【论文精读】Transformer论文逐段精读

一、整篇论文的总结

引言：论文介绍了一种新的模型架构——Transformer，这是第一个完全依赖于自注意力机制来处理序列转换任务的模型。这种架构避免了RNN和CNN，可以处理长距离依赖性，并且可以并行计算，从而提高了训练效率。

背景：论文详细介绍了自注意力机制和编码器-解码器架构的基本概念，这两个概念是Transformer模型的基础。

模型架构：Transformer模型由编码器和解码器组成，每个部分都是由多个相同的层堆叠而成。每一层都有两个子层：自注意力层和前馈神经网络。为了进行残差连接，每个子层的输出都会与其输入相加并进行层归一化。

自注意力机制：自注意力机制允许模型在序列的所有位置上进行注意力计算，从而更好地处理长距离依赖性。论文详细介绍了自注意力机制的计算过程，并引入了多头注意力的概念，通过这种方式，模型可以关注来自不同位置的信息。

位置编码：由于Transformer模型没有明确的顺序性，因此需要添加位置编码来保留序列中的词的顺序信息。论文使用了正弦和余弦函数的组合来生成位置编码。

训练细节：论文使用了WMT 2014英德和英法翻译任务进行训练，并详细介绍了训练过程中的各种设置，包括优化器的选择，学习率的调整，以及正则化的方法。

实验结果：Transformer模型在WMT 2014英德和英法翻译任务上都取得了新的最高水平。在英德翻译任务中，Transformer模型的表现甚至超过了所有以前报告的集成模型。

模型变化：论文还探讨了对Transformer模型进行各种变化的效果，包括改变注意力键的大小，增大模型大小，以及使用学习的位置嵌入等。

在其他任务上的泛化能力：除了机器翻译任务，论文还在英语成分解析任务上测试了Transformer模型的泛化能力。结果表明，Transformer模型在这个任务上的表现也非常出色，超过了所有以前报告的模型，除了递归神经网络语法。

注意力头的行为：论文最后探讨了注意力头的行为，发现许多注意力头表现出与句子结构相关的行为，这表明这些头显然学会了执行不同的任务。

二、常见面试题

2.1 什么是Transformer？为什么要使用Transformer？如何使用Transformer？

什么是Transformer？：Transformer是一种基于自注意力机制的序列处理模型，它由编码器和解码器组成，每个部分都包含多个相同的层，每个层都使用自注意力机制和前馈神经网络。

为什么要使用Transformer？：传统的序列处理模型，如RNN和CNN，在处理长序列时，往往会遇到长距离依赖问题。而Transformer模型通过自注意力机制，可以直接关注到序列中的任何位置，有效地处理了长距离依赖问题。此外，Transformer模型的并行性更强，训练效率更高。

如何使用Transformer？：使用Transformer模型，首先需要定义模型的结构，包括编码器和解码器的层数、注意力头的数量等参数。然后，需要准备输入数据，包括源序列和目标序列，并进行必要的预处理，如词嵌入和位置编码。最后，通过前向传播和反向传播，训练模型，并在测试数据上验证模型的性能。

2.2 Transformer的主要组成部分是什么？

Transformer的主要组成部分包括编码器、解码器、自注意力机制、前馈神经网络和位置编码。

编码器：编码器由多个相同的层堆叠而成，每一层都包含两个子层：自注意力机制和前馈神经网络。编码器的任务是将输入序列转换为一系列连续的表示，这些表示同时包含了输入序列的每个元素的上下文信息。

解码器：解码器也由多个相同的层堆叠而成，每一层有三个子层：两个自注意力层和一个前馈神经网络。第一个自注意力层只允许关注到前面的位置，第二个自注意力层则接收编码器的输出作为K和V。

自注意力机制：自注意力机制是Transformer的核心，它允许模型在所有位置的输入序列中同时关注不同位置的信息。

前馈神经网络：前馈神经网络在每个注意力层之后进行处理，对注意力层的输出进行进一步的变换。

位置编码：位置编码是通过添加到输入嵌入中来给模型提供序列中单词的位置信息。

2.3 什么是自注意力机制？为什么要使用自注意力机制？如何使用自注意力机制？

什么是自注意力机制？：自注意力机制是一种能够处理序列数据的机制，它可以计算序列中每个元素对其他元素的影响。

为什么要使用自注意力机制？：自注意力机制可以直接关注到序列中的任何位置，有效地处理了长距离依赖问题。此外，自注意力机制的并行性更强，训练效率更高。

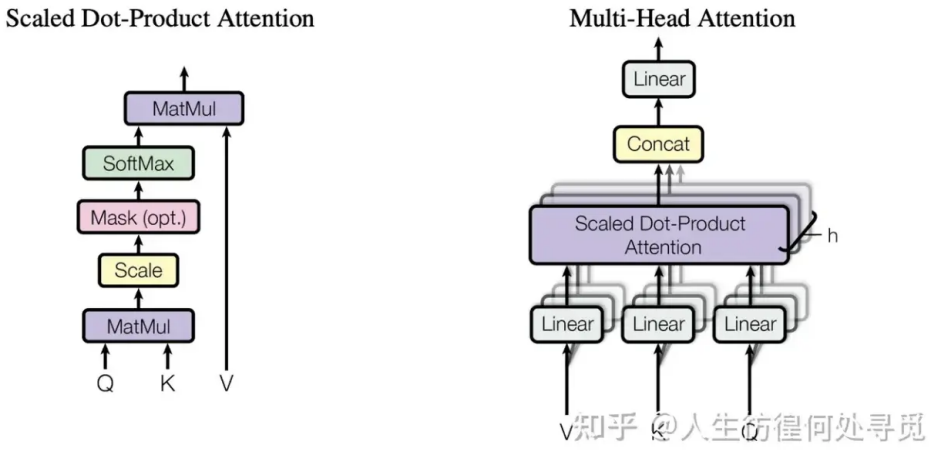
如何使用自注意力机制？：在自注意力机制中，首先需要计算每个元素的查询（Query）、键（Key）和值（Value）。然后，通过计算查询和所有键的点积，得到每个元素对其他元素的注意力分数。最后，用这些注意力分数对值进行加权求和，得到最终的输出。

