9 综述 —— The Transformer

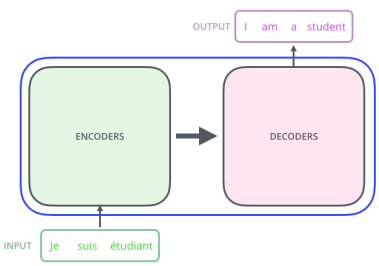
本章概述

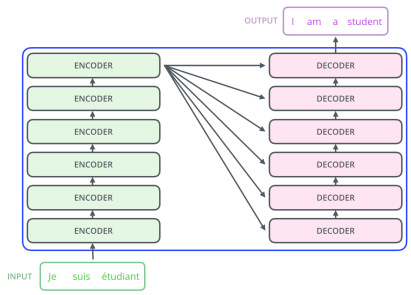
Transformer模型的出现犹如一场具有颠覆性意义的技术革命，重新定义了对**序列数据**处理的理解。过去，RNN和LSTM凭借其**递归结构**在处理时间序列方面取得了一定成绩，但在面对**长序列**时，这些模型常常遭遇瓶颈。当从一段冗长的文章中提取关键信息时，传统模型可能因为记忆和信息流动的限制而无法有效捕捉到核心内容，这种情况不仅影响了结果的准确性，更可能让人对NLP技术感到沮丧。而Transformer的引入恰恰解决了这个问题。**自注意力机制**可联想到人类的思维方式。当阅读文本时，思维并不是线性推进，而是瞬间关联和抓取与当前词相关的上下文信息。这种灵活的关注方式让Transformer在理解文本时不再受到序列长度的限制。通过对每个词的**查询、键和值**的计算，模型能**够并行处理**所有词的信息，就像构建了一张**动态的语义网络**，这种设计使得信息的传递和理解变得更加高效而丰富。

Transformer的成功不是偶然。人们最初还没有想到它在处理大量数据和复杂语义时展示出的潜力，尤其是在预训练的环境下，通过**自监督学习**，模型能从海量语料中汲取上下文知识，这或许可以说数据不仅仅是模型的输入，更是其学习的养分。丰富的背景信息能够帮助模型展现更深层次的理解和推理能力，这正是其在下游任务中表现出色的原因。然而，Transformer并不是完美的解决方案。尽管其自注意力机制在处理长序列时表现优越，但**计算复杂度的平方增长**仍然使得处理极长序列时面临挑战。虽然技术的进步让看到了更高效的处理方式，但科学的探索永无止境。为了解决这一问题，已经有许多研究者们纷纷尝试开发**稀疏注意力**和分层架构等更高效的变种，但Transformer的迭代改进依然在路上。

总览

首先，Transformer 就是个 Sequence-to-Sequence 模型，由 Encoders（多个 Encoder 堆叠）和 Decoders（多个 Decoder 堆叠）两部分组成。





简单拆解

编码器

这些编码器在结构上都是相同的（但它们不共享权重），每一层又分为两个子层，如图 1：

1. 编码器的输入首先流经自注意力层，该层帮助编码器在对特定单词进行编码时查看输入句子中的其他单词。
2. 自注意力层的输出被馈送到前馈神经网络。完全相同的前馈网络独立应用于每个位置。

