

# XLNet/Transformer-XL

**Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context.** Zihang Dai, Zhilin Yang, et al. ACL 2019 [PDF]

**XLNet: Generalized Autoregressive Pretraining for Language Understanding.** Zhilin Yang, Zihang Dai, et al. NeurIPS 2019 [PDF]

BERT掀起NLP预训练模型的热潮之后，很多人都在BERT上修修补补，但也有少数工作敏锐地观察到，BERT之所以成功，最大的功臣其实是Transformer。本文介绍的Transformer-XL和XLNet就是比较成功的对Transformer进行改进的工作：Transformer-XL在Transformer基础上，延长上下文长度的同时也提高了推理速度；XLNet结合自编码语言模型和自回归语言模型二者之长，并引入Transformer-XL作为backbone，对比BERT成绩有较大的提升。

正是由于XLNet和Transformer-XL的一脉相承（两篇论文的作者都是来自CMU的博士戴自航、杨植麟），所以这里将它们放在一起。

## 1 Transformer-XL

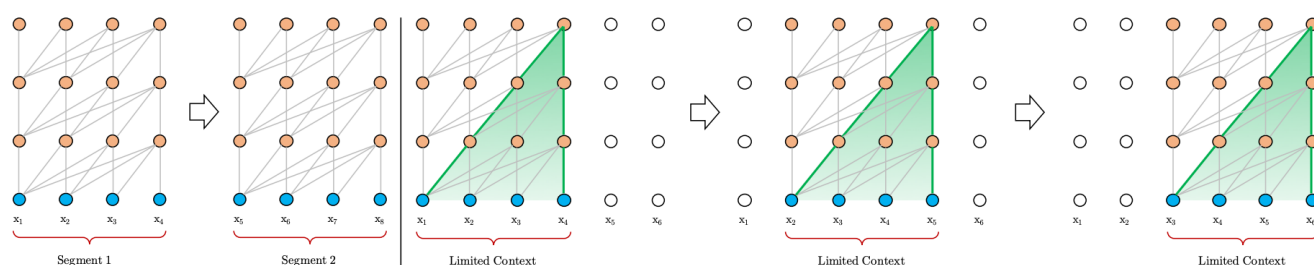
### 1.1 总览

在捕捉长期依赖信息时，Transformer上下文的长度是固定的。于是Transformer-XL提出**片段级循环机制**（segment-level recurrence mechanism）和**相对位置编码**（relative positional encoding）在不破坏时间连续性的前提下捕捉更长的上下文依赖信息。实验结果表明Transformer-XL的上下文长度比RNN长80%，比vanilla Transformer长450%，无论是短序列还是长序列，Transformer-XL都取得了更好的成绩，并且在评估时推导速度达到了vanilla Transformer的1800多倍。

### 1.2 上下文分裂问题（context fragmentation problem）

语言模型的核心问题是有效地将任意长度的上下文编码成固定长度的表征。

一个简单暴力的办法就是将文本语料切成可以处理的等长的片段（segment），然后在依次在各个片段上进行训练，这种模型称为vanilla model. 如下图所示，(a)展示了训练的过程，(b)展示了推导的过程。



vanilla model的弊端就是上下文分裂。简单地将序列切割成固定长度的片段，会无视上一个片段的所有上下文信息，这样，片段的长度就成为能够学习到依赖信息的最大长度的上界。此外，由于各个片段之间是分开训练的，所以每次都需要自底向上逐层计算，这种计算方式低效且耗时。

