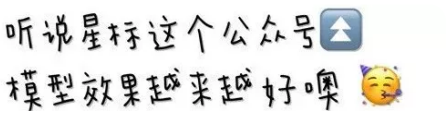
Transformer家族简史（PART I）

原创 kaiyuan NewBeeNLP 2020-12-04

收录于话题
#BERT巨人肩膀

44个



> NewBeeNLP原创出品
> 作者| kaiyuan

经过之前一段时间的 NLP Big Bang，现在相对比较平静了，Transformer 派已经占据了绝对的主导地位，在各类应用中表现出色。看标题大家也可以猜个差不多，整理了一系列自《Attention is all you need》之后的对 Vanilla Transformer 的改进论文，和大家一起梳理整个发展过程。这篇是第一趴，都来自ICLR。

OK，来看看今天的 Transformers：

1. 「 from UTS，ICLR2018」
2. 「 from UVA&Google，ICLR2019」
3. 「 from Google，ICLR2020」

我们建立了**自然语言处理、机器学习等**讨论组，欢迎大家加入讨论（人数达到上限，添加下方好友手动邀请）注意一定要**备注**姓名噢~

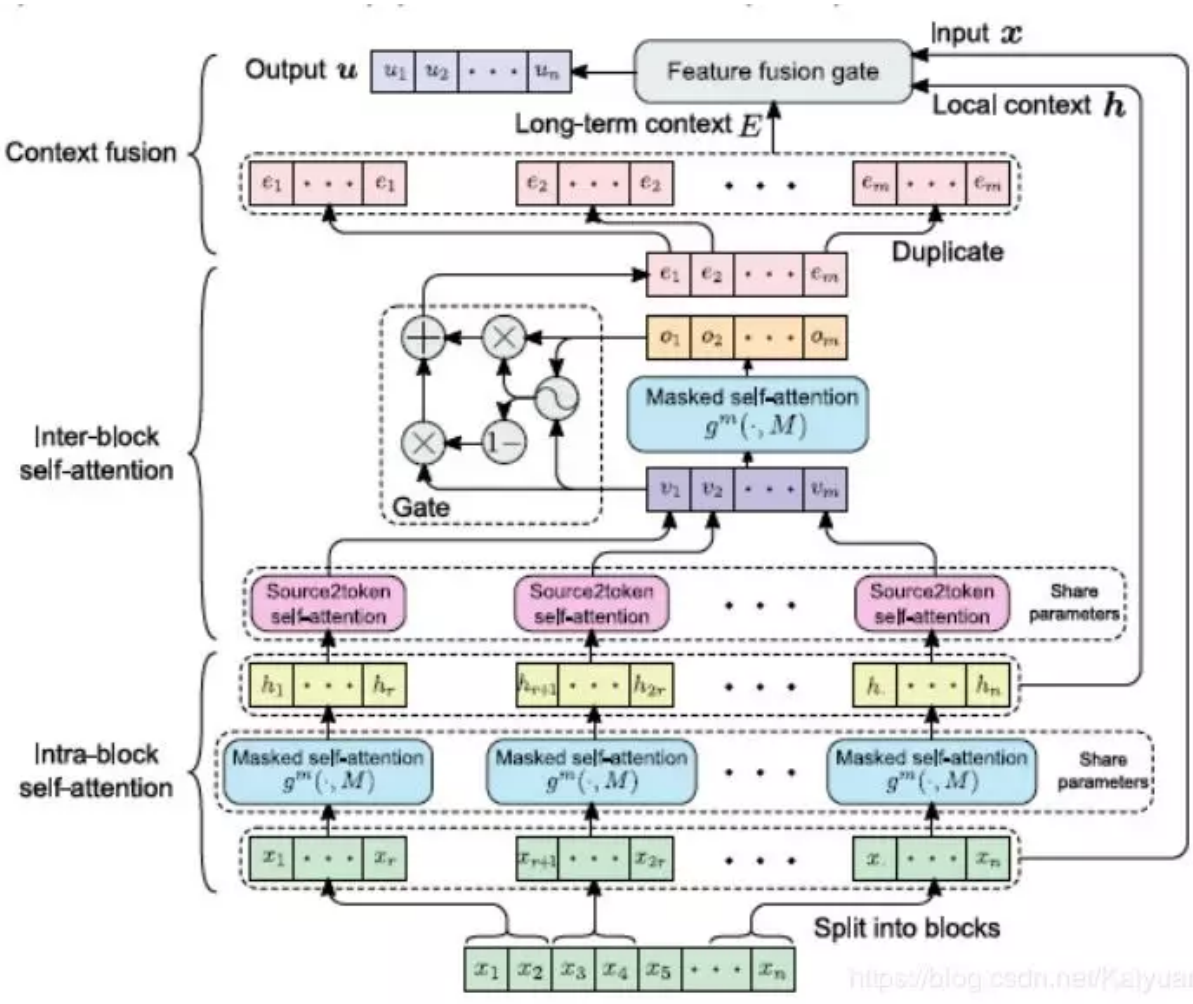
BI-DIRECTIONAL BLOCK SELF-ATTENTION FOR FASTAND MEMORY-EFFICIENT SEQUENCE MODELING

