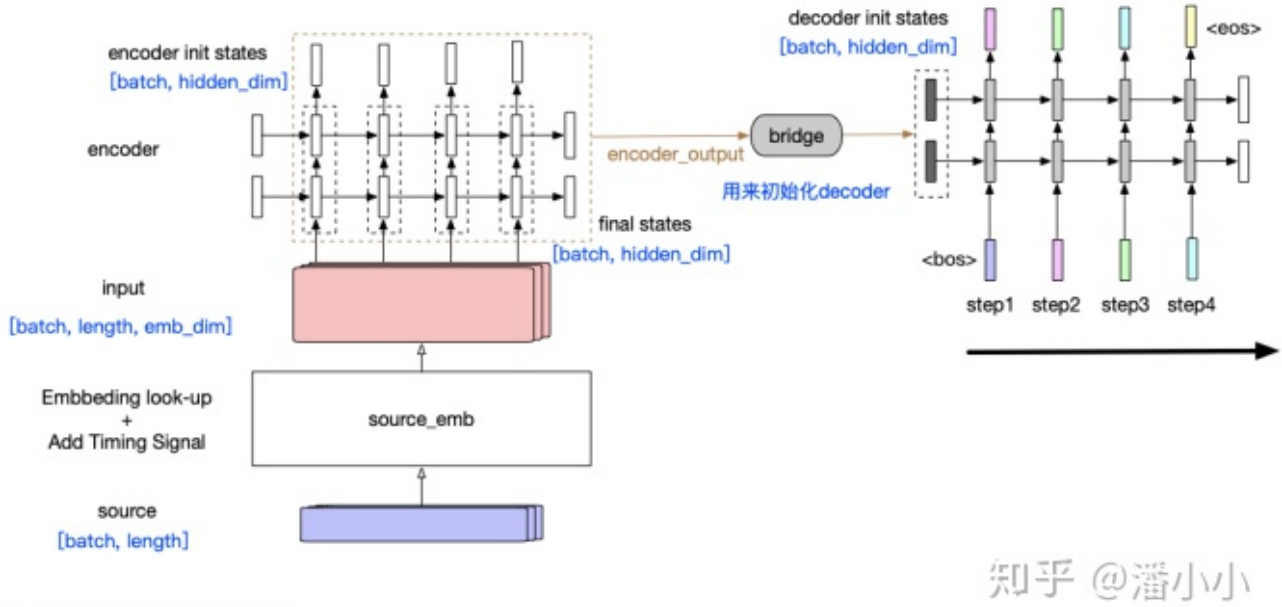
1 Attention是什么?

首先从机器翻译领域的模型演进历史切入。机器翻译是从RNN开始跨入神经网络机器翻译时代的，几个比较重要的阶段分别是: Simple RNN -> Contextualize RNN -> Contextualized RNN with attention -> Transformer(2017)，下面来一一介绍。

1.1 Simple RNN

这个encoder-decoder模型结构中，encoder将整个源端序列(不论长度)压缩成一个向量(encoder output)，源端信息和decoder之间唯一的联系只是: encoder output会作为decoder的initial states的输入。这样带来一个显而易见的问题就是，随着decoder长度的增加，encoder output的信息会衰减。这种模型有2个主要的问题:

1. 源端序列不论长短，都被统一压缩成一个固定维度的向量，并且显而易见的是这个向量中包含的信息中，关于源端序列末尾的token的信息更多，而如果序列很长，最终可能基本上“遗忘”了序列开头的token的信息。
2. 第二个问题同样由RNN的特性造成: 随着decoder timestep的信息的增加，initial hidden states中包含的encoder output相关信息也会衰减，decoder会逐渐“遗忘”源端序列的信息，而更多地关注目标序列中在该timestep之前的token的信息。

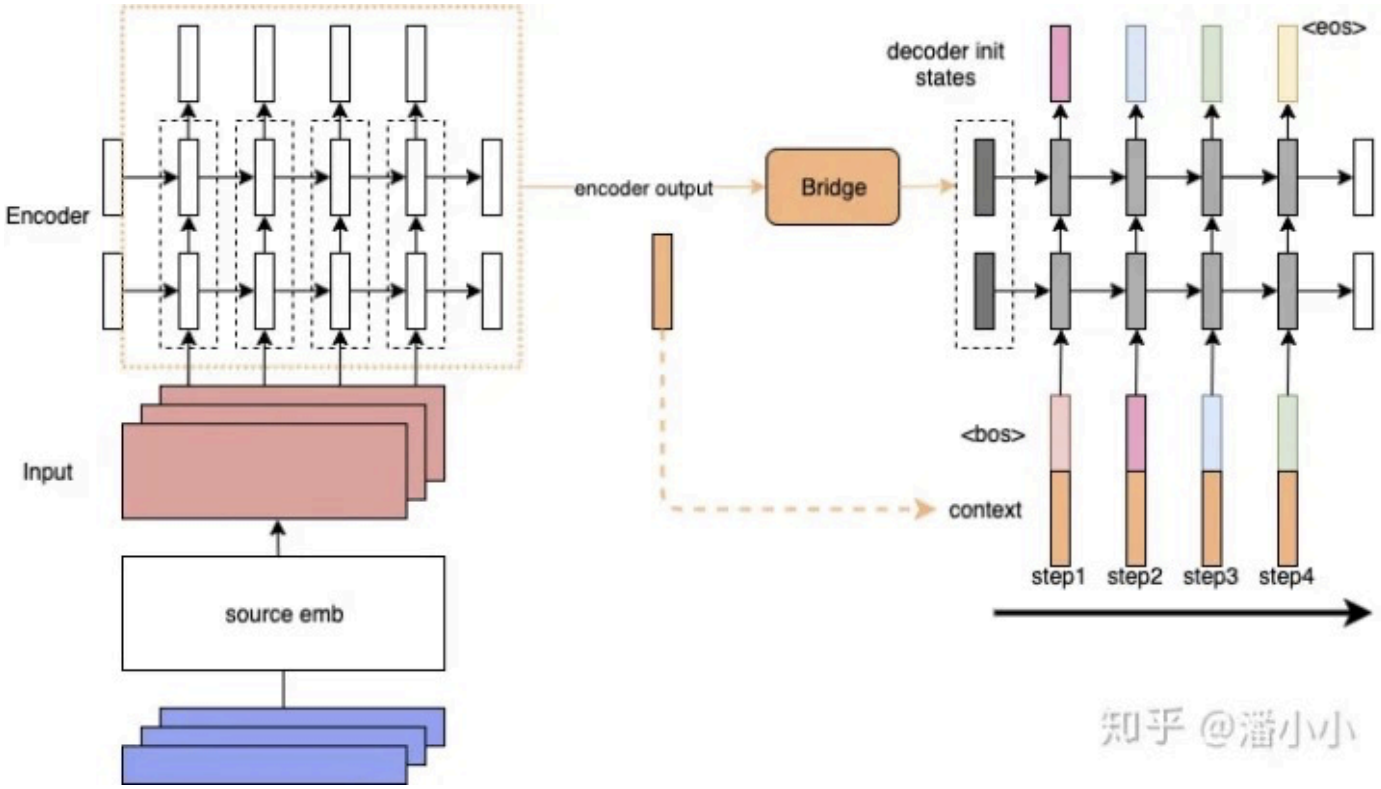


首先要统一同一个batch里的sentences的长度

1.2 Contextualized RNN

为了解决上述第二个问题，即encoder output随着decoder timestep增加而信息衰减的问题，有人提出了一种加了context的RNN sequence to sequence模型：decoder在每个timestep的input上都会加上一个context。为了方便理解，我们可以把这看作是encoded source sentence。这样就可以在decoder的每一步，都把源端的整个句子的信息和target端当前的token一起输入到RNN中，防止源端的context信息随着timestep的增长而衰减。