### 1.3 状态复用的片段级循环（segment-level recurrence with state reuse）

Transformer-XL提出，每个片段内的计算都会复用上一个片段的隐藏状态（如下图绿色连线），这样当前片段内部就不用全部重头计算了，大大缩减了计算量。

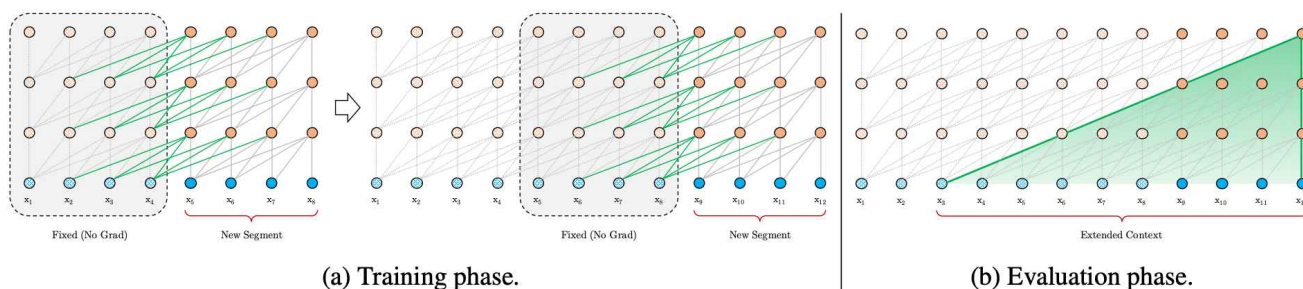
假设  $s_\tau = [x_{\tau,1}, \dots, x_{\tau,L}]$  和  $s_{\tau+1} = [x_{\tau+1,1}, \dots, x_{\tau+1,L}]$  是两个长度为  $L$  的连续片段， $h_{\tau+1}^n$  是片段  $s_\tau$  第  $n$  层的隐藏状态。那么  $h_{\tau+1}^n$  的计算过程为：

$$\begin{aligned}\tilde{h}_{\tau+1}^{n-1} &= [SG(h_\tau^{n-1}) \circ h_{\tau+1}^{n-1}] \\ q_{\tau+1}^n, k_{\tau+1}^n, v_{\tau+1}^n &= h_{\tau+1}^{n-1} W_q^T, \tilde{h}_{\tau+1}^{n-1} W_k^T, \tilde{h}_{\tau+1}^{n-1} W_v^T \\ h_{\tau+1}^n &= TransformerLayer(q_{\tau+1}^n, k_{\tau+1}^n, v_{\tau+1}^n)\end{aligned}$$

其中  $SG(\cdot)$  表示停止梯度， $[h_u \circ h_v]$  表示沿着序列长度所在轴的拼接。

有个细节，Q和KV不同，它仅来源于当前片段。

由此可见，Transformer-XL将上一个片段的隐藏状态固定并缓存，然后用于下一个片段的预测。



如上图所示，在这种情况下，Transformer-XL的真实上下文长度（绿色阴影）是和层数成线性关系的，即  $O(N \times L)$ 。除此之外，由于状态的复用，计算效率也得到了提高。

片段级循环，实现了上下文的扩展，解决了上下文分裂问题，同时还提高了计算效率。

### 1.3 相对位置编码

状态复用的片段级循环通虽然解决了上下文分裂问题，但是也带来了一个新的问题，那就是之前的位置编码不再适用了，模型无法区分不同片段中相同位置的字符。

假设  $U \in \mathbb{R}^{L_{max} \times d}$  为位置编码， $E_{s_\tau}$  表示片段  $s_\tau$  的字符嵌入，Transformer的输入是字符嵌入和位置编码的和，隐藏状态的计算过程为：

$$\begin{aligned} h_{\tau+1} &= f(h_\tau, E_{s_{\tau+1}} + U_{1:L}) \\ h_\tau &= f(h_{\tau-1}, E_{s_\tau} + U_{1:L}) \end{aligned}$$

此时模型无法区分  $x_{\tau,j}$  和  $x_{\tau+1,j}$  之间的位置差异， $j = 1, \dots, L$ 。

在原先的Transformer中，位置  $i$  处的query和位置  $j$  处的key之间的注意力分数计算过程为：

$$\begin{aligned} A_{i,j}^{abs} &= [W_q(E_{x_i} + U_i)]^T W_k(E_{x_j} + U_j) \\ &= (E_{x_i}^T + U_i^T) W_q^T W_k (E_{x_j} + U_j) \\ &= \underbrace{E_{x_i}^T W_q^T W_k E_{x_j}}_{(a)} + \underbrace{E_{x_i}^T W_q^T W_k U_j}_{(b)} + \underbrace{U_i^T W_q^T W_k E_{x_j}}_{(c)} + \underbrace{U_i^T W_q^T W_k U_j}_{(d)} \end{aligned}$$

