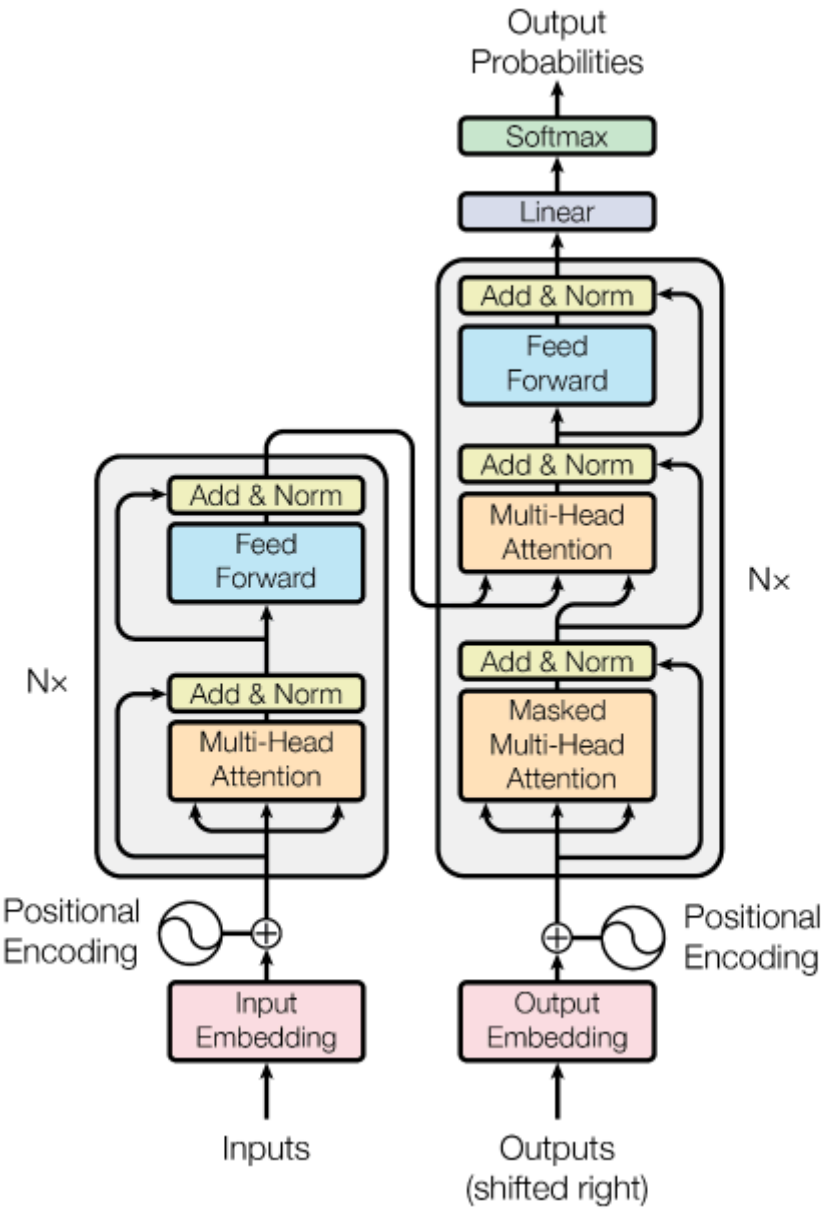
Transformer架构细节



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**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，比单个head提取特征的效果更佳。
  - 前馈全连接层由两个全连接层组成，线性变换中间增加一个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`。

> **Note**
>
> 虽然更大的`d_ff`可以带来更好的模型表现，但也会显著增加计算量和内存消耗。因此，在实际工程实现中，通常会采用一个合理的放大倍数（如 4 倍），既保证了模型的表达能力，又不会造成太大的资源负担。

(2) Decoder模块

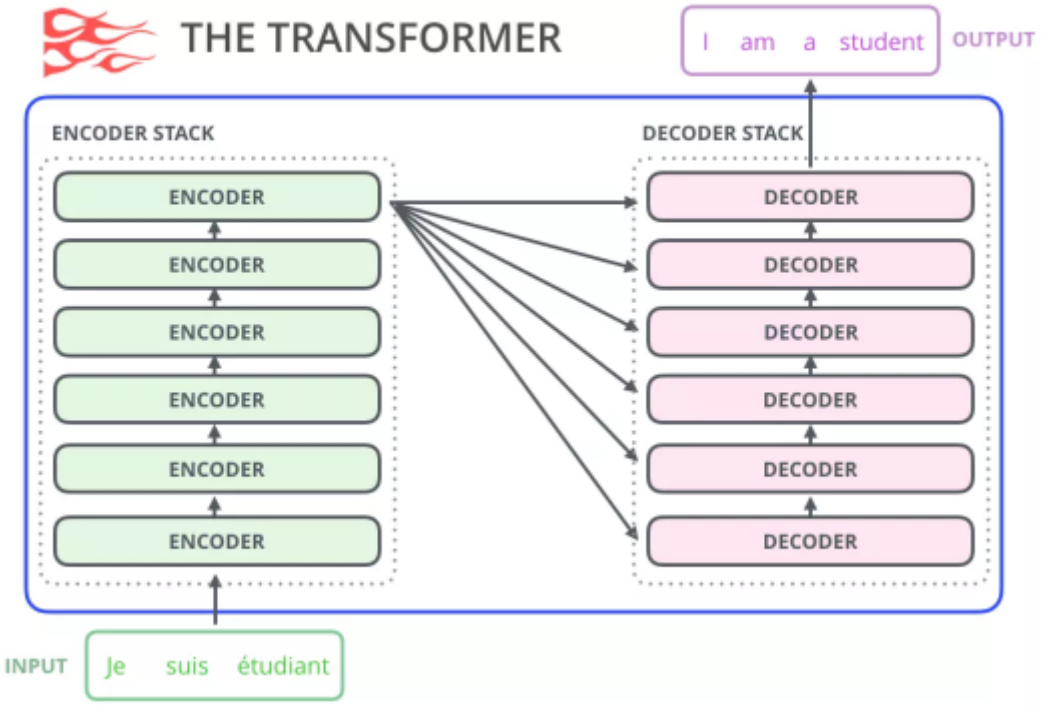
- 经典的Transformer架构中的Decoder模块包含6个Decoder Block。
- 每个Decoder Block包含3个子模块，分别是**多头自注意力层、Encoder-Decoder Attention层和前馈全连接层。**
  - 多头自注意力层采用和Encoder模块一样的**Scaled Dot-Product Attention**的计算方式，最大的区别在于**需要添加look-ahead-mask**，即遮掩"未来的信息"。
  - Encoder-Decoder Attention层和上一层多头自注意力层最主要的区别在于`Q ≠ K = V`。矩阵`Q`来源于上一层Decoder Block的输出，而`K`和`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地传递到更深的层次，来增强模型的拟合能力。
- **Norm**表示LayerNorm，是层级别的数值标准化操作，作用是防止参数过大或过小导致的学习过程异常，使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

Transformer 的 Decoder 在训练时使用 **真实标签作为输入（ 上一时刻的真实标签）** （Teacher Forcing），而在推理时使用 **上一步的预测结果作为输入** （Auto-regressive）。这是为了在训练时加快收敛，在推理时模拟真实场景，但也带来了训练与推理之间输入分布的差异，需要特别注意。

阶段	输入机制	目标	特点
训练	使用真实标签作为下一个输入（左移一位）	加快训练速度、提高模型稳定性	Teacher Forcing
推理	使用上一步预测结果作为下一步输入	模拟真实场景下的解码过程	自回归生成

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.

   ```
   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。**

   ```
   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.