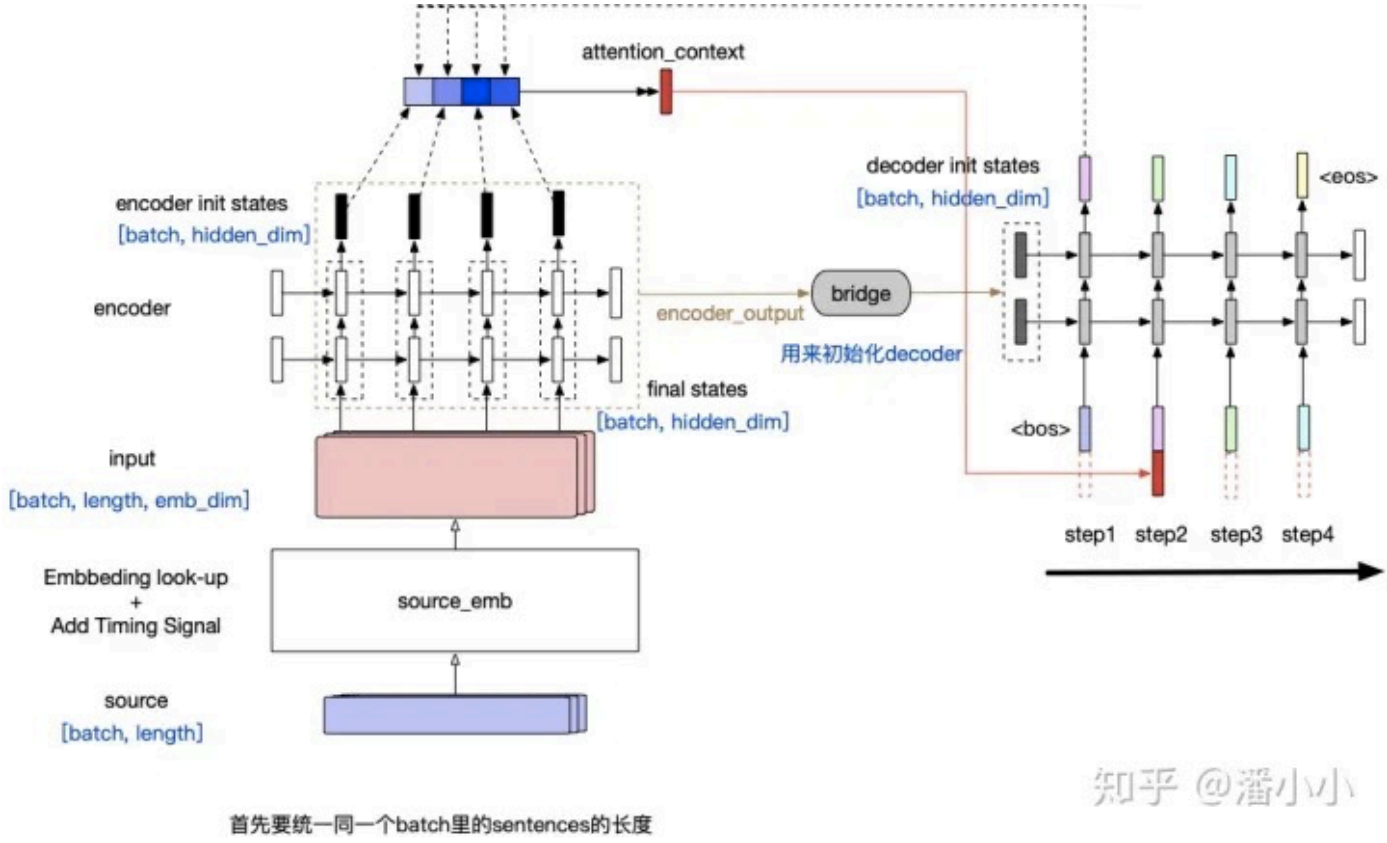
但是这样依然有一个问题: context对于每个timestep都是静态的(encoder端的final hidden states，或者是所有timestep的output的平均值)。但是每个decoder端的token在解码时用到的context真的应该是一样的吗？在这样的背景下，Attention就应运而生了。



1.3 Contextualized RNN with soft align (Attention)

在每个timestep输入到decoder RNN结构中之前，会用当前的输入token的vector与encoder output中的每一个position的vector作一个"attention"操作，这个"attention"操作的目的就是计算当前token与每个position之间的"相关度"，从而决定每个position的vector在最终该timestep的context中占的比重有多少。最终的context即encoder output每个位置vector表达的加权平均。

$$att(q, X) = \sum_{i=1}^{N} \alpha_i X_i$$



Attention与全连接层的区别何在?

> - Attention的最终输出可以看成是一个“在关注部分权重更大的全连接层”。但是它与全连接层的区别在于，注意力机制可以利用输入的特征信息来确定哪些部分更重要
> - 全连接的作用的是对一个实体进行从一个特征空间到另一个特征空间的映射，而注意力机制是要对来自同一个特征空间的多个实体进行整合
> - 全连接的权重对应的是一个实体上的每个特征的重要性，而注意力机制的输出结果是各个实体的重要性。全连接层中的权重，对于所有的样本，都是固定的、相同的；attention模式下的权重，对于不同的样本，则是不同的
> - Attention摆脱了全连接死板的位置依赖，通过输入的特征自动学习关注点，既考虑输入特征自身，又考虑输入特征（空间和通道）之间的相对位置关系

2 Transformer

在没有Transformer以前，神经机器翻译用的最多的是基于RNN的Encoder-Decoder模型：

Transformer中抛弃了传统的CNN和RNN，整个网络结构完全是由Attention机制组成。 采用Attention机制的原因是考虑到RNN（或者LSTM，GRU等）的计算限制为是顺序的，也就是说RNN相关算法只能从左向右依次计算或者从右向左依次计算，这种机制带来了两个问题：

1. 时间片 t 的计算依赖 $t-1$ 时刻的计算结果，这样限制了模型的并行能力
2. 顺序计算的过程中信息会丢失，尽管LSTM等门机制的结构一定程度上缓解了长期依赖的问题，但是对于特别长期的依赖现象，LSTM依旧无能为力

Transformer的提出解决了上面两个问题：

1. 首先它使用了Attention机制，将序列中的任意两个位置之间的距离缩小为一个常量，更好地解决长时依赖问题。当然如果计算量太大，比如序列长度N大于序列维度D这种情况，也可以用窗口限制Self-Attention的计算数量。不管当前词和其他词的空间距离有多远，包含其他词的信息不取决于距离，而是取决于两者的相关

