机器学习与深度学习面试系列十九（Transformer）

什么是注意力机制?

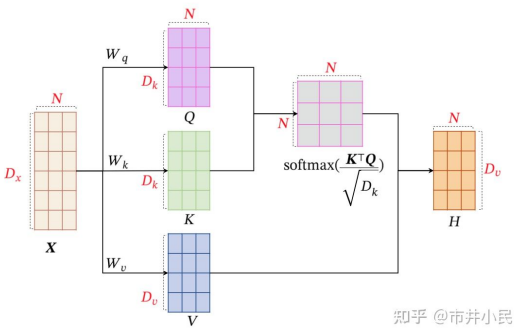
注意力一般分为两种：自下而上的无意识的注意力，称为**基于显著性的注意力**。自上而下的有意识的注意力，称为**聚焦式注意力**。一个和注意力有关的例子是鸡尾酒会效应。当一个人在吵闹的鸡尾酒会上和朋友聊天时，尽管周围噪音干扰很多，他还是可以听到朋友的谈话内容，而忽略其他人的声音(聚焦式注意力)。同时，如果未注意到的背景声中有重要的词(比如他的名字)，他会马上注意到(显著性注意力)。

对于基于显著性的注意力，其实我们并不陌生，在之前CNN中的最大池化，LSTM和GRU中的门控机制，其本质都是基于显著性的注意力（关注超过明显超过周围的或者高于一定阈值的输入）。后面我们所说的都是聚焦式注意力，主动去关心与之相关的部分输入，尽量忽略无关的部分。其实听起来很高端的想法，实现起来非常朴素：

$$Attetntion(Q,K,V)=softmax(\frac{QK^T}{\sqrt{D_k}})V=AV$$

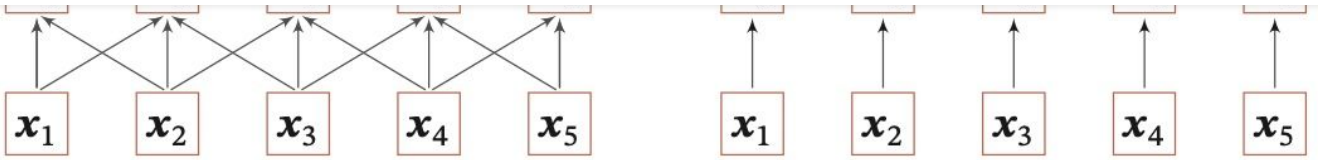
给定Q(查询)-K(键)-V(值)，使用Q和K的(放缩)点积作为系数(A)，这个系数A就是基于这一组Q和K形成的注意力分布下对V中各个分量受关注的程度，利用这个关注程度作为权重对V进行加权平均。

在实际操作中我们只有输入X，那么Q, K, V从哪来的呢？实际上我们通常使用的是自注意力机制。



对于输入X，我们先由 W_q, W_k, W_v 矩阵相乘得到Q, K, V，然后利用上述注意力机制公式计算加权平均的V。由于Q和K都是从X自身得来的，所以我们把这种方法叫做自注意力机制。

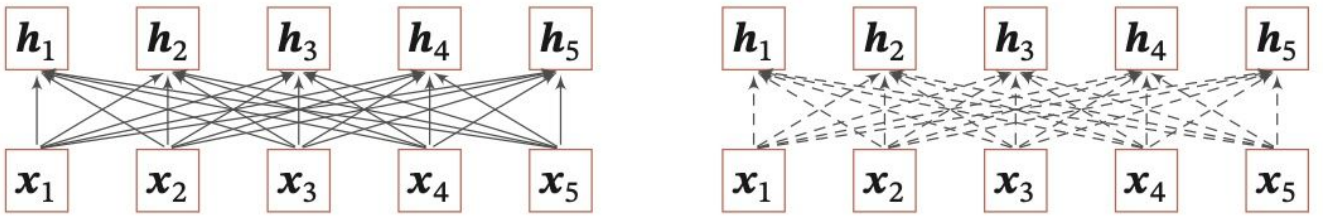
注意力机制到底解决了什么问题?