   ```

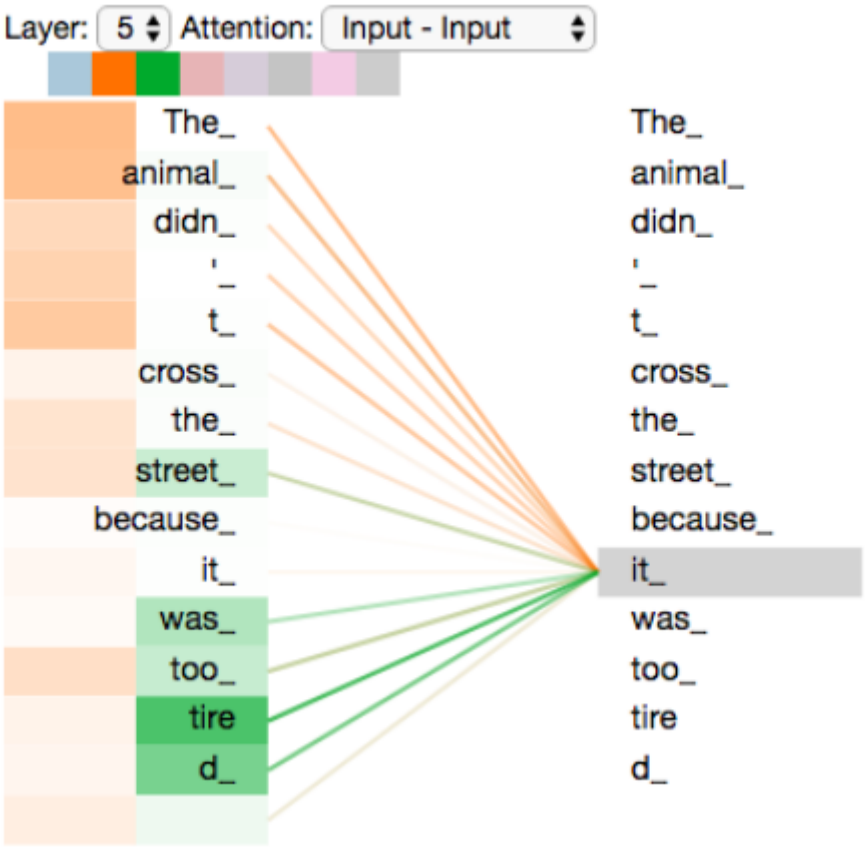
3.Self-attention

> 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 是一种通过自身与自身进行关联的 Attention 机制，从而获得更好的表示（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**，即查询（Query）、键（Key）和值（Value）都来自于同一个输入序列。 在这种机制下，**序列中的每个单词（token）都会与该序列中的其他所有单词进行 Attention 规则的计算。**

Attention 机制的一个显著特点是：它可以**直接跨越一句话中不同距离的 token，从而远距离地学习到序列中的语义依赖和语序结构**



从图示中可以看到，Self-Attention 可以远距离地捕捉到语义层面的信息，例如代词 "it" 的指代对象是 "animal"。

相比之下，传统的 RNN 和 LSTM 在获取长距离语义信息和结构信息时，需要按照序列顺序依次计算。随着距离的增加，信息传递过程中的损耗会越来越大，导致模型难以有效提取和捕获远距离的依赖关系。**而使用 Self-Attention 时，计算过程中会将句子中任意两个 token 的联系通过一个计算步骤直接建立起来，因此能够更高效地建模全局依赖关系。**

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

从设计和分析的角度来看，Self-Attention 使用 **Query、Key 和 Value** 这三个不同的向量具有明确的意义：

- **Query（查询）**：代表当前 token 对其他 token 的关注程度；
- **Key（键）**：用于与其他 Query 进行匹配，表示每个 token 被关注的可能性；
- **Value（值）**：真正携带语义信息的部分，在注意力权重的作用下被加权聚合。

这种结构使得模型能够更好地建模“我需要什么信息”（Query）、“谁提供了这些信息”（Key），以及“这些信息具体是什么”（Value）之间的关系，从而在数学上具有更强的表达能力和完备性。相比之下，仅使用 `(K, V)` 或者 `(V)` 的形式虽然在理论上也可以构建某种注意力机制，但缺乏对查询能力的独立建模，可能会限制模型捕捉复杂依赖关系的能力。

目前，并没有严格的理论推导或论文明确指出为何必须使用三元组而非其他组合形式，也缺乏系统性的对比实验来验证不同组合的效果差异。因此，**QKV 三元组的设计更多是经验性的选择**。可以将这个问题视为一个开放性的研究方向，未来仍有探索空间。现阶段只需明确：在经典的 Self-Attention 实现中（如 Transformer 中），使用的是 Query、Key、Value 三元组的形式。

4. Self-Attention 归一化和放缩

（1）Self-Attention 中的归一化概述

在训练过程中，随着词嵌入维度 d_k 的增大，Query 和 Key 的点积结果也会随之变大。这会带来一个训练上的问题： 当这些点积结果输入到 `softmax` 函数中时，会将输入值推到梯度非常小的区域（例如接近饱和区），从而可能导致**梯度消失**，影响模型的收敛效果。

从数学角度分析这一现象：假设 Query 和 Key 的元素是独立同分布的标准正态随机变量，即满足均值为 0、方差为 1 的分布。 那么它们的点积结果将是一个均值为 0、方差为 d_k 的分布。