[1]

这篇论文首先分析了目前几大类特征抽取器 CNN、RNN、Attention 的优缺点，针对其不足提出了一种对 self-attention 的改进，**「双向分块自注意力机制（bidirectional block self-attention (Bi-BloSA)）」**。

1.1 Masked Block Self-Attention

其最主要的组件是**「掩码分块自注意力机制（masked block self-attention (mBloSA)）」**，基本思想是将序列分成几个等长的 block（必要时进行 padding），对每个单独的 block 内应用 self-attention（**「intra-block SAN」**）捕获局部特征，然后对所有 block 输出再应用 self-attention（**「inter-block SAN」**）捕获全局特征。这样，每个 attention 只需要处理较短的序列，与传统的 attention 相比可以节省大量的内存。最后通过 Context fusion 模块将块内 SAN、块间 SAN 与原始输入结合生成最终上下文表示。整体框架如下图所示：



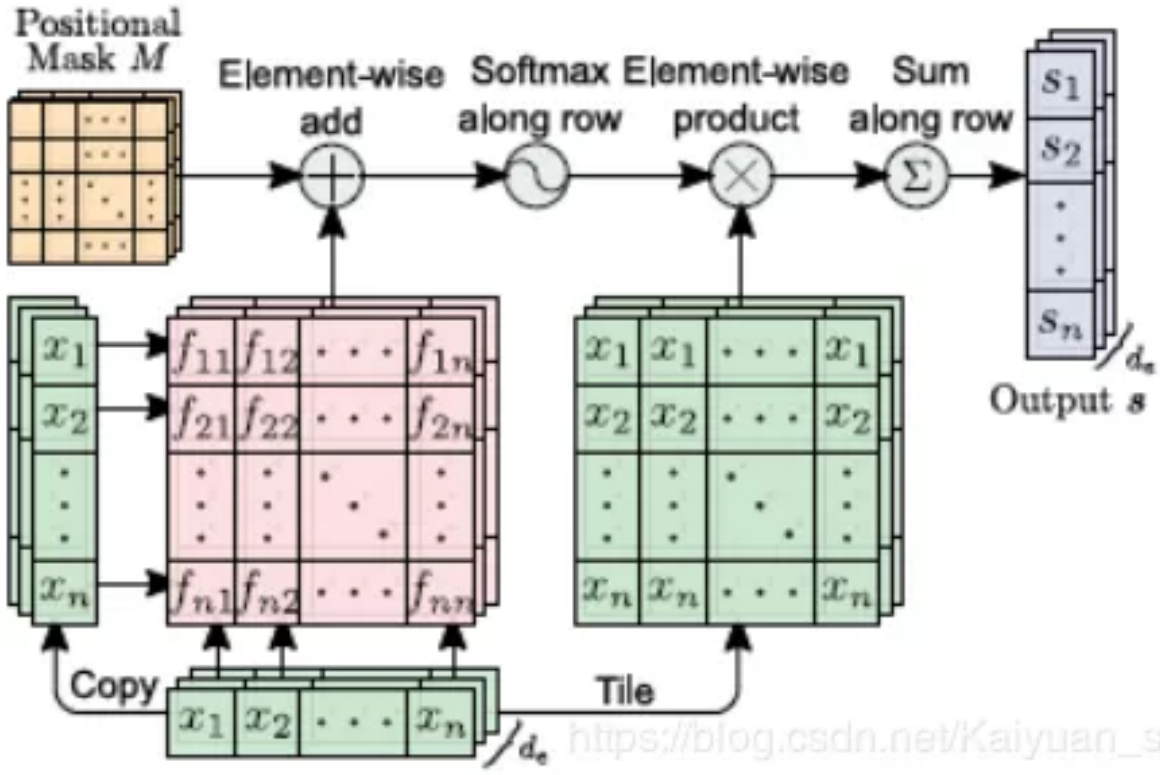
1.2 Masked Self-Attention

在传统的 self-attention 中时序信息是很难编码进去的，通过对某些位置进行掩码可以实现位置信息的融合，流程如下图。

$$f(x_i, x_j) = c \cdot \tanh\left(\left[W^{(1)}x_i + W^{(2)}x_j + b^{(1)}\right]/c\right) + M_{ij}1$$