(a) 卷积网络　　(b) 双向循环网络

正如前面文章中提到的，很多时候深度学习都是一种表示学习。通过神经网络，再结合一些归纳偏置，自动的从复杂样本中学习到特征之间的关系。如上图所示，是我们所熟悉的卷积网络和循环网络。卷积网络其实假定了样本的空间上局部不变形，利用多个卷积核来提取样本空间上的局部依赖关系，这种结构只能学习到短距离的依赖关系。循环神经网络其实是假设了时间上的不变性，通过改进为LSTM或GRU，理论上可以学习到更长距离的依赖，但由于存在容量限制和梯度消失问题，实际上也只能建立短距离依赖关系。如果要想学习到长距离的依赖关系，通常需要需要堆叠更多层，如CNN中更多的卷积层可以不断扩大感受野，学习更长距离的依赖关系。

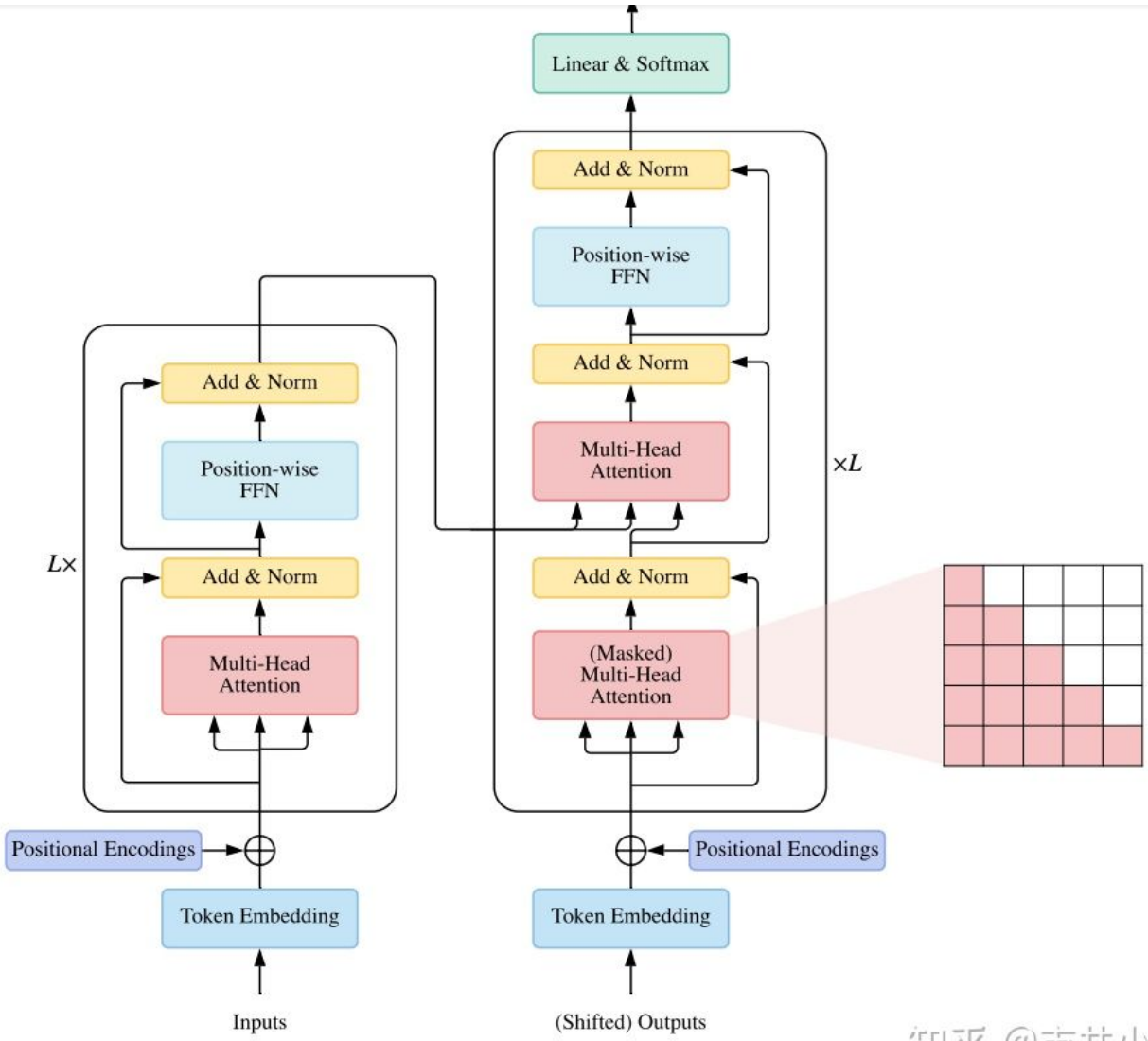
另一种解决长距离依赖关系的方法是受全连接网络的启发，其实本质上全连接网络是可以学习任意远距离的依赖关系的（想象一下，所有节点都与下一层每个节点连接，再远的距离都可以被汇总）。但苦于全连接网络无法处理变长的输入序列。而不同的输入长度，其连接权重大小也是不同的。自注意力机制正是允许"动态"地生成不同连接的权重，来解决长距离依赖问题。



(a) 全连接模型　　(b) 自注意力模型

自注意力机制可以说抛弃了CNN和RNN的归纳偏置，让模型足够通用灵活解决了远距离依赖关系的学习问题，所以Google在原论文[1]中用了个"很大口气"的标题：Attention is all you need!