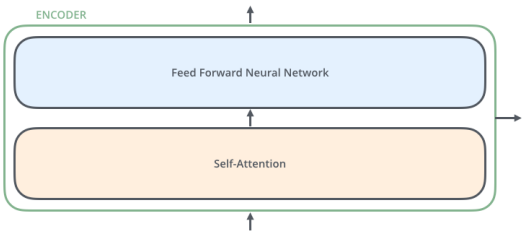


图 1

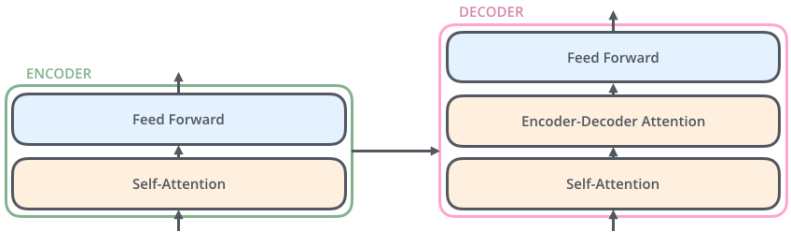


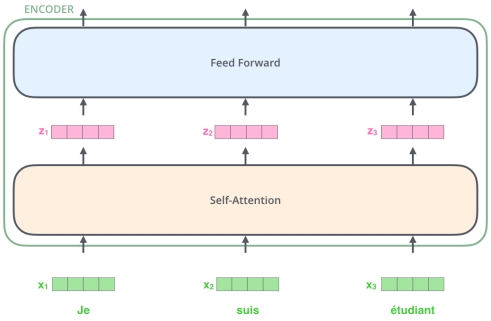
图 2

解码器

解码器同样有自注意力层和前馈神经网络层，但它们之间还有一个注意力层，帮助解码器关注输入句子的相关部分（类似于 seq2seq 模型中注意力的作用）。如图 2。

从输入到输出

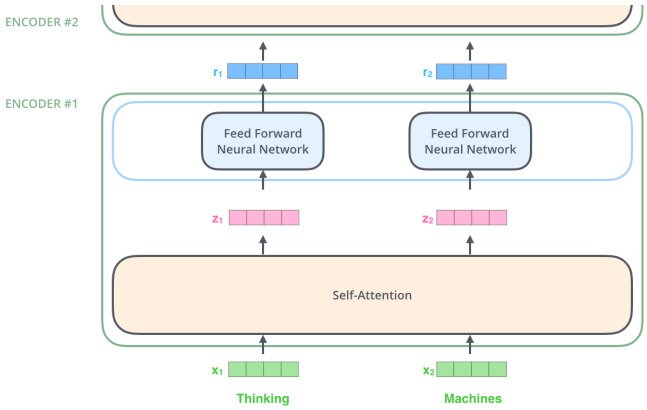
与 NLP 各种任务中的一样，首先使用嵌入算法将每个输入单词转换为向量。所有编码器接收一组向量作为输入，论文中的输入向量的维度是512。最底下的那个编码器接收的是嵌入向量，之后的编码器接收的是前一个编码器的输出。向量长度这个超参数是可以设置的，一般来说是训练集中最长的那个句子的长度。当输入序列经过词嵌入之后得到的向量会依次通过编码器组件中的两个层。



在这里，可以看到 Transformer 的一个关键属性，即每个位置上的单词在编码器中有各自的流通方向。在自注意力层中，这些路径之间存在依赖关系。 然而，前馈神经网络中没有这些依赖关系，因此各种路径可以在流过前馈神经网络层的时候并行计算。

编码器

上边提到每个编码器组件接受一组向量作为输入。在其内部，输入向量先通过一个自注意力层，再经过一个前馈神经网络，最后将其将输出给下一个编码器组件。



每个位置的单词都会经过一个自注意力过程。然后，它们各自通过前馈神经网络——完全相同的网络，每个向量分别流经该网络。

自注意力

假设要翻译下边这句话：

> ”The animal didn't cross the street because it was too tired”

这里 it 指的是什么？是 street 还是 animal？人理解起来很容易，但是对算法来讲就不那么容易了。当模型处理 it 这个词的时候，自注意力会让 it 和 animal 关联起来。当模型编码每个位置上的单词的时候，自注意力的作用就是：看一看输入句子中其他位置的单词，试图寻找一种对当前单词更好的编码方式。RNN 如何处理当前时间步的隐藏状态？将之前的隐藏状态与当前位置的输入结合起来。在 Transformer 中，自注意力机制则可以将其他相关单词的“理解”融入到当前处理的单词中。

细说自注意力机制

1. 对编码器的每个输入向量都计算三个向量，就是对每个输入向量都算一个 query、key、value 向量。

   怎么算的？把输入的词嵌入向量与三个权重矩阵相乘。权重矩阵是模型训练阶段训练出来的。

注意，这三个向量维度是64，比嵌入向量的维度小，嵌入向量、编码器的输入输出维度都是512。这三个向量**不是必须**比编码器输入输出的维数小，这样做主要是为了让多头注意力的计算更稳定。