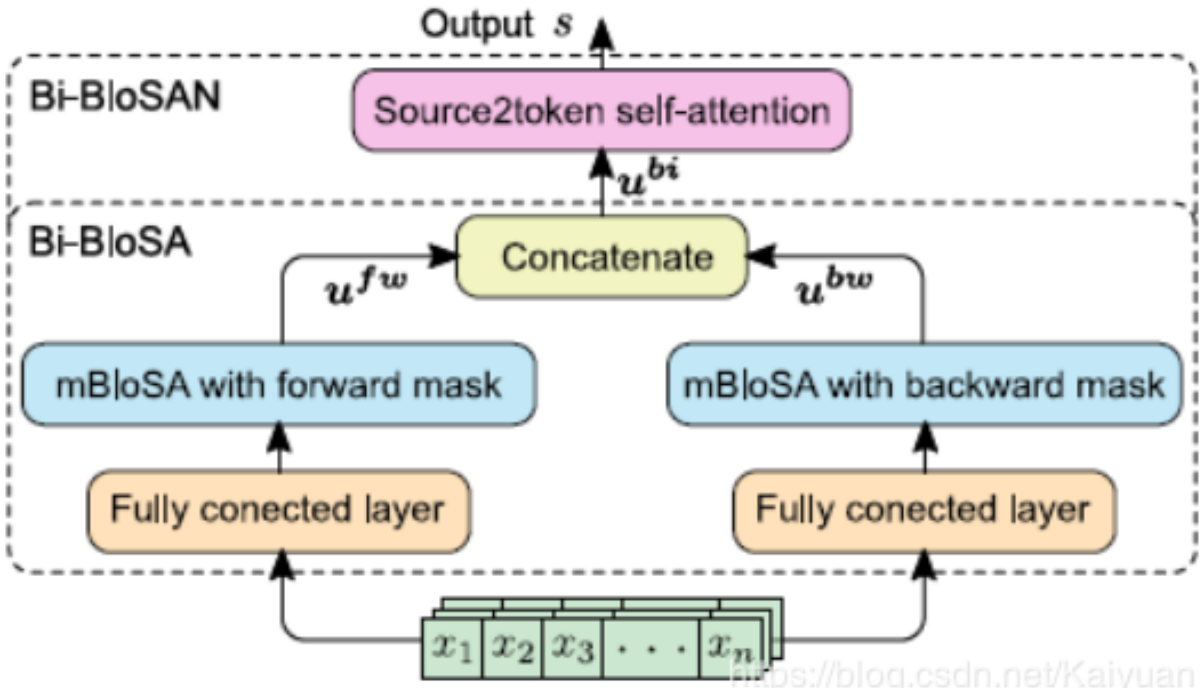
而建模双向信息，只需要将上式最后一项修改即可得到前向和后向 SAN

$$M_{ij}^{fw} = \begin{cases} 0, & i < j \\ -\infty, & \text{otherwise} \end{cases}$$
$$M_{ij}^{bw} = \begin{cases} 0, & i > j \\ -\infty, & \text{otherwise} \end{cases}$$



1.3 Bi-directional Block Self-Aattention Network

最终将上述模块整合后得到的网络框架如下图。



实验工作也非常充分，在九大公开数据集的多类 NP 任务中取得 SOTA，同时模型的计算和内存更高效。

1.4 Reference

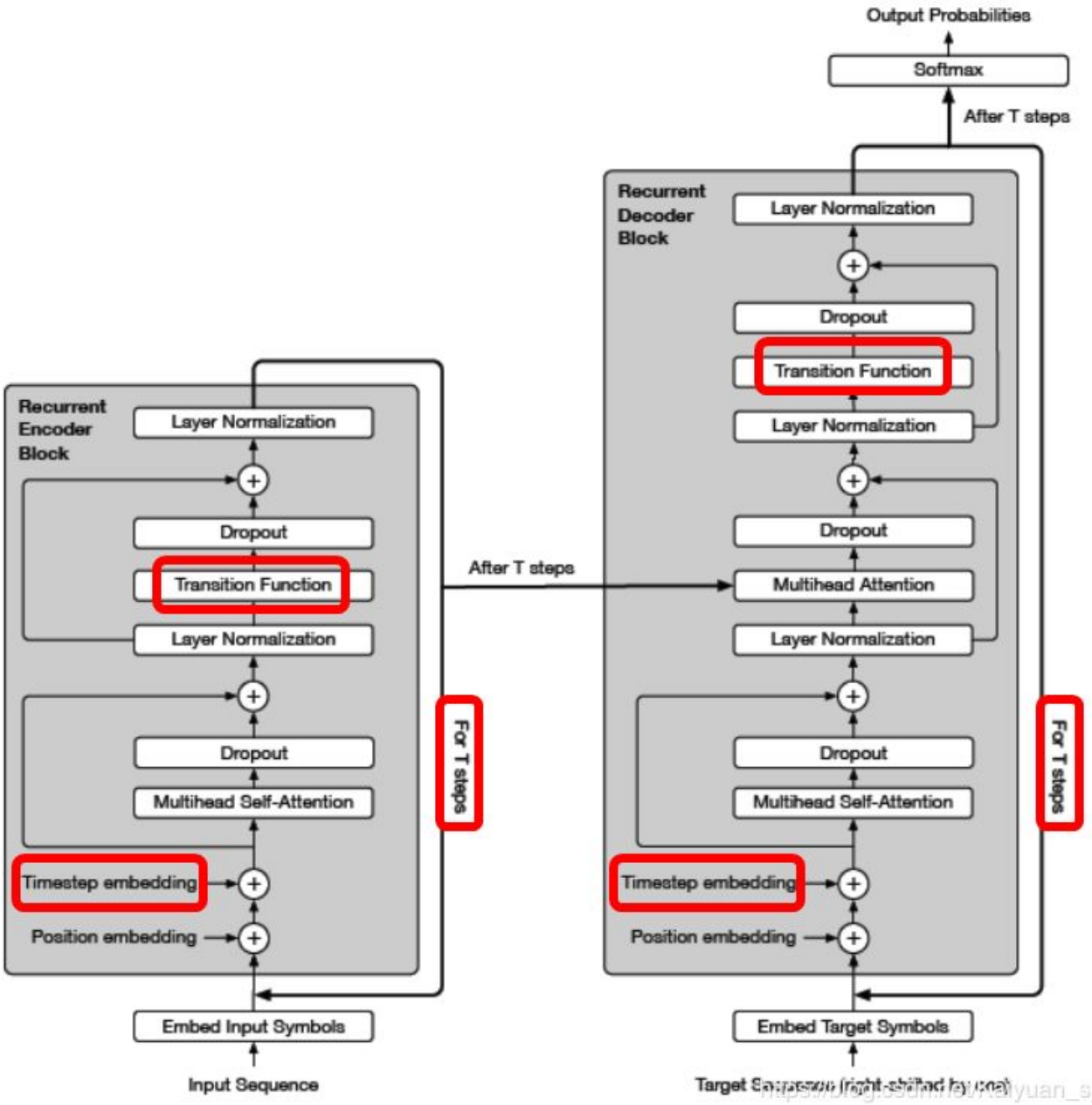
- Code Here[2]

