多头掩码自注意力机制

多头掩码自注意力机制（Multi-Head Masked Self-Attention）大语言模型架构中的一个核心组件。它帮助模型同时关注到文本序列中的多个不同位置的信息，从而捕捉更丰富的上下文信息。

相关概念

自注意力机制

自注意力机制计算输入文本序列中的每个token都与序列中所有其他token的相关性得分(即注意力分数)，从而得到加权组合这些token的权重。它的核心思想是能够使序列中的每个元素关注到该序列中的所有其他元素，并根据这些元素的相关性来调整自身表示。

自注意力机制的工作原理为：

- 接受输入表示：经过嵌入层的输入序列，每个token被表示为一个向量，结果是一个张量 X。
- 线性变换：为了计算注意力得分，原始输入 X 会被线性变换生成 3 个不同的向量集合——查询向量(Query, Q)、键向量(Key, K)和值向量(Value, V):

$$Q = XW^Q, K = XW^K, V = XW^V$$

- 计算注意力得分: 使用查询向量和键向量来计算注意力得分。具体来说，对于每个元素，它会基于其查询向量与其他所有元素的键向量之间的相似度（通常是点积）来计算得分。然后，这些得分通过Softmax函数进行归一化，确保它们可以作为加权系数。
- 加权求和: 使用上一步得到的归一化得分作为权重，对所有的值向量 V 进行加权求和，从而得到每个元素的新表示。这反映了每个元素在考虑了序列中所有其他元素的情况下更新后的表示。公式表达为：

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

其中，d_k 是键向量的维度，分母中的 $\sqrt{d_k}$ 用于缩放点积结果，防止过大导致Softmax函数饱和。

多头自注意力机制

多头自注意力(MHA)

多头自注意力机制（Multi-Head Self-Attention Mechanism）是自注意力机制的一种扩展形式，主要用于增强模型处理序列数据时捕捉信息的能力。它允许模型并行地执行多个自注意力过程，每个过程都专注于输入的不同部分或不同的表示子空间。

多头自注意力机制的工作原理如为：

- 分头处理：原始的自注意力机制只执行一次注意力计算。而在多头自注意力机制中，这个过程被重复 h 次(头的数量)，即每次使用不同的权重矩阵 $W_i^Q, W_i^K, W_i^V, i \in \{1, 2, ..., h\}$ 对输入进行变换。这意味着每“头”都会学习到输入序列的不同方面。
- 线性变换：对于每一头，输入 X 通过 3 个独立的线性变换生成查询(Query)、键(Key)和值(Value)向量：

$$Q_i = XW_i^Q, K_i = XW_i^K, V_i = XW_i^V$$

- 计算注意力得分：每“头”根据其特定的查询、键和值向量计算注意力得分，并应用Softmax函数得到加权系数。
- 加权求和：使用这些加权系数对值向量 V_i 进行加权求和，得到该头的输出：

$$Attention(Q_i, K_i, V_i) = softmax(\frac{Q_i K_i^T}{\sqrt{d_k}})V_i$$

- 合并与线性变换：将所有头的输出拼接在一起，并通过一个额外的线性变换整合这些信息(W^O 是一个用于组合不同头的信息的可学习权重矩阵)，形成最终的输出：

$$MHA(Q, K, V) = \text{Concat}[Attention(Q_1, K_1, V_1), \ldots, Attention(Q_h, K_h, V_h)]W^O$$

多头自注意力的优点有：

- 捕捉更丰富的信息：多头机制使得模型能够同时关注序列中的多个不同位置，从而捕捉序列中元素间更复杂的依赖关系。
- 提高表达能力：通过在不同的表示子空间中学习，模型可以学习到数据的不同特征，提高整体表达能力。
- 增强灵活性：各头可以独立学习不同的模式，增加了模型的灵活性和适应性。

上面介绍的多头自注意力机制简称 MHA，是最广泛使用的形式。除了MHA，还有多查询注意力(MQA)、分组查询注意力(GQA)等其他变体，它们在特定场景下有其优势。

多查询注意力(MQA)

MQA 是 MHA 的一种简化形式 ，旨在减少计算开销 。在 MQA 中 ，多个查询共享同一个键和值 。

工作原理 ：查询矩阵 Q 被分为多个头 ，但键矩阵 K 和值矩阵 V 在所有头之间共享 。这样可以显著减少 K 和 V 的存储和计算需求:

$$MQA(Q, K, V) = \text{Concat}[Attention(Q_1, K, V), \ldots, Attention(Q_h, K, V)]W^O$$