2.4 什么是多头注意力？为什么要使用多头注意力？如何使用多头注意力？

什么是多头注意力？：多头注意力是一种扩展了自注意力机制的方法，它将输入的维度分割成多个部分，然后对每个部分分别进行自注意力操作，最后再将所有的结果拼接起来。

为什么要使用多头注意力？：多头注意力可以让模型同时关注来自不同位置的信息，且每个头可以学习到不同的注意力模式。

如何使用多头注意力？：在多头注意力中，首先需要将输入的维度分割成多个部分，然后对每个部分分别进行自注意力操作，最后再将所有的结果拼接起来。



Scaled Dot-Product Attention & Multi-Head Arrention

2.5 什么是位置编码？为什么要使用位置编码？如何使用位置编码？

什么是位置编码？：位置编码是一种提供序列中单词位置信息的方法。由于Transformer模型没有明确的顺序感知能力，因此需要通过位置编码来提供这种信息。

为什么要使用位置编码？：在处理序列数据时，元素的位置信息往往非常重要。位置编码可以让Transformer模型利用这种信息。

如何使用位置编码？：在使用位置编码时，首先需要计算每个位置的编码，然后将这些编码添加到对应位置的输入嵌入中。

2.6 Transformer如何处理长距离依赖？

Transformer通过自注意力机制处理长距离依赖问题。自注意力机制可以直接关注到序列中的任何位置，因此，无论两个元素之间的距离有多远，Transformer都能够直接建立它们之间的联系。

2.7 Transformer的优点是什么？

Transformer的优点主要包括：

处理长距离依赖：通过自注意力机制，Transformer可以直接关注到序列中的任何位置，有效地处理了长距离依赖问题。

并行性强：由于自注意力机制可以同时处理所有位置的信息，因此Transformer的并行性更强，训练效率更高。

灵活性强：Transformer模型的结构非常灵活，可以通过调整层数、注意力头的数量等参数，来适应不同的任务和数据。

2.8 Transformer的缺点是什么？

Transformer的缺点主要包括：

计算复杂度高：由于自注意力机制需要计算所有位置之间的关系，因此Transformer的计算复杂度较高，尤其是在处理长序列时。

缺乏显式的顺序感知能力：虽然Transformer通过位置编码引入了位置信息，但它仍然缺乏显式的顺序感知能力，这在某些任务中可能会成为问题。

需要大量的训练数据：Transformer是一种参数量非常大的模型，因此需要大量的训练数据才能充分学习到有效的模式。

2.9 Transformer为何使用多头注意力机制？（为什么不使用一个头）

Transformer使用多头注意力机制的主要原因是，多头注意力可以让模型同时关注来自不同位置的信息，且每个头可以学习到不同的注意力模式。如果只使用一个头，那么模型只能学习到一种固定的注意力模式，这可能会限制模型的表达能力。

2.10 Transformer为什么Q和K使用不同的权重矩阵生成，为何不能使用同一个值进行自身的点乘？（注意和第一个问题的区别）

在自注意力机制中，Q（Query）和K（Key）分别代表了查询和键，它们的作用是不同的。Q用于表示当前的查询目标，而K用于表示可以被查询的内容。如果Q和K使用同一

个值进行自身的点乘，那么就相当于每个元素只关注自己，而忽视了其他元素的信息，这与自注意力机制的初衷相悖。

2.11 Transformer计算attention的时候为何选择点乘而不是加法？两者计算复杂度和效果上有什么区别？

Transformer选择点乘来计算attention的主要原因是，点乘可以更好地衡量两个元素之间的相似性。如果使用加法，那么无论两个元素是否相似，它们的加法结果都是一样的，这无法反映出元素之间的相似性。在计算复杂度上，点乘和加法的复杂度是相同的，都是O(n)。但在效果上，由于点乘可以更好地衡量相似性，因此通常会比加法得到更好的结果。

2.12 为什么在进行softmax之前需要对attention进行scaled（为什么除以dk的平方根），并使用公式推导进行讲解

在进行softmax之前对attention进行scaled的主要原因是，当点乘的结果很大时，softmax函数可能会进入饱和区，导致梯度消失。通过除以dk的平方根，可以使得点乘的结果在一个合理的范围内，避免softmax函数进入饱和区。具体的公式推导如下：

$$softmax(QK^T/sqrt(dk)) = exp(QK^T/sqrt(dk))/sum(exp(QK^T/sqrt(dk)))$$

```
1 其中，Q和K是查询和键，dk是它们的维度，sqrt(dk)是dk的平方根。
```