UNIVERSAL TRANSFORMERS[3]

上一篇论文针对的是 Transformer 内存占用大、对长序列输入不友好的缺陷，除此之外，这篇论文指出其还存在着以下几个问题：

- 丢弃了 RNN 的归纳偏置（Inductive Bias），而这对一些任务至关重要（EMNLP 2018[4]的一篇文章对比了 Transformer 和 LSTM 得出该结论）。Inductive Bias 也可以理解为「**Prior Knowledge**」，可以用来预测从未遇到的输入的输出，即模型泛化的能力。（refer: Inductive Bias in Recurrent Neural Networks[5]）。
- 由于固定了 Transformer 的层数，使得模型计算低效。对于一个输入，有些部分是比较好理解的，有些部分是比较含糊不清的，如果采取相同的计算会导致无效计算。
- 原始 Transformer 是非图灵完备的。（这块没仔细看下面不会涉及）

针对以上存在的不足，论文提出了一种 Transformer 的改进模型：「**Universal Transformer（UT）**」，如下图所示。乍一看与 Transformer 非常相似（不同的地方在下图用红框框起来会在下文介绍），下面我们来仔细看看 UT 是如何解决上述缺陷的。



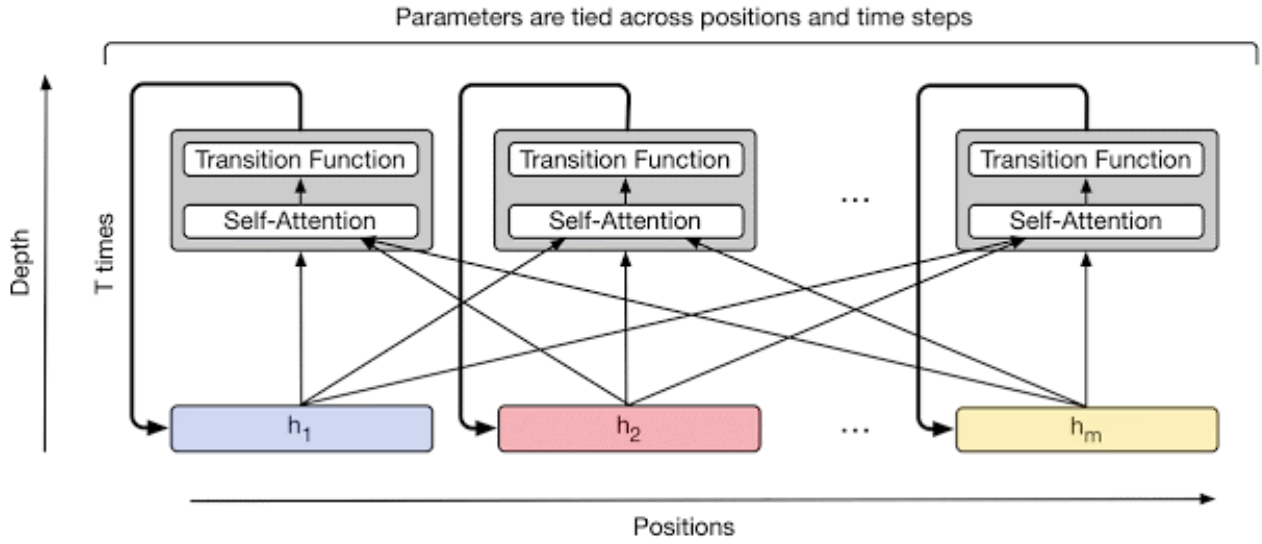
2.1 Parallel-in-Time Recurrence Mechanism

