超硬核 ICML'21 | 如何使自然语言生成提速五倍，且显存占用减低99%

原创 炼丹学徒 夕小瑶的卖萌屋 2021-06-09 12:25

文 | 炼丹学徒
编 | 小轶

我们忽略掉引言和介绍，直接把工作的效果丢上来，相信就足够令自然语言生成的相关同学心动——对于任何一个已有的Transformer生成模型，只需根据本文算法更改attention的计算顺序，就可以实现

- **成倍速度提升！**
- **显存使用量降低到原来百分之个位数！**
- **不需要重新训练！**
- **保证输出结果与原来完全一致！**

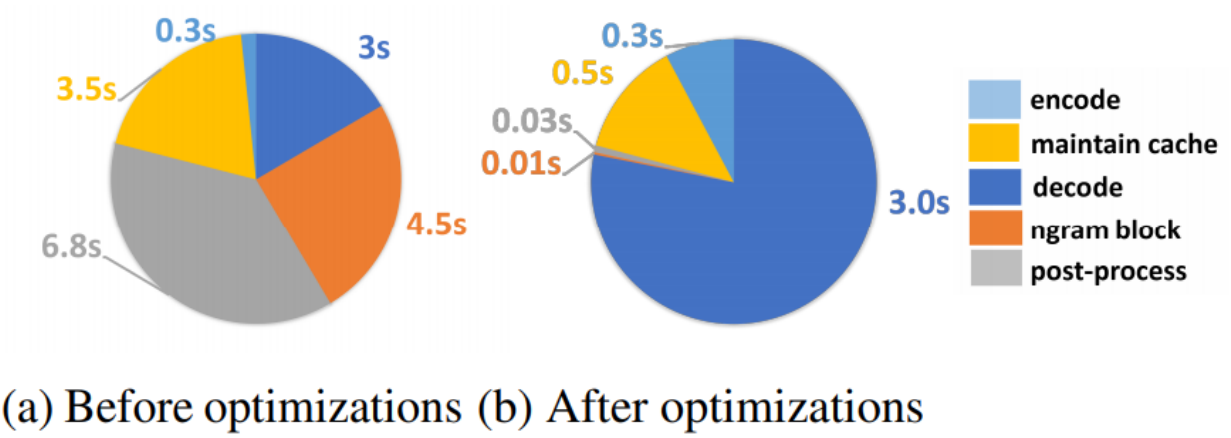
以BART为例，本文方法可以把显存使用率降低为原来的96分之一！是的，不需要在效率和质量中做权衡！无脑地将本文策略应用到你的Transformer里，庞大的自回归预训练的生成模型速度也会变得可以接受！你甚至可以大胆地去和蒸馏模型、剪枝模型、（半）非自回归模型比较速度。

仔细想想，我们自然语言生成的过程中，其实只有编码和解码是必须要计算的开销，而作者们发现，**显卡计算的时间远小于CPU操作和显存IO的时间，并最终进行理论分析降低显存的耗时、优化代码降低CPU操作的耗时，显著降低显存占用和提升生成速度**。本文正是聚焦在了显存优化的部分。

对于该方法的理论描述先是发表在了ICML 2021上。之后文章作者又将EL-Attention等相关技术封装成了一个工具包供大家一键调用，项目名称FastSeq，在2021 ACL Demo paper里获得了5 5 4的高分，并被两位审稿人推荐为best demo paper。

感兴趣的读者可以直接安装FastSeq工具包，**仅需要一行代码引入该库函数，只要你用的是常见的Facebook Fairseq或者Huggingface Transformers中的模型，import 完 FastSeq，甚至不需要**