2.13 为什么在进行多头注意力的时候需要对每个head进行降维？

在进行多头注意力的时候，需要对每个head进行降维的主要原因是，每个头的输出维度都是原始维度的一部分，如果不进行降维，那么在多头注意力后的输出维度将会是原始维度的n倍（n是头的数量），这会导致模型的参数量和计算复杂度大大增加。通过降维，可以保持模型的输出维度不变，同时还能让每个头关注到不同的特征。

2.14 大概讲一下Transformer的Encoder模块？

Transformer的Encoder模块主要由两部分组成：自注意力层和前馈神经网络层。自注意力层负责处理输入序列，通过自注意力机制，每个元素可以关注到序列中的任何位置，有效地处理了长距离依赖问题。前馈神经网络层则是一个普通的神经网络，用于进一步处理自注意力层的输出。这两部分都是通过残差连接和层归一化结合起来的，这样可以防止梯度消失，使得模型可以进行深层的训练。

2.15 为何在获取输入词向量之后需要对矩阵乘以embedding size的开方？意义是什么？

在获取输入词向量之后，需要对矩阵乘以embedding size的开方，主要是为了防止词向量的值过大。词向量的值过大可能会导致softmax函数进入饱和区，使得梯度消失。通过乘以embedding size的开方，可以使得词向量的值在一个合理的范围内，避免softmax函数进入饱和区。

2.16 简单介绍一下Transformer的位置编码？有什么意义和优缺点？

Transformer的位置编码是一种用于表示序列中元素位置信息的方法。由于Transformer的自注意力机制没有考虑元素的位置信息，因此需要通过位置编码来引入这种信息。Transformer使用的是一种基于正弦和余弦函数的位置编码，它可以处理任意长度的序列，并且具有良好的泛化能力。但是，这种位置编码的缺点是，它并不能直接表示出元素之间的相对位置关系。

2.17 还了解哪些关于位置编码的技术，各自的优缺点是什么？

除了Transformer使用的正弦和余弦函数的位置编码，还有一种常见的位置编码是位置嵌入（Position Embedding）。位置嵌入是将位置信息直接嵌入到词向量中，这种方法的优点是可以直接学习到元素之间的相对位置关系，但是缺点是只能处理固定长度的序列，且泛化能力较差。

2.18 在计算attention score的时候如何对padding做mask操作？

在计算attention score的时候，对padding做mask操作的方法是将padding位置的attention score设置为一个非常大的负数。这样，在进行softmax操作时，这些位置的权重将接近于0，从而实现了忽略padding位置的效果。

2.19 简单讲一下Transformer中的残差结构以及意义。

Transformer中的残差结构是指在每个子层（如自注意力层和前馈神经网络层）的输出上添加一个残差连接，即将子层的输入直接加到其输出上。这种结构的意义在于，它可以防止梯度消失，使得模型可以进行深层的训练。同时，残差结构也可以使得模型更容易学习到恒等映射，这有助于提高模型的性能。

2.20 为什么transformer块使用LayerNorm而不是BatchNorm？LayerNorm 在Transformer的位置是哪里？

Transformer块使用LayerNorm而不是BatchNorm的主要原因是，LayerNorm是对单个样本进行归一化，而BatchNorm是对整个批次的样本进行归一化。在处理序列数据时，由于每个序列的长度可能不同，因此使用BatchNorm可能会导致问题。而LayerNorm则没有这个问题，因此更适合用于Transformer。在Transformer中，LayerNorm通常位于每个子层的输出和残差连接之后。

2.21 简答讲一下BatchNorm技术，以及它的优缺点。

BatchNorm是一种用于加速神经网络训练的技术，它通过对每一层的输入进行归一化，使得输入的均值为0，方差为1，从而解决了梯度消失和梯度爆炸的问题。BatchNorm的优点是可以加速训练，提高模型的性能，同时也有一定的正则化效果。但是，BatchNorm的缺点是它依赖于批次的大小，如果批次太小，那么BatchNorm的效果可能会变差。

2.22 简单描述一下Transformer中的前馈神经网络？使用了什么激活函数？相关优缺点？

Transformer中的前馈神经网络是一个普通的全连接神经网络，它由两个线性变换和一个非线性激活函数组成。在Transformer中，通常使用ReLU或者GELU作为激活函数。前馈神经网络的优点是可以进一步提取特征，增加模型的复杂性。但是，前馈神经网络的缺点是增加了模型的参数量和计算复杂度。

2.23 Encoder端和Decoder端是如何进行交互的？（在这里可以问一下关于seq2seq的attention知识）

在Transformer中，Encoder端和Decoder端的交互主要通过注意力机制实现。具体来说，Decoder在生成每一个输出元素时，都会对Encoder的所有输出进行注意力计算，从而得到当前位置的上下文信息。这种交互方式使得Decoder可以考虑到输入序列的全局信息，从而生成更准确的输出。

2.24 Decoder阶段的多头自注意力和encoder的多头自注意力有什么区别？（为什么需要decoder自注意力需要进行 sequence mask)

Decoder阶段的多头自注意力和Encoder的多头自注意力的主要区别在于，Decoder需要进行序列掩码（sequence mask）。这是因为在序列生成任务中，当前位置的输出只能依赖于前面的位置，不能依赖于后面的位置。通过序列掩码，可以确保在计算注意力时，当前位置只关注到前面的位置，从而满足序列生成任务的要求。

