Attention

1.1 讲讲对Attention的理解?

Attention机制是一种在处理时序相关问题的时候常用的技术，主要用于处理序列数据。

核心思想是在处理序列数据时，网络应该更关注输入中的重要部分，而忽略不重要的部分，它通过学习不同部分的权重，将输入的序列中的重要部分显式地加权，从而使得模型可以更好地关注与输出有关的信息。

在序列建模任务中，比如机器翻译、文本摘要、语言理解等，输入序列的不同部分可能具有不同的重要性。传统的循环神经网络（RNN）或卷积神经网络（CNN）在处理整个序列时，难以捕捉到序列中不同位置的重要程度，可能导致信息传递不够高效，特别是在处理长序列时表现更明显。

Attention机制的关键是引入一种机制来动态地计算输入序列中各个位置的权重，从而在每个时间步上，对输入序列的不同部分进行加权求和，得到当前时间步的输出。这样就实现了模型对输入中不同部分的关注度的自适应调整。

1.2 Attention的计算步骤是什么?

具体的计算步骤如下：

- **计算查询（Query）**：查询是当前时间步的输入，用于和序列中其他位置的信息进行比较。
- **计算键（Key）和值（Value）**：键表示序列中其他位置的信息，值是对应位置的表示。键和值用来和查询进行比较。
- **计算注意力权重**：通过将查询和键进行内积运算，然后应用softmax函数，得到注意力权重。这些权重表示了在当前时间步，模型应该关注序列中其他位置的重要程度。
- **加权求和**：根据注意力权重将值进行加权求和，得到当前时间步的输出。

Q	**Query 向量，表示当前关注点（query）**
K	Key 向量，表示输入中每个位置的“标签”或“索引”
V	Value 向量，表示输入中每个位置的实际内容

在Transformer中，Self-Attention 被称为"Scaled Dot-Product Attention"，其计算过程如下：

1. 对于输入序列中的每个位置，通过计算其与所有其他位置之间的相似度得分（通常通过点积计算）。
2. 对得分进行缩放处理，以防止梯度爆炸。
3. 将得分用softmax函数转换为注意力权重，以便计算每个位置的加权和。
4. 使用注意力权重对输入序列中的所有位置进行加权求和，得到每个位置的自注意输出。

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

1.3 Attention机制和传统的Seq2Seq模型有什么区别?

Seq2Seq模型是一种基于编码器-解码器结构的模型，主要用于处理序列到序列的任务，例如机器翻译、语音识别等。

传统的Seq2Seq模型只使用编码器来捕捉输入序列的信息，而**解码器只从编码器的最后状态中获取信息，并将其用于生成输出序列。**

而**Attention机制则允许解码器在生成每个输出时，根据输入序列的不同部分给予不同的注意力，从而使得模型更好地关注到输入序列中的重要信息。**

1.4 self-attention 和 target-attention的区别?

self-attention是指在序列数据中，**将当前位置与其他位置之间的关系建模。**它通过计算每个位置与其他所有位置之间的相关性得分，从而为每个位置分配一个权重。这使得模型能够根据输入序列的不同部分的重要性，自适应地选择要关注的信息。

target-attention则是指将**注意力机制应用于目标（或查询）和一组相关对象之间的关系。**它用于将目标与其他相关对象进行比较，并将注意力分配给与目标最相关的对象。这种类型的注意力通常用于任务如机器翻译中的编码-解码模型，其中需要将源语言的信息对齐到目标语言。

因此，**自注意力主要关注序列内部的关系，而目标注意力则关注目标与其他对象之间的关系。**这两种注意力机制在不同的上下文中起着重要的作用，帮助模型有效地处理序列数据和相关任务。