性

2. 其次它不是类似RNN的顺序结构，因此具有更好的并行性，符合现有的GPU框架

整体框架如下：

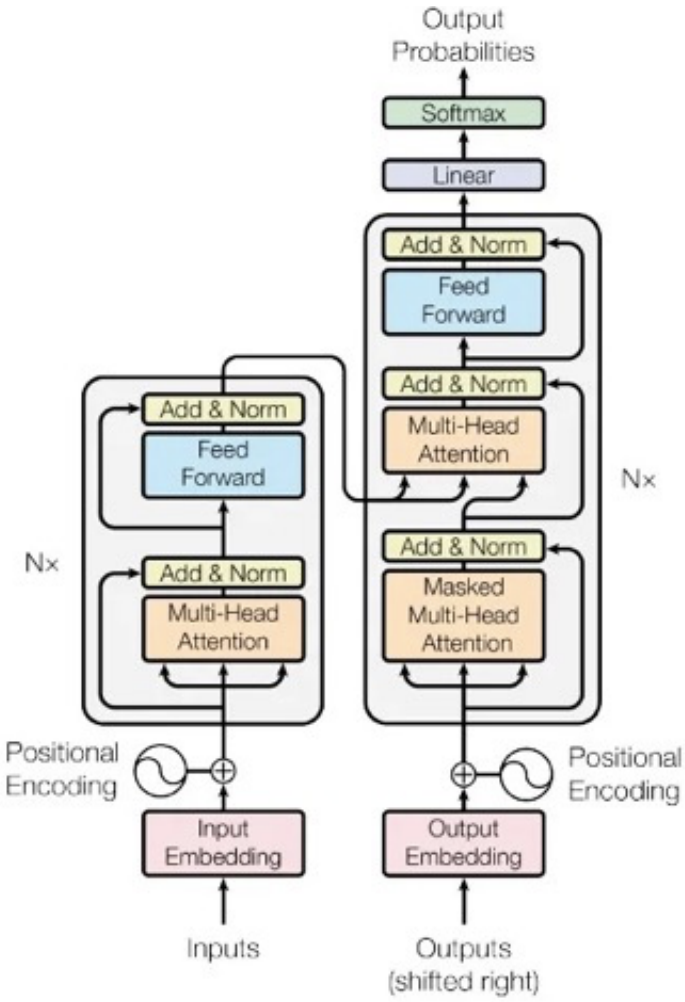


Figure 1: The Transformer - model architecture.

2.1 Self-Attention机制

这个绿色的框（Encoder #1）就是Encoder里的一个独立模块。下面绿色的输入的是两个单词的embedding。这个模块想要做的事情就是想把 x_1 转换为另外一个向量 r_1，这两个向量的维度是一样的。然后就一层层往上传。

转化的过程分成几个步骤，第一个步骤就是Self-Attention，第二个步骤就是普通的全连接神经网络。但是注意，Self-Attention框里是所有的输入向量共同参与了这个过程，也就是说，x_1 和 x_2 通过某种信息交换和杂糅，得到了中间变量 z_1 和 z_2 。而全连接神经网络是割裂开的，z_1 和 z_2 各自独立通过全连接神经网络，得到了r_1 和 r_2。

x_1 和 x_2 互相不知道对方的信息，但因为在第一个步骤Self-Attention中发生了信息交换，所以 r_1 和 r_2各自都有从 x_1 和 x_2 得来的信息了。

Attention和Self-Attention的区别?

> - 以Encoder-Decoder框架为例，输入Source和输出Target内容是不一样的，比如对于英-中机器翻译来说，Source是英文句子，Target是对应的翻译出的中文句子，Attention发生在Target的元素Query和Source中的所有元素之间。
> - Self Attention，指的不是Target和Source之间的Attention机制，而是Source内部元素之间或者Target内部元素之间发生的Attention机制，也可以理解为Target=Source这种特殊情况下的Attention。

自注意力则是注意力机制的一种特殊形式。尽管其输出结果还是输入的加权组合，但是权重是由输入本身得到。它本质上是一个函数，带有三个参数：$\boldsymbol{Q}$、$\boldsymbol{K}$ 和 $\boldsymbol{V}$, 都是 $T \times d$ 大小的矩阵。

$$\boldsymbol{Q}=\begin{bmatrix} - & \boldsymbol{q}_1^\top & - \\ - & \boldsymbol{q}_2^\top & - \\ & \vdots & \\ - & \boldsymbol{q}_T^\top & - \end{bmatrix}, \boldsymbol{K}=\begin{bmatrix} - & \boldsymbol{k}_1^\top & - \\ - & \boldsymbol{k}_2^\top & - \\ & \vdots & \\ - & \boldsymbol{k}_T^\top & - \end{bmatrix}, \boldsymbol{V}=\begin{bmatrix} - & \boldsymbol{v}_1^\top & - \\ - & \boldsymbol{v}_2^\top & - \\ & \vdots & \\ - & \boldsymbol{v}_T^\top & - \end{bmatrix}$$

对于最原始的输入 $\boldsymbol{X}$，这三个矩阵是输入分别经过一个线性变换得到

$$\boldsymbol{Q}=\boldsymbol{X}\boldsymbol{W}_Q, \boldsymbol{K}=\boldsymbol{X}\boldsymbol{W}_K, \boldsymbol{V}=\boldsymbol{X}\boldsymbol{W}_V$$

线性变换的过程保持了输入各行的独立性，没有让不同时间步的表示混杂起来。得到这三个矩阵以后，就可以定义自注意力的计算方法

$$\text{SelfAttention}\left(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}\right)=\text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d_k}}\right)\boldsymbol{V}$$

其最终会把不同时间步的不同值混合在一起。记这个中间矩阵 $\boldsymbol{Q}\boldsymbol{K}^\top$ 为 $\boldsymbol{A}$, 则 $A_{ij}=\boldsymbol{k}_i^\top \boldsymbol{q}_j$，可以理解为是计算 $\boldsymbol{k}_i$ 和 $\boldsymbol{q}_j$ 之间的相似度。经过 softmax以后，得到的矩阵 $\tilde{\boldsymbol{A}}$ 尺寸不变，但是可以让每行的值的和都为1。最后结果和各输入参数尺寸一致，第 i 行可以看做是 $\boldsymbol{V}$ 由权重矩阵第 i 行加权得到的结果。Transformer就是利用了一系列注意力机制（以及前馈层）来处理时序输入。即：

$$\boldsymbol{Z}^{(i+1)}=\text{TransformerLayer}\left(\boldsymbol{Z}^{(i)}\right)$$

Multi-Head Attention 上方还包括一个 Add & Norm 层，Add 表示残差连接 (Residual Connection) 用于防止网络退化，Norm 表示 Layer Normalization，用于对每一层的激活值进行归一化。考虑注意力机制可能对复杂过程的拟合程度不够, 通过增加两层网络来增强模型的能力。因此Encoder端和Decoder端每个子模块实际的输出为：

$$\begin{aligned}\tilde{\boldsymbol{Z}}=&\ \text{SelfAttention}\left(\boldsymbol{Z}^{(i)}\boldsymbol{W}_Q, \boldsymbol{Z}^{(i)}\boldsymbol{W}_K, \boldsymbol{Z}^{(i)}\boldsymbol{W}_V\right)\\ =&\ \text{softmax}\left(\frac{\boldsymbol{Z}^{(i)}\boldsymbol{W}_Q\boldsymbol{W}_K^\top\boldsymbol{Z}^{(i)^\top}}{\sqrt{d_k}}\boldsymbol{Z}^{(i)}\boldsymbol{W}_V\right)\\ \tilde{\boldsymbol{Z}}=&\ \text{LayerNorm}\left(\boldsymbol{Z}^{(i)}+\tilde{\boldsymbol{Z}}\right)\\ \boldsymbol{Z}^{(i+1)}=&\ \text{LayerNorm}\left(\text{ReLU}(\tilde{\boldsymbol{Z}}\boldsymbol{W})+\tilde{\boldsymbol{Z}}\right)\end{aligned}$$

1. 上式为什么要除以 $\sqrt{d_k}$？

> 缩放因子的作用是归一化。为了防止维数过高时 QK^T的值过大导致softmax函数反向传播时发生梯度消失。那为什么是 $\sqrt{d_k}$ 而不是 d_k 呢？假设 Q、K 里的元素的均值为0，方差为1，那么 $A=QK^T$ 中元素的均值为0，方差为d。当 d_k 变得很大时，A 中的元素的方差也会变得很大，如果 A 中的元素方差很大，那么$softmax(A)$ 的分布会趋于陡峭(分布的方差大，分布集中在绝对值大的区域)。总结一下就是 $softmax(A)$ 的分布会和 d_k 有关。因此 A 中每一个元素除以 $\sqrt{d_k}$ 后，方差又变为1。这使得 $softmax(A)$ 的分布“陡峭”程度与 d_k 解耦，从而使得训练过程中梯度值保持稳定。

> X、Y 相互独立，$EX=0$、$EY=0$ 时：
> $$
> \begin{aligned}
> E(XY) &= EX \cdot EY \\
> D(X) &= E^2(X-EX) = EX^2 - E^2X \\
> D(X+Y) &= D(X) + D(Y) \\
> \Rightarrow D(XY) &= E^2[XY - E(XY)] \\
> &= E\left[(XY)^2\right] - E^2(XY) \\
> &= EX^2 \cdot EY^2 - E^2X \cdot E^2Y \\
> &= DX \cdot DY
> \end{aligned}
> $$
>
> $$
> A_{ij} = \sum_k^{d_k} Q_{ik} \cdot K_{jk}
> $$
>
> $$
> D\left(A_{ij}\right) = D\left(\sum_k^{d_k} Q_{ik} \cdot K_{jk}\right) = \sum_k^{d_k} D\left(Q_{ik}\right) \cdot D\left(K_{jk}\right) = d_k
> $$

2. 为什么在分类层（最后一层），使用非 scaled 的 softmax?