2.25 Transformer的并行化提现在哪个地方？Decoder端可以做并行化吗？

Transformer的并行化主要体现在Encoder端。由于Encoder端处理的是整个输入序列，且每个位置的处理都是独立的，因此可以将整个序列的处理并行化，大大提高了计算效率。而Decoder端由于在生成每个位置的输出时，都需要依赖于前面的位置，因此不能直接进行并行化。但是，可以通过一些技巧，如缓存已经计算过的结果，来提高Decoder端的计算效率。

2.26 简单描述一下wordpiece model 和 byte pair encoding，有实际应用过吗？

wordpiece model和byte pair encoding都是一种用于处理开放词汇表问题的方法。wordpiece model是将词分割成多个子词，这些子词可以是单个字符，也可以是多个字符组成的词片。byte pair encoding则是通过统计字符对出现的频率，将频率高的字符对合并为一个新的字符，从而得到一个更大的词汇表。这两种方法都被广泛应用于各种NLP任务中，如机器翻译，文本分类等。

2.27 Transformer训练的时候学习率是如何设定的？Dropout是如何设定的，位置在哪里？Dropout 在测试的需要有什么需要注意的吗？

在Transformer的训练中，学习率通常使用一个特殊的学习率调度策略，即先线性增加学习率，然后再逐渐减小学习率。这种策略可以在训练初期快速收敛，同时在训练后期保持稳定的性能。Dropout是一种正则化技术，用于防止过拟合。在Transformer中，Dropout通常设置在每个子层的输出和残差连接之后，以及在前馈神经网络中。在测试时，需要关闭Dropout，即将Dropout的比率设置为0，以确保模型的输出是确定的。

2.28 引申一个关于bert问题，bert的mask为何不学习transformer在attention处进行屏蔽score的技巧？

BERT的mask操作是在输入层进行的，即在输入序列中随机选择一些位置进行mask，然后让模型预测这些被mask的位置的原始值。这种方法的优点是可以让模型在训练时就考虑到序列中的所有位置，从而更好地学习到上下文信息。而在attention处进行mask的方法虽然也可以实现类似的效果，但是它需要在每个位置都进行mask操作，这会增加模型的计算复杂度。因此，BERT选择在输入层进行mask操作，以达到更好的效果和更高的计算效率。

三、实现Transformer的步骤 & 代码

步骤1：实现Transformer中的自注意力机制

1.定义查询（Q）、键（K）和值（V）：这些是自注意力机制的输入，它们是输入序列的线性变换。
2.计算注意力分数：注意力分数是查询（Q）和键（K）的点积，然后除以缩放因子（通常是根号下的维度数），最后应用softmax函数。
3.计算输出：输出是注意力分数和值（V）的乘积。
4.多头注意力：为了充分利用模型的表示能力，我们通常会使用多头注意力，即将输入分割成多个部分，然后分别进行自注意力计算，最后将结果拼接在一起。

步骤2：使用PyTorch实现自注意力机制

```
class SelfAttention(nn.Module):
    def __init__(self, embed_size, heads):
        super(SelfAttention, self).__init__()
        self.embed_size = embed_size
        self.heads = heads
        self.head_dim = embed_size // heads
        assert (
            self.head_dim * heads == embed_size
        ), "Embedding size needs to be divisible by heads"
        self.values = nn.Linear(self.head_dim, self.head_dim, bias=False)
        self.keys = nn.Linear(self.head_dim, self.head_dim, bias=False)
        self.queries = nn.Linear(self.head_dim, self.head_dim, bias=False)
        self.fc_out = nn.Linear(heads * self.head_dim, embed_size)
    def forward(self, values, keys, query, mask):
        N = query.shape[0]
        value_len, key_len, query_len = values.shape[1], keys.shape[1], quer
        # Split the embedding into self.heads different pieces
        values = values.reshape(N, value_len, self.heads, self.head_dim)
        keys = keys.reshape(N, key_len, self.heads, self.head_dim)
        queries = query.reshape(N, query_len, self.heads, self.head_dim)
        values = self.values(values)  # (N, value_len, heads, head_dim)
        keys = self.keys(keys)  # (N, key_len, heads, head_dim)
        queries = self.queries(queries)  # (N, query_len, heads, heads_dim)
        # Get the dot product between queries and keys, and then apply the s
        energy = torch.einsum("nqhd,nkhd->nhqk", [queries, keys])
        if mask is not None:
            energy = energy.masked_fill(mask == 0, float("-1e20"))
        attention = torch.softmax(energy / (self.embed_size ** (1 / 2)), dim
        out = torch.einsum("nhql,nlhd->nqhd", [attention, values]).reshape(
            N, query_len, self.heads * self.head_dim
        )
        out = self.fc_out(out)
        return out
```

步骤3：实现Transformer中的位置编码

1.定义位置编码：位置编码是一个二维的矩阵，其中行代表序列中的位置，列代表编码的维度。对于每一行（位置p），其偶数列的编码为sin(p / 10000^(2i/d_model))，奇数列的编码为cos(p / 10000^(2i/d_model))，其中i为列的索引，d_model为模型的维度。
2.添加位置编码：位置编码会被添加到模型的输入嵌入中，这样模型就可以区分序列中的不同位置。