对比维度	Self-Attention（自注意力）	Target-Attention（目标注意力 / Cross-Attention）
定义	关注序列内部不同位置之间的关系	关注目标（Query）与其他对象（Key/Value）之间的关系
建模对象	同一序列内部的不同位置之间	查询（如 Decoder 的状态）与键值对（如 Encoder 输出）之间
Q、K、V 来源	均来自同一输入序列（如 Encoder 输出）	Q 来自目标（如 Decoder），K/V 来自相关对象（如 Encoder）
主要用途	捕捉长距离依赖，增强模型对序列内部结构的理解	在生成过程中，让模型关注最相关的输入信息（如翻译时对齐）
常见场景	Transformer Encoder、BERT、GPT 等	Transformer Decoder、机器翻译、文本摘要等

1.5 在常规attention中，一般有k=v，那self-attention可以吗?

self-attention实际只是attention中的一种特殊情况，因此k=v是没有问题的，也即K，V参数矩阵相同。实际上，在Transformer模型中，Self-Attention的典型实现就是k等于v的情况。Transformer中的Self-Attention被称为"Scaled Dot-Product Attention"，其中通过将词向量进行线性变换来得到Q、K、V，并且这三者是相等的。

1.6目前主流的attention方法有哪些?

- **Scaled Dot-Product Attention**: 这是Transformer模型中最常用的Attention机制，用于计算查询向量（Q）与键向量（K）之间的相似度得分，然后使用注意力权重对值向量（V）进行加权求和。
- **Multi-Head Attention**: 这是Transformer中的一个改进，通过同时使用多组独立的注意力头（多个QKV三元组），并在输出时将它们拼接在一起。这样的做法允许模型在不同的表示空间上学习不同类型的注意力模式。

- **Relative Positional Encoding**: 传统的Self-Attention机制在处理序列时并未直接考虑位置信息，而相对位置编码引入了位置信息，使得模型能够更好地处理序列中不同位置之间的关系。
- **Transformer-XL**: 一种改进的Transformer模型，通过使用循环机制来扩展Self-Attention的上下文窗口，从而处理更长的序列依赖性。

1.7 self-attention 在计算的过程中，如何对padding位做mask?

在 Attention 机制中，同样需要忽略 padding 部分的影响，这里以transformer encoder中的self-attention为例：self-attention中，Q和K在点积之后，需要先经过mask再进行softmax，因此，**对于要屏蔽的部分，mask之后的输出需要为负无穷**，这样softmax之后输出才为0。

> **Note**
>
> 虽然 Transformer Encoder 可以看到整个输入序列（即非自回归），但为了防止模型关注到无意义的填充部分（padding）（padding是由切分batch的时候引入），在 Self-Attention 计算中仍然需要对 padding 位进行 **mask 处理** 。这个 mask 的作用是让 softmax 忽略这些位置。

1.8 深度学习中attention与全连接层的区别何在?

> **Note**
>
> **全连接层是“固定权重，平均处理”，而 Attention 是“动态选择，重点聚焦”。它通过 Query 和 Key 的比对，找出最相关的 Value，从而让模型更聪明地理解上下文。 全连接层虽然不像 Attention 那样动态地分配注意力，但它确实可以看作是一种“静态注意力机制” —— 它给输入的不同维度分配了固定的权重，而不是像 Attention 那样根据上下文动态决定“该注意谁”。**

Transformer 论文重新定义了 Attention 机制，并引入了 **QKV 的概念** ，也就是 **Query（查询）** 、**Key（键）** 和 **Value（值）** 。

如果我们用这套 QKV 框架去理解传统的 RNN 中使用的 Attention 机制，就会发现它的过程是这样的：

- RNN 编码器每一步都会输出一个隐藏状态；
- 在解码阶段，RNN 解码器当前时刻的状态被当作 **Query** ；
- 这个 Query 去“查看”编码器所有时间步的隐藏状态，其中每个隐藏状态作为一个 **Key** ；
- Query 和 Key 相互作用，计算出一组 **Attention Score** ，表示哪些位置的信息与当前任务最相关；
- 最后，这些 Attention Score 被用来对对应的 **Value** （也就是编码器各时间步的隐藏状态）进行加权求和，得到最终的上下文向量（context vector）。

那么如果我们不用 Attention，而只使用全连接层呢?