原始的 Transformer 堆叠层数是固定的（6 层或 12 层），为了结合 RNN 的优势，对每一个 token 设置了一个 Transition Function（**「参数共享」**），并且可以一次一次循环迭代计算（Transformer 是对 attention 层后直接输入一个 Dense Layer）。如下图，横坐标是输入序列的 token，纵坐标是循环迭代次数 Depth。每一次迭代是在整个序列上进行的，不同于 RNN 需要从左往右依次计算，大大提高了计算效率。当迭代次数 Depth 人为固定时，就变成了原始 Transformer。

假设输入为长度为 m 的序列，第 t 次循环后有：

$$H^t = \text{LAYER NORM}\left(A^t + \text{TRANSITION}\left(A^t\right)\right)$$
$$A^t = \text{LAYERNORM}\left(\left(H^{t-1} + P^t\right) + \text{MULTIHEADSELFATTENTION}\left(H^{t-}\right.\right.$$

其中 Transition Function 可以是深度可分离卷积[6] 也可以是全连接。



2.2 Coodinate Embeddings

在上面公式中有一个 P，指的是两种 embedding，（position，time）：

1. 「**position embedding**」和原始 Transformer 的一样；
2. 「**timestep embedding**」，其实本质上和 position embedding 一样，只不过这里是关注于循环步数 t，因为不同的循环次数学习的是不同层面的知识，这一信息可以提供给模型学习了几次，什么时候该停止；

$$P_{i,2j}^t = \sin\left(i/10000^{2j/d}\right) + \sin\left(t/10000^{2j/d}\right)$$
$$P_{i,2j+1}^t = \cos\left(i/10000^{2j/d}\right) + \cos\left(t/100000^{2j/d}\right)$$

2.3 Dynamic Halting

正如前面所说的，输入中的某些部分需要更多次的计算模型才可以理解。例如最常见的 I went to the bank to deposit money 里 bank 一词就需要更多次的运算。在 UT 中引入了Adaptive Computation Time (ACT)[7]来实现，ACT 可以基于模型在每步循环中预测的标量「**halting probability**」，动态调制处理每个输入 token 所需的计算步骤数（即响应时间「**ponder time**」）。UT 将动态 ACT 暂停机制分别应用于每个位置， 一旦特定的循环块停止，它将其状态复制到下一个步骤，直到所有块都停止，或者直到达到最大步数为止， 编码器的最终输出是以此方式产生的最后一层表示。Dynamic Halting 策略流程可以概括为：

1. 在第 t 步循环时，有第 t 步的 halting probability、第 $t-1$ 步的 token state 以及提前设置的超参停止阈值；
2. 利用 UT 计算当前 token state；
3. 利用全连接层和 sigmoid 激活函数计算 ponder time，ponder time 表示模型对每个 token 还需要多少次运算的预测；
4. 判断每个 token 当前是否需要停止，规则为：当 `(halting probability + pondering) > halt threshold` 时，停止；
   当 `(halting probability + pondering) ≤ halt threshold` 时，继续运算；
5. 对于那些仍然在运算的 token，更新停止概率：`halting probability += pondering`；
6. 当所有位置都停止或者模型达到循环最大步数时，结束。

整体流程如下面动图所示。

2.4 Reference

- Code Here[8]
- Moving Beyond Translation with the Universal Transformer[9]

REFORMER: THE EFFICIENT TRANSFORMER[10]

传统的 Transformer 仍然存在着一些问题，比如内存占用大、计算复杂度大、无法较好处理长文本等。针对以上不足，本文作者提出了一种更为高效的 Transformer，该方法的主要目

标是能够处理更长的上下文窗口，同时减少计算量和提高内存效率，具体做法如下:

- :boom:局部敏感哈希注意力替代传统注意力，将复杂度从 $O(L^2)$ 降低为 $O(L)$;
- 使用可逆层（reversible layers），在训练过程中只存储单层激活值，而不是 N 次;
- 将 FF 层中的激活进行切分并分块处理

3.1 关于注意力机制

Scaled Dot-Product Attention

Transformer 中最关键的部分就是其注意力机制，原始自注意力机制通过三个不同的映射矩阵将输入表示为 Q、K、V 三部分，然后计算下式:

$$\text{Attention}\left(Q, K, V\right)=\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