Transformer的结构是怎样的?



这幅图诠释了整个Transformer结构，它分为左右两侧，分别是Encoder部分和Decoder部分。两边结构有相同的部分，如Token Embedding、FFN、残差连接(Add)、层归一化(Norm)、Postional Encodings。也有不同的部分，如注意力机制。我们分别来看看每个部分的作用。

Token Embedding：没啥可说的，将输入以高维的one-hot编码映射到低维空间，可以节约空间，也让输入语义更清楚。

Postional Encodings：RNN天然是有序的(输入样本依次输入到模型中)，而Transformer解除了时序依赖(输入样本一次性喂入模型中)，因此需要引入位置编码来编码词序信息，简单的说就是输入“I like this movie because it does not have an overhead history.” 和 “I do not like this movie because it has an overhead history." 这两个句子模型应当作出不同的响应。这一点在卷积神经网络中也是常见的操作，Uber在2018年发过一篇文章[2]， 他们发现卷积神经网络由于天生具有局部空间不变性，而导致很难完成一些简单的任务如监督回归，解决方案就是给输入图像每个像素增加位置信息，让卷积网络学习目标像素点的“绝对位置”。

$$H' = LayerNorm(SelfAttention(X) + X)$$

$$H = LayerNorm(H')$$

残差连接和层归一化可以有效的克服梯度消失问题，提高模型训练效率。

FFN：这一层就是两层全连接层，中间加一个激活层，可以写为:

$$FFN(X) = ReLU(W_1 X + b_1)W_2 + b_2$$

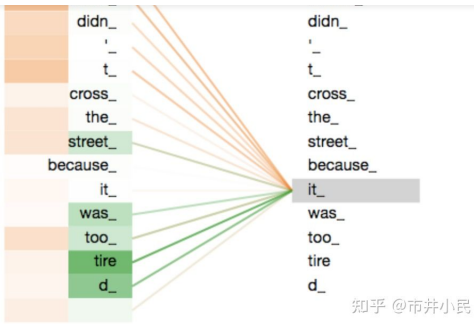
值得注意的一点是，这里全连接层不是作用在所有的输入上的，而是分别作用在每个word上。即这一层并不能学习word之间的相关依赖关系，只是对单个word的embedding做全连接。

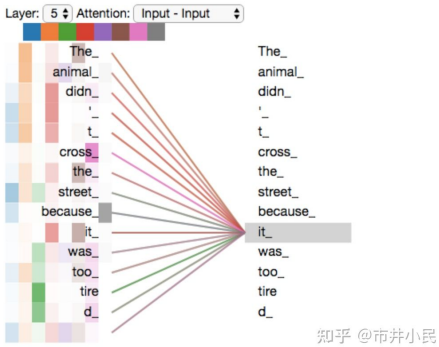
Attention: 我们注意到，在Transformer中共有三块Attention，它们互相之间都有一点区别。

- Encoder侧的Multi-Head Attention。这跟我们在上文中的自注意力机制基本完全相同，只是引入了“多头机制”。对于输入的句子 $X = (x_1, x_2, \dots x_n)$ ，分别利用 W_q, W_k, W_v 矩阵相乘得到Q, K, V，然后使用自注意力公式来计算各个 v_n 在 q_n 和K形成的注意力分布下，得到的加权平均 $\hat{v_n}$ 。$\hat{v_n}$ 可以看作是在“关注了与它相关的其他word”后得到的编码。
- Decoder侧的Masked Multi-Head Attention。由于在Decoder侧是输入的是目标语句，在位置n时，只应该注意到 $x_{1:n-1}$ ， $x_{n+1:N}$ 的word此时视作看不到，所以引入掩码来保证计算注意力的时候只使用该位置以前的信息。
- Decoder侧的Multi-Head Attention。它使用的Query使用的是Decoder的输出映射得到的，而Key和Value使用的是Encoder输出映射得到的。

Transformer为何使用多头注意力机制?