> 最后一层 softmax 通常和交叉熵联合求导，在某个目标类别 i 上的整体梯度变为 $y_i' - y_i$，即预测值和真值的差。当出现某个极大的元素值，softmax 的输出概率会集中在该类别上。如果预测正确，整体梯度接近于0，抑制参数更新；如果预测错误，则整体梯度接近于1，给出最大程度的负反馈。也就是说，这个时候的梯度形式改变，不会出现极大值导致梯度消失的情况了。

3. self-attention中词向量不乘 Q、K、V 参数矩阵，会有什么问题?

> Self-Attention的核心是用文本中的其它词来增强目标词的语义表示，从而更好的利用上下文的信息。如果不乘，$Q=K=V=X$ 是完全一样的，会导致 $A_{ij}=A_{ji}$。另外在相同量级的情况下，q_i 与 k_i 点积的值会是最大的。那在softmax后的加权平均中，该词本身所占的比重将会是最大的，使得其他词的比重很少，无法有效利用上下文信息来增强当前词的语义表示。

4. Self-Attention 的时间复杂度是怎么计算的?

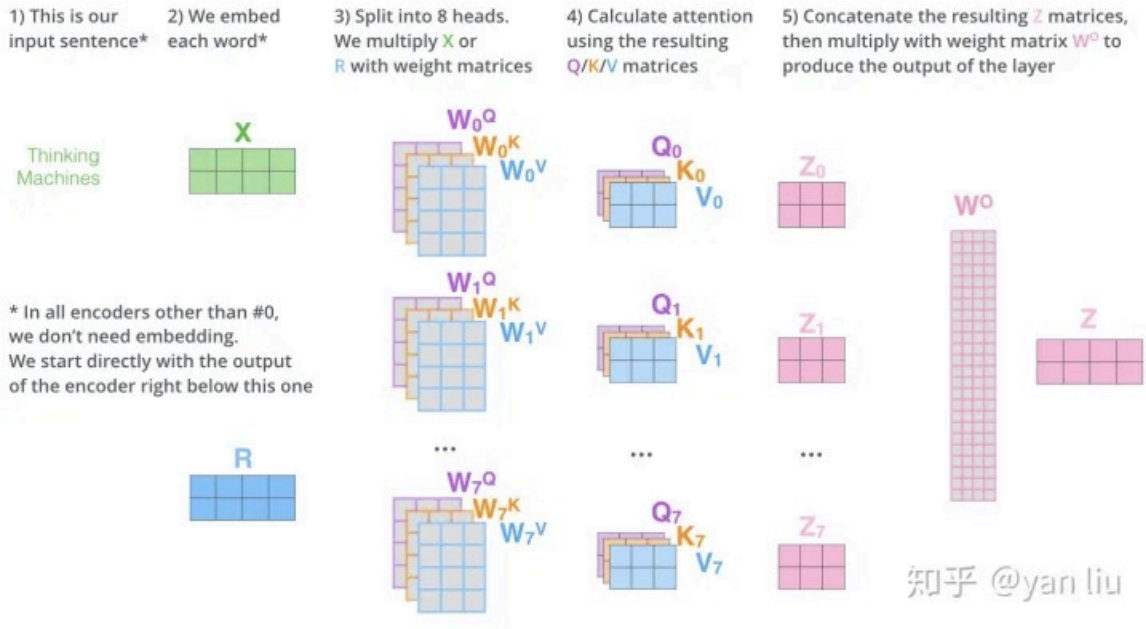
> Self-Attention时间复杂度：$O\left(T^2 \cdot d\right)$，这里，$T$ 是序列的长度，d 是embedding的维度。Self-Attention包括三个步骤：相似度计算，softmax 和加权平均，它们分别的时间复杂度是：
>
> - 相似度计算可以看作大小为 (T,d) 和 (d,T) 的两个矩阵相乘，时间复杂度：$O\left(T^2 \cdot d\right)$，得到一个 (T,T) 的矩阵
> - softmax就是直接计算了，时间复杂度为 $O(T^2)$
> - 加权平均可以看作大小为 (T,T) 和 (T,d) 的两个矩阵相乘，时间复杂度：$O\left(T^2 \cdot d\right)$

2.2 Multi-Head Attention

Multi-Head Attention相当于 h 个不同的self-attention的集成（ensemble），在这里我们以 $h=8$ 举例说明。Multi-Head Attention的输出分成3步：

1. 将数据 $\boldsymbol{X}$ 分别输入到 8 个self-attention中，得到8个加权后的特征矩阵 $Z_i, i \in \{1,2,\ldots,8\}$
2. 将 8 个 Z_i 按列拼成一个大的特征矩阵
3. 特征矩阵经过一层全连接后得到输出 Z

$$
\begin{aligned}
\mathrm{MultiHead}(Q,K,V) &= \mathrm{Contact}\left(\mathrm{head}_1,\ldots,\mathrm{head}_2\right)W^O \\
\text{where } \mathrm{head}_i &= \mathrm{Attention}\left(QW_i^Q, KW_i^K, VW_i^V\right)
\end{aligned}
$$



它给出了注意力层的多个“表示子空间”（representation subspaces）。这些集合中的每一个都是随机初始化的，在训练之后，每个集合都被用来将输入词嵌入投影到不同的表示子空间中。可以让Attention有更丰富的层次。有多个Q、K、V 的话，可以分别从多个不同角度来看待Attention。这样的话，输入X，对于不同的multi-headed Attention，就会产生不同的Z。这样可以在不改变参数量的情况下增强每一层attention的表现力。举一个不一定妥帖的例子：当你浏览网页的时候，你可能在颜色方面更加关注深色的文字，而在字体方面会去注意大的、粗体的文字。这里的颜色和字体就是两个不同的表示子空间。同时关注颜色和字体，可以有效定位到网页中强调的内容。使用多头注意力，也就是综合利用各方面的信息/特征。

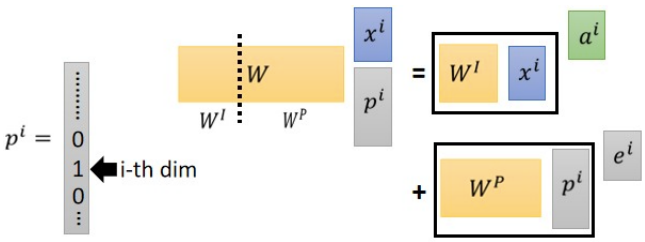
Multi-head Attention的本质是，在参数总量保持不变的情况下，借鉴CNN多核的思想，将同样的query, key, value映射到原来的高维空间的不同子空间中进行attention的计算，在最后一步再合并不同子空间中的attention信息。这样降低了计算每个head的attention时每个向量的维度，在某种意义上防止了过拟合；由于Attention在不同子空间中有不同的分布，Multi-head Attention实际上是寻找了序列之间不同角度的关联关系，并在最后concat这一步骤中，将不同子空间中捕获到的关联关系再综合起来。简而言之，就是希望每个注意力头，只关注最终输出序列中一个子空间，互相独立，其核心思想在于，抽取到更加丰富的特征信息。

2.3 位置编码

Transformer 本身是不能利用单词的顺序信息的，因此需要在输入中添加位置 Embedding，否则 Transformer 就是一个词袋模型了。输入的时候，不仅有单词的向量 x，还要加上Positional Encoding，即输入模型的整个Embedding是Word Embedding与Positional Embedding直接相加之后的结果。这是想让网络知道这个单词所在句子中的位置是什么，是想让网络做自注意力的时候，不但要知道注意力要聚焦在哪个单词上面，还想要知道单词之间的互相距离有多远。

不同位置的Positional Encoding是独特的。但是计算Positional Encoding的方式不是唯一的，甚至Positional Encoding也可以是train出来的，并不是必须用作者说的 sin、cos。只要能相互计算距离就可以。

Embedding 直接相加会对语义有影响吗?