改代码，就可以获得如下的加速效果：



(a) Before optimizations (b) After optimizations

论文题目：

EL-Attention: Memory Efficient Lossless Attention for Generation

FastSeq项目链接：

https://github.com/microsoft/fastseq

论文链接：

https://arxiv.org/pdf/2105.04779.pdf

Arxiv访问慢的小伙伴也可以在 **【夕小瑶的卖萌屋】**订阅号后台回复关键词 **【0609】** 下载论文PDF~

简单回顾Transformer的注意力计算

注意力层中，输入是Q，K，V即query、key、value矩阵，输出是Q、K、V隐状态维度相同，与Q的批大小、序列长度相同的隐状态矩阵。训练过程中，自注意力层Q=K=V=隐状态H；编码器-解码器注意力层中，Q=解码器的隐状态H，K=V=编码器的隐状态H。推断过程中，自注意力层K=V=H是已经输出的前文隐状态，Q是预测的下一个词；编码器-解码器注意力层中，K=V=H是编码器的隐状态，Q是解码器里预测的下一个词。

计算时，我们先把输入的Q，K，V线性变换，得到多头的隐状态变小一些的Q_i, K_i, V_i（i代表第几个头），然后对于每一个头，Q_i, K_i点乘归一获得注意力分布，用这个加权把V_i的值取过来，再把这个个头的低纬度信息线性方便换到之前Q，K，V的维度上作为这个头的隐状态计算结果，把每个头i的隐状态结果相加获得最终结果。在自回归推断时，无论在自注意力层，还是编码器-解码器注意力层中，Q都是一个单词，而K和V为输入编码后的隐状态或者已经解码的前文，都是比较长的内容。为了表达方便，后续描述中，$H = K = V$为经过线性变换前的隐状态，经过线性变换后的记为K_i, V_i，表示多头注意力中第i头的内容。（详细的公式描述可以看**推导章节**）

推断过程中，由于需要进行beam search，所以往往把编码器的输出重复beam size份并cache起来。其次，因为每次只能预测下一个词，所以自注意力层、编码器解码器注意力层里的K_i, V_i会被cache起

来避免重复计算，而Q_i是要预测的下一个词的隐状态，因此不可能被cache起来，是我们想要计算的东西。

Transformer Beam Search为啥这么慢

我们回忆一下Transformer生成训练的时候，forward一次的速度是非常快的，但是 **为什么真正去beam search 然后推断 inference 的时候却很慢**。

首先我们知道，训练和推断的时候，编码器端的运行是相同的，所以**变慢的原因都在解码器端**。即使我们设置beam search增加了一些计算量，但是实际上我们等待的时间远远大于理论上增加的计算量，把常见的生成任务的测试集完整的生成一遍结果，动辄几个小时的等待时间，到底花在哪里了呢？通过每个调用函数的时间消耗分析，作者得出了结论是：推断的代码中，把完整的矩阵运算打散成了每次只能预测后续一个词，零散的运算（从训练时 teacher forcing 的完整矩阵的Q,K,V计算，变成了推断时每次Q都只有一个词，去和K,V自回归地计算若干遍）从而显存的带宽成为了推断速度的瓶颈。

由于有cache技术的存在，beam search 时我们往往把计算过的隐状态存起来反复使用以避免重复计算（如果不cache，会更慢，因为要反复计算重复内容。后续分析会告诉我们，cache的速度瓶颈在显存IO，不cache的速度瓶颈在计算速度），**频繁的显存内容搬运和粗放的显存使用，导致GPU memory IO的时间超过了计算时间，显卡一直在等显存内容的搬运。如果再加上去除重复的输出等等CPU的操作，速度就更慢了。**

我们可以再看一下本篇推送引言部分的推断时长分布图。左侧优化前的推断时间里，CPU相关的后处理占用了最多的时间，消耗了6.8秒；其次是库函数中往往支持去除相邻的连续的多少个词的连续出现的问题，也就是图中的ngram block函数，去处理反复生成相同单词短语的问题，消耗了4.5秒。显存的搬运也是时间的大头，3.5秒，比真正解码计算的时间3s要多。编码只用了最少的时间，因为只有一次简单的forward。**所有的这些时间里，只有编码和解码是必须消耗的，EL-Attention解决掉了cache的问题，FaseSeq项目的其他部分解决了CPU相关计算的问题，最终把不必要的计算去除，优化达到耗时最少。本篇后续只介绍EL-Attention部分的提升。**