很简单：全连接层中并没有 Query 和 Key 的概念，它只有一个类似 Value 的输入。也就是说，它只是简单地给每个 Value 分配一个固定的权重，然后加权求和。如果是 Self-Attention 场景，甚至这个加权过程都可以省略，因为 Value 本身就已经是从原始输入中通过加权聚合得来的。由此可见，**Attention 和全连接层之间最大的区别就在于有没有 Query 和 Key** 。而这两者正是生成 **Attention Score** 的核心机制 —— 也是整个 Attention 最关键的部分。 而在 Query 和 Key 之间，**Query 更加关键** 。因为它是整个注意力机制的“锚点”，所有的 Attention Score 都是基于这个 Query 与其他 Key 的相似性或距离计算出来的。 可以说，在任何基于 Attention 的算法中，都离不开 Query 这个核心概念，而这一点在全连接层中是完全不存在的。

Transformer

2.1 transformer中multi-head attention中每个head为什么要进行降维?

在Transformer的Multi-Head Attention中，对每个head进行降维是**为了增加模型的表达能力和效率。**

每个head是独立的注意力机制，它们可以学习不同类型的特征和关系。通过使用多个注意力头，Transformer可以并行地学习多种不同的特征表示，从而增强了模型的表示能力。然而，在使用多个注意力头的同时，注意力机制的计算复杂度也会增加。原始的Scaled Dot-Product Attention的计算复杂度为$O(d^2)$，其中d是输入向量的维度。如果使用h个注意力头，计算复杂度将增加到$O(hd^2)$。这可能会导致Transformer在处理大规模输入时变得非常耗时。

为了缓解计算复杂度的问题，Transformer中在每个head上进行降维。在每个注意力头中，输入向量通过线性变换被映射到一个较低维度的空间。这个降维过程使用两个矩阵：一个是查询（Q）和键（K）的降维矩阵W_Q和，W_K另一个是值（V）的降维矩阵W_V。通过降低每个head的维度，Transformer可以在**保持较高的表达能力的同时，大大减少计算复杂度。**降维后的计算复杂度为：$(h\hat{d}^2)$，其中 $\hat{d}$ 是降维后的维度。通常情况下，$\hat{d}$ 会远小于原始维度 d，这样就可以显著提高模型的计算效率。

2.2 transformer在哪里做了权重共享，为什么可以做权重共享?

1. 词嵌入层与输出层的权重共享

在标准 Transformer 的语言模型中（如 BERT、T5），词嵌入层（Embedding）和输出层（Output Projection）的权重共享是一种常见优化策略。具体来说，输入的 token 首先通过一个嵌入矩阵 W_e 转换为向量（即词嵌入），而在模型生成输出时，最终的线性层（用于预测下一个 token）会直接复用这个嵌入矩阵 W_e 的转置（或本身）。例如，输出 logits 的计算公式为：

$$\text{Logits} = \text{Decoder}(\text{output}) \times W_e^T$$

这种共享方式的核心优势在于：

- **减少参数量**：嵌入层和输出层通常是模型中参数量最大的部分，共享权重能显著降低内存占用。
- **统一语义空间**：输入和输出的 token 共享相同的语义表示，有助于模型更好地理解上下文。

但这一策略并非标准 Transformer 的默认设计，而是特定模型（如 BERT、T5）的优化手段，且要求词嵌入维度与隐藏层维度一致。

2. 位置编码的权重共享

标准 Transformer 中的位置编码（Positional Encoding）是固定或可学习的向量，用于为输入序列添加位置信息。其共享机制如下：

- **同一层内部**：所有 token 的位置编码是共享的，**所有 token 使用同一个参数来源（如嵌入矩阵或函数）来生成它们的位置编码向量**（例如，所有位置的 token 使用相同的正弦/余弦函数生成的位置编码，或通过可学习的嵌入矩阵共享权重）。
- **跨层或跨模块（编码器 ↔ 解码器）**：**不共享。**编码器和解码器的位置编码是独立设计的，因为它们处理的上下文不同（编码器处理源语言，解码器处理目标语言）。