为了抵消这种随维度增长而带来的方差放大效应，在计算注意力得分时引入了一个**缩放因子** $\frac{1}{\sqrt{d_k}}$。这样可以使得缩放后的点积结果仍然保持均值为 0、方差为 1 的标准正态分布，从而避免 softmax 输入过大带来的数值不稳定性。

（2）Softmax 的梯度变化

我们通过以下三个步骤来解释 softmax 梯度的变化机制：

第一步：Softmax 输入分布如何影响输出

对于一个输入向量 x，Softmax 函数将其映射为一个概率分布：

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

在这个过程中，Softmax 会先用自然指数函数 e^x 放大输入之间的差异，然后再进行归一化。如果输入的数值整体偏大，会导致某些输出的概率值趋近于 1，而其他则趋近于 0，形成“one-hot-like”分布。这会使梯度变得非常小，影响参数更新。

举个例子： 设输入向量为 $x = [a, a, 2a]$，观察不同 a 值下第 3 个位置的 Softmax 输出 y_3 变化情况：

a 值	y_3 值
1	0.5761
10	0.9999
100	1.0

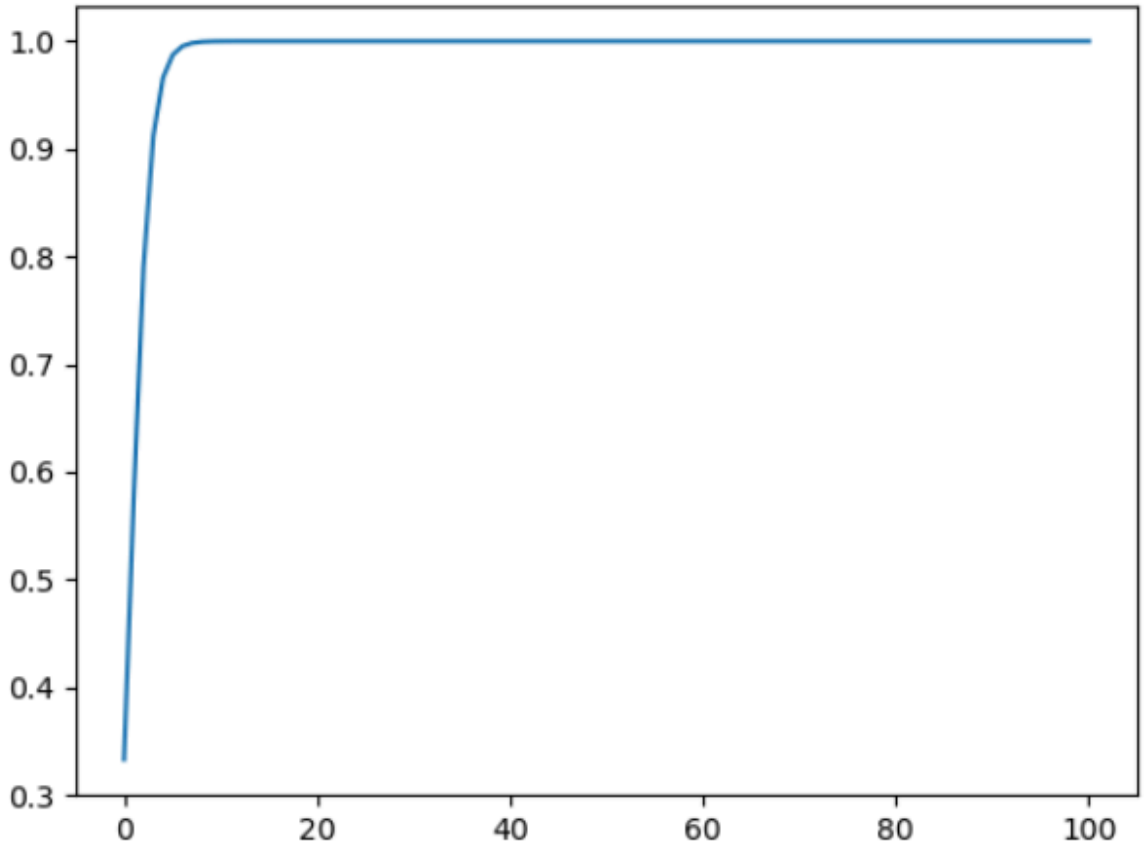
可以看出，随着 a 增大，y_3 趋近于 1，Softmax 输出趋于极端。我们可以用 Python 绘制出 y_3 随 a 变化的曲线图：

```
from math import exp
import numpy as np
import matplotlib.pyplot as plt
```

```
def f(x):
    return exp(2 * x) / (exp(x) + exp(x) + exp(2 * x))

x_values = np.linspace(0, 100, 100)
y_values = [f(x) for x in x_values]

plt.plot(x_values, y_values)
plt.title("Softmax Output y₃ vs Input Scaling Factor a")
plt.xlabel("a")
plt.ylabel("y₃")
plt.grid()
plt.show()
```



从上图可以很清楚地看到，输入元素的数量级对 Softmax 最终的输出分布影响非常显著。

结论：当输入元素的数量级较大时，Softmax 函数几乎将全部的概率质量都分配给了最大值对应的类别标签。

第二步：Softmax 函数在反向传播中的梯度求导

首先，定义神经网络的输入输出，设输入向量为：

$$X = [x_1, x_2, \ldots, x_n]$$

输出经过 Softmax 激活后为：

$$Y = \text{softmax}(X) = [y_1, y_2, \ldots, y_n]$$

其中：

$$y_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Softmax 的作用是将输入 X 映射成一个概率分布，满足：

$$\sum_{i=1}^{n} y_i = 1, \quad y_i > 0$$

反向传播过程中，我们需要计算损失函数对输入 X 的偏导数，即：$\frac{\partial y_i}{\partial x_j}$。我们分两种情况讨论：

（1）当 $i = j$ 时：