步骤4：使用PyTorch实现位置编码

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super(PositionalEncoding, self).__init__()
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * -(math.lo
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0).transpose(0, 1)
        self.register_buffer('pe', pe)
    def forward(self, x):
        x = x + self.pe[:x.size(0), :]
        return x
```

步骤5：实现Transformer中的编码器

1.定义编码器层：编码器由多个相同的层堆叠而成，每一层都有两个子层：自注意力层和前馈神经网络。为了进行残差连接，每个子层的输出都会与其输入相加并进行层归一化。
2.实现自注意力机制：自注意力机制是编码器中的关键部分。你需要实现这个机制，使得模型可以在序列的所有位置上进行注意力计算。
3.实现前馈神经网络：前馈神经网络是编码器中的另一个关键部分。你需要实现这个网络，它由两个线性层和一个ReLU激活函数组成。
4.堆叠编码器层：最后，你需要将多个编码器层堆叠在一起，形成完整的编码器。

步骤6：使用PyTorch实现编码器

```
class TransformerBlock(nn.Module):
    def __init__(self, embed_size, heads, dropout, forward_expansion):
        super(TransformerBlock, self).__init__()
        self.attention = SelfAttention(embed_size, heads)
        self.norm1 = nn.LayerNorm(embed_size)
        self.norm2 = nn.LayerNorm(embed_size)

        self.feed_forward = nn.Sequential(
            nn.Linear(embed_size, forward_expansion * embed_size),
            nn.ReLU(),
            nn.Linear(forward_expansion * embed_size, embed_size),
        )

        self.dropout = nn.Dropout(dropout)

    def forward(self, value, key, query, mask):
        attention = self.attention(value, key, query, mask)

        # 添加第一个残差连接，然后进行层归一化
        x = self.norm1(attention + query)
```

```
        x = self.dropout(x)

        # 通过前馈神经网络
        forward = self.feed_forward(x)

        # 添加第二个残差连接，然后进行层归一化
        out = self.norm2(forward + x)
        out = self.dropout(out)
        return out
```

步骤7：实现Transformer中的解码器

1.定义解码器层：解码器也由多个相同的层堆叠而成，每一层有三个子层：两个自注意力层和一个前馈神经网络。第一个自注意力层只允许关注到前面的位置，第二个自注意力层则接收编码器的输出作为K和V。
2.实现自注意力机制：解码器中的自注意力机制与编码器中的类似，但需要进行一些修改，使得在生成每个词时只能使用前面的词。
3.实现前馈神经网络：前馈神经网络与编码器中的相同。
4.堆叠解码器层：最后，你需要将多个解码器层堆叠在一起，形成完整的解码器。

步骤8：使用PyTorch实现解码器

```
class DecoderBlock(nn.Module):
    def __init__(self, embed_size, heads, forward_expansion, dropout, device
        super(DecoderBlock, self).__init__()
        self.norm = nn.LayerNorm(embed_size)
        self.attention = SelfAttention(embed_size, heads=heads)
        self.transformer_block = TransformerBlock(
            embed_size, heads, dropout, forward_expansion
        )
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, value, key, src_mask, trg_mask):
        attention = self.attention(x, x, x, trg_mask)
        query = self.dropout(self.norm(attention + x))
        out = self.transformer_block(value, key, query, src_mask)
        return out
```

步骤9：使用PyTorch实现Transformer模型

```
class Transformer(nn.Module):
    def __init__(
        self,
        src_vocab_size,
        trg_vocab_size,
```

```
        src_pad_idx,
        trg_pad_idx,
        embed_size=512,
        num_layers=6,
        forward_expansion=4,
        heads=8,
        dropout=0,
        device="cpu",
        max_length=100,
    ):

        super(Transformer, self).__init__()

        self.encoder = Encoder(
            src_vocab_size,
            embed_size,
            num_layers,
            heads,
            device,
            forward_expansion,
            dropout,
            max

_length,
        )

        self.decoder = Decoder(
            trg_vocab_size,
            embed_size,
            num_layers,
            heads,
            forward_expansion,
            dropout,
            device,
            max_length,
        )

        self.src_pad_idx = src_pad_idx
        self.trg_pad_idx = trg_pad_idx
        self.device = device

    def make_src_mask(self, src):
        src_mask = (src != self.src_pad_idx).unsqueeze(1).unsqueeze(2)
        return src_mask.to(self.device)

    def make_trg_mask(self, trg):
        N, trg_len = trg.shape
        trg_mask = torch.tril(torch.ones((trg_len, trg_len))).expand(
            N, 1, trg_len, trg_len
```

```
        )

        return trg_mask.to(self.device)