MQA 显著减少了内存占用和计算成本 ，特别是在解码器中的缓存优化场景中表现优异 。在某些任务中（如生成任务） ，性能接近于 MHA (略逊于 MHA) 。

分组查询注意力(GQA)

GQA 是介于 MHA 和 MQA 之间的一种折中方案 。它将注意力头分成若干组 ，每组内的头共享同一个键(Key)和值(Value) ，但不同组之间的 K 和 V 是独立的 。

工作原理 ：

- 将 h 个头分成 g 组($g < h$) ，每组包含 h/g 个头 。
- 每组内的头共享一组 K 和 V ，但不同组之间的 K 和 V 是独立的 。

这种设计在保持一定表达能力的同时 ，降低了计算和存储开销:

$$GQA(Q, K, V) = \text{Concat}[head_{group_1}, head_{group_2}, \ldots, head_{group_g}]W^O$$

其中 ，

$$head_{group_i} = \text{Concat}[Attention(Q_{1,i}, K_i, V_i), \ldots, Attention(Q_{h/g,i}, K_i, V_i)]$$

GQA 平衡了 MHA 和 MQA 的优缺点 ，在减少计算和存储开销的同时保留了较强的表达能力 ，其实现复杂度高于 MQA ，但仍比 MHA 简单 。

多头掩码自注意力机制

掩码(Masking)是一种重要的技术 ，用于控制模型在处理序列数据时只能关注特定范围内的信息 。例如 ，在自回归语言模型中 ，掩码可以防止模型在生成当前词时“看到”未来的词 。掩码的作用有 ：

- 防止未来信息泄露 ：在自回归任务(如文本生成)中 ，模型只能依赖当前词及其前面的词来预

测下一个词。如果不加掩码，模型可能会利用未来的信息，从而导致训练和推理不一致。

- 屏蔽无效信息：在某些场景下(如填充序列)，可以通过掩码屏蔽掉无意义的填充部分。

掩码的实现

掩码通常通过一个额外的掩码矩阵或张量来实现，该矩阵会在计算注意力得分时被加入到得分矩阵中，通过修改注意力分数，使得在计算注意力时某些位置的权重接近于 0。

上三角掩码(Causal Mask)

上三角掩码是最常见的掩码类型，用于自回归任务。它的目的是确保每个位置只能关注它之前的位置。假设输入序列的长度为 s，上三角掩码矩阵 $Mask$ 的形状为一个大小 $s \times s$ 的矩阵，其中对角线以下的元素为 0，其余位置为无穷小($-inf$)。

在计算注意力得分时，将上三角掩码矩阵与得分矩阵相加：

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}} + Mask)V$$

填充掩码

在自然语言处理任务中，输入序列通常会被填充到固定长度(padding)。为了屏蔽这些填充部分的影响，可以使用填充掩码。填充掩码矩阵 P 的形状与输入序列相同，填充位置为 0，非填充位置为无穷小($-inf$)。在计算注意力分数时，将此掩码加到得分矩阵上。

掩码矩阵和多头自注意力相加时的维度问题

多头注意力得分矩阵的维度

假设：批次大小 b，序列长度 s，头数 h，每个头的维度为 d。

注意力得分矩阵计算公式：

$$Score = QK^T/\sqrt{d}$$

其中，Q 和 K 的形状分别为 (b, h, s, d) (最初的时候形状应该是 (b, s, h, d)，但会被交换 1、2维，重塑为 (b, h, s, d))。在计算 QK^T 时，Q 和 K 的最后 2 个维度进行点积操作，结果是一个形状为 (b, h, s, s) 的张量，亦即 $s \times s$ 表示每个注意力头的得分矩阵，表示序列中每个位置对其他位置的关注程度。

掩码矩阵的维度

(1) 上三角掩码

上三角掩码矩阵 $Mask$ 的形状为一个大小 $s \times s$ 的矩阵 °为了与多头注意力得分矩阵兼容 ’上三角掩码需要广播到批次和头的维度 ’最终形状为 ：(b, h, s, s) °

(2) 填充掩码

填充角掩码矩阵 $Mask$ 的形状为一个大小 $b \times s$ 的矩阵 °为了与多头注意力得分矩阵兼容 ’填充掩码需要广播到头和序列长度的维度 ’最终形状为 ：(b, h, s, s) °

