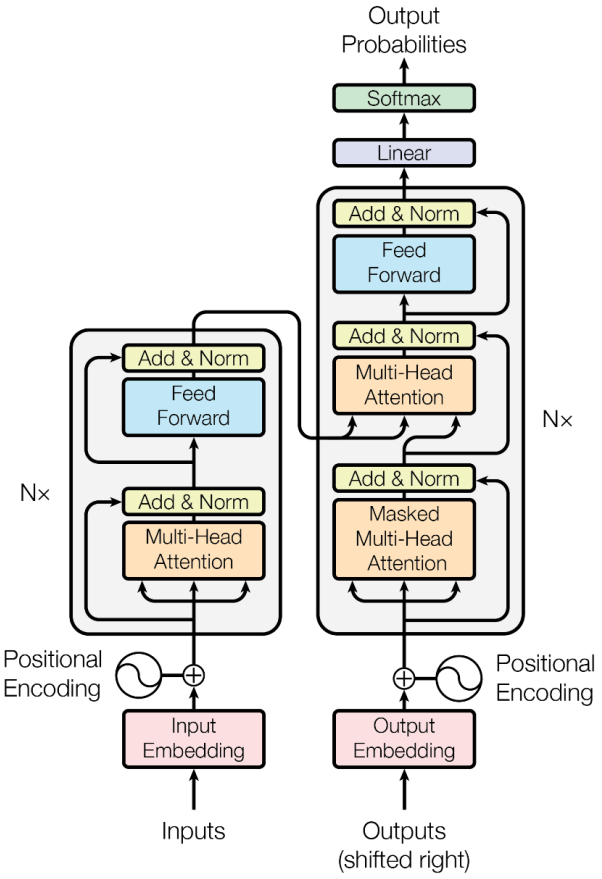
Transformer 架构细节

1.Transformer 各个模块的作用

（1）Encoder 模块

- 经典的 Transformer 架构中的 Encoder 模块包含 6 个 Encoder Block.
- 每个 Encoder Block 包含两个子模块, 分别是多头自注意力层, 和前馈全连接层.



- 多头自注意力层采用的是一种 Scaled Dot-Product Attention 的计算方式, 实验结果表 明, Multi-head 可以在更细致的层面上提取不同 head 的特征, 比单一 head 提取特征的 效果更佳.
- 前馈全连接层是由两个全连接层组成, 线性变换中间增添一个 Relu 激活函数, 具体的 维度采用 4 倍关系, 即多头自注意力的 d_model=512, 则层内的变换维度 d_ff=2048.

（2）Decoder 模块

- 经典的 Transformer 架构中的 Decoder 模块包含 6 个 Decoder Block.
- 每个 Decoder Block 包含 3 个子模块, 分别是多头自注意力层, Encoder-Decoder Attention 层, 和前馈全连接层.

- 多头自注意力层采用和 Encoder 模块一样的 Scaled Dot-Product Attention 的计算方 式, 最大的 区别在于**需要添加 look-ahead-mask,** 即遮掩"未来的信息".
- Encoder-Decoder Attention 层和上一层多头自注意力层最主要的区别在于 Q != K = V, 矩阵 Q 来源于上一层 Decoder Block 的输出, 同时 K, V 来源于 Encoder 端的输出.
- 前馈全连接层和 Encoder 中完全一样.

（3）Add & Norm 模块

- Add & Norm 模块接在每一个 Encoder Block 和 Decoder Block 中的每一个子层的后面.
- 对于每一个 Encoder Block, 里面的两个子层后面都有 Add & Norm.
- 对于每一个 Decoder Block, 里面的三个子层后面都有 Add & Norm.
- Add 表示残差连接, 作用是为了将信息无损耗的传递的更深, 来增强模型的拟合能力.
- Norm 表示 LayerNorm, 层级别的数值标准化操作, 作用是防止参数过大过小导致的学习过程异常 , 模型收敛特别慢的问题.

（4）位置编码器 Positional Encoding

- Transformer 中采用三角函数来计算位置编码.
- 因为三角函数是周期性函数, 不受序列长度的限制, 而且这种计算方式可以对序列中不同位置的编码的重要程度同等看待

2.Decoder 端训练和预测的输入

1. 在 Transformer 结构中的 Decoder 模块的输入, 区分于不同的 Block, 最底层的 Block 输入有其特殊的地方。第二层到第六层的输入一致, 都是上一层的输出和 Encoder 的输出。
2. 最底层的 Block 在**训练阶段**, 每一个 time step 的输入是上一个 time step 的输入加上真实标 签序列向后移一位. 具体来看, 就是每一个 time step 的输入序列会越来越长, 不断的将之前的 输入融合进来.

```
纯文本
假设现在的真实标签序列等于"How are you?",
当time step=1时, 输入张量为一个特殊的token, 比如"SOS";
当time step=2时, 输入张量为"SOS How";
当time step=3时, 输入张量为"SOS How are";
以此类推...
```