Transformer-XL提出使用相对位置编码来对绝对位置进行替换：

$$A_{i,j}^{rel} = \underbrace{E_{x_i}^T W_q^T \mathbf{W}_{k,E} E_{x_j}}_{(a)} + \underbrace{E_{x_i}^T W_q^T \mathbf{W}_{k,R} R_{i-j}}_{(b)} + \underbrace{u^T \mathbf{W}_{k,E} E_{x_j}}_{(c)} + \underbrace{v^T \mathbf{W}_{k,R} R_{i-j}}_{(d)}$$

修改的地方有三处：

- 将所有绝对位置编码  $U_j$  换成对应的相对位置编码  $R_{i-j}$ ，和Transformer的绝对位置编码一样，这里的相对位置编码也是个无需学习的正弦编码矩阵
- 将(c)中的  $U_i^T W_q^T$  换成可训练参数  $u^T$ ，将(d)中的  $U_i^T W_q^T$  换成可训练的参数  $v^T$ ，考虑到  $U_i W_q^T$  应该和位置  $i$  无关，每个位置对应的  $q$  应该相同
- 和  $E$  相乘的  $W_k$  换成  $W_{k,E}$ ，表示基于内容的关键字向量（content-based key vector），和  $U$  相乘的  $W_k$  换成  $W_{k,R}$ ，表示基于位置的关键字向量（position-based key vector）

综合以上状态复用的片段级循环和相对位置编码，得到Transformer-XL完整计算过程：

$$\begin{aligned} \tilde{h}_\tau^{n-1} &= [SG(m_\tau^{n-1}) \circ h_\tau^{n-1}] \\ q_\tau^n, k_\tau^n, v_\tau^n &= h_\tau^{n-1} W_q^{nT}, \tilde{h}_\tau^{n-1} W_{k,E}^{nT}, \tilde{h}_\tau^{n-1} W_v^{nT} \\ A_{\tau,i,j}^n &= q_{\tau,i}^{nT} k_{\tau,j}^n + q_{\tau,i}^{nT} W_{k,R}^n R_{i-j} + u^T k_{\tau,j}^n + v^T W_{k,R}^n R_{i-j} \\ a_\tau^n &= \text{Masked-Softmax}(A_\tau^n) v_\tau^n \\ o_\tau^n &= \text{LayerNorm}(\text{Linear}(a_\tau^n) + h_\tau^{n-1}) \\ h_\tau^n &= \text{Positionwise-Feed-Forward}(o_\tau^n) \end{aligned}$$

## 2 XLNet

## 2.1 总览

XLNet主要包含三个点：

- 为了同时捕获到上下文左右两边的信息，自回归模型XLNet在所有可能的因子分解顺序（factorization order）上最大化期望对数似然，并通过双流自注意力（two-stream self-attention）架构来弥补对位置感知能力的缺失，从而完美地将自编码语言模型和自回归语言模型的优点结合在一起；
- 在模型方面，XLNet采用了比Transformer性能更好的Transformer-XL，并通过调整Transformer-XL的计算过程来更好地适应多种因子分解顺序；
- 除了算法提升，XLNet在数据上也做了提升：在BERT的基础上加了三个数据集

实验结果表明，和BERT相比，XLNet在多个任务上均取得较大的提升。



XLNet的成功经验：预训练领域提升成绩的主流方法都是算法和数据双管齐下。

## 2.2 自编码和自回归

给定文本序列  $x = [x_1, \dots, x_T]$ ，自回归（autoregressive, AR）语言模型根据上文（或下文）预测当前字符，模型最大化如下的对数似然：

$$\max_{\theta} \log p_{\theta}(x) = \sum_{t=1}^T \log p_{\theta}(x_t | x_{<t})$$

而自编码（autoencoding, AE）语言模型对上下文进行建模，以BERT为例，AE通过破坏-重构的思路来预测序列缺失的字符：

$$\max_{\theta} \log p_{\theta}(\bar{x} | \hat{x}) \approx \sum_{t=1}^T m_t \log p_{\theta}(x_t | \hat{x})$$