2. 计算注意力得分。

假设现在在计算输入中第一个单词 Thinking 的自注意力。需要使用自注意力给输入句子中的每个单词打分，这个分数决定当编码某个位置的单词的时候，应该对其他位置上的单词给予多少关注度。这个得分是 query 和 key 的点乘积得出来的。例如，要算第一个位置的注意力得分的时候就要将第一个单词的 query 和其他单词的 key 依次相乘，在这里就是 $q_1 \cdot k_1$ ， $q_1 \cdot k_2$

3. 将计算获得的注意力分数除以8。

为什么选 8？是因为 key 向量的维度是 64，取其平方根，这样让梯度计算的时候更稳定。默认是这么设置的，当然也可以用其他值。

4. 除 8 之后将结果扔进 softmax 计算，使结果归一化，softmax 之后注意力分数相加等于 1，并且都是正数。

经过softmax 后得到注意力分数，其表示在计算当前位置的时候，其他单词受到的关注度的大小。显然在当前位置的单词肯定有一个高分，但是有时候也会注意到与当前单词相关的其他词汇。

5. 将每个 value 向量乘以注意力分数。

   这是为了留下想要关注的单词的 value，并把其他不相关的单词丢掉。在第一个单词位置得到新的 v_1 。

6. 将上一步的结果相加，输出本位置的注意力结果。

   第一个单词的注意力结果就是 z_1 。

这就是自注意力的计算。计算得到的向量直接传递给前馈神经网络。但是为了处理的更迅速，实际是用矩阵进行计算的。接下来尝试用矩阵计算。

用矩阵计算 self-attention

计算 Query, Key, Value 矩阵。直接把输入的向量打包成一个矩阵 X ，再把它乘以训练好的 W^Q 、 W^K 、 W^V 。X 矩阵中的一行相当于输入句子中的一个单词。其中维度的差异：原文中嵌入矩阵的长度为 512 , q、k、v 矩阵的长度为 64。

因为现在用矩阵处理，所以可以直接将之前的第二步到第六步压缩到一个公式中一步到位获得最终的注意力结果 Z 。

多头注意力

Transformer 进一步改进了自注意力层，增加了一个机制，也就是多头注意力机制。这样做有两个好处：

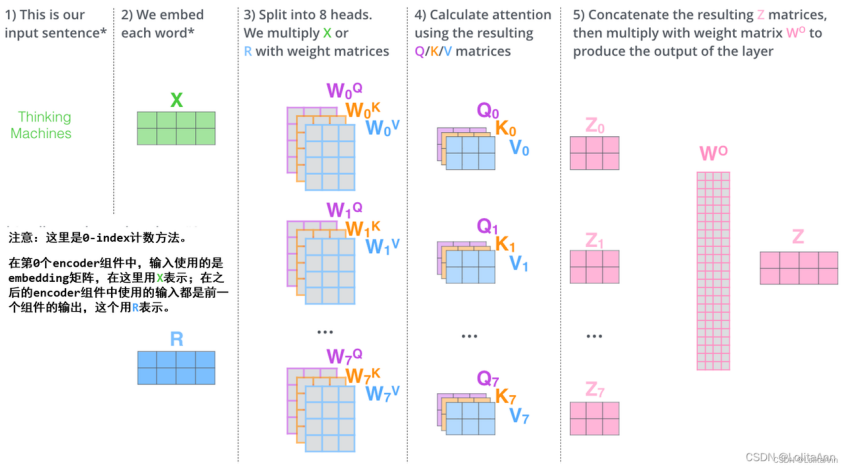
1. 它扩展了模型专注于不同位置的能力。

   在上面例子里只计算一个自注意力的的例子中，编码“Thinking”的时候，虽然最后 Z_1 或多或少包含了其他位置单词的信息，但是它实际编码中还是被 “Thinking” 单词本身所支配。如果翻译一个句子，比如 “The animal didn’ t cross the street because it was too tired”，会想知道 “it” 指的是哪个词，这时模型的“多头”注意力机制会起到作用。