具体实现

初始化

(1) 初始化注意力机制所需的参数

- 多头自注意力机制中键 `值头的个数(与注意力权重的计算方式('MHA', 'MQA', 'GQA')相关) ’即Query的分组数量(组内共享键和值 ’Query分几组 ’就需要几个键和值头): g .
- 多头自注意力机制中头的个数(即Query的头的个数): h .
- 要重复多少次Key `Value头才能与Query头数量对应: h/g .
- 每个头的维度: $d//h$. 其中 ’ d 是嵌入维度 °

(2) 初始化注意力机制所需的层

- 初始化不带偏置的线性变换层: W_q, W_k, W_v, W_o .
- 初始化注意力分数计算使用的Dropout层: Dropout_attention.
- 初始化注意力结果计算使用的Dropout层: Dropout_output.

(3) 初始化掩码张量

- 生成一个形状为 $(1, 1, s, s)$ `值全为 $-inf$ 的张量 °
- 对生成的张量应用了上三角函数 torch.triu ’保留主对角线以上的部分(不包括主对角线) ’并将其他位置的值设置为 0 °这用于确保注意力是因果的⌒即 ’每个位置只能关注到其前面的位置 ’后面的是 -inf 被屏蔽掉了⌣ °
- 生成的掩码张量注册为模型的一个缓冲区 °在 PyTorch 中 ’缓冲区是一种特殊的张量 ’它不是模型的可训练参数 ’但与模型相关联 ’会在模型的前向传播中使用 °这里的掩码张量是

一个固定值 ’不需要参与梯度计算 ’因此适合注册为缓冲区 。

前向传播函数

(1) 入参

前向传播函数的入参包括 ：

- 输入张量: x ’形状为 (b, s, d) 。
- 位置编码张量: $freqs_cis$ ’ 形状为 $(s, d//g//2)$ 。
- 否使用缓存: use_kv_cache ’为布尔值 。
- 缓存的 Key 张量: $cache\_k$ 。
- 缓存的 Value 张量: $cache\_v$ 。

(2) 计算查询 `键和值的线性变换结果 ’并进行形状重塑

- 分别计算查询 `键和值的线性变换结果 ：

$$xq, xk, xv = W_q(x), W_k(x), W_v(x)$$

- 分别重塑查询 `键和值的线性变换结果:

$$xq = xq.\text{view}(b, s, h, d//h)$$

$$xk = xk.\text{view}(b, s, h/g, d//h)$$

$$xv = xv.\text{view}(b, s, h/g, d//h)$$

- 对 Query 和 Key 应用旋转位置编码 ’得到新的 xq, xk 。¬参见位置编码┴ .

(3) 使用和更新 Key 和 Value 缓存

- 使用缓存中的 Key 和 Value ($cache\_k, cache\_v$) ’与当前输入的 Key 和 Value (xk, xv) 进行拼接 ’并将拼接后的结果赋值给 xk, xv 。
- 如果 $use\_kv\_cache = \text{TRUE}$ ’则使用新的 xk, xv 更新 Key 和 Value 缓存 $cache\_k, cache\_v$ 。

(4) 重塑 Query, Key, Value 张量的形状

- 将查询张量 xq 的第 1 维和第 2 维进行交换(转置) ’转置后的形状为 $(b, h, s, d//h)$ 。

- 将键和值张量 xk, xv 进行复制操作 ’然后将复制结果的第 1 维和第 2 维进行交换(转置) ’转置后的形状为 $(b, h, s, d//h)$ 。

(5) 计算注意力分数

使用如下公式计算注意力分数:

$$output = \text{Dropout\_attention}[softmax(\frac{QK^T}{\sqrt{d_k}} + Mask)]V$$

其中 ’Q 的取值为是 xq ；K 的取值为是 xk ；V 的取值为是 xv ；$Mask$ 的取值为初始化的掩码张量 ；d_k 的取值为 $d//h$ ；$softmax$ 应用于张量的最后一个维度上 ；$softmax$ 的结果又被施加了一个 Dropout 层 ’然后才与 V 相乘 。

最后 ’将输出张量重塑(transpose和reshape操作)为 (b, s, d) 的形状 。

(6) 应用线性变换及Dropout

对 $output$ 应用线性变换 ’并施加 Dropout 层:

$$output = \text{Dropout\_output}[W_o(output)]$$

最终 ’返回注意力计算结果 $output$ 及更新后的Key和Value缓存 $cache\_k, cache\_v$ 。