    def forward(self, src, trg):
        src_mask = self.make_src_mask(src)
        trg_mask = self.make_trg_mask(trg)
        enc_src = self.encoder(src, src_mask)
        out = self.decoder(trg, enc_src, src_mask, trg_mask)
        return out
```

四、十个问题 & 回答

Q1 论文试图解决什么问题？

这篇论文试图解决的问题是如何有效地处理序列数据，特别是在处理长序列时，如何保持高效和准确。为此，作者提出了一种新的模型——Transformer，它完全基于注意力机制，摒弃了传统的RNN和CNN结构，能够更好地处理长距离依赖问题。

Q2 这是否是一个新的问题？

这并不是一个新的问题。处理序列数据，特别是处理长序列数据的问题，一直是自然语言处理和其他序列任务中的一个重要问题。然而，这篇论文提出的Transformer模型，提供了一种全新的解决方案。

Q3 这篇文章要验证一个什么科学假设？

这篇文章的科学假设是：通过使用全注意力机制，可以构建一个更加高效且准确的序列处理模型，这个模型能够处理长距离依赖问题，并且在许多任务上超越现有的最先进的模型。

Q4 有哪些相关研究？如何归类？谁是这一课题在领域内值得关注的研究员？

相关的研究主要包括各种序列处理模型，如RNN、LSTM、GRU和CNN等。这些模型都试图解决序列数据的处理问题，但在处理长序列时，都存在一定的问题。这篇论文的作者包括Vaswani等人，他们在自然语言处理领域有很高的影响力，是值得关注的研究员。

Q5 论文中提到的解决方案之关键是什么？

论文中提到的解决方案的关键是全注意力机制。通过这种机制，模型可以直接关注到序列中的任何位置，从而有效地处理长距离依赖问题。此外，模型还使用了多头注意力和位置编码等技术，进一步提升了模型的性能。

Q6 论文中的实验是如何设计的？

论文中的实验主要包括两部分：一部分是在机器翻译任务上的实验，作者使用了WMT 2014 English-to-German和English-to-French数据集；另一部分是在英语句子解析任务上的实验，作者使用了Penn Treebank数据集。通过这些实验，作者验证了Transformer模型的有效性。

Q7 用于定量评估的数据集是什么？代码有没有开源？

用于定量评估的数据集包括WMT 2014 English-to-German、English-to-French和Penn Treebank数据集。论文中并没有明确提到代码是否开源，但现在有许多开源的Transformer模型实现，如Hugging Face的Transformers库。

Q8 论文中的实验及结果有没有很好地支持需要验证的科学假设？

是的，论文中的实验结果很好地支持了需要验证的科学假设。在机器翻译任务上，Transformer模型超越了当时的最先进模型。在英语句子解析任务上，Transformer也取得了很好的结果。

Q9 这篇论文到底有什么贡献？

这篇论文的主要贡献是提出了Transformer模型，这是一种全新的基于注意力机制的序列处理模型。这个模型在处理长序列数据时表现出了优越的性能，且在多个任务上超越了当时的最先进模型。

Q10 下一步呢？有什么工作可以继续深入？

下一步，可以在Transformer模型的基础上进行更多的研究和探索，如改进模型结构，提出新的注意力机制，或者将Transformer模型应用到更多的任务中。此外，还可以研究如何更好地理解和解释Transformer模型，以及如何提高模型的训练效率和推理速度。

五、每页摘要

第1页摘要

1.标题：注意力就是你所需要的
2.作者：Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, Illia Polosukhin
3.摘要：这篇论文提出了一种新的网络架构——Transformer，它完全基于注意力机制，而不需要复杂的循环或卷积神经网络，也不需要编码器和解码器。实验结果显示，这种模型在质量上优于其他模型，同时具有更好的并行性，训练时间大大减少。在WMT 2014英-德翻译任务上，该模型达到了28.4 BLEU，比现有的最佳结果（包括集成模型）提高了2 BLEU。在WMT 2014英-法翻译任务上，该模型在8个GPU上训练3.5天后，达到了41.8 BLEU的新的单模型最佳成绩，这是文献中最佳模型训练成本的一小部分。此外，该模型在英语句法解析任务上也表现出良好的泛化能力。
4.引言：循环神经网络，特别是长短期记忆（LSTM）和门控循环神经网络（GRU），已经被确定为序列建模和转换任务的最佳方法。

第2页摘要

1.序列建模和转换问题：循环神经网络，特别是长短期记忆（LSTM）和门控循环神经网络（GRU），已经被确定为序列建模和转换任务的最佳方法，例如语言建模和机器翻译。然而，这些模型的计算过程通常沿着输入和输出序列的符号位置进行，这种固有的序列性质阻止了在训练样本中的并行化，这在长序列长度时变得关键，因为内存限制限制了跨样本的批处理。
2.注意力机制：注意力机制已经成为各种任务中引人注目的序列建模和转换模型的一个重要部分，它允许在不考虑输入或输出序列中的距离的情况下建模依赖关系。然而，在除了少数几种情况外，这种注意力机制都是与循环网络一起使用的。
3.Transformer模型：在这项工作中，我们提出了Transformer模型，这种模型架构完全放弃了循环，而完全依赖于注意力机制来在输入和输出之间建立全局依赖关系。Transformer模型允许更多的并行化，并且在8个P100 GPU上训练了12小时后，可以达到翻译质量的新的最佳水平。