3. 最底层的 Block 在**训练阶段**, 真实的代码实现中, 采用的是 MASK 机制来模拟输入序列不断添 加的过程.
4. 最底层的 Block 在**预测阶段**, 每一个 time step 的输入是从 time step=0 开始, 一直到上一个 time step 的预测值的累积拼接张量. 具体来看, 也是随着每一个 time step 的输入序列会越来越长. 相比于训练阶段最大的不同是这里不断拼接进来的 token 是每一个 time step 的预测值, 而不是训练阶段每一个 time step 取得的 groud truth 值

```
纯文本
当time step=1时, 输入的input_tensor="SOS", 预测出来的输出值是output_tensor="What";
当time step=2时, 输入的input_tensor="SOS What", 预测出来的输出值是output_tensor="is";
当time step=3时, 输入的input_tensor="SOS What is", 预测出来的输出值是output_tensor="the";
当time step=4时, 输入的input_tensor="SOS What is the", 预测出来的输出值是output_tensor="matter";
当time step=5时, 输入的input_tensor="SOS What is the matter", 预测出来的输出值是output_tensor="?";
当time step=6时, 输入的input_tensor="SOS What is the matter ?", 预测出来的输出值是output_tensor="EOS", 代表句子的结束符, 说明解码结束, 预测结束.
```

3.Self-attention

> Transformer 中一直强调的 self-attention 是什么? 为什么能 发挥如此大的作用? 计算的时候如果不使用三元组(Q, K, V), 而 仅仅使用(Q, V)或者(K, V)或者(V)行不行?

（1）self-attention 的机制和原理

self-attention 是一种通过自身和自身进行关联的 attention 机制, 从而得到更好的 representation 来表达自身.

self-attention 是 attention 机制的一种特殊情况: 在 self-attention 中, Q=K=V, **序列中的每个单词(token)都和该序列中的其他所有单词 (token)进行 attention 规则的计算.**

attention 机制计算的特点在于, 可以**直接跨越一句话中不同距离的 token, 可以远距离的学习到序列的知识依赖和语序结构.**

- 从上图中可以看到, self-attention 可以远距离的捕捉到语义层面的特征(it 的指代对象是 animal).
- 应用传统的 RNN, LSTM, 在获取长距离语义特征和结构特征的时候, **需要按照序列顺序依次 计算, 距离越远的联系信息的损耗越大, 有效提取和捕获的可能性越小.**
- 但是应用 self-attention 时, 计算过程中会直接将句子中任意两个 token 的联系通过一个计算 步骤直接联系起来,

（2）关于 self-attention 为什么要使用(Q, K, V)三元组而不是其他形式

首先一条就是从分析的角度看, 查询 Query 是一条独立的序列信息, 通过关键词 Key 的提示作用, 得到最终语义的真实值 Value 表达, 数学意义更充分, 完备.

这里不使用(K, V)或者(V)没有什么必须的理由, 也没有相关的论文来严格阐述比较试验的结果差异, 所以可以作为开放性问题未来去探索, 只要明确在经典 self-attention 实现中用的是三元组就好

4.Self-attention 归一化和放缩

（1）self-attention 中的归一化概述

训练上的意义: 随着词嵌入维度 d_k 的增大, q * k 点积后的结果也会增大, 在训练时会将 softmax 函数推入梯度非常小的区域, 可能出现梯度消失的现象, 造成模型收敛困难.

数学上的意义: 假设 q 和 k 的统计变量是满足标准正态分布的独立随机变量, 意味着 q 和 k 满足均 值为 0, 方差为 1。 **那么 q 和 k 的点积结果就是均值为 0, 方差为 d_k , 为了抵消这种方差被放大 d_k 倍的影响, 在计算中主动将点积缩放 $\frac{1}{\sqrt{(d_k)}}$** , 这样点积后的结果依然满足均值为 0, 方差为 1。

（2）维度与点积大小的关系

针对为什么维度会影响点积的大小, 原始论文中有这样的一点解释如下:

分两步对其进行一个推导, 首先就是假设向量 q 和 k 的各个分量是相互独立的随机变量, $X = q_i$, $Y = k_i$, X 和 Y 各自有 d_k 个分量, 也就是向量的维度等于 d_k, 有 $E(X) = E(Y) = 0$, 以及 $D(X) = D(Y) = 1$.