$$\frac{\partial y_i}{\partial x_i} = \frac{\partial}{\partial x_i}\left(\frac{e^{x_i}}{\sum_{k=1}^{n} e^{x_k}}\right) = \frac{(e^{x_i})' \cdot \sum_{k=1}^{n} e^{x_k} - e^{x_i} \cdot (\sum_{k=1}^{n} e^{x_k})'}{(\sum_{k=1}^{n} e^{x_k})^2}$$

由于 $(e^{x_i})' = e^{x_i}$，且 $(\sum_{k=1}^{n} e^{x_k})' = e^{x_i}$，代入得：

$$= \frac{e^{x_i} \cdot \sum_{k=1}^{n} e^{x_k} - e^{x_i} \cdot e^{x_i}}{(\sum_{k=1}^{n} e^{x_k})^2} = \frac{e^{x_i}}{\sum_{k=1}^{n} e^{x_k}} - \left(\frac{e^{x_i}}{\sum_{k=1}^{n} e^{x_k}}\right)^2 = y_i - y_i^2 = y_i(1 - y_i)$$

（2）当 $i \neq j$ 时：

$$\frac{\partial y_i}{\partial x_j} = \frac{\partial}{\partial x_j}\left(\frac{e^{x_i}}{\sum_{k=1}^{n} e^{x_k}}\right) = \frac{0 \cdot \sum_{k=1}^{n} e^{x_k} - e^{x_i} \cdot e^{x_j}}{(\sum_{k=1}^{n} e^{x_k})^2} = -\frac{e^{x_i} \cdot e^{x_j}}{(\sum_{k=1}^{n} e^{x_k})^2} = -y_i \cdot y_j$$

经过对两种情况分别的求导计算, 可以得出最终的结论如下：**Softmax 的导数矩阵等于它的输出组成的对角矩阵减去它的外积矩阵**综上所述，Softmax 的 Jacobian 矩阵可表示为：

$$\frac{\partial y_i}{\partial x_j} = \begin{cases} y_i - y_i \cdot y_i, & i = j \\ 0 - y_i \cdot y_i, & i \neq j \end{cases}$$

所以：

$$\frac{\partial Y}{\partial X} = \mathrm{diag}(Y) - Y^\top \cdot Y$$

其中，Y 是形状为 $(1, n)$ 的行向量。

第三步: softmax函数出现梯度消失现象的原因

根据第二步中softmax函数的求导结果, 可以将最终的结果以矩阵形式展开如下:

$$\frac{\partial g(X)}{\partial X} \approx \begin{bmatrix} \hat{y}_1 & 0 & \cdots & 0 \\ 0 & \hat{y}_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \hat{y}_d \end{bmatrix} - \begin{bmatrix} \hat{y}_1^2 & \hat{y}_1\hat{y}_2 & \cdots & \hat{y}_1\hat{y}_d \\ \hat{y}_2\hat{y}_1 & \hat{y}_2^2 & \cdots & \hat{y}_2\hat{y}_d \\ \vdots & \vdots & \ddots & \vdots \\ \hat{y}_d\hat{y}_1 & \hat{y}_d\hat{y}_2 & \cdots & \hat{y}_d^2 \end{bmatrix}$$