> 
>

> 在原始输入向量上concat一个代表位置信息的向量，再经过一个全连接网络，最终的效果等价于：先对原始输入向量做变换(token embedding)，然后再加上位置嵌入(position embedding)。

2.4 Encoder模块

通过上面描述的 Multi-Head Attention, Feed Forward, Add & Norm 就可以构造出一个 Encoder block，Encoder block 接收输入矩阵 $X_{T\times d}$，并输出一个矩阵 $O_{T\times d}$。通过多个 Encoder block 叠加就可以组成 Encoder。

第一个 Encoder block 的输入为句子单词的表示向量矩阵，后续 Encoder block 的输入是前一个 Encoder block 的输出，最后一个 Encoder block 输出的矩阵就是编码信息矩阵 C，这一矩阵后续会用到 Decoder 中。

2.5 Mask

mask表示掩码，它对某些值进行掩盖，使其在参数更新时不产生效果。Transformer模型里面涉及两种mask，分别是 padding mask和sequence mask。 其中，padding mask在所有的scaled dot-product attention 里面都需要用到，而sequence mask只有在Decoder的Self-Attention里面用到。

因为每个批次输入序列长度是不一样的也就是说，我们要对输入序列进行对齐。具体来说，就是给在较短的序列后面填充 0。但是如果输入的序列太长，则是截取左边的内容，把多余的直接舍弃。因为这些填充的位置，其实是没什么意义的，所以我们的Attention机制不应该把注意力放在这些位置上。具体的做法是，把这些位置的值加上一个非常大的负数(负无穷)，这样的话，经过softmax，这些位置的概率就会接近0！ 而我们的padding mask 实际上是一个张量，每个值都是一个Boolean，值为false的地方就是我们要进行处理的地方。

sequence mask是为了使得Decoder不能看见未来的信息。也就是对于一个序列，在time_step为t的时刻，我们的解码输出应该只能依赖于t时刻之前的输出，而不能依赖t之后的输出。因此我们需要想一个办法，把t之后的信息给隐藏起来。 具体做法：产生一个上三角矩阵，对角线以及对角线左下都是1。把这个矩阵作用在每一个序列上，就可以达到我们的目的。

对于Decoder的Self-Attention，里面使用到的scaled dot-product attention，同时需要padding mask和sequence mask作为attn_mask，具体实现就是两个mask相加作为attn_mask。其他情况，attn_mask一律等于padding mask。

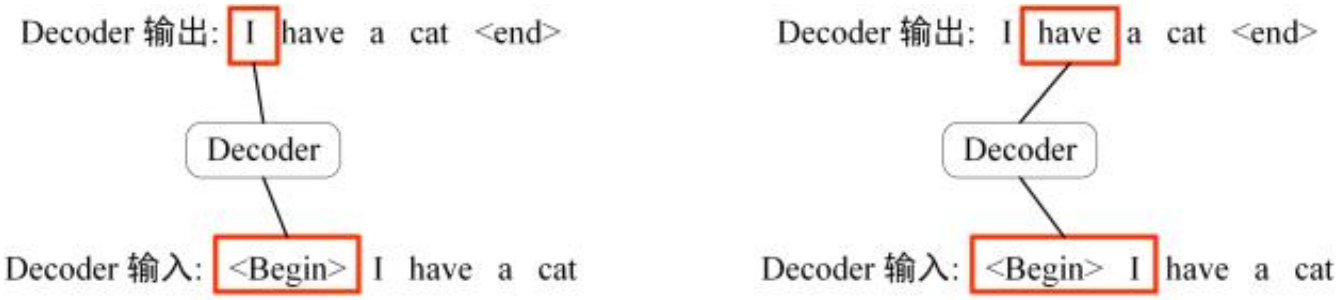
2.6 Decoder模块

上图红色部分为 Transformer 的 Decoder block 结构，与 Encoder block 相似，但是存在一些区别：

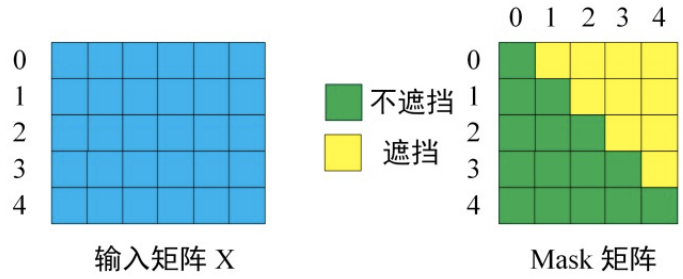
- 包含两个 Multi-Head Attention 层
- 第一个 Multi-Head Attention 层采用了 Masked 操作
- 第二个 Multi-Head Attention 层的 K, V 矩阵使用 Encoder 的编码信息矩阵 C 进行计算，而 Q 使用上一个 Decoder block 的输出计算
- 最后有一个 Softmax 层计算下一个翻译单词的概率

2.7 Decoder 中的 Self-Attention

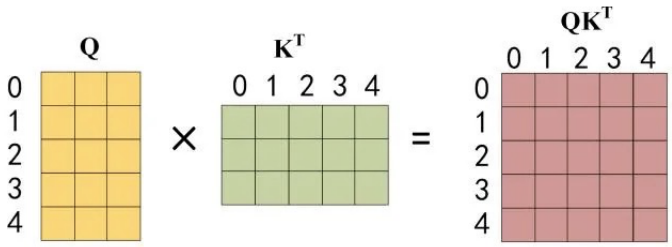
Decoder block 的第一个 Multi-Head Attention 采用了 Masked 操作，因为在翻译的过程中是顺序翻译的，即翻译完第 i 个单词，才可以翻译第 i+1 个单词。通过 Masked 操作可以防止第 i 个单词知道 i+1 个单词之后的信息。



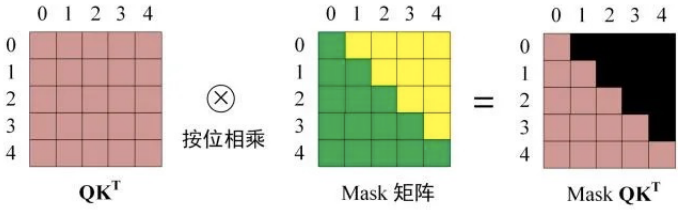
1. Decoder 的输入矩阵和 Mask 矩阵



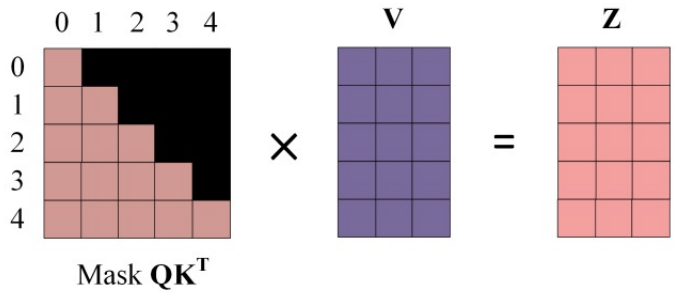
2. 接下来的操作和之前的 Self-Attention 一样，通过输入矩阵 X 计算得到 Q, K, V 矩阵。然后计算 QK^T