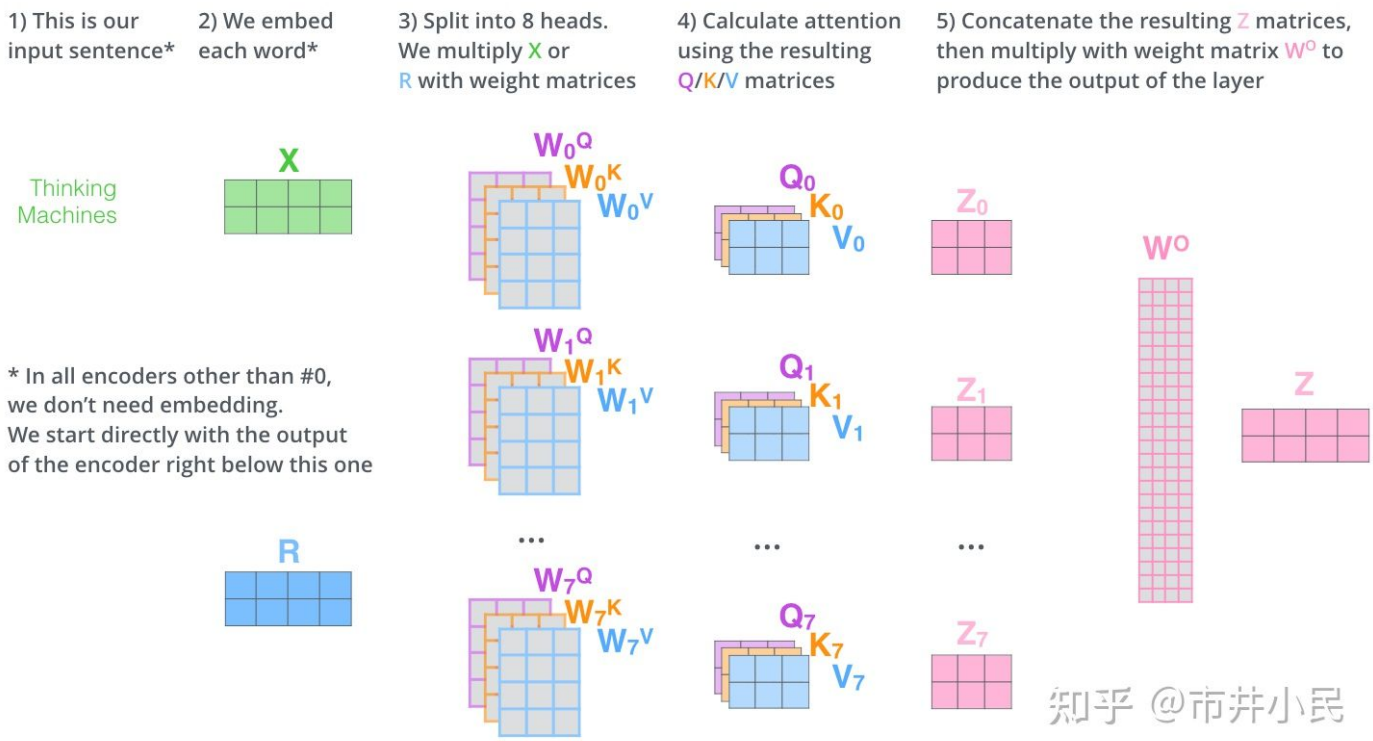
在原论文中，作者给出的解释是多头注意力模型可以从多个子空间获取不同位置之间的信息，这听起来类似于在CNN中我们使用多个卷积核分别对图片进行卷积以提取不同的特征信息。其实关于为什么要用多头注意力机制，原作者解释不清楚，业内也还在研究中。一种比较靠谱的直观解释是在这篇文章[3]中作者给出的：





当我们对“it”这个词进行编码时，一个注意力头最关注“animal”，而另一个注意力头关注“trie”——从某种意义上说，模型在对同一个语句中“it”这个词的表示进行学习时，可以关注不同方面的特征。

多头注意力机制是怎么做的?



就是：

$$\begin{aligned}\text{MultiHead}(Q,K,V) &= \text{Concat}(\text{head}_1,...,\text{head}_\text{h})W^O \\ \text{where head}_\text{i} &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)\end{aligned}$$

Transformer为什么Q和K使用不同的权重矩阵生成?