3. Encoder 和 Decoder 的词嵌入共享

标准 Transformer 中，编码器和解码器的词嵌入是独立的。例如英德翻译任务中，编码器处理英文，解码器生成德文，词表不同无法共享。但在多语言模型（如 mBART、mT5）中，若源语言和目标语言使用统一词表（如 BPE 子词），词嵌入层可共享。这对多语言任务有益（如共享数字、标点等通用子词），但对差异大的语言对（如中英）可能引入噪声，且共享会增加 softmax 层计算负担。

4. 解码器自注意力层的权重共享

在标准 Transformer 中，解码器的自注意力层（Self-Attention）并未共享 QKV（Query、Key、Value）的权重矩阵。每层的 Q、K、V 均通过独立的线性变换生成：

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$

这种设计允许模型捕捉丰富的上下文信息。然而，**在轻量级变体模型（如 ALBERT）中，部分层的 QKV 权重可能被共享，以压缩模型规模。例如，ALBERT 的每一层共享相同的 QKV 权重，从而大幅减少参数数量，但代价是模型表达能力受限。**因此，解码器自注意力层的权重共享并非标准 Transformer 的默认行为，而是特定优化策略，需结合任务需求谨慎使用。

2.3 transformer的点积模型做缩放的原因是什么?

使用缩放的原因是为了控制注意力权重的尺度，以避免在计算过程中出现**梯度爆炸**的问题。

Attention的计算是在内积之后进行softmax，主要涉及的运算是 $e^{q\cdot k}$，可以大致认为内积之后、softmax之前的数值在 $-3\sqrt{d}$ 到 $3\sqrt{d}$ 这个范围内，由于 d 通常都至少是64，所以 $e^{3\sqrt{d}}$ 比较大而 $e^{-3\sqrt{d}}$ 比较小，因此经过softmax之后，Attention的分布非常接近一个one hot分布了，这带来严重的梯度消失问题，导致训练效果差。（例如 $y=\mathrm{softmax}(x)$ 在 $|x|$ 较大时进入了饱和区，x 继续变化 y 值也几乎不变，即饱和区梯度消失）

相应地，解决方法就有两个:

像NTK参数化那样，在内积之后除以 $\sqrt{d}$，使 $q\cdot k$ 的方差变为1，对应 e^3, e^{-3} 都不至于过大过小，这样softmax之后也不至于变成one hot而梯度消失了，这也是常规的Transformer如BERT里边的Self Attention的做法

另外就是不除以 $\sqrt{d}$，但是初始化 q,k 的全连接层的时候，其初始化方差要多除以一个 d，这同样能使得使 $q\cdot k$ 的初始方差变为1，T5采用了这样的做法

BERT

3.1 BERT用字粒度和词粒度的优缺点有哪些?

BERT可以使用字粒度（character-level）和词粒度（word-level）两种方式来进行文本表示，它们各自有优缺点：

字粒度（Character-level）：

- **优点**：处理未登录词（Out-of-Vocabulary，OOV）：字粒度可以处理任意字符串，包括未登录词，不需要像词粒度那样遇到未登录词就忽略或使用特殊标记。对于少见词和低频词，字粒度可以学习更丰富的字符级别表示，使得模型能够更好地捕捉词汇的细粒度信息。
- **缺点**：计算复杂度高：使用字粒度会导致输入序列的长度大大增加，进而增加模型的计算复杂度和内存消耗。需要更多的训练数据：字粒度模型对于少见词和低频词需要更多的训练数据来学习有效的字符级别表示，否则可能会导致过拟合。