3. 在得到 QK^T 之后需要进行 Softmax，计算 attention score，我们在 Softmax 之前需要使用Mask矩阵遮挡住每一个单词之后的信息，遮挡操作如下：



4. 使用 $Mask(QK^T)$ 与矩阵 V 相乘，得到输出 Z。则单词 1 的输出向量 Z_1 是只包含单词 1 信息的



5. 通过上述步骤就可以得到一个 Mask Self-Attention 的输出矩阵 Z_i，然后和 Encoder 类似，通过 Multi-Head Attention 拼接多个输出 Z_i 然后计算得到第一个 Multi-Head Attention 的输出 Z

2.8 Encoder-Decoder Attention

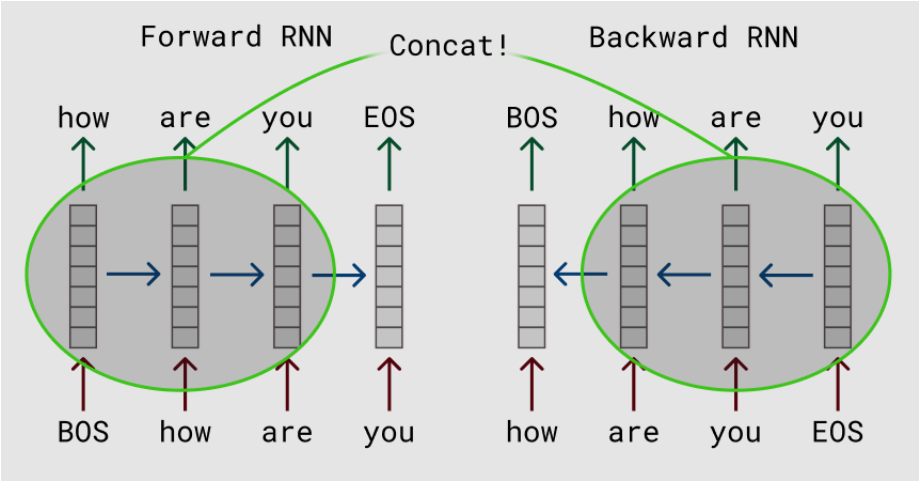
主要的区别在于其中 Self-Attention 的 K, V 矩阵不是使用上一个 Decoder block 中的 Self-Attention 输出计算的，而是使用 Encoder 的编码信息矩阵 C 计算的。根据 Encoder 的输出 C 计算得到 K, V。

根据上一个 Decoder block 中的 Self-Attention 的输出 Z 计算 Q (如果是第一个 Decoder block 则使用输入矩阵 X 进行计算)，后续的计算方法与之前描述的一致。这样做的好处是在 Decoder 的时候，每一位单词都可以利用到 Encoder 所有单词的信息 (这些信息无需 Mask)。

3 ELMo

之前的Word2Vec本质上是个静态的方式，训练好之后每个单词的Word Embedding不会跟着上下文场景的变化而改变。

ELMo是Embeddings from Language Models的缩写。它的思路非常简单，就是用海量的数据训练双向LSTM的语言模型。训练好的模型就不动了，对于下游的任务，我们用这个LSTM来提取文本的特征，然后在它基础上加一些全连接层，用监督数据微调这些全连接层的参数。



- 对于每个方向上的单词来说，因为两个方向彼此独立训练，故在一个方向被encoding的时候始终是看不到它另一侧的单词的，从而避免 see it self 的问题
- 考虑到句子中有的单词的语义会同时依赖于它左右两侧的某些词，仅仅从单方向做encoding是不能描述清楚的，所以再来一个反向encoding，故称双向

4 BERT

ELMo和GPT最大的问题：

1. 语言模型是单向的，无法利用整个(下文)句子的信息。即使ELMo训练了双向的两个RNN，但是一个RNN只能看一个方向，因此也是无法"同时"利用前后两个方向的信息的。
2. 训练的时候输入是一个句子，但是很多下游任务比如相似度计算和问答等输入是多个句子

4.1 Mask LM

随机的Mask掉15%的词，然后让BERT来预测这些Mask的词，通过调整模型的参数使得模型预测正确的概率尽可能大，这等价于交叉熵的损失函数。这样的Transformer在编码一个词的时候会(必须)参考上下文的信息。

但是这有一个问题：在Pretraining Mask LM时会出现特殊的Token [MASK]，但是在后面的fine-tuning时却不会出现，这会出现Mismatch的问题。因此BERT中，如果某个Token在被选中的15%个Token里，则按照下面的方式随机的执行。例如在这个句子"my dog is hairy"中，它选择的token是"hairy"：

- 80%的概率替换成[MASK]，比如my dog is hairy → my dog is [MASK]
- 10%的概率替换成随机的一个词，比如my dog is hairy → my dog is apple
- 10%的概率替换成它本身，比如my dog is hairy → my dog is hairy

这样做的好处是，BERT并不知道[MASK]替换的是哪一个词，而且任何一个词都有可能是被替换掉的，比如它看到的apple可能是被替换的词。这样强迫模型在编码当前时刻的时候不能太依赖于当前的词，而要考虑它的上下文，甚至根据上下文进行"纠错"。相当于告诉模型，我可能给你答案，也可能不给你答案，也可能给你错误的答案，有[MASK] 的地方我会检查你的答案，没 [MASK] 的地方我也可能检查你的答案，所以 [MASK] 标签对你来说没有什么特殊意义。

4.2 预测句子关系

NSP(Next Sentence Prediction)给定一个句子对，判断在原文中第二个句子是否紧接第一个句子。正例来自于原文中相邻的句子；负例来自于不同文章的句子。RoBERTa,ALBERT,XLNet都做过实验表明NSP没有必要。

BERT 预训练阶段实际上是将上述两个任务结合起来，同时进行，然后将所有的 Loss 相加。

bert的mask为何不学习transformer在attention处进行屏蔽score的技巧?

> - 用［MASK］多了一个位置信息，position embedding是有的，表示原文这个位置有一个词。如果按attention mask的方式，等于这个被遮掩的token永远不会参与计算，他就少了一个信息："位置x有一个词"。每个词的一词多义的语义不仅来自上下文，同样来自这个词在文本中所处的位置，所以必须标示出来
> - 第一层【其他词】看到了【本词】，【本词】不知道本词；第二层【本词】会去索引【其他词】。因为其他词都知道【本词】是【本词】，第二层【本词】必然也知道了，不适合Bert的目的

5 ALBERT

ALBERT 的结构和 BERT 基本一样，具体的创新部分有三个：

1. embedding 层参数因式分解：ALBERT 不直接将原本的 one-hot 向量映射到 H，而是分解成两个矩阵，原本参数数量为 $V * H$，V 表示的是 Vocab Size。分解成两步则减少为 $V * E + E * H$，当 H 的值很大时，能大幅降低参数数量
2. 跨层参数共享：ALBERT 作者尝试将所有层的参数进行共享，相当于只学习第一层的参数，并在剩下的所有层中重用该层的参数，而不是每个层都学习不同的参数
3. 将 NSP 任务改为 SOP(Sentence Order Prediction) 任务：NSP作为一项任务，和MLM相比，没有什么难度。具体而言，由于正例来自于同样一篇文章，而负例来自于不同的文章，很可能判别正负例仅仅需要判别两个句子的主题是否一致，而这些信息已经在MLM中学习过了。作为NSP的替代，SOP显然更有挑战性。它使用同一篇文章的相邻的两个句子作为正例，把句子的位置互换作为负例。任务是判断句序是否正确。而这仅属于一致性预测，大大提高了预测的难度。