Transformer 推断过程显存IO瓶颈

根据论文作者的分析，Transformer自然语言生成时的显存IO瓶颈主要由以下三个问题组成：

1）在解码器中的*编码器-解码器注意力子层*，把编码器的输出经过每个子层不同的线性变换得到每一层都不一样的多头矩阵K_i, V_i矩阵存储。这就导致，**L层的解码器，需要把encoded hidden states存$2 \times L$遍**。甚至由于开了beam search，**当前Transformer的各个库函数中，解码器中的每一层都还把自己层计算出来的编码器K,V又要再重复beam size遍**，占用了大量的显存空间。解码器中的 *自注意力子层* 也有相同的问题，存储的同样是经过线性变换后的多头矩阵K_i, V_i。

2）在beam search过程中，因为**每一步的宽度搜索，都会导致beam candidates的得分发生变化从而导致重新排序**，以及生成结束符时从candidates队列向finished队列搬运的过程，从而**导致大量的**

memory IO消耗。

3）在显卡中，如果两个三维矩阵运算时，他们的第一维大小相同，则运算通过并行运算其中的各个二维矩阵运算完成。推断过程中，Q只是下一个词的隐状态，而K，V则显存占用比Q大得多，描述整个上文/输入信息。Q对K和V的运算，**反复加载大量显存占用的K和V，增大IO吞吐量负担。**（EL-Attention后面则减小query的第一维，增大query第二维，从而通过一次矩阵运算得到完整的各个头的计算结果，避免了反复加载key的值）