其中  $m_t = 1$  表示  $x_t$  被遮掩。

二者的优缺点，可以从三个角度去分析：

- 独立性假设：AE假设给定其他未遮掩字符，预测字符之间是独立的，这种独立性假设使得每个遮掩字符都是分开重构的。AE正是基于这种独立性假设来对联合概率  $p_{\theta}(\bar{x} | \hat{x})$  进行因子分解。而AR则是按照乘法规则对  $p_{\theta}(x)$  进行因子分解，字符之间是存在依赖关系的；
- 输入噪声：AE，以BERT为例，会引入一些特殊字符（噪声），然而在微调的时候这些特殊字符却不会出现，从而导致预训练和微调的不一致；反观AR不依赖破坏输入序列，所以不存在这个问题；
- 上下文依赖：AR预测当前字符仅仅依赖于上文（或下文），属于单向的上下文，而AE则是可以利用上下文两边的信息。

📌 在双向上下文建模比单向好已经成为共识的时候，GPT系列模型却始终奉行单向建模。

基于以上的分析，AE和AR各有优劣，于是XLNet考虑将二者的优点结合。

## 2.3 排列语言建模 (permutation language modeling)

XLNet在本质上仍旧属于AR模型，但是它在保留AR模型的优势的基础上，通过排列语言建模目标函数来捕获双向的上下文。

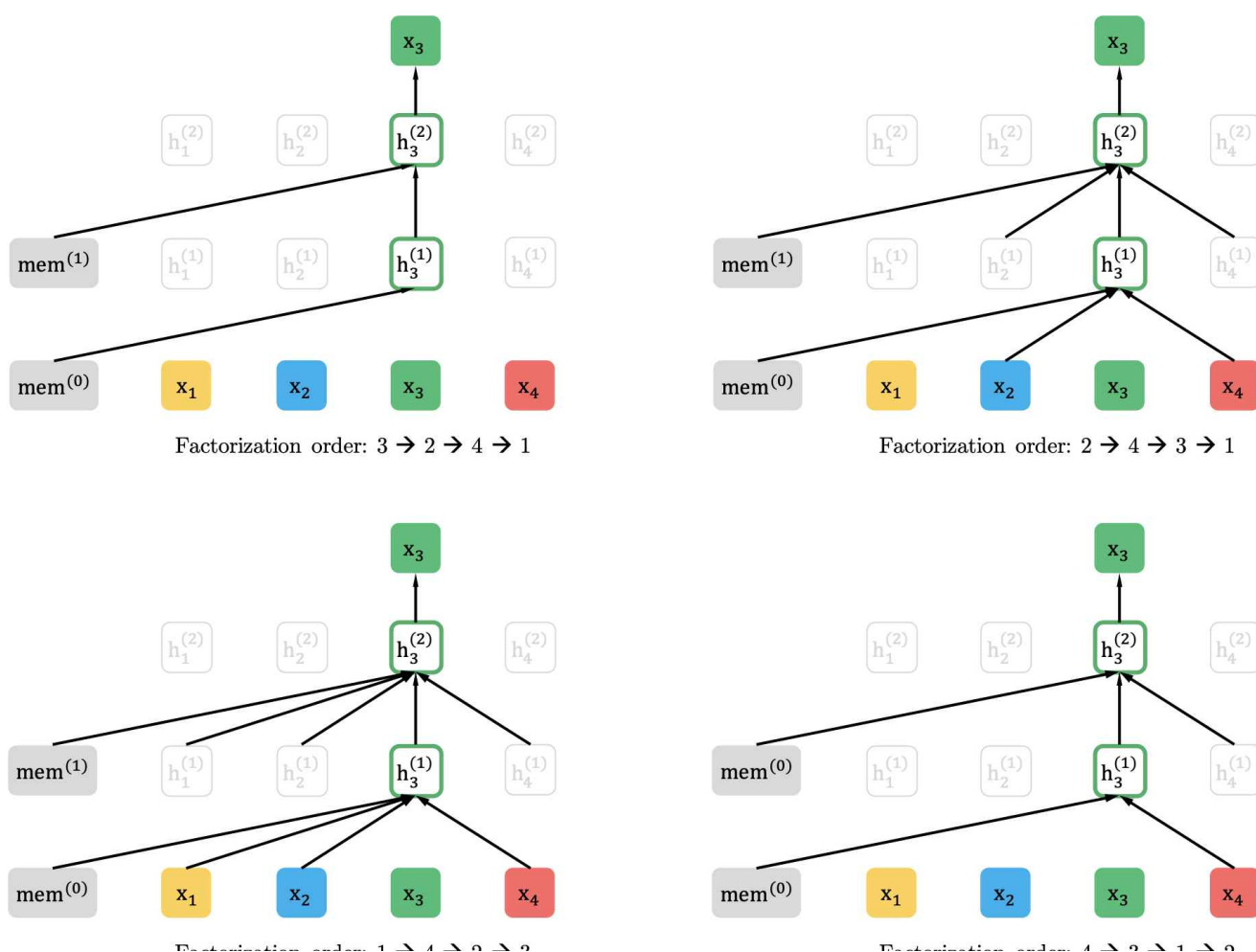
排列语言建模的核心思想就是，长度为  $T$  的序列，其因子分解顺序有  $T!$  种，如果模型在所有不同的因子分解顺序上共享参数，那么模型就能够学习收集所有位置的上下文信息。