6 XLNet

自回归(Autoregressive, AR)语言模型和自编码(autoencoding)模型的优缺点分别为：

- 独立假设：BERT假设在给定 $\hat{\mathbf{x}}$ 的条件下被Mask的词是独立的(没有关系的)，这个显然并不成立。输入：[New York] is a city；[Los Angle] is a city。模型学习完之后，认为[New Angle] is a city的概率也会很高。没有考虑New和York之间的关系，也没有考虑Los和Angle之间的关系，单词和单词之间是独立的
- 输入噪声：BERT的在预训练时会出现特殊的[MASK]，但是它在下游的fine-tuning中不会出现，这就是出现了不匹配。
- AR语言模型只能参考一个方向的上下文

与其说XLNet解决了BERT的问题，不如说它基于AR采用了一种新的方法实现双向编码，因为AR方法不存在上述两个痛点。

6.1 排列(Permutation)语言模型

给定长度为 T 的序列 x，总共有 $T!$ 种排列方法，也就对应 $T!$ 种链式分解方法。比如假设 $\mathbf{x}=x_1x_2x_3$，那么总共用 $3!=6$ 种分解方法:

$$
\begin{aligned}
p(\mathbf{x}) &= p\left(x_1\right)p\left(x_2 \mid x_1\right)p\left(x_3 \mid x_1x_2\right) \Rightarrow 1 \rightarrow 2 \rightarrow 3 \\
p(\mathbf{x}) &= p\left(x_1\right)p\left(x_2 \mid x_1x_3\right)p\left(x_3 \mid x_1\right) \Rightarrow 1 \rightarrow 3 \rightarrow 2 \\
p(\mathbf{x}) &= p\left(x_1 \mid x_2\right)p\left(x_2\right)p\left(x_3 \mid x_1x_2\right) \Rightarrow 2 \rightarrow 1 \rightarrow 3 \\
p(\mathbf{x}) &= p\left(x_1 \mid x_2x_3\right)p\left(x_2\right)p\left(x_3 \mid x_2\right) \Rightarrow 2 \rightarrow 3 \rightarrow 1 \\
p(\mathbf{x}) &= p\left(x_1 \mid x_3\right)p\left(x_2 \mid x_1x_3\right)p\left(x_3\right) \Rightarrow 3 \rightarrow 1 \rightarrow 2
\end{aligned}
$$

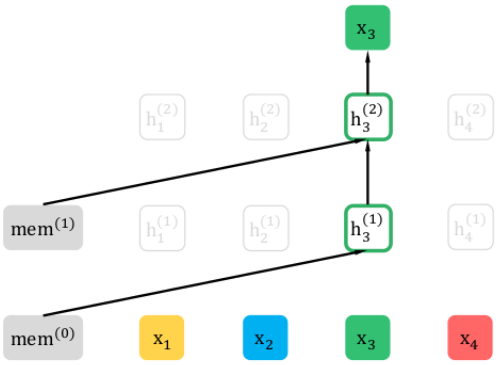
- 注意 $p\left(x_2 \mid x_1x_3\right)$ 指的是第一个词是 x_1 并且第三个词是 x_3 的条件下第二个词是 x_2 的概率，也就是说原来词的顺序是保持的。如果理解为第一个词是 x_1 并且第二个词是 x_3 的条件下第三个词是 x_2，那么就不对了

因此我们可以遍历 $T!$ 种路径，然后学习语言模型的参数，但是这个计算量非常大，实际我们只能随机的采样部分排列。排列语言模型的目标是从所有的排列中采样一种，然后根据这个排列来分解联合概率成条件概率的乘积，然后加起来。调整模型参数使得下面的似然概率最大：

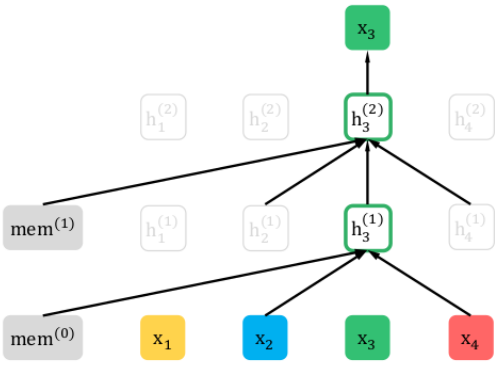
$$
\max_{\theta} \mathbb{E}_{z \sim \mathcal{Z}_T}\left[\sum_{t=1}^{T} \log p_\theta\left(x_{z_t} \mid \mathbf{x}_{z_{<t}}\right)\right]
$$

- $\mathcal{Z}_T$：长度为 T 的序列的所有排列组成的集合
- $z \in \mathcal{Z}_T$：是一种排列方法
- z_t：表示排列的第 t 个元素
- $z_{<t}$：z 的第1到第 $\mathrm{t}-1$ 个元素

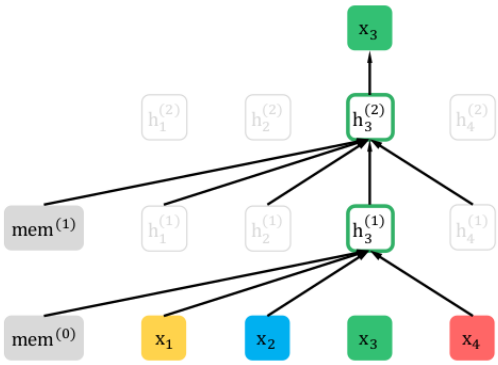
上面的模型只会遍历概率的分解顺序，并不会改变原始词的顺序。实现是通过Attention的Mask来对应不同的分解方法。排列语言模型在预测 x_3 时不同排列的情况如下：



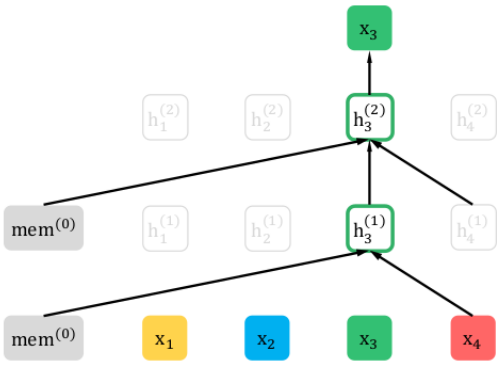
Factorization order: 3 → 2 → 4 → 1



Factorization order: 2 → 4 → 3 → 1



Factorization order: 1 → 4 → 2 → 3



Factorization order: 4 → 3 → 1 → 2

6.2 Two-Stream Self-Attention

如果我们使用标准的Transformer来实现时会有问题。假设输入的句子是"I like New York"，并且一种排列为 $z=[1,3,4,2]$，假设我们需要预测 $z_3=4$，那么根据公式：

$$p_\theta\left(X_{z_3}=x \mid x_{z_1 z_2}\right)=p_\theta\left(X_4=x \mid x_1 x_3\right)=\frac{\exp\left(e(x)^T h_\theta\left(x_1 x_3\right)\right)}{\sum_{x'} \exp\left(e(x')^T h_\theta\left(x_1 x_3\right)\right)}$$

- 表示第一个词是I，第3个词是New的条件下第4个词是York的概率

另外我们再假设一种排列为 $z'=[1,3,2,4]$，我们需要预测 $z_3=2$，那么：

$$p_\theta\left(X_{z_3}=x \mid x_{z_1 z_2}\right)=p_\theta\left(X_2=x \mid x_1 x_3\right)=\frac{\exp\left(e(x)^T h_\theta\left(x_1 x_3\right)\right)}{\sum_{x'} \exp\left(e(x')^T h_\theta\left(x_1 x_3\right)\right)}$$