词粒度（Word-level）：

- **优点**：计算效率高：使用词粒度可以大大减少输入序列的长度，从而降低模型的计算复杂度和内存消耗。学习到更加稳定的词级别表示：词粒度模型可以学习到更加稳定的词级别表示，特别是对于高频词和常见词，有更好的表示能力。
- **缺点**：处理未登录词（OOV）：词粒度模型无法处理未登录词，遇到未登录词时需要采用特殊处理（如使用未登录词的特殊标记或直接忽略）。对于多音字等形态复杂的词汇，可能无法准确捕捉其细粒度的信息。

3.2 BERT的Encoder与Decoder掩码有什么区别?

Encoder主要使用自注意力掩码和填充掩码，**Decoder 使用自注意力掩码（Causal Mask）来防止未来信息泄露；同时使用编码器-解码器注意力掩码来屏蔽 Encoder 输入中的填充部分（padding），而不是为了防止未来信息泄露。**这些掩码操作保证了Transformer在处理自然语言序列时能够准确、有效地进行计算，从而获得更好的表现。

> **Note**
>
> **Causal Mask 是一种上三角掩码，在 Decoder 的自注意力中使用，用来防止当前词看到未来的词，从而模拟语言模型"从左到右"的生成过程。 它是一个上三角矩阵（Upper Triangular Matrix），形状为 [seq_len, seq_len]：False 表示允许关注的位置。**
> **True 表示不允许关注的位置（即被屏蔽）**
>
> ```
> tensor([[False,  True,  True,  True,  True],
>         [False, False,  True,  True,  True],
>         [False, False, False,  True,  True],
>         [False, False, False, False,  True],
>         [False, False, False, False, False]])
> ```

3.3 BERT用的是transformer里面的encoder还是decoder?

BERT使用的是Transformer中的**Encoder部分**，而不是Decoder部分。**专注于从输入文本中提取强大的上下文化表示，适用于各种理解类任务，但不具备生成能力。**Transformer模型由Encoder和Decoder两个部分组成。Encoder用于将输入序列编码为一系列高级表示，而Decoder用于基于这些表示生成输出序列。
在BERT模型中，只使用了Transformer的Encoder部分，并且对其进行了一些修改和自定义的预训练任务，而没有使用Transformer的Decoder部分。

3.4 为什么BERT选择mask掉15%这个比例的词，可以是其他的比例吗?

BERT选择mask掉15%的词是一种经验性的选择，是原论文中的一种选择，并没有一个固定的理论依据，实际中当然可以尝试不同的比例，15%的比例是由BERT的作者在原始论文中提出，并在实验中发现对于BERT的训练效果是有效的。

3.5 为什么BERT在第一句前会加一个[CLS] 标志?

BERT在第一句前会加一个 [CLS] 标志，**最后一层该位对应向量可以作为整句话的语义表示，从而用于下游的分类任务等。**为什么选它？因为与文本中已有的其它词相比，这个无明显语义信息的符号会更"公平"地融合文本中各个词的语义信息，从而更好的表示整句话的语义。

具体来说，self-attention是用文本中的其它词来增强目标词的语义表示，但是目标词本身的语义还是会占主要部分的，因此，经过BERT的12层，每次词的embedding融合了所有词的信息，可以去更好的表示自己的语义。**而 [CLS] 位本身没有语义，经过12层，得到的是attention后所有词的加权平均，相比其他正常词，可以更好的表征句子语义。**

> **Note**
>
> `[CLS]` **之所以能代表整句话的语义，是因为它本身没有语义信息，在 BERT 的多层自注意力机制下，它不断地从其他 token 获取信息，最终成为一个融合整句话信息的"全局语义表示"**