优化方案

后续的一切优化和计算的更改都是保证计算结果与原始Transformer完全一致的情况下展开和推导的

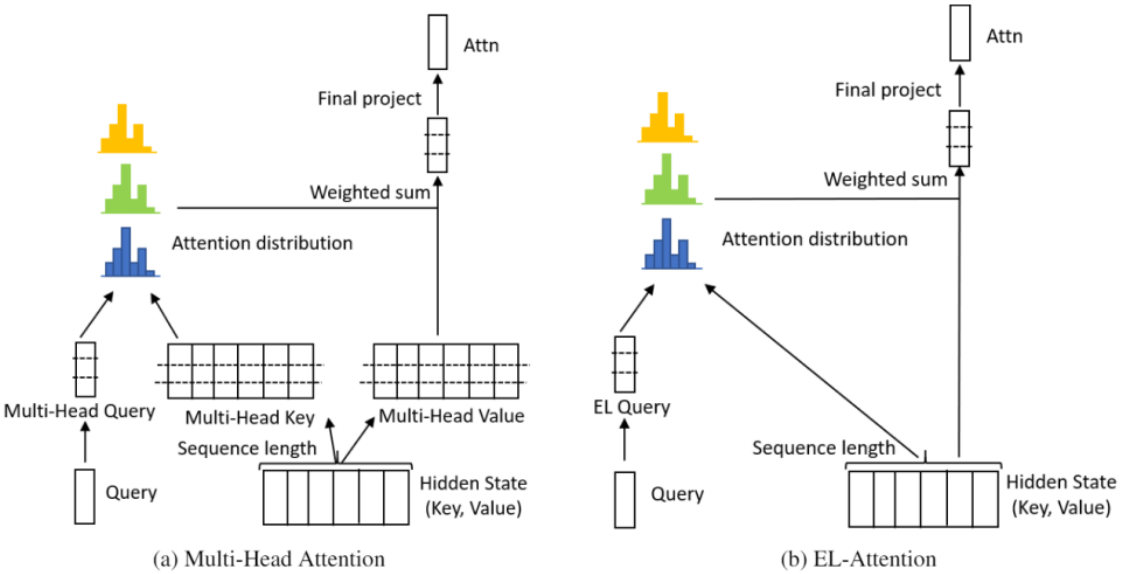


Figure 2. (a) Multi-head attention: query, key, and value are converted to their multi-head formats respectively, then the attention probability and attention result for each head are calculated; at last, results are aggregated to get the final output. (b) EL-attention: it only applies linear conversion to query. The hidden state, which is encoder output or previous layer output, is directly used as both key and value and shared for all heads.

为了推导出更适合推断过程的计算顺序，降低存储量，让矩阵的运算更高效，还能保持输出结果一致，本章节介绍EL-Attention如何进行MultiHead Attention（多头注意力计算）的等效替换。本章节里，仍然是使用Q, K ， V代表线性变换前的隐状态,Q_i, K_i, V_i代表线性变换后的低多头注意力里第i头的结果，$H = K = V$。

相比于存储解码器段每层计算过的K_i, V_i，EL-Attention只缓存经过线性变换之前的隐状态H，由当前要预测词的Q和线性变换前的H，直接计算得到注意力层的结果，从而将原始的注意力计算

$$MultiHead(Q, K, V) = \sum_{i=1}^{h} softmax(\frac{Q_i K_i^T}{\sqrt{d_k}})V_i W_i^O Q_i = QW_i^Q + b_i^Q, K_i = KW_i^K + b_i^K V_i$$

变成：

$$EL\_Attention(Q, K, V) = \sum_{i=1}^{h} FFN_i^O(softmax(\frac{FFN_i^Q(Q)K^T}{\sqrt{d_k}})V)H = K = V$$

很明显的我们看到，**原始计算里使用的是经过线性变换后的多头K_i, V_i进行计算，而EL-Attention中，则直接使用输入的$K, V(H=K=V)$进行计算。这就是本文的核心做法，只cache隐状态H而非多个低维度的多头K_i, V_i，从而进行更高效的矩阵运算，显著减少显存占用。**

其中，FFN_i^O和FFN_i^Q是两个线性变换。为了表达方便，我们略去了部分计算比如矩阵运算中的bias。完整的计算方法可以看下面的**推导**章节。此时，我们可以抛弃所有的计算过的K_i, V_i缓存，从而只缓存一份隐状态 $H=K=V$ 即可。其中，FFN_i^Q无需把Q计算到隐状态变小的多头状态进行零散矩阵运算，而是直接在原本的hidden size进行更加完整的矩阵运算，详细内容见推导章节如下：

推导

本章节我们一起看一下推导，**确保EL-Attention的计算结果是和MultiHead Attention完全一致的。**回顾传统的注意力计算方法，是将输入的 Q,K,V 线性变换得到维度更小，但是多份的多头隐状态Q_i, K_i, V_i，对于每一个头i，进行注意力计算，然后再用 W_i^O 线性变换到之前隐状态的大维度，把每个头的隐状态加起来。

$$MultiHead(Q,K,V)=\sum_{i=1}^{h} softmax(\frac{Q_iK_i^T}{\sqrt{d_k}})V_iW_i^OQ_i=QW_i^Q+b_i^Q, K_i=KW_i^K+b_i^KV_i$$

我们假设原本 H 的隐状态是m维的（$H=K=V$ 均为 m 维），多头数为 h，每一头的隐状态是 k 维的，则 $Q\in R^{1\times d_m}$，$H\in R^{n\times d_m}$，$W_i^Q, W_i^K, W_i^V\in R^{d_m\times d_k}$，$W_i^O\in R^{d_k\times d_m}$。

$$\begin{aligned}Q_iK_i^K&=\left(QW_i^Q+b_i^Q\right)\left(KW_i^K+b_i^K\right)^T\\&=\left(QW_i^Q+b_i^Q\right)(KW_i)^T+\left(QW_i^Q+b_i^Q\right)\left(b_I^K\right)^T\\&=FFN_i^Q(Q)K^T+Q_i\left(b_i^K\right)^T\end{aligned}$$