- 表示是第一个词是I，第3个词是New的条件下第2个词是York的概率

仔细对比一下公式会发现这两个概率是相等的，问题的关键是模型并不知道要预测的那个词在原始序列中的位置。为了解决这个问题，我们把预测的位置 z_t 放到模型里：

$$p_\theta\left(X_{z_t}=x \mid \mathbf{x}_{z_{<t}}\right)=\frac{\exp\left(e(x)^T g_\theta\left(\mathbf{x}_{z_{<t}}, z_t\right)\right)}{\sum_{x'} \exp\left(e(x')^T g_\theta\left(\mathbf{x}_{z_{<t}}, z_t\right)\right)}$$

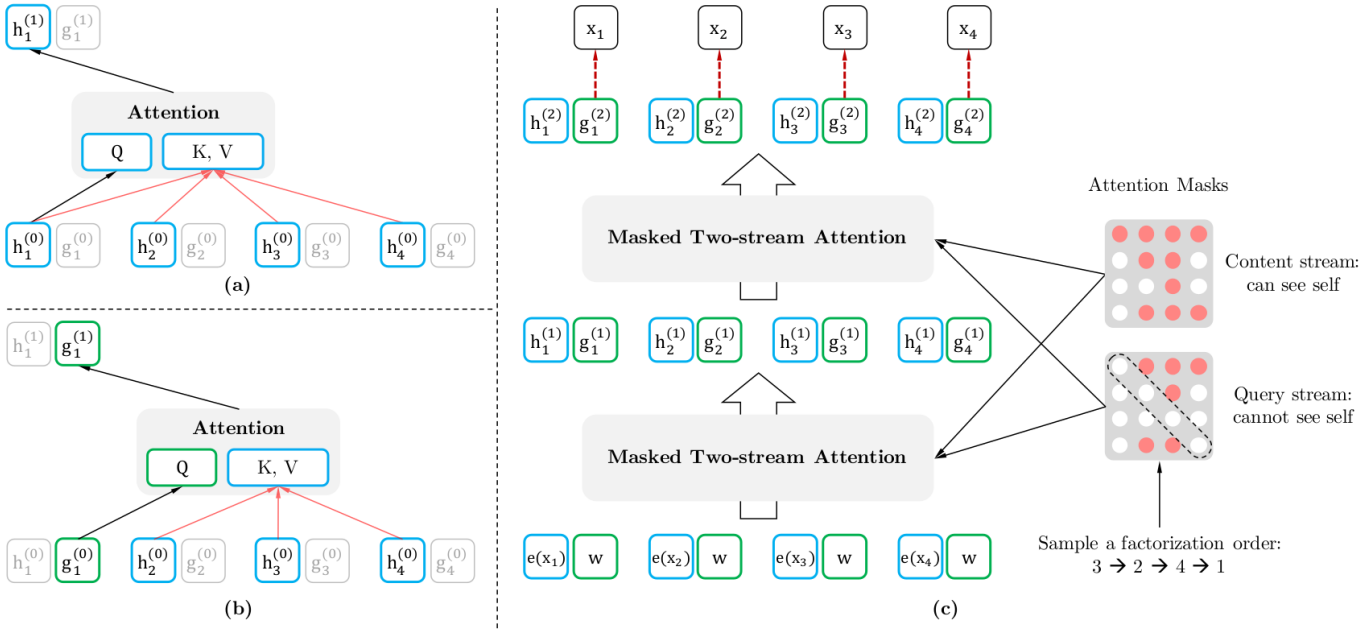
- $g_\theta\left(\mathbf{x}_{z_{<t}}, z_t\right)$：这是一个新的模型 g，并且它的参数除了之前的词 $\mathbf{x}_{z_{<t}}$，还有要预测的词的位置 z_t

接下来的问题是用什么模型来表示 $g_\theta\left(\mathbf{x}_{z<t}, z_t\right)$。它需要满足如下两点要求：

1. 为了预测 $\mathbf{x}_{z_t}$，$g_\theta\left(\mathbf{x}_{z_{<t}}, z_t\right)$ 只能使用位置信息 z_t 而不能使用 $\mathbf{x}_{z_t}$ 。你预测一个词当然不能知道要预测的是什么词
2. 为了预测 z_t 之后的词，$g_\theta\left(\mathbf{x}_{z_{<t}}, z_t\right)$ 必须编码了 x_{z_t} 的信息

为了解决这个问题，论文引入了两个Stream，也就是两个隐状态：

- Content流 $h_\theta\left(\mathbf{x}_{z<t}\right)$，简写为 h_{zt}：它和标准的Transformer一样，既编码上下文(context)也编码 x_{z_t} 的内容
- Query流 $g_\theta\left(\mathbf{x}_{z_{<p}} z_t\right)$，简写为 g_{z_t}：它只编码上下文和要预测的位置 z_t，但是不包含 x_{z_t}



假设排列为 $3 \rightarrow 2 \rightarrow 4 \rightarrow 1$，并且我们现在预测第 1 个位 置的词的概率：

- Content流计算：我们可以参考所有 4 个词的信息，因此 $KV=\left[h_1^{(0)}, h_2^{(0)}, h_3^{(0)}, h_4^{(0)}\right]$，而 $Q=h_1^{(0)}$
- Query流的计算：因为不能参考自己的内容，因此 $KV=\left[h_2^{(0)}, h_3^{(0)}, h_4^{(0)}\right]$，而 $Q=g_1^{(0)}$
- Query流与Content流共享参数
- Query Mask和Content Mask的区别就是不能attend to自己，因此对角线都是白点。

上面两个流分别使用自己的Query向量 g_{z_t} 和 h_{zt}，但是Key和Value向量用的都是内容 h。

Two Stream排列模型的计算过程：

1. 把Query流 $g_i^{(0)}$ 初始化为一个变量 w，把Content流 $h_i^{(0)}$ 初始化为词的Embedding $e\left(x_i\right)$ 。这里的上标0表示第0层(不存在的层，用于计算第一层)。因为Content流可以编码当前词，因此初始化为词的Embedding比较合适
2. 然后Content Mask和Query Mask计算第一层的输出 $h^{(1)}$ 和 $g^{(1)}$
3. 然后计算第二层.....
4. 在计算损失函数的时候，我们使用最上面一层的Query向量 $g_{z_t}^{(M)}$
5. 在fine-tuning的时候，我们可以丢弃掉Query流而只用Content流

6.3 XLNet与BERT的对比

XLNet和BERT都是预测一个句子的部分词。BERT使用的是Mask语言模型，因此只能预测部分词；XLNet预测部分词是出于性能考虑。

除此之外，它们最大的区别其实条件独立的假设：那些被MASK的词在给定非MASK的词的条件下是独立的，但是这个假设并不(总是)成立。下面我们通过一个例子来说明：

假设输入是[New, York, is, a, city]，并且假设恰巧XLNet和BERT都选择使用[is, a, city]来预测New和York。同时我们假设XLNet的排列顺序为[is, a, city, New, York]。那么它们优化的目标函数分别为：

$$\begin{aligned}\mathcal{J}_{\mathrm{BERT}} &= \log p(\ \mathrm{New}\ \mid\ \mathrm{is\ a\ city}\ ) + \log p(\ \mathrm{York}\ \mid\ \mathrm{is\ a\ city}\ ) \\ \mathcal{J}_{\mathrm{XLNet}} &= \log p\ (\mathrm{New}\ \mid\ \mathrm{is\ a\ city}\ ) + \log p(\ \mathrm{York}\ \mid\ \mathrm{New, is\ a\ city}\ )\end{aligned}$$

从上面可以发现，XLNet可以在预测York的使用利用New的信息，因此它能学到”New York”经常出现在一起而且它们出现在一起的语义和单独出现是完全不同的。