在之前很多人认为，Q和K不能用相同的权重矩阵，因为会丧失模型的灵活性和表达能力。知乎上有很多相关的讨论：transformer中为什么使用不同的K 和 Q， 为什么不能使用同一个值? 但是2020年，Google再次提出Reformer，认为K和Q是可以共享的，并且实验效果也不差。（手动捂脸）只能再次说，人工智能中很多东西都是凭大佬的直觉在做，然后通过实验验证，很多东西仍然不可解释。

为什么在进行softmax之前对attention进行scaled（除以 d_k 的平方根）？

因为在Softmax括号中，我们先做了 $\boldsymbol{QK^T}$ 这样的点积运算，如果维度很高，就容易产生较大的值。而Softmax会将几乎全部的概率分布都分配给了最大值对应的标签，这意味着Softmax进入到平坦区域，而这个区域梯度很小，无法训练。可以参考这个知乎回答[4] 。

假设Q和K的各个分量 $\boldsymbol{Q_i}$ 和 $\boldsymbol{K_i}$ 都是互相独立的随机变量，均值是0，方差是 $\boldsymbol{\sigma}$ ，那么

$$\begin{aligned}\boldsymbol{D(Q_iK_i^T)} &= \boldsymbol{E(Q_i^2 \cdot K_i^2) - [E(Q_iK_i^2)]^2} \\ &= \boldsymbol{E(Q_i^2)E(K_i^2) - [E(Q_i)E(K_i)]^2} \\ &= \boldsymbol{E(Q_i^2 - 0^2)E(K_i^2 - 0^2) - [E(Q_i)E(K_i)]^2} \\ &= \boldsymbol{E(Q_i^2 - [E(Q_i)]^2)E(K_i^2 - [E(K_i)]^2) - [E(Q_i)E(K_i)]^2} \\ &= \boldsymbol{D(Q_i)D(K_i) - [E(Q_i)E(K_i)]^2} \\ &= \boldsymbol{\sigma \times \sigma - (0 \times 0)^2} \\ &= \boldsymbol{\sigma^2}\end{aligned}$$

$$\boldsymbol{D(QK^T) = D(\sum_{i=1}^{d_k} Q_iK_i^T) = \sum_{i=1}^{d_k} D(Q_iK_i^T) = d_k\sigma^2}\ ,$$

啊），则 $D(\frac{q_i k_i}{\sqrt{d_k}}) = \frac{1}{\sqrt{d_k}^2} D(q_i k_i) = \frac{1}{\sqrt{d_k}^2} D(\sum_{j=1}^{d_k} q_j k_j)$。

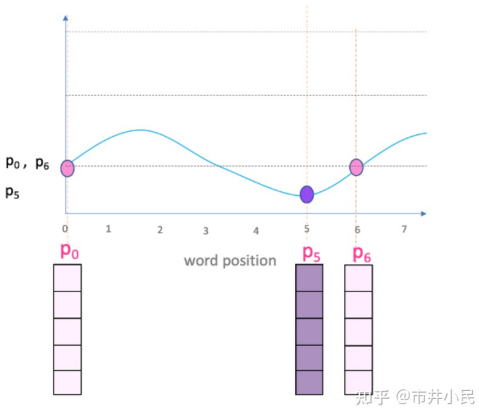
简单介绍一下Transformer的位置编码?

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

知乎上有很多解释transformer的位置编码为什么这么设计的文章，原文作者也对此进行了解释，但个人觉得最直观的解释还是stackexchange上的这个回答[5]。

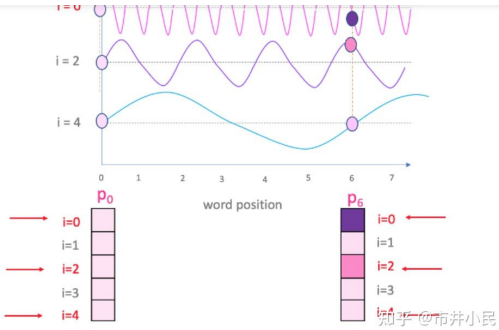
位置编码的目的就是给不同位置的word不同的位置编码，最直接的方式就是用 $[1, 2, \ldots N]$ 位置索引，但这样的方式不好，因为输入序列的长度会很长，这样编码到最后，N会非常大，以至于覆盖了Embedding本身的语义信息，所以我们希望这里的位置编码是给Embedding添加一个与word位置唯一对应的"微小"的扰动信息。那么三角函数是个不错的选择，它可以将 $[1, 2, \ldots N]$ 映射到[0, 1]区间内。