以输入序列  $[x_1, \dots, x_T]$  为例， $Z_T$  是输入序列  $x$  所有可能的因子分解顺序，那么排列语言建模目标函数可以表示为：

$$\max_{\theta} \mathbb{E}_{z \sim Z_T} \left[ \sum_{t=1}^T \log p_{\theta}(x_{z_t} | x_{z < t}) \right]$$

对于序列  $x$ ，每次都取样出一个因子分解顺序，然后根据顺序来分解似然  $p_{\theta}(x)$ 。于是，通过对所有因子分解顺序的建模，在任意位置上可以看到所有可能的上下文元素（即  $x_i \neq x_t$ ）。


那么如何在保证序列顺序不变的前提下，更改序列的因子分解顺序呢？方法就是引入合适的注意力掩码（attention mask），如下图所示：



左上角的图表示因子分解的顺序是3->2->4->1，那么此时  $x_3$  处，它只能看到初始值。

右上角的图表示因子分解的顺序是2->4->3->1，那么此时  $x_3$  处，它可以看到之前的2和4，即依赖于  $x_2$  和  $x_4$  。

左下和右下同理，于是通过设置注意力掩码可以实现在输入序列不变的情况下对序列建立不同的因子分解顺序。

 这里将自回归和自编码结合，就不再是传统的自回归了，传统的自回归是把上一个位置的输出当做下一个位置的输入，而这里稍微改变了下形式，在上文的作用下是采用不同分解顺序。

## 2.4 双流注意力（two-stream self-attention）

通过设置注意力掩码，可以很好地利用Transformer来根据上文预测字符：

$$p_{\theta}(x_{z_t} | x_{z < t}) = \frac{\exp(e(x)^T h_{\theta}(x_{z < t}))}{\sum_{x'} \exp(e(x')^T h_{\theta}(x_{z < t}))}$$

其中  $h_{\theta}$  表示基于注意力生成的上文表征。但是上面这个预测的式子缺失了对位置的感知能力，也就是预测token的概率分布不会因为目标位置而有所不同。