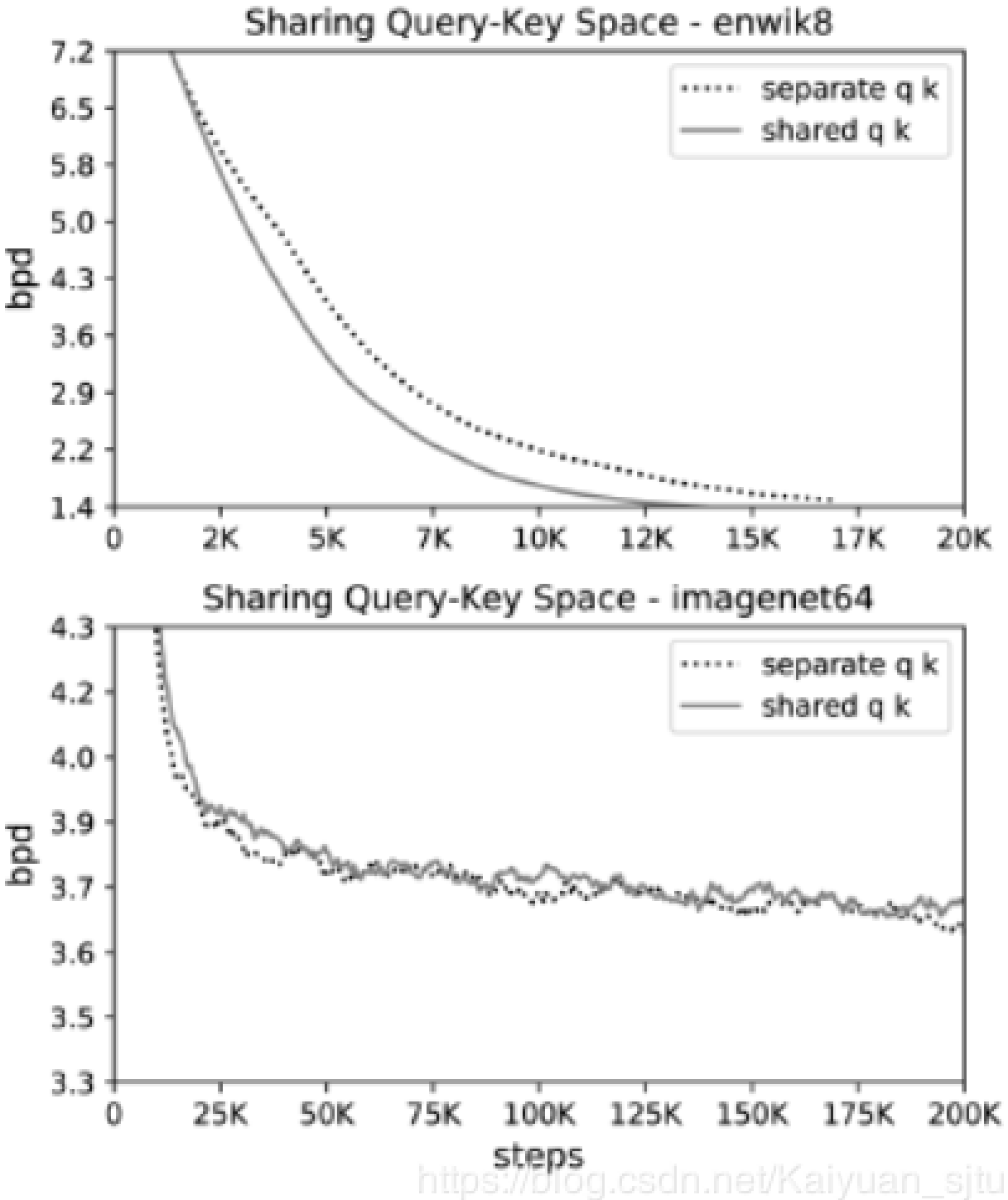
详细解释可以参考**illustrated-transformer**[11]、**illustrated-self-attention**[12]。

Memory-Efficient Attention

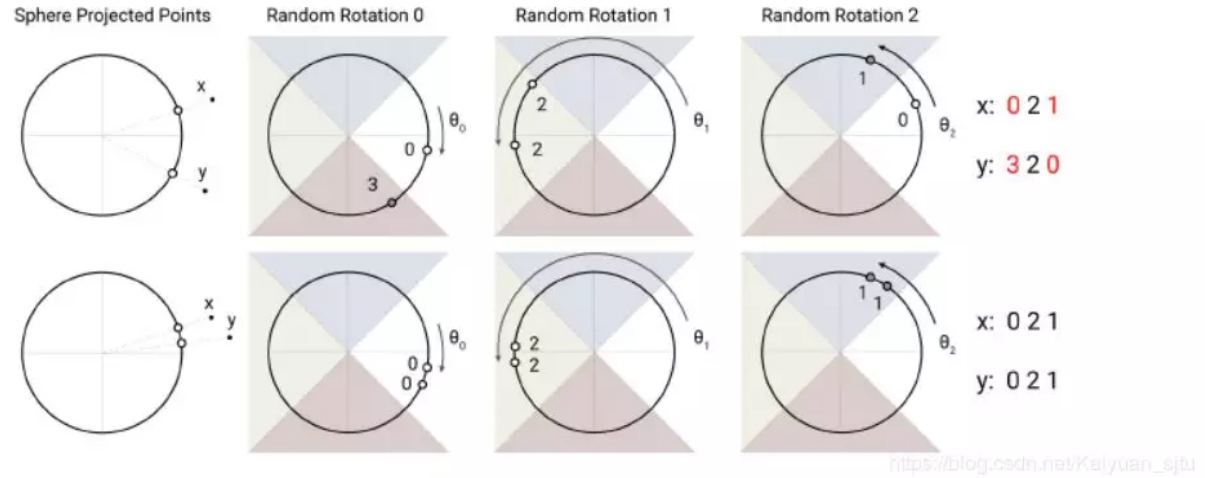
但是根据上式计算会发现模型占用内存非常大，主要问题在于 QK^T，假定 Q、K、V 的形状为 $[batch\_size, length, d_{model}]$，那么 QK^T 项的维度为 $[batch\_size, length, length]$。举个栗子，对于一个长度为 64K 的序列，即使 batc*size 为 1，这也会是一个 $64K*64K$ 的矩阵，并且序列越长，占用的内存会越大，这无疑会极大地限制 Transformer 的性能。但是注意，我们并不需要计算整个 Q，而可以选择对 Q 中的每一个 queryq_i 单独计算即可，然后在需要 BP 的时候再算一次（emmmm 感觉像是工程实现上的 trick）。虽然这样做不那么 computation-efficient，但是比较 memory-efficient，可以处理更长的序列。

