy
    attn = MultiHeadedAttention(h, d_model)
```

```
    ff = PositionwiseFeedForward(d_model, d_ff, dropout)
    position = PositionalEncoding(d_model, dropout)
    model = EncoderDecoder(
        Encoder(EncoderLayer(d_model, c(attn), c(ff), dropout), N),
        Decoder(DecoderLayer(d_model, c(attn), c(attn), c(ff), dropout), N),
        nn.Sequential(Embeddings(d_model, src_vocab), c(position)),
        nn.Sequential(Embeddings(d_model, tgt_vocab), c(position)),
        Generator(d_model, tgt_vocab))

    # This was important from their code.
    # Initialize parameters with Glorot / fan_avg.
    for p in model.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform_(p)
    return model
```

7 实战案例

下面我们用一个人造的玩具级的小任务，来实战体验下Transformer的训练，加深我们的理解，并且验证我们上面所述代码是否work。

任务描述：针对数字序列进行学习，学习的最终目标是使模型学会输出与输入的序列删除第一个字符之后的相同的序列，如输入[1,2,3,4,5]，我们尝试让模型学会输出[2,3,4,5]。

显然这对模型来说并不难，应该简单的若干次迭代就能学会。

代码实现的基本的步骤是：

第一步：构建并生成人工数据集

第二步：构建Transformer模型及相关准备工作

第三步：运行模型进行训练和评估

第四步：使用模型进行贪婪解码

篇幅的原因，这里就不对数据构造部分的代码进行介绍了，感兴趣欢迎大家查看项目的源码，并且亲自运行起来跑跑看：https://github.com/datawhalechina/dive-into-cv-pytorch/tree/master/code/chapter06_transformer/6.1_hello_transformer

训练的大致流程如下：

```
# Train the simple copy task.
device = "cuda"
nrof_epochs = 20
batch_size = 32
V = 11    # 词典的数量
sequence_len = 15  # 生成的序列数据的长度
nrof_batch_train_epoch = 30    # 训练时每个epoch多少个batch
nrof_batch_valid_epoch = 10    # 验证时每个epoch多少个batch
criterion = LabelSmoothing(size=V, padding_idx=0, smoothing=0.0)
```

```
model = make_model(V, V, N=2)
optimizer = torch.optim.Adam(model.parameters(), lr=0, betas=(0.9, 0.98), eps=1e-9)
model_opt = NoamOpt(model.src_embed[0].d_model, 1, 400, optimizer)
if device == "cuda":
    model.cuda()


for epoch in range(nrof_epochs):
    print(f"\nepoch {epoch}")
    print("train...")
    model.train()
    data_iter = data_gen(V, sequence_len, batch_size, nrof_batch_train_epoch, device)
    loss_compute = SimpleLossCompute(model.generator, criterion, model_opt)
    train_mean_loss = run_epoch(data_iter, model, loss_compute, device)
    print("valid...")
    model.eval()
    valid_data_iter = data_gen(V, sequence_len, batch_size, nrof_batch_valid_epoch, device)
    valid_loss_compute = SimpleLossCompute(model.generator, criterion, None)
    valid_mean_loss = run_epoch(valid_data_iter, model, valid_loss_compute, device)
    print(f"valid loss: {valid_mean_loss}")
```

训好模型后，使用贪心解码的策略，进行预测。

推理得到预测结果的方法并不是唯一的，贪心解码是最常用的，我们在 6.1.2 模型输入的小节中已经介绍过，其实就是先从一个句子起始符开始，每次推理解码器得到一个输出，然后将得到的输出加到解码器的输入中，再次推理得到一个新的输出，循环往复直到预测出句子的终止符，此时将所有预测连在一起便得到了完整的预测结果。

贪心解码的代码如下：

```
# greedy decode
def greedy_decode(model, src, src_mask, max_len, start_symbol):
    memory = model.encode(src, src_mask)
    # ys代表目前已生成的序列，最初为仅包含一个起始符的序列，不断将预测结果追加到序列最后
    ys = torch.ones(1, 1).fill_(start_symbol).type_as(src.data)
    for i in range(max_len-1):
        out = model.decode(memory, src_mask,
                           Variable(ys),
                           Variable(subsequent_mask(ys.size(1)).type_as(src.data)))
        prob = model.generator(out[:, -1])
        _, next_word = torch.max(prob, dim = 1)
        next_word = next_word.data[0]
        ys = torch.cat([ys, torch.ones(1, 1).type_as(src.data).fill_(next_word)], dim=1)
    return ys


print("greedy decode")
model.eval()
src = Variable(torch.LongTensor([[1,2,3,4,5,6,7,8,9,10]])).cuda()
```

```
src_mask = Variable(torch.ones(1, 1, 10)).cuda()
pred_result = greedy_decode(model, src, src_mask, max_len=10, start_symbol=1)
print(pred_result[:, 1:])
```

运行我们的训练脚本，训练过程与预测结果打印如下：

```
...

epoch 18
train...
Epoch Step: 1 Loss: 0.078836 Tokens per Sec: 13734.076172
valid...
Epoch Step: 1 Loss: 0.029015 Tokens per Sec: 23311.662109
valid loss: 0.03555255010724068

epoch 19
train...
Epoch Step: 1 Loss: 0.042386 Tokens per Sec: 13782.227539
valid...
Epoch Step: 1 Loss: 0.022001 Tokens per Sec: 23307.326172
valid loss: 0.014436692930758

greedy decode
tensor([[ 2,  3,  4,  5,  6,  7,  8,  9, 10]], device='cuda:0')
```

可以看到，由于任务非常简单，通过20epoch的简单训练，loss已经收敛到很低。

测试用例[1,2,3,4,5,6,7,8,9,10] 的预测结果为[2,3,4,5,6,7,8,9,10]，符合预期，说明我们的Transformer模型搭建正确了，成功～

小结

本次我们介绍了Transformer的基本原理，并且由外向内逐步拆解出每个模块进行了原理和代码的讲解，最后通过一个玩具级的demo实践了Transformer的训练和推理流程。希望通过这些内容，能够让初学者对Transformer有了更清晰的认知。

本文参考：http://nlp.seas.harvard.edu/2018/04/03/attention.html