3.6 BERT非线性的来源在哪里?

主要来自两个地方：**前馈层的gelu激活函数**和**self-attention**。

前馈神经网络层：在BERT的Encoder中，每个自注意力层之后都跟着一个前馈神经网络层。前馈神经网络层是全连接的神经网络，通常包括一个线性变换和一个非线性的激活函数，如**gelu**。这样的非线性激活函数引入了非线性变换，使得模型能够学习更加复杂的特征表示。

self-attention layer：在自注意力层中，查询（Query）、键（Key）、值（Value）之间的点积得分会经过softmax操作，形成注意力权重，然后将这些权重与值向量相乘得到每个位置的自注意输出。这个过程中涉及了softmax操作，使得模型的计算是非线性的。

3.7 BERT训练时使用的学习率 warm-up 策略是怎样的？为什么要这么做?

在BERT的训练中，使用了学习率warm-up策略，这是**为了在训练的早期阶段增加学习率，以提高训练的稳定性和加快模型收敛。**

学习率warm-up策略的具体做法是，在训练开始的若干个步骤（通常是一小部分训练数据的迭代次数）内，**将学习率逐渐从一个较小的初始值增加到预定的最大学习率。**在这个过程中，学习率的变化是线性的，即学习率在warm-up阶段的每个步骤按固定的步幅逐渐增加。学习率warm-up的目的是为了解决BERT在训练初期的两个问题：

- **不稳定性**：在训练初期，由于模型参数的随机初始化以及模型的复杂性，模型可能处于一个较不稳定的状态。此时使用较大的学习率可能导致模型的参数变动太大，使得模型很难收敛，学习率warm-up可以在这个阶段将学习率保持较小，提高模型训练的稳定性。
- **避免过拟合**：BERT模型往往需要较长的训练时间来获得高质量的表示。如果在训练的早期阶段就使用较大的学习率，可能会导致模型在训练初期就过度拟合训练数据，降低模型的泛化能力。通过学习率warm-up，在训练初期使用较小的学习率，可以避免过度拟合，等模型逐渐稳定后再使用较大的学习率进行更快的收敛。

> **Note**
>
> **学习率（Learning Rate）** 是深度学习中最重要的超参数之一。它决定了在每次梯度下降更新时，模型参数调整的"步长"大小。
>
> 数学上，参数更新公式如下：
>
> $$\theta_{t+1} = \theta_t - \eta \cdot \nabla J(\theta_t)$$
>
> 其中：
>
> - θ_t：第 t 步的模型参数；
> - η：学习率（Learning Rate）；
> - $\nabla J(\theta_t)$：损失函数对参数的梯度；

> - θ_{t+1}：更新后的参数。
>
> 简单来说，学习率控制着模型在训练过程中每一步朝着“更优解”迈进的幅度。

3.8 在BERT应用中，如何解决长文本问题?

在BERT应用中，处理长文本问题有以下几种常见的解决方案：

- **截断与填充**：将长文本截断为固定长度或者进行填充。BERT模型的输入是一个固定长度的序列，因此当输入的文本长度超过模型的最大输入长度时，需要进行截断或者填充。通常，可以根据任务的要求，选择适当的最大长度，并对文本进行截断或者填充，使其满足模型输入的要求。
- **Sliding Window**：将长文本分成多个短文本，然后分别输入BERT模型。这种方法被称为Sliding Window技术。具体来说，将长文本按照固定的步长切分成多个片段，然后分别输入BERT模型进行处理。每个片段的输出可以进行进一步的汇总或者融合，得到最终的表示。 **（在切片时让相邻片段部分重叠，避免信息断裂。）**
- **Hierarchical Model**：使用分层模型来处理长文本，其中底层模型用于处理短文本片段，然后将不同片段的表示进行汇总或者融合得到整个长文本的表示。这样的分层模型可以充分利用BERT模型的表示能力，同时处理长文本。 **（将长文本切分为多个短文本片段（segments），允许重叠（sliding window）以保留上下文连续性。用 BERT 编码每个片段，通常的做法是取 `[CLS]` 向量作为该片段的语义表示。再通过一个上层模型（如 BiLSTM、Transformer、Attention 等）对这些片段的表示进行融合，从而得到整个长文本的表示。）**
- **Longformer、BigBird等模型**：使用专门针对长文本的模型，如Longformer和BigBird。这些模型采用了不同的注意力机制，以处理超长序列，并且通常在处理长文本时具有更高的效率。
- **Document-Level Model**：将文本看作是一个整体，而不是将其拆分成句子或段落，然后输入BERT模型进行处理。这样的文档级模型可以更好地捕捉整个文档的上下文信息，但需要更多的计算资源。