其中，$FFN_i^Q(Q)=(QW_i^Q+b_i^Q)(W_i^K)^T$，$Q_i=QW_i^Q+b_i^Q$。我们记 $Prob_i=softmax(\frac{Q_iK_i^T}{\sqrt{d_k}})$，则：

$$Prob_i\cdot V_i\cdot W_i^O=Prob_i(VW_i^V+b_i^V)W_i^O=Prob_i(VW_i^VW_i^O)+Prob_i\cdot Repeart(b_i^V)\cdot W_i^O$$

其中，$FFN_i^O(X)=XW_i^VW_i^O$

最终我们得到，

$$MultiHead(Q,K,V)=\sum_{i=1}^{h}FFN_i^O(Prob_i\cdot V)+\sum_{i=1}^{h}b_i^VW_i^O=EL\_Attention(Q,K,V)$$ 其中

在推断过程中，$H=K=V$。

降低了多少

直观的减少显存使用

显存占用：假设**编码器-解码器注意力层** beam search 的 size 大小为b，解码器L层，则原始的beam search会缓存 $2 \times b \times L$ 倍的encoded隐状态。其中的b倍是因为当前的库函数实现不佳，重复beam size份造成的，可以简单的优化掉，剩下的$b \times L$倍通过EL-Attention优化掉。**即，编码器-解码器注意力子层中，把encoded hidden states的显存占用降为**$\frac{1}{2 \times b \times L}$。类似的，**解码器的自注意力子层中，可以把显存占用降低1/2。**

详细的计算复杂度和显存优化

进一步分析，EL-Attention分析注意力计算中三个步骤的计算复杂度和显存占用复杂度。它把注意力的计算分解成三部分进行分析，第一部分是**Build Key and Value**(即原本计算中的把H线性变换到多头的K_i, V_i)，第二部分是**Build Query**（即原本计算中的把Q线性变换到多头的Q_i），第三部分是进行**注意力的计算**。

首先看**Build Key and Value**，传统的做法中，如果不cache，则需要每次进行$O(nd_m^2)$的计算，然后把计算结果存起来（存储复杂度$O(nd_m)$）。他的计算复杂度高，需要反复重新计算，cache则相反。而EL-Attention中，由于直接使用原始的输入K，V进行计算，无需计算出多头的那些K_i, V_i，因此计算和显存都为0。

其次是**Build Query**，对于要预测的下一个词的计算是绕不开的，所以无论传统做法中是否cache，Q都要被计算到多头的Q_i，因此计算复杂度和显存使用相同。EL-Attetnion的这一步是$FFN_i^Q(Q)$函数，由于多乘了$(W_i^K)^T$将多头的低hidden size隐状态变成原本的高hidden size计算，因此此处显存多使用了h（多头数）倍。然而这个其实很小，因为毕竟Q只有后续要预测的那一个单词的隐状态。最终是**注意力计算部分**，可以看到，因为没有缓存那些计算过的K_i, V_i，EL-Attention的计算复杂度增大为h倍，与此同时，显存消耗降低了。

Table 1. Computational and memory complexity for three groups of operations in attention. When cache key and value, multi-head attention only calculates key and value for the first time and re-uses in the following steps. We mark its computational complexity as 0 for simplicity. Our EL-attention does not depend on multi-head key and value from group 1. It has lower memory complexity for group 3 when using beam search, assuming sequence length is bigger than the number of heads. For notations, n is sequence length, d_m is model dimension, h is number of heads and x is beam size.