我们考虑直接使用 $sin(pos)$ 来对位置进行编码：



可以看到，由于三角函数具有周期性，不同的位置可能会被映射成同样的值，这样就失去了位置编码的意义，例如：图中pos=0和pos=6被映射成同样的值，如果这两个位置上的word一样，模型也就无法区分了。

解决思想也很简单，这个位置编码结果也是一个向量，且维度与Embedding的维度一致，这样方便直接将Embedding和位置编码直接按位相加。在位置编码向量的每一位上，我们采用不同频率的三角函数，这样就算有些位因为三角函数的周期性导致不同位置的映射结果一样，但是其他位还是不一样。也就是说我们用了一系列频率不同的三角函数来对同一个位置进行编码。



而之所以选择sin和cos两种三角函数来配合进行解析，是为了满足“相对位置”，即两个位置之间的关系可以通过他们位置编码间的仿射变换来获得[6]。 $sin(x+\alpha)$ 和 $cos(x+\alpha)$ 都不能单独由 $sin(x)$ 和 $cos(x)$ 导出。

为什么transformer块使用LayerNorm而不是BatchNorm?

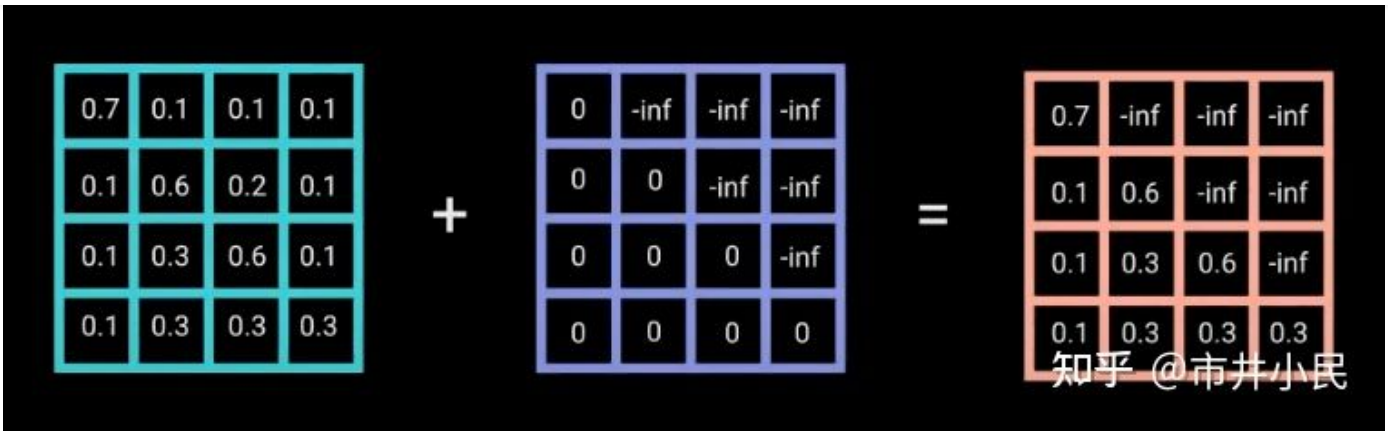
在NLP任务中，由于各个批次内各个句子长度不同是很常见的，如果我们使用批量归一化方法，那么对每个word都有单独的批量归一化参数。如果预测时的句子比训练时的句子长，模型就不知道该如何处理了。所以在自然语言处理任务中，通常使用层归一化，具体可以参考这篇论文[7]