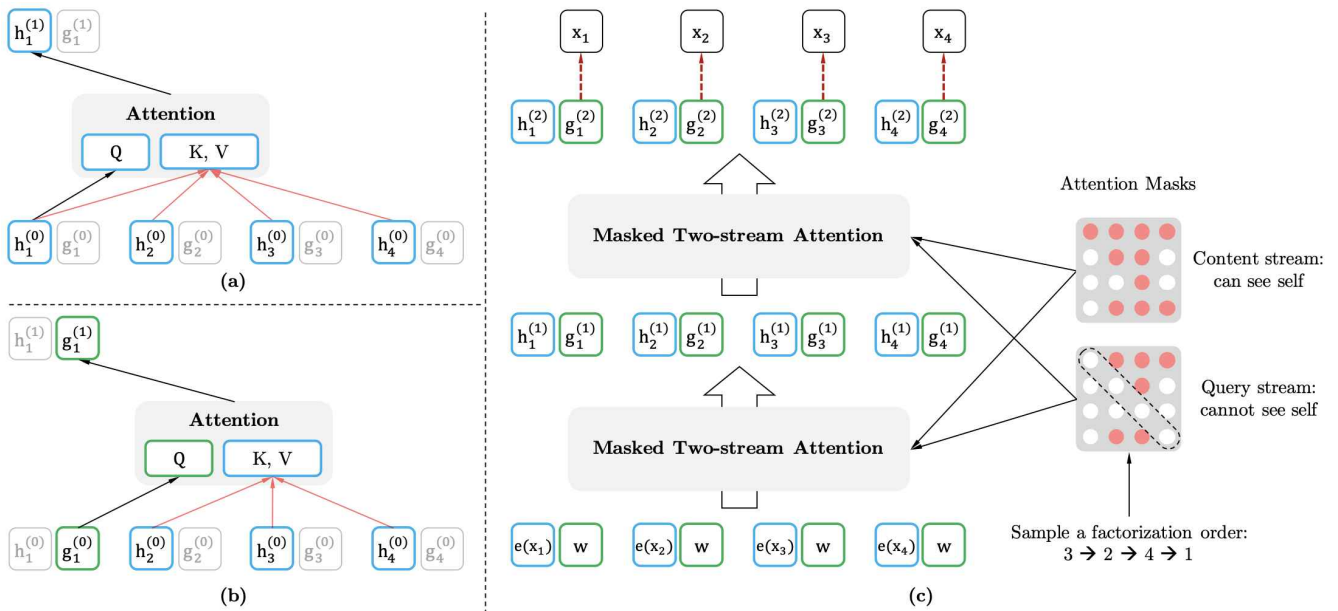
## 疑问

上文相同时，当前位置不就唯一确定了吗，难道存在两个因子分解顺序 $z_1, z_2$ ， $z_{1<t} = z_{2<t}$  但是  $z_{1t} \neq z_{2t}$  ？关键是论文在appendix中，还真的举了这个例子。。。

为了弥补对位置的感知能力，现把位置作为输入，并把当前位置的输入剔除掉，得到目标感知的表征（target-aware representation）：

$$p_{\theta}(x_{z_t} | x_{z<t}) = \frac{\exp(e(x)^T g_{\theta}(x_{z<t}, z_t))}{\sum_{x'} \exp(e(x')^T g_{\theta}(x_{z<t}, z_t))} \quad (*)$$

于是将以上二者结合得到如下所示的双流自注意力：



(a) 图表示内容流自注意力（content stream self-attention），也就是常规的注意力方法，它对所有上文以及当前位置的隐藏状态进行编码，得到  $h_{\theta}(x_{z \leq t})$ ，目的是预测下文的字符（ $z_{>t}$ ）；

(b) 图表示查询流自注意力（query stream self-attention），对上文隐藏状态以及当前位置进行编码，输出  $g_{\theta}(x_{z < t}, z_t)$ ，进而预测当前字符。

(c) 图则是整个架构的展示，左边表示了三层的Transformer，每一层都包含了右边的两个注意力流。可以看到两个注意力掩码方阵的主对角是有区别的：内容流是全满的，表示可以看到自身，而查询流是全空的，表示看不到自身。另外，内容流的初始层（第0层输入层）的参数就是embedding  $e(x)$ ，查询层的初始层的参数是可训练向量  $w$ ，其余层都是共享参数的。层与层之间的参数更新过程如下：

$$\begin{aligned} g_{z_t}^{(m)} &\leftarrow \text{Attention}(Q = g_{z_t}^{(m-1)}, KV = h_{z<t}^{(m-1)}; \theta) & \text{query stream} \\ h_{z_t}^{(m)} &\leftarrow \text{Attention}(Q = h_{z_t}^{(m-1)}, KV = h_{z \leq t}^{(m-1)}; \theta) & \text{content stream} \end{aligned}$$