	① Build Key and Value Compute bound		② Build Query Compute bound		③ Calculate Attention Memory bound	
	Multi-Head Attention	EL Attention	Multi-Head Attention	EL Attention	Multi-Head Attention	EL Attention
Cache Key/Value	✗	✓	✗	✗ or ✓	✗	✗ or ✓	✗
Compute Complexity	$\mathcal{O}(nd_m^2)$	0	0	$\mathcal{O}(d_m^2)$	$\mathcal{O}(d_m^2)$	$\mathcal{O}(xnd_m)$	$\mathcal{O}(hxnd_m)$
Memory Complexity	$\mathcal{O}(nd_m)$	$\mathcal{O}(nd_m)$	0	$\mathcal{O}(d_m)$	$\mathcal{O}(hd_m)$	$\mathcal{O}(xnd_m)$	$\mathcal{O}(nd_m + hxd_m)$

为了比较上述三个步骤，用计算换取减少显存的操作是否收益大于付出，EL-Attention使用下面的图来表示这种权衡的收益。下图中，横轴是显存的使用量，纵轴是计算量，面积代表时间消耗。传统做法的时间消耗由三部分组成，图中为无边框的蓝色的大圈，灰色的大圈和橙色的小圈。EL-Attention的时间消耗由两部分组成，虚线边框的灰色小圈和橙色圈，可以看到，由于重新平衡了指令密度，显存消耗和计算消耗，总时间消耗（两个虚线边框圆的总面积）明显小于传统做法（三个无边框圆的总面积）。