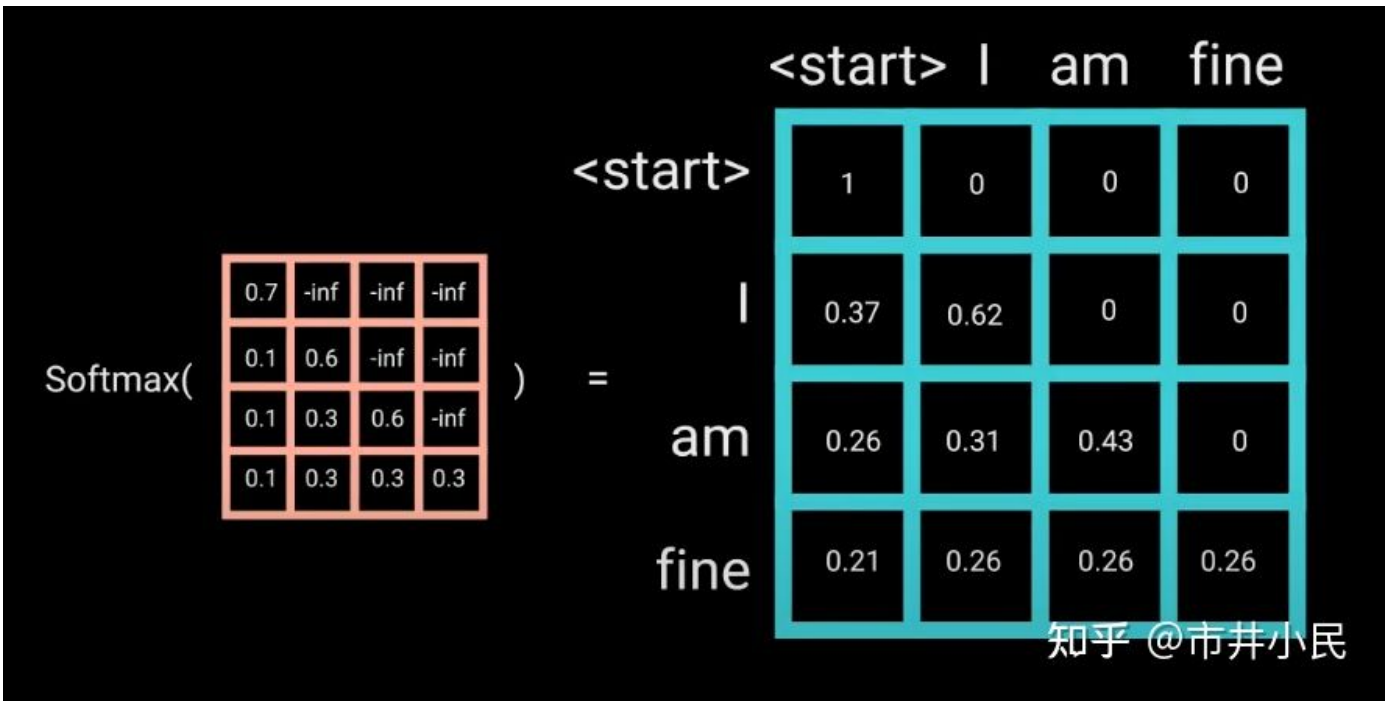
Transformer中有几处mask操作？如何做的?

Transformer中有两处mask操作：

1. Padding。对于同一批次输入句子不一样长的情况，我们会使用padding将其补其到最长句子的长度，对于补的<pad>位置，在计算注意力时是可以用mask忽略的。
2. Masked Self-attention。在Decoder侧，由于解码时，我们只应该关注当前word之前的序列，而忽略其之后的序列，所以需要用masked遮盖后面的序列，以此“欺骗”Decoder。

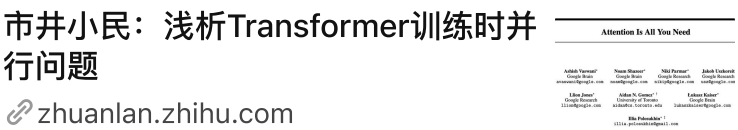
两者的实现方式是基本一样的，都是在计算 QK^T 后，做Softmax前，将Padding位置或者预测位置后续序列的attention score设置为-inf。经过Softmax后，-inf位置会变为0。[8]