MHA & MQA & MGA

4.1MHA（Multi-Head Attention）

从多头注意力的结构图中，貌似这个所谓的多个头就是指多组线性变换层，其实并不是，只有使用了一组线性变化层，即三个变换张量对 Q、K、V 分别进行线性变换，**这些变换不会改变原有张量的尺寸**，因此每个变换矩阵都是方阵。**得到输出结果后，多头的作用才开始显现，每个头开始从词义层面分割输出的张量，也就是每个头都想获得一组 Q、K、V 进行注意力机制的计算，但是句子中的每个词的表示只获得一部分，也就是只分割了最后一维的词嵌入向量。**这就是所谓的多头，将每个头获得的输入送到注意力机制中，就形成多头注意力机制。Multi-head attention 允许模型共同关注来自不同位置的不同表示子空间的信息，如果只有一个 attention head，它的平均值会削弱这个信息。

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

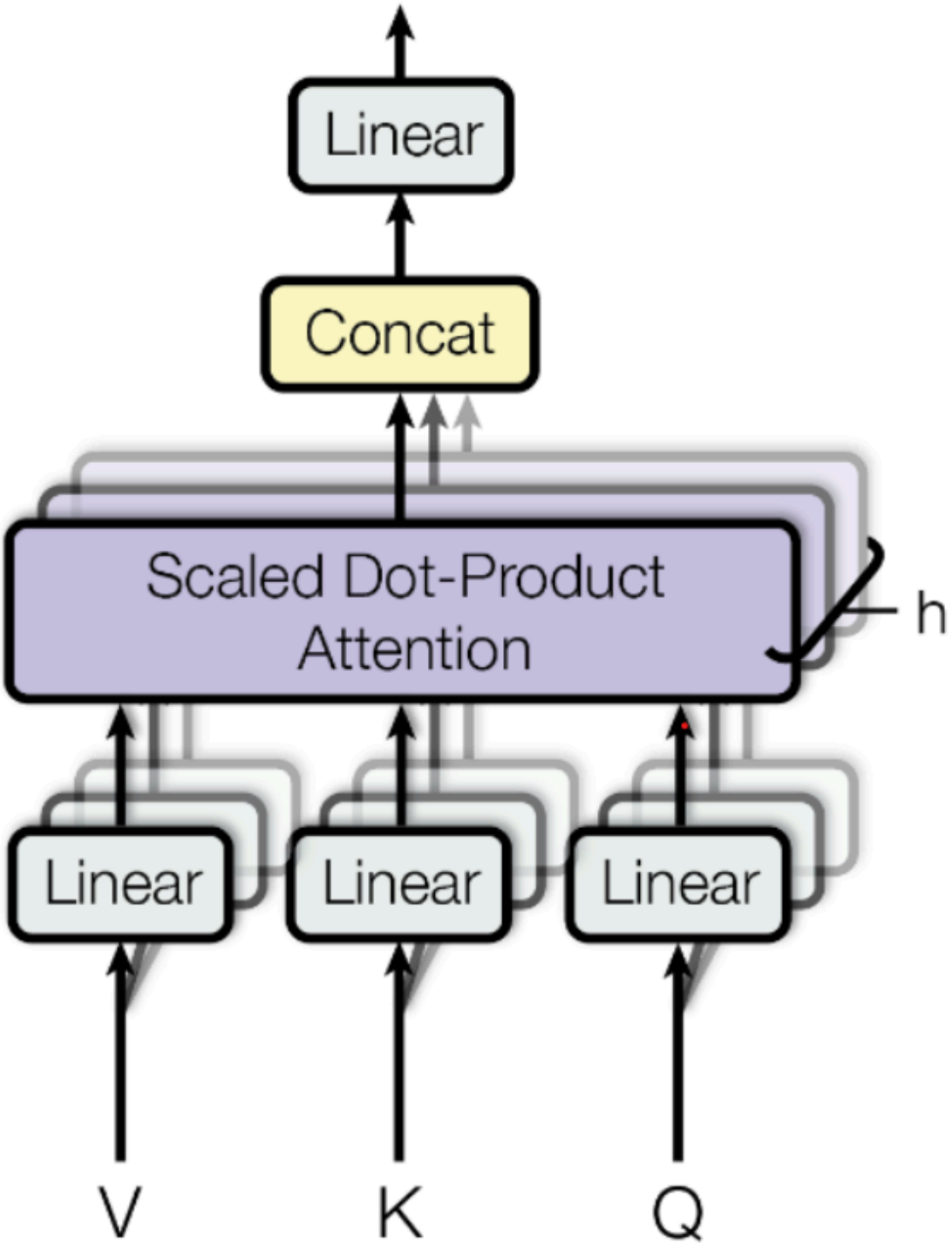
其中，每一个头 h_i 是把Q, K, V通过 可以学习的W_i^Q, W_i^K, W_i^V 投影到d_v 上，再通过注意力函数，得到$head_i$。