4.背景：减少序列计算的目标也是扩展神经GPU、ByteNet和ConvS2S的基础，所有这些都使用卷积神经网络作为基本的构建块，为所有的输入和输出位置并行计算隐藏表示。在这些模型中，将两个任意输入或输出位置的信号关联起来所需的操作数量随着位置之间的距离的增长而增长，对于ConvS2S是线性的，对于ByteNet是对数的。这使得学习远距离位置之间的依赖关系变得更加困难。在Transformer中，这被减少到了一个常数数量的操作，尽管由于平均注意力加权位置的效果，有效的分辨率降低了，我们用多头注意力来抵消这种效果。

第3页摘要

1.模型架构：Transformer模型的架构如图1所示。
2.编码器和解码器堆栈：
3.编码器：编码器由N=6个相同的层堆叠而成。每一层都有两个子层。第一个是多头自注意力机制，第二个是简单的位置全连接前馈网络。我们在每个子层周围使用残差连接，然后进行层归一化。也就是说，每个子层的输出是LayerNorm(x+ Sublayer(x))，其中Sublayer(x)是子层本身实现的函数。为了方便这些残差连接，模型中的所有子层以及嵌入层都产生维度为dmodel = 512的输出。
4.解码器：解码器也由N=6个相同的层堆叠而成。除了每个编码器层中的两个子层外，解码器还插入了第三个子层，该子层对编码器堆栈的输出执行多头注意力。与编码器类似，我们在每个子

层周围使用残差连接，然后进行层归一化。我们还修改了解码器堆栈中的自注意力子层，以防止位置关注到后续位置。这种掩蔽，结合输出嵌入被偏移一个位置的事实，确保了位置i的预测只能依赖于位置小于i的已知输出。

5.注意力：注意力函数可以被描述为将一个查询和一组键值对映射到一个输出，其中查询、键、值和输出都是向量。输出是值的加权和，其中每个值的权重是通过查询与相应键的兼容性函数计算的。

第4页摘要

1.缩放点积注意力：我们将我们特定的注意力称为"缩放点积注意力"（图2）。输入包括维度为dk的查询和键，以及维度为dv的值。我们计算查询与所有键的点积，将每个点积除以√dk，然后应用softmax函数得到值的权重。在实践中，我们同时对一组查询进行注意力函数的计算，这些查询被打包到一个矩阵Q中。键和值也被打包到矩阵K和V中。我们计算输出矩阵为：Attention(Q;K;V）= softmax(QKT/√dk)V。

2.多头注意力：我们发现，与其使用dmodel维的键、值和查询执行单一的注意力函数，不如将查询、键和值线性投影h次，使用不同的、学习得到的线性投影到dk、dk和dv维度。然后我们在这些投影版本的查询、键和值上并行执行注意力函数，得到dv维的输出值。这些输出值被连接起来，然后再次投影，得到最终的值，如图2所示。

第5页摘要

1.多头注意力：多头注意力允许模型在不同位置的不同表示子空间中共同关注信息。对于单一的注意力头，平均会抑制这种效果。我们在这项工作中使用h=8个并行的注意力层，或者说头。对于这些头，我们使用dk=dv=dmodel/h=64。由于每个头的维度减小，总的计算成本与全维度的单头注意力相似。

2.注意力在我们的模型中的应用：Transformer在三种不同的方式中使用多头注意力：

3.在"编码器-解码器注意力"层中，查询来自前一个解码器层，记忆键和值来自编码器的输出。这允许解码器中的每个位置关注输入序列中的所有位置。

4.编码器包含自注意力层。在自注意力层中，所有的键、值和查询都来自同一个地方，即编码器中前一层的输出。编码器中的每个位置可以关注到编码器前一层的所有位置。

5.同样，解码器中的自注意力层允许解码器中的每个位置关注到解码器中包括该位置在内的所有位置。我们需要防止解码器中的左向信息流动，以保持自回归性质。我们在缩放点积注意力中通过掩蔽（设置为-∞）所有在softmax输入中对应于非法连接的值来实现这一点。

6.位置全连接前馈网络：除了注意力子层外，我们的编码器和解码器中的每一层都包含一个全连接前馈网络，它被单独且相同地应用于每个位置。这包括两个线性变换，中间有一个ReLU激活函数。

7.嵌入和Softmax：与其他序列转换模型类似，我们使用学习的嵌入将输入令牌和输出令牌转换为dmodel维的向量。我们还使用通常的学习的线性变换和softmax函数将解码器输出转换为预测的下一个令牌概率。

8.位置编码：由于我们的模型不包含递归和卷积，为了让模型利用序列的顺序，我们必须注入一些关于相对或绝对位置的信息。

第6页摘要

1.位置编码：我们在编码器和解码器堆栈底部的输入嵌入中添加"位置编码"。位置编码具有与嵌入相同的维度dmodel，因此两者可以相加。有许多位置编码的选择，包括学习和固定的。在这项工作中，我们使用不同频率的正弦和余弦函数：

$$PE(pos;2i) = sin(pos/100002i/d_{model})$$
$$PE(pos;2i+1) = cos(pos/100002i/d_{model}):$$

其中，pos是位置，i是维度。也就是说，位置编码的每个维度对应一个正弦波。波长形成一个从2π到10000*2π的几何级数。我们选择这个函数，因为我们假设它会让模型更容易学习通过相对位置进行关注，因为对于任何固定的偏移k，PEpos+k可以被表示为PEpos的线性函数。

2.为什么使用自注意力：在这一部分，我们将自注意力层与常用的用于映射一个可变长度的符号表示序列（x1;:::;x n）到另一个等长序列（z1;:::;z n）的循环和卷积层进行比较，其中xi;zi∈Rd，例如典型的序列转换编码器或解码器中的隐藏层。我们使用自注意力的动机是考虑三个期望。

3.一个是每层的总计算复杂性。

4.另一个是可以并行化的计算量，用最少的顺序操作数来衡量。

5.第三个是网络中长距离依赖关系的路径长度。学习长距离依赖关系是许多序列转换任务的关键挑战。影响学习这种依赖关系的一个关键因素是网络中前向和后向信号必须遍历的路径的长度。输入和输出序列中任何位置组合之间的路径越短，学习长距离依赖关系就越容易。

第7页摘要