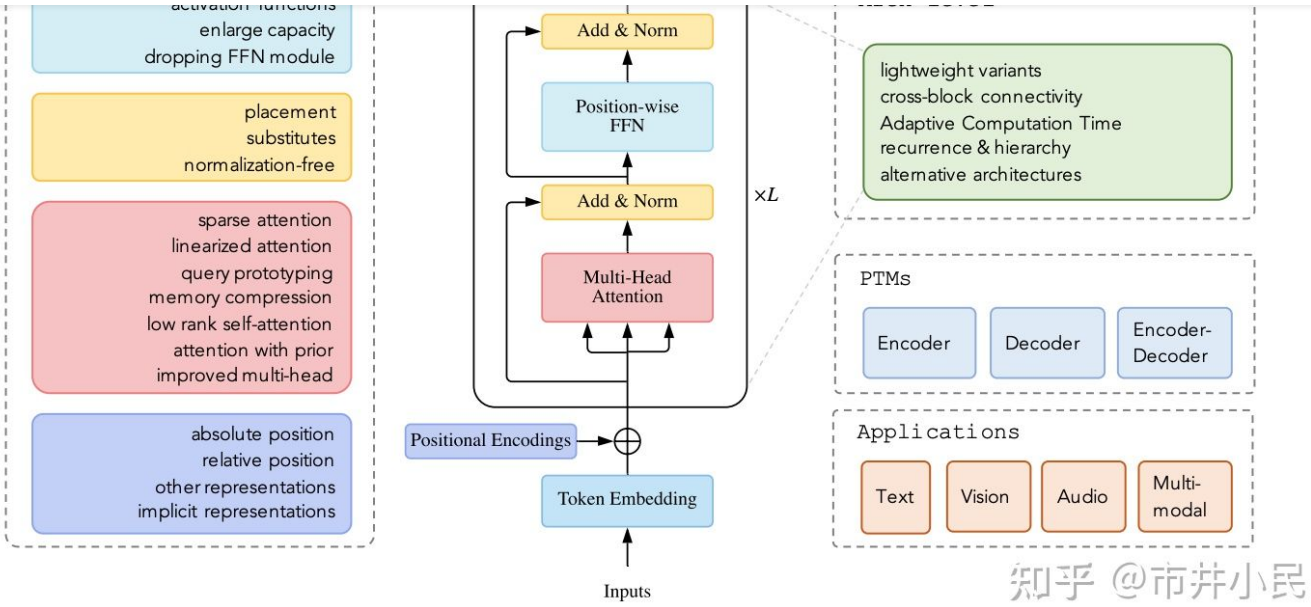


Transformer如何支持并行化?

在Encoder端Transformer可以并行处理整个序列，并得到整个输入序列经过Encoder端的输出，不像RNN只能从前到后的按序列依次执行。在Decoder侧，Transformer在训练时可以并行化处理，在预测时只能一步一步迭代处理。具体可以参考我的这篇博客：



Transformer有哪些改进模型?



自16年Google提出Transformer以来，很多人都对这个模型提出了各种改进方案，最近 @邱锡鹏 大牛老师团队的新作[9] 对这些改进做了很好的总结。这些改进大都是在架构级别的，或者是模块级别的。对于上述各个模块，Positional Encodings、Attention、FFN、LayerNorm都有各种各样的改进方案，其中最耗时最瓶颈的方案个人理解还是Attention部分，这部分的参数量和计算量实在是太大了。具体各个改进方案可以拜读邱老师的大作。

这里我们简单介绍一种Google自己家2020年在顶会ICLR上发表的新作：Reformer[10]。Reformer的优化都是在模块层面的，大多数是为了节约内存开销的，因为工程上，Transfomer的内存消耗实在是太大了。

- **Axial Positional Encodings**。这个是通过用一个坐标对(r_i, c_j)来代替之前的直接对位置向量的编码，可以大大的降低内存开销。
- **Locality Sensitive Hashing Attention**。它的基本思想就是不再计算全局的注意力机制，转向只做与之关系最近的若干个word的注意力。那么哪些word与之最接近呢，Reformer采用的是本地敏感哈希来判断。同时Reformer的Q和K使用的是同一个矩阵，这样大大的降低了参数量。
- **Reversible layers**。这一层是为了解决残差连接层数加深后，我们需要储存每一层的activations（即每一层的输入），导致内存消耗过大的问题。我们采用这种方式的话，不需要我们记录中间层的activations，而只需要我们储存最后一层的输出，从而通过模型的特定结构，反推出中间层的结果。
- **Chunking FFN layers**。将FFN分段处理，因为FFN中的输入之间互相独立，进行分段的处理可以降低空间消耗。

具体可以参考这篇英文博客[11]和这篇知乎中文博客[12]。

参考

3. ^ http://jalammar.github.io/illustrated-transformer/
4. ^ https://www.zhihu.com/question/339723385/answer/782509914
5. ^ https://datascience.stackexchange.com/questions/51065/what-is-the-positional-encoding-in-the-transformer-model
6. ^ https://kazemnejad.com/blog/transformer_architecture_positional_encoding/
7. ^ https://arxiv.org/pdf/1607.06450.pdf
8. ^ https://towardsdatascience.com/illustrated-guide-to-transformers-step-by-step-explanation-f74876522bc0
9. ^ https://arxiv.org/pdf/2106.04554.pdf
10. ^ https://openreview.net/forum?id=rkgNKkHtvB
11. ^ https://huggingface.co/blog/reformer
12. ^ https://zhuanlan.zhihu.com/p/357628257