根据第一步中的讨论结果, 当输入x的分量值较大时, softmax函数会将大部分概率分配给最大的元素, 假设最大元素是x1, 那么softmax的输出分布将产生一个接近one-hot的结果张量y_ = [1, 0, 0,..., 0], 此时结果矩阵变为:

$$\frac{\partial g(X)}{\partial X} \approx \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix} - \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix} = 0$$

结论：综上可以得出, **所有的梯度都消失为0(接近于0), 参数几乎无法更新, 模型收敛困难。**

（3）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:

> To illustrate why the dot products get large, assume that the components of q and k are independent random variables with mean 0 and variance 1. Then their dot product, $q \cdot k = q_1 k_1 + q_2 k_2 + \cdots + q_{d_k} k_{d_k}$, has mean 0 and variance d_k.

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 $D(X) = D(Y) = 1$。

可以得到:

$$E(XY) = E(X)E(Y) = 0 \times 0 = 0$$

同理, 对于 $D(XY)$ 推导如下:

$$\begin{aligned} D(XY) &= E(X^2 \cdot Y^2) - [E(XY)]^2 \\ &= E(X^2)E(Y^2) - [E(X)E(Y)]^2 \\ &= E(X^2 - 0^2)E(Y^2 - 0^2) - [E(X)E(Y)]^2 \\ &= E(X^2 - [E(X)]^2)E(Y^2 - [E(Y)]^2) - [E(X)E(Y)]^2 \\ &= D(X)D(Y) - [E(X)E(Y)]^2 \\ &= 1 \times 1 - (0 \times 0)^2 = 1 \end{aligned}$$

根据期望和方差的性质, 对于互相独立的变量满足下式:

如果你有一组互不干扰（独立）的随机数 $Z_1, Z_2, \ldots, Z_n$，那么它们加在一起的平均值（期望），就是它们各自平均值的总和；它们加在一起的波动范围（方差），也是各自波动范围的总和。

$$E\left(\sum_i Z_i\right) = \sum_i E(Z_i), \quad D\left(\sum_i Z_i\right) = \sum_i D(Z_i)$$

根据上面的公式, 可以很轻松的得出 $q \cdot k$ 的均值为

$$E(qk) = 0$$

$$D(qk) = d_k$$

所以向量维度越大，方差越大, 点积结果越“不稳定”。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:

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

最终的结论：通过数学上的技巧将方差控制在 1，也就有效地控制了点积结果的发散，从而解决了对应的梯度消失问题!