1.自注意力的优势：自注意力层可以在输入序列中的任何两个位置之间建立直接的依赖关系，这使得最大路径长度为O(1)。如果我们允许在输出序列中的每个位置关注输入序列中的所有位置，那么最大路径长度将增加到O(n/r)。我们计划在未来的工作中进一步研究这种方法。

2.训练：我们在标准的WMT 2014英德数据集上进行训练，该数据集包含大约450万个句子对。句子使用字节对编码进行编码，共享源-目标词汇表大约有37000个令牌。对于英法，我们使用了更大的WMT 2014英法数据集，该数据集包含3600万个句子，并将令牌分割成32000个词片词汇表。句子对按照大致的序列长度进行批处理。每个训练批次包含一组句子对，包含大约25000个源令牌和25000个目标令牌。

3.硬件和计划：我们在一台装有8个NVIDIA P100 GPU的机器上训练我们的模型。对于我们的基础模型，使用本文描述的超参数，每个训练步骤大约需要0.4秒。我们训练基础模型总共100,000步，或者12小时。对于我们的大模型，步骤时间是1.0秒。大模型训练了300,000步（3.5天）。

4.优化器：我们使用Adam优化器，其中β1=0.9，β2=0.98，ε=10^-9。我们在训练过程中改变学习率，根据公式：lrate = d_model^-0.5 * min(step_num^-0.5, step_num * warmup_steps^-1.5)。这对应于在前warmup_steps训练步骤中线性增加学习率，然后按照步骤数量的平方根的倒数比例减少。我们使用warmup_steps = 4000。

5.正则化：我们在训练过程中使用三种类型的正则化：

6.残差Dropout：我们在每个子层的输出上应用dropout，然后将其添加到子层输入并进行归一化。此外，我们在编码器和解码器堆栈中的嵌入和位置编码的和上应用dropout。对于基础模型，我们使用的dropout率为P_drop=0.1。

第8页摘要

1.标签平滑：在训练过程中，我们使用了值为0.1的标签平滑。这会损害困惑度，因为模型学会更加不确定，但会提高准确性和BLEU分数。
2.结果：
3.机器翻译：在WMT 2014英德翻译任务中，大型Transformer模型（表2中的Transformer (big)）比以前报告的最好的模型（包括集成模型）高出2.0以上的BLEU，建立了新的最高BLEU分数28.4。这个模型的配置列在表3的底部。训练在8个P100 GPU上进行，花费3.5天。即使是我们的基础模型也超过了所有以前发布的模型和集成模型，训练成本只是任何竞争模型的一小部分。
4.在WMT 2014英法翻译任务中，我们的大模型达到了41.0的BLEU分数，超过了所有以前发布的单一模型，训练成本不到之前最先进模型的1/4。用于英法训练的Transformer (big)模型使用了dropout率P_drop=0.1，而不是0.3。
5.模型变化：为了评估Transformer的不同组件的重要性，我们以不同的方式改变了我们的基础模型，测量了在开发集newstest2013上的英德翻译性能的变化。我们使用了上一节描述的波束搜索，但没有检查点平均。我们在表3中展示了这些结果。

第9页摘要

1.Transformer架构的变化：表3列出了对Transformer架构的各种变化。未列出的值与基础模型相同。所有的指标都是在英德翻译开发集newstest2013上。列出的困惑度是每个词片，根据我们的字节对编码，不应与每词困惑度进行比较。
2.Transformer在英语成分解析中的泛化能力：为了评估Transformer是否能泛化到其他任务，我们在英语成分解析上进行了实验。这个任务提出了特定的挑战：输出受到强烈的结构约束。在表3的(B)行，我们观察到减少注意力键大小dk会损害模型质量。这表明确定兼容性并不容易，比点积更复杂的兼容性函数可能是有益的。我们在(C)和(D)行进一步观察到，如预期，更大的模型更好，dropout在避免过拟合中非常有帮助。在(E)行，我们用学习的位置嵌入替换了我们的正弦位置编码，观察到的结果与基础模型几乎相同。

第10页摘要

1.Transformer在英语成分解析中的泛化能力：我们在Penn Treebank的Wall Street Journal (WSJ)部分上训练了一个4层的Transformer，大约有40K的训练句子。我们还在半监督设置中进行了训练，使用了大约1700万句子的更大的高置信度和BerkleyParser语料库。我们对WSJ只设置使用了16K的词汇表，对半监督设置使用了32K的词汇表。我们在Section 22开发集上进行了少量的实验来选择dropout，注意力和残差，学习率和束宽，所有其他参数都保持不变，与英德基础翻译模型相同。在推理过程中，我们将最大输出长度增加到输入长度+300。我们对WSJ只和半监督设置都使用了21的束宽和0.3的β。
2.结果：我们的结果表明，尽管缺乏任务特定的调整，我们的模型表现出令人惊讶的好，比所有以前报告的模型都要好，除了递归神经网络语法。与RNN序列到序列模型不同，Transformer在只对40K句子的WSJ训练集进行训练时，就超过了Berkeley-Parser。
3.结论：在这项工作中，我们提出了Transformer，这是第一个完全基于注意力的序列转换模型，用多头自注意力替换了编码器-解码器架构中最常用的递归层。对于翻译任务，Transformer可以比基于递归或卷积层的架构更快地进行训练。在WMT 2014英德和WMT 2014英法翻译任务中，我们都达到了新的最高水平。在前者的任务中，我们的最好模型甚至超过了所有以前报告的集成模型。我们对基于注意力的模型的未来感到兴奋，计划将它们应用到其他任务。我们计划将Transformer扩展到涉及除文本以外的输入和输出方式的问题，并研究局

部，受限的注意力机制，以有效地处理大的输入和输出，如图像，音频和视频。使生成更少序列化是我们的另一个研究目标。