根据上面的公式, 可以很轻松的得出 q*k 的均值为 $E(qk) = 0$, $D(qk) = d_k$ 。所以方差越大, **对应的 qk 的点积就越大, 这样 softmax 的输出分布就会更偏向最大值所在的分量**。一个技巧就是将点积除以 $\sqrt{d_k}$ 将方差在数学上重新"拉回 1", 如下所示

$$D(\frac{q \cdot k}{\sqrt{d_k}}) = \frac{d_k}{(\sqrt{d_k})^2} = 1$$

最终的结论: **通过数学上的技巧将方差控制在 1, 也就有效的控制了点积结果的发散, 也就控制了对应的梯度消失的问题**!

5.Multi-head Attention

（1）采用 Multi-head Attention 的原因

1. 原始论文中提到进行 Multi-head Attention 的原因是将模型分为多个头, **可以形成多个子空间间, 让模型去关注不同方面的信息**, 最后再将各个方面的信息综合起来得到更好的效果.
2. 多个头进行 attention 计算最后再综合起来, 类似于 CNN 中采用多个卷积核的作用, 不同的卷 积核提取不同的特征, **关注不同的部分, 最后再进行融合**.
3. 直观上讲, **多头注意力有助于神经网络捕捉到更丰富的特征信息**.

（2）Multi-head Attention 的计算方式

1. Multi-head Attention 和单一 head 的 Attention 唯一的区别就在于, **其对特征张量的最后一个维度进行了分割, 一般是对词嵌入的 embedding_dim=512 进行切割成 head=8,** 这样每一个 head 的嵌入维度就是 512/8=64, 后续的 Attention 计算公式完全一致, 只不过是在 64 这个维度上进行一系列的矩阵运算而已.
2. 在 head=8 个头上分别进行注意力规则的运算后, 简单采用拼接 **concat** 的方式对结果张量进 行融合就得到了 Multi-head Attention 的计算结果.

6.Transformer 和 RNN

（1）Transformer 的并行计算

对于 Transformer 比传统序列模型 RNN/LSTM 具备优势的第一大原因就是强大的并行计算能力.

对于 RNN 来说, 任意时刻 `t` 的输入是时刻 t 的输入 `x(t)` 和上一时刻的隐藏层输出 `h(t-1)`, 经过运算后得到当前时刻隐藏层的输出 `h(t)`, 这个 `h(t)` 也即将作为下一时刻 `t+1` 的输入的一部分. 这个计算过程是 RNN 的本质特征, RNN 的历史信息是需要通过这个时间步一步一步向后传递的. 而**这就意味着 RNN 序列后面的信息只能等到前面的计算结束后, 将历史信息通过 hidden state 传递给后面才能开始计算, 形成链式的序列依赖关系, 无法实现**并行.

对于 Transformer 结构来说, 在 self-attention 层, 无论序列的长度是多少, 都可以一次性计算所有单词之间的注意力关系, 这个 attention 的计算是同步的, 可以实现并行.

（2）Transformer 的特征抽取能力

对于 Transformer 比传统序列模型 RNN/LSTM 具备优势的第二大原因就是强大的特征抽取能力 。

Transformer 因为采用了 Multi-head Attention 结构和计算机制, 拥有比 RNN/LSTM 更强大的特征抽取能力, 这里并不仅仅由理论分析得来, 而是大量的试验数据和对比结果, 清楚的展示了 Transformer 的特征抽取能力远远胜于 RNN/LSTM.

注意: 不是越先进的模型就越无敌, 在很多具体的应用中 RNN/LSTM 依然大有用武之地, 要具体问题具体分析

7.Transformer 代替 seq2seq?

（1）seq2seq 的两大缺陷

1. seq2seq 架构的第一大缺陷是将 Encoder 端的所有信息**压缩成一个固定长度的语义向量中，用这个固定的向量来代表编码器端的全部信息. 这样既会造成信息的损耗**, 也无法让 Decoder 端在解码的时候去用注意力聚焦哪些是更重要的信息.
2. seq2seq 架构的第二大缺陷是**无法并行**, 本质上和 RNN/LSTM 无法并行的原因一样.

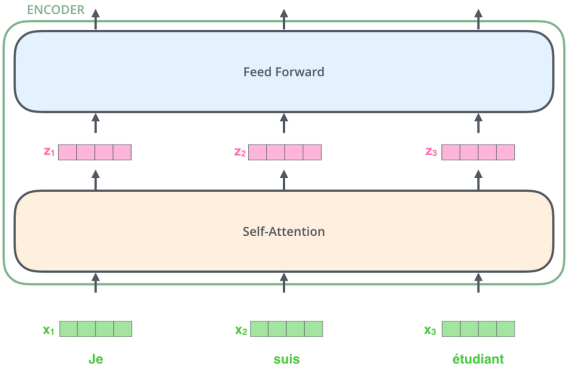
（2）Transformer 的改进

Transformer 架构同时解决了 seq2seq 的两大缺陷, 既可以并行计算, 又应用 Multi-head Attention 机制来解决 Encoder 固定编码的问题, 让 Decoder 在解码的每一步可以通过注意力去 关注编码器输出中最重要的那些部分.