Locality-Sensitive Hashing Attention

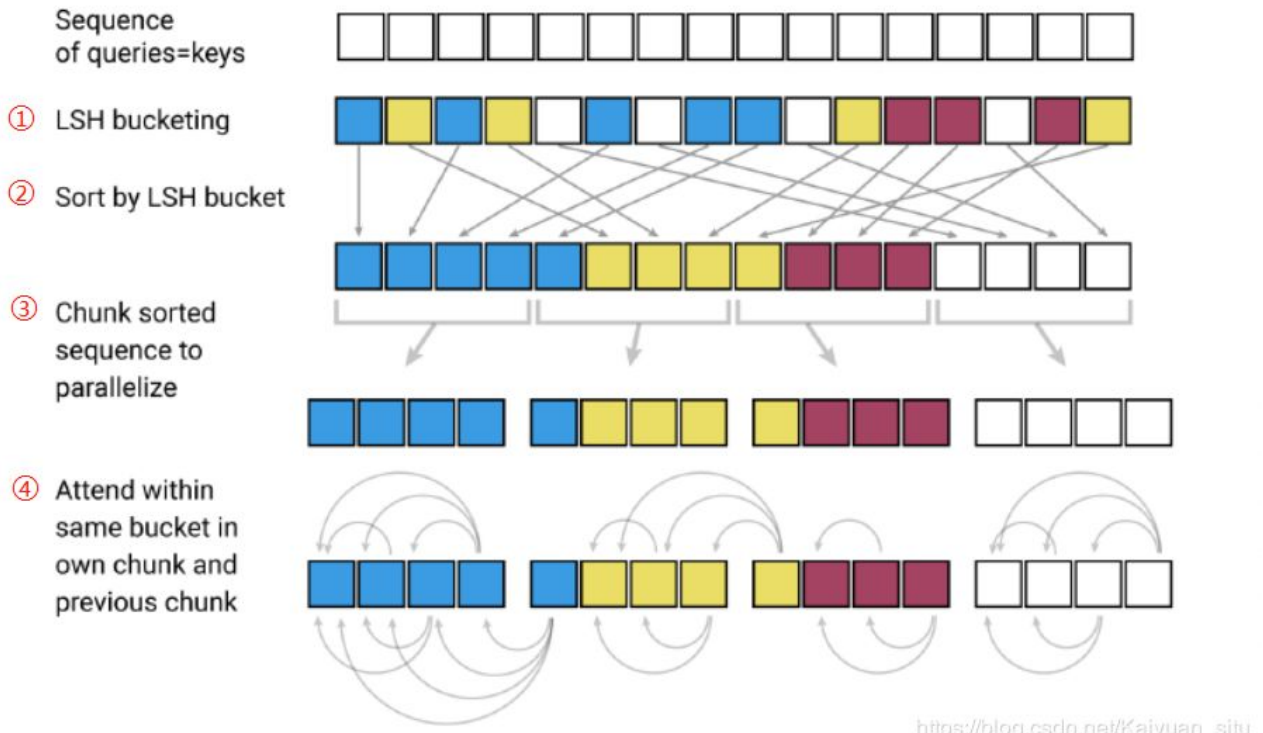
LSH attention 使用的是共享 Q、K 策略，这样可以有效降低内存占用，并且下图实验显示共享策略并不会比原始策略效果差。



注意到最终 QK^T 是会经过一个 softmax 函数，而 softmax 结果的关键在于那些比较大的输入，因此对于每一个 query q_i，我们只需要关注 K 中与其最接近的元素即可。那么问题是如何找到最接近的那些呢？这里就需要用到**「局部敏感哈希」**，简单来说就是使用哈希函数将邻近的向量以高概率映射到同一个哈希值，而距离远的向量高概率映射不到同一个哈希值。更具体地，文中使用的 LSH 哈希策略如下图：



最终 LSH Attention 的整体流程如下所示，① 局部敏感哈希分桶，不同的颜色代表不同的桶，相同的颜色代表相近的向量；② 序列重排，将具有相同颜色的向量放在一起；③ 切分序列，方便并行化计算；④ 在分出来的更小的区块中应用 attention，为了防止溢出向量，会考虑前一个 chunk 中的元素进行 attention