5. Multi-head Attention

（1）采用 Multi-head Attention 的原因

原始论文中提到进行 Multi-head Attention 的原因是：将模型分为多个头，可以在不同的子空间中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，从而得到更好的效果。

多个头进行 Attention 计算后最终再综合起来，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。**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

Multi-head Attention 和单一 head 的 Attention 唯一的区别在于，**它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` 切分成 `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。

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。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，清楚地展示了 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 能否代替 Seq2Seq?

（1）Seq2Seq 的两大缺陷

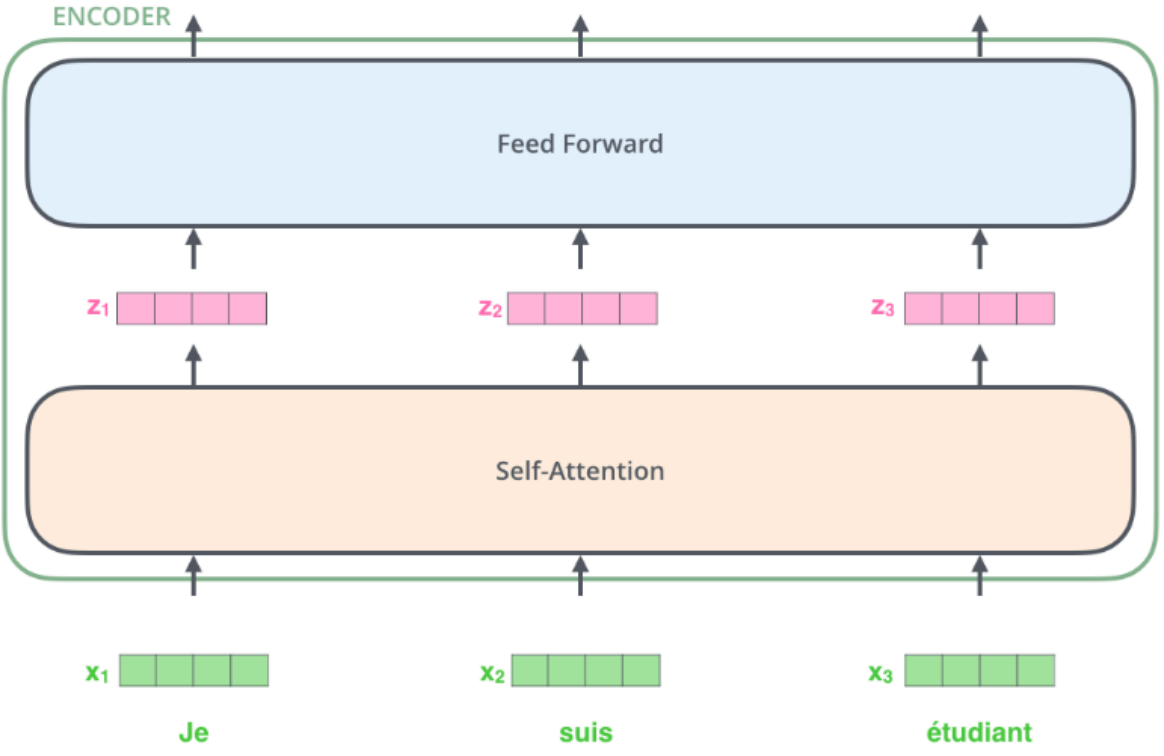
1.Seq2Seq 架构的第一大缺陷是：将 Encoder 端的所有信息压缩成一个固定长度的语义向量，用这个固定的向量来代表编码器端的全部信息。这样做既会造成信息的损耗，也无法让 Decoder 端在解码的时候通过注意力机制去聚焦哪些是更重要的信息。
2.Seq2Seq 架构的第二大缺陷是：无法并行计算。本质上和 RNN/LSTM 无法并行的原因是一样的——序列依赖、逐步推进的结构导致无法同时处理多个时间步的信息。

（2）Transformer 的改进

Transformer 架构同时解决了 Seq2Seq 的两大缺陷：
1.可以并行计算：得益于 Self-Attention 机制，模型可以一次性处理整个序列；
2.应用 Multi-head Attention 机制：解决了 Encoder 固定编码的问题，让 Decoder 在解码的每一步都可以通过注意力机制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