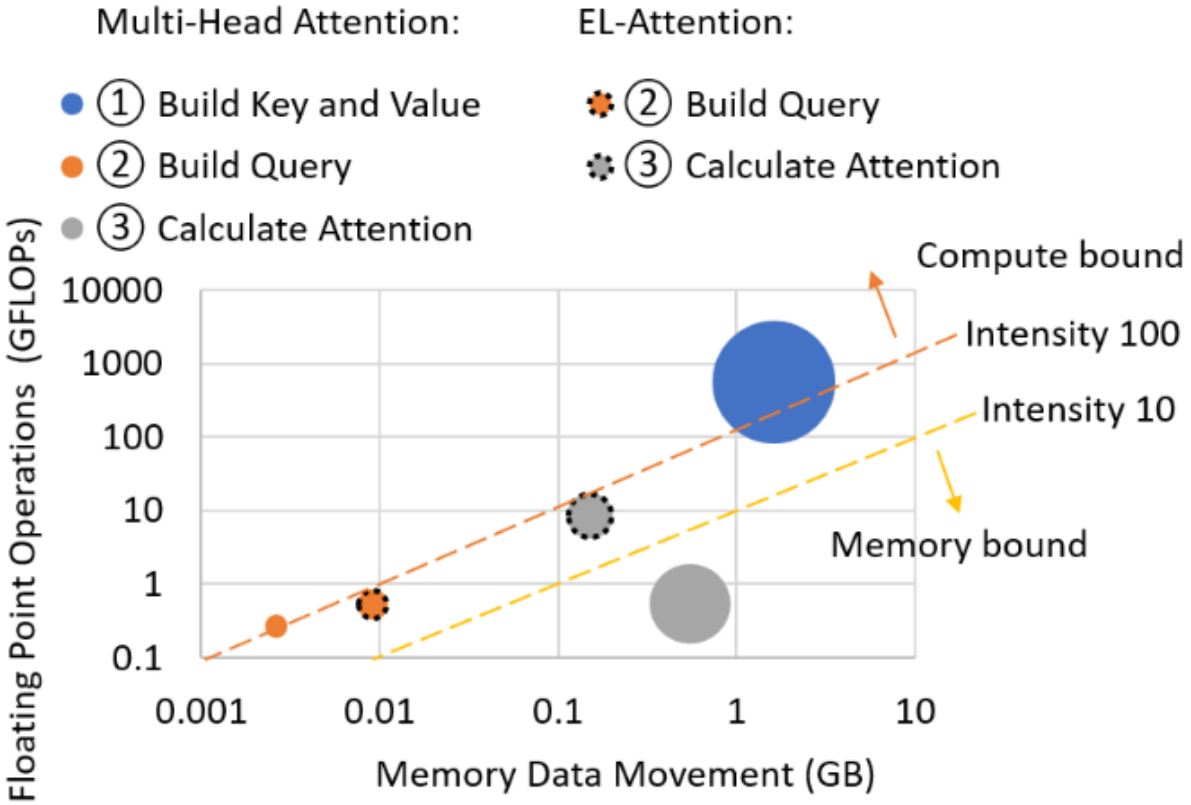


Figure 3. Performance data for operations in attention. X-axis is floating-point operations, y-axis is memory data movement, both are log scaled. Bubble area size represents execution time. EL-attention does not have group 1 operations for building key and value. See § 4.1.1 for details.

实验结果

首先，因为EL-Attention优化后的输出结果与优化前的Transformer模型完全一致，不需要重新训练，只需要优化推断的计算顺序，因此performance和输出结果，原始论文中没有展示。为了分析速度，首先，EL-Attention使用固定的假输入去分析速度影响，他固定了编码器端输入1024长，然后尝试不同的解码器段长度、不同的beam size去比较EL-Attention和原始attention的速度。我们可以看到，cache机制虽然增加了显存使用，但因为避免了重复计算，明显比不cahe的速度快，而EL-Attention则又明显的优于带cache的beam search生成。

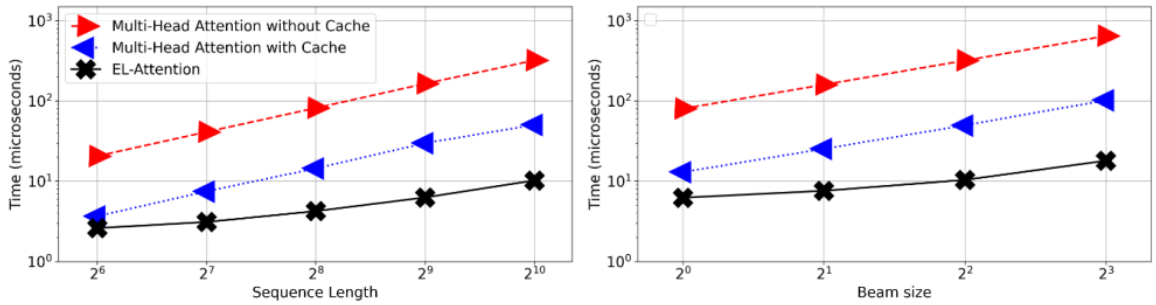


Figure 4. Comparison of attention execution time on various sequence lengths and beam sizes for EL-attention, multi-head attention without cache, and multi-head attention with cache. Axes are log-log scaled. Our method EL-attention has the fastest speed in all settings. Full details of this experiment can be found in § 4.1.2.

此外，EL-Attention在真实模型和数据集上开展试验。它使用Transformer，BART，GPT-2作为实验模型，其中Transformer和BART为编码器-解码器结构，GPT-2为只有解码器的结构，在SQuAD 1.1问题生成、XSum摘要任务、CNN/DM摘要任务上开展试验。**beam size越大，EL-Attention的加速效果越明显，EL-Attention作者很保守的把所有模型的beam size都开的比较小，只有4，就有了若干倍的速度提升。**

Table 2. Inference speed (samples/second) comparison across decoding methods, models, tasks, and computation precision. Speed up ratio is calculated for the same precision. Single precision marked as fp32 and half precision marked as fp16. See details in § 4.3.