预训练时利用最后一层的  $g_{z_t}^{(M)}$  来预测公式 (\*)；在微调时，简单地把查询流注意力删除即可。

为了让模型快速收敛，选择一个切分点，选择让模型只预测尾部的一些字符。

## 2.5 引入Transformer-XL

Transformer-XL存在片段的缓存，所以KV是来源自多个片段的，这里对其计算过程进行调整（当前片段只取上文的隐藏状态）：

$$h_{z_t}^{(m)} \leftarrow \text{Attention}(Q = h_{z_t}^{(m-1)}, KV = [\tilde{h}^{m-1}, h_{z \leq t}^{m-1}]; \theta)$$

这里的[,]是按照序列长度进行的拼接。

## 2.6 多片段建模

对于句子对任务，需要输入多个句子。

## 2.7 最后

Transformer-XL和XLNet是基于BERT改进最“成功”的模型，也确实是从模型上进行了创新，进步很大。机器之心对二人做了一场[报道](#)，其中有几个观点具有参考性：

- 依靠算力解决问题是当前研究 AI 的王道：让计算机去做它的强项——计算；如果算力解决不了的问题，再用算法去做。“我读过人工智能先驱 Richard Sutton几个月前的文章《苦涩的教训》，它的大意是说：你如果纵观 70 年的 AI 发展历程，就会发现以算力为杠杆的通用方法是最有效的。从深蓝、AlphaGo 到 NLP最近的进展都遵循了这个思路。所以我们要做的事情就是：一方面把算力推到极致，另一方面发明和提升通用算法，解决更难的问题。XLNet 可以理解成这两方面的一个结合；
- 把算力推到极致的好处是知晓当前算法的边界，避免在算力可以解决的问题上做一些不必要的算法创新，让大家关注最重要的研究问题。但同时大算力带来的弊端是提升了研究门槛，比如一般的学校和实验室可能没有资源做预训练。这个问题我觉得短时间内要通过不同的分工来解决，资源多的研究者利用资源做大算力研究，资源少的研究者做基于小算力的研究；
- 早几天 XLNet 团队公平地对比了 BERT-Large 和 XLNet-Large 两个模型，他们表示尽管 XLNet 的数据是 BERT 的 10倍，但算法带来的提升相比于数据带来的提升更大。杨植麟说：“我认为并不是数据越多越好，我们的 XLNet 基本上将手头有的数据都加上去了，但我们需要做更仔细的分析，因为很可能数据质量和数量之间会有一个权衡关系。BERT所采用的 BooksCorpus 和 English Wikipedia 数据集质量都非常高，它们都是专业作者书写的文本。但是后面增加的 Common Crawl 或 ClueWeb数据集都是网页，虽然它们的数据量非常大，但质量会相对较低。所以它们的影响值得进一步探索，如何在数据数量和质量之间取得一个好的平衡是一个重要的课题。另外，细分领域的训练数据是十分有限的，如何在预训练的框架下做domain adaptation 也是一个重要问题；
- 在预训练语言模型上，杨植麟表示还有 3 个非常有潜力的方向。首先即怎样在 Transformer 架构基础上构建更强的长距离建模方式，例如这个月 Facebook提出的Adaptive Attention Span和杨植麟之前提出的 Transformer-XL 都在积极探索。其次在于怎样加强最优化的稳定性，因为研究者发现在训练Transformer时，Adam 等最优化器不是太稳定。例如目前在训练过程中，一定要加上 Warm up机制，即学习率从0开始逐渐上升到想要的值，如果不加的话Transformer甚至都不会收敛。这表明最优化器是有一些问题的，warm up之类的技巧可能没有解决根本问题。最后模型可提升的地方在于训练效率，怎样用更高效的架构、训练方式来提升预训练效果。例如最近天



津大学提出的Tensorized Transformer，他们通过张量分解大大降低Muti-head Attention 的参数量，从而提高Transformer 的参数效率。