上图最底层绿色的部分，整个序列中所有的 token 可以并行地进行 Embedding 操作，这一层的处理是没有依赖关系的。

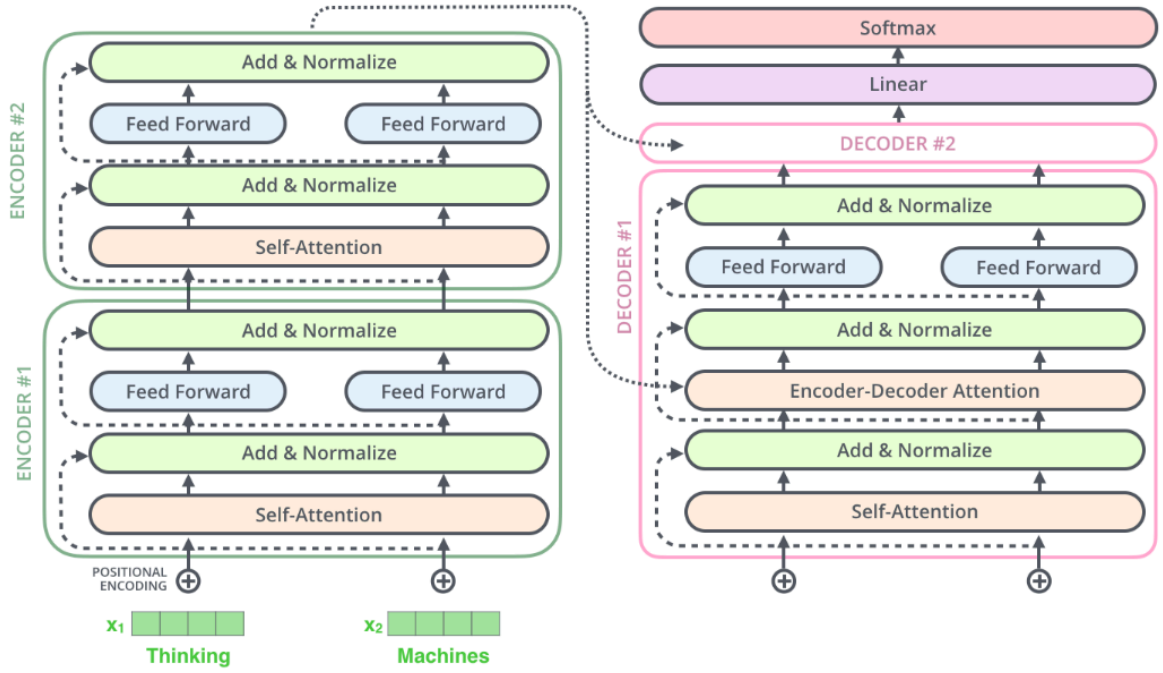
上图第二层土黄色的部分，也就是 Transformer 中最重要的 self-attention 部分。这里对于任意一个单词（比如 x_1），要计算 x_1 对其他所有 token 的注意力分布，得到 z_1。这个过程是具有依赖性的，必须等到序列中所有的单词完成 Embedding 才可以进行。因此这一步是不能并行处理的。**但是从另一个角度看，我们真实计算注意力分布的时候，采用的都是矩阵运算，也就是可以一次性地计算出所有 token 的注意力张量。从这个角度看也算是实现了并行，只是矩阵运算的“并行”和词嵌入的“并行”在概念上有所不同而已。**

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

在这一层是可以实现并行化处理的。也就是说，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。

当然可以！以下是你提供的内容，**不做任何文字修改**，仅将其中的数学公式用 `$$` 包裹起来，便于你做笔记和阅读：

(2) Decoder 的并行化



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通过矩阵乘法实现的，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。

Decoder 模块在预测阶段**基本上不被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预测出的 token。

> **Note**
>
> **注意**：最重要的区别是—— 在训练阶段，如果目标文本有 20 个 token，那么在训练过程中是一次性输入给 Decoder 端的，可以做到一些子层的并行化处理； 但是在预测阶段，如果生成的结果语句总共有 20 个 token，则需要重复处理 20 次循环的过程，每次的输入序列比上一次多一个 token，因此**不被认为是并行处理。**

(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 模块在预测阶段不能并行化处理。**