Model	Parameter Number	Task	Multi-Head Attention fp16 (fp32)	EL-Attention fp16 (fp32)	Speed Up Ratio fp16 (fp32)
			Beam Search		
Transformer	270M	SQuAD 1.1	170.9 (86.6)	458.1 (173.6)	**2.7x** (2.0x)
$BART_{large}$	400M	XSum	14.7 (6.8)	69.4 (26.3)	**4.7x** (3.9x)
$BART_{large}$	400M	CNN/DailyMail	5.7 (3.4)	28.6 (12.2)	**5.0x** (3.6x)
$GPT\text{-}2_{small}$	117M	CNN/DailyMail	2.1 (1.5)	3.8 (2.5)	**1.8x** (1.7x)
$GPT\text{-}2_{medium}$	345M	CNN/DailyMail	0.9 (0.6)	2.0 (1.1)	**2.2x** (1.8x)
			Diverse Beam Search		
Transformer	270M	SQuAD 1.1	162.3 (82.1)	454.1 (171.8)	**2.8x** (2.1x)
$BART_{large}$	400M	XSum	15.8 (7.2)	71.9 (27.5)	**4.6x** (3.8x)
$BART_{large}$	400M	CNN/DailyMail	5.4 (3.2)	28.5 (12.0)	**5.3x** (3.8x)
			Greedy Search		
Transformer	270M	SQuAD 1.1	436.4 (190.3)	699.7 (260.3)	**1.6x** (1.4x)
$BART_{large}$	400M	XSum	42.6 (15.0)	107.8 (44.9)	**2.5x** (3.0x)
$BART_{large}$	400M	CNN/DailyMail	13.0 (7.0)	40.0 (19.5)	**3.1x** (2.8x)
$GPT\text{-}2_{small}$	117M	CNN/DailyMail	14.7 (9.0)	26.2 (13.6)	**1.8x** (1.5x)
$GPT\text{-}2_{medium}$	345M	CNN/DailyMail	5.9 (3.6)	10.4 (5.9)	**1.8x** (1.6x)

EL-Attention由于显著地减少了显存的占用，所以可以在有限的显存里，把batch size开大很多倍。通过增大batch size的方法，继续提高GPU的使用率和推断吞吐量（**下表的显存占用对比令人吃惊**）：

Table 3. Inference speed (samples/second) on different batch sizes. OOM means out of memory. All speedup ratios are compared to the same cell value of no cache under batch size 32.

	Multi-Head Attention		EL-Attention
Batch size	No Cache	Has Cache	
32	1.9 (1x)	5.7 (3x)	8.0 (4.2x)
64	1.9 (1x)	OOM	12.6 (6.6x)
128	OOM	OOM	21.3 (11.2x)
320	OOM	OOM	28.6 (15.1x)

Table 4. Comparison of memory sizes used for storing input related model states, see § 4.4 for detail.

Sequence Length	Multi-Head Attention	EL Attention
Batch size 32		

256	1.5 GB	0.02 GB
1024	6 GB	0.06 GB
Batch size 64		
256	3 GB	0.03 GB
1024	12 GB	0.13 GB
Batch size 320		
256	15 GB	0.15 GB
1024	60 GB	0.63 GB

总结

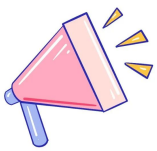
EL-Attention通过分析自然语言生成中的速度瓶颈，精确定位到了显存IO的问题，然后通过理论分析显存的计算方案，找到了若干致命问题，**对于已经训练好的模型，通过提出新的计算顺序和算法来在对原输出无损的情况下**，优化计算量和显存使用，从而达到了降显存、加速生成的效果。

萌屋作者：炼丹学徒

在微软搬砖的联培博士在读生，擅长烹饪和摸鱼，被迫掌握丰富的增肥和减肥经验。祝大家吃好喝好，减肥成功。

作品推荐

1. 把数据集刷穿是什么体验？MetaQA已100%准确率
2. Transformer太大了，我要把它微调成RNN



喜欢此内容的人还喜欢

若被制裁，中国AI会雪崩吗?

夕小瑶的卖萌屋