2. 它给了注意层多个“表示子空间”。

   就是在多头注意力中同时用多个不同的 W^Q 、 W^K 、 W^V 权重矩阵（Transformer 使用8个头部，因此最终会得到8个计算结果），每个权重都是随机初始化的。经过训练每个 W^Q 、 W^K 、 W^V 都能将输入的矩阵投影到不同的表示子空间。

Transformer 中的一个多头注意力（有8个head）的计算，就相当于用自注意力做 8 次不同的计算，并得到 8 个不同的结果 Z 。但是这会存在一点问题，多头注意力出来的结果会进入一个前馈神经网络，这个前馈神经网络可不能一下接收8个注意力矩阵，它的输入需要是单个矩阵（矩阵中每个行向量对应一个单词），所以需要一种方法把这8个压缩成一个矩阵。



使用位置编码表示序列的位置

> 强烈安利一个详细解释位置编码的文章：Transforme 结构：位置编码详解。

如何表示输入序列中词汇的位置？Transformer 在每个输入的嵌入向量中添加了位置向量。这些位置向量遵循某些特定的模式，这有助于模型确定每个单词的位置或不同单词之间的距离。将这些值添加到嵌入矩阵中，一旦它们被投射到 Q、K、V 中，就可以在计算点积注意力时提供有意义的距离信息。位置编码向量和嵌入向量的维度是一样的。

残差

编码器中的一个细节：每个编码器中的每个子层（自注意力层、前馈神经网络）都有一个残差连接，之后是做了一个层归一化（layer-normalization）。

解码器

因为 encoder 的 decoder 组件差不多，所以基本上也介绍了解码器的组件是如何工作的。接下来直接看看二者是如何协同工作的。

编码器首先处理输入序列，将最后一个编码器组件的输出转换为一组注意向量 K 和 V。每个解码器组件将在“encoder-decoder attention”层中使用编码器传过来的 K 和 V，这有助于解码器将注意力集中在输入序列中的适当位置。在解码阶段每一轮计算都只往外蹦一个输出。输出步骤会一直重复，直到遇到句子结束符 表明transformer的解码器已完成输出。每一步的输出都会在下一个时间步喂给给底部解码器，解码器会像编码器一样运算并输出结果（每次往外蹦一个词）。

跟编码器一样，在解码器中我们也为其添加位置编码，以指示每个单词的位置。

解码器中的自注意力层和编码器中的不太一样：在解码器中，自注意力层只允许关注已输出位置的信息。实现方法是在自注意力层的 softmax 之前进行 mask，将未输出位置的信息设为极小值。**“encoder-decoder attention”层的工作原理和前边的多头自注意力差不多，但是 Q、K、V 的来源不同，Q 是从下层创建的（比如解码器的输入和下层decoder组件的输出），但是其 K 和 V 是来自最后一个编码器的输出结果。**（因为编码器是对整个输入序列进行编码嘛，然后将其结果转化成 K 和 V 传给解码器，这个 K、V 包含整个句子的所有信息。但是解码器的输入是什么？是拼接之前解码器的输出的单词。所以解码器造出来的 Q 仅包含已经输出的内容。）

最后的线性层和 softmax 层

Decoder 输出的是一个浮点型向量，如何把它变成一个词？这就是最后一个线性层和softmax要做的事情。线性层就是一个简单的全连接神经网络，它将解码器生成的向量映射到 logits 向量中。假设模型词汇表是10000个英语单词，它们是从训练数据集中学习的。那 logits 向量维数也是 10000，每一维对应一个单词的分数。然后，softmax 层将这些分数转化为概率（全部为正值，加起来等于 1.0），选择其中概率最大的位置的词汇作为当前时间步的输出。

本章问题

1. 尽管Transformer在许多自然语言处理任务中表现优异，但其依赖大量数据和计算资源的特性不得不让我们对于存在疑问。在某些特定的应用场景中，Transformer是否真的具备必要的有效性和可行性？
2. 在实际应用中，Transformer的非图灵完备性是否限制了其在某些复杂任务中的表现？