8.Transformer 并行化

（1）Encoder 并行化

1. 上图最底层绿色的部分, 整个序列所有的 token 可以并行的进行 Embedding 操作, 这一层的处理是没有依赖关系的.

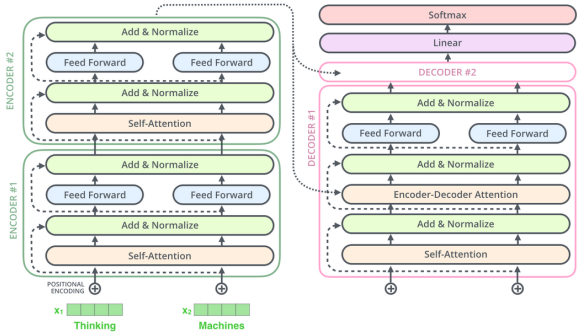


2. 上图第二层土黄色的部分, 也就是 Transformer 中最重要的 self-attention 部分, 这里对于任意一个单词比如 x1, 要计算 x1 对于其他所有 token 的注意力分布, 得到 z1. 这个过程是具有依赖性的, 必须等到序列中所有的单词完成 Embedding 才可以进行。因此这一步是不能并行处理的。 但是从另一个角度看, 我们真实计算注意力分布的时候, 采用的都是矩阵运算, 也就是可以一次性的计算出所有 token

的注意力张量, 从这个角度看也算是实现了并行, 只是矩阵运算的"并行"和词嵌入的"并行"概念上不同而已.

3. 上图第三层蓝色的部分, 也就是前馈全连接层, 对于不同的向量 z 之间也是没有依赖关系的, 所以这一层是可以实现并行化处理的. 也就是所有的向量 z 输入 Feed Forward 网络的计算可以同步进行, 互不干扰

（2）Decoder 的并行化



1. Decoder 模块在训练阶段采用了并行化处理。 其中 Self-Attention 和 Encoder-Decoder Attention 两个子层的并行化也是在进行矩阵乘法, 和 Encoder 的理解是一致的.

在进行 Embedding 和 Feed Forward 的处理时, 因为各个 token 之间没有依赖关系, 所以也是可以完全并行化处理的, 这里和 Encoder 的理解也是一致的.

2. Decoder 模块在预测阶段基本上不认为采用了并行化处理. 因为第一个 time step 的输入只是一个"SOS", 后续每一个 time step 的输入也只是依次添加之前所有的预测 token.

3. **注意:** 最重要的区别是训练阶段目标文本如果有 20 个 token, 在训练过程中是一次性的输入给 Decoder 端, 可以做到一些子层的并行化处理. 但是在预测阶段, 如果预测的结果语句总共有 20 个 token, 则需要重复处理 20 次循环的过程, 每次的输入添加进去一个 token, 每次的输入序列比上一次多一个 token, 所以不认为是并行处理.

（3）总结

Transformer 架构中 Encoder 模块的并行化机制

- **Encoder 模块在训练阶段和测试阶段都可以实现完全相同的并行化.**
- Encoder 模块在 Embedding 层, Feed Forward 层, Add & Norm 层都是可以并行化的.
- Encoder 模块在 self-attention 层, 因为各个 token 之间存在依赖关系, 无法独立计算, 不是真正意义上的并行化.

- Encoder 模块在 self-attention 层, 因为采用了矩阵运算的实现方式, 可以一次性的完成所有注意力张量的计算, 也是另一种"并行化"的体现.

Transformer 架构中 Decoder 模块的并行化机制

- **Decoder 模块在训练阶段可以实现并行化**.
- Decoder 模块在训练阶段的 Embedding 层, Feed Forward 层, Add & Norm 层都是可以并行化的.
- Decoder 模块在 self-attention 层, 以及 Encoder-Decoder Attention 层, 因为各个 token 之间存在依赖关系, 无法独立计算, 不是真正意义上的并行化.
- Decoder 模块在 self-attention 层, 以及 Encoder-Decoder Attention 层, 因为采用了矩阵运算的实现方式, 可以一次性的完成所有注意力张量的计算, 也是另一种"并行化"的体现.
- **Decoder 模块在预测计算不能并行化处理.**