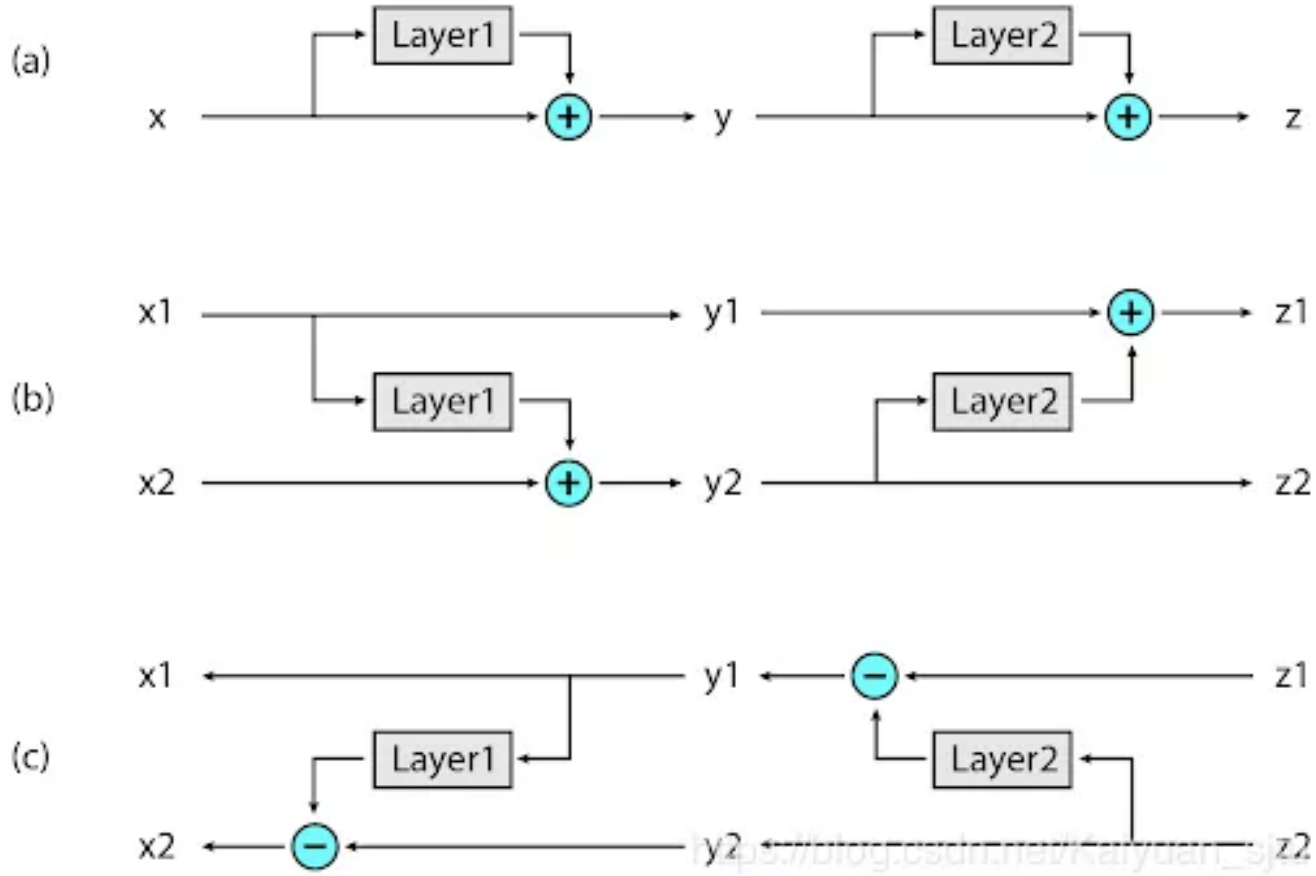


3.2 关于内存

虽然通过 LSH 策略降低了 attention 的复杂度，但是由于模型通常包含好多层，导致在训练过程中保存的中间值数量也非常巨大。为此，Reformer 使用了可逆 Transformer，不在内存中保存所有的值，而是在反向传播时按需计算得出，思想借鉴于RevNet[13]的可逆层。

（a）传统残差网络：上一层的激活用于更新下一层的输入。$y = x + F(x)$ （b）可逆残差网络：维护两组激活值，每层只更新其中的一组。$y_2 = x_2 + F(x_1)$ ；$z_1 = x_1 + G(y_2)$ （c）反

向：只需要减去激活值就可以恢复任意中间值



明白了上述思想，在 Reformer 中就是将 Attention 层和 FeedForward 层带入 F 和 G 即可。

3.3 reference

- Code Here[14]
- Reformer 应用[15]
- Reformer: The Efficient Transformer[16]
- Reddit 上关于 Reformer 的讨论[17]

本来还打算再加上 Transformer-XL 的，啊太长了就算了吧....

本文参考资料

[1] **BI-DIRECTIONAL BLOCK SELF-ATTENTION FOR FASTAND MEMORY-EFFICIENT SEQUENCE MODELING:** https://arxiv.org/abs/1804.00857

[2] **Code Here:** https://github.com/taoshen58/BiBloSA