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

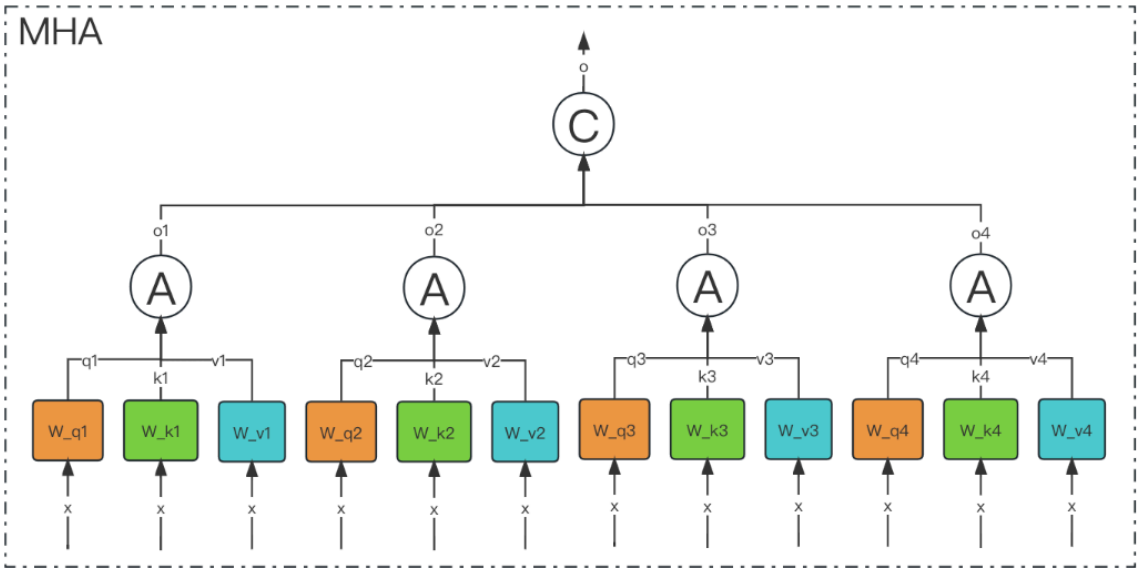
各个映射由权重矩阵完成：

- $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$
- $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$
- $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$

- 最终输出的投影矩阵为：$W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$



> **Note**
>
> 多头注意力机制的本质是将词向量划分到多个子空间中并行处理，让模型能够从多个角度理解语言结构，增强其表达能力和泛化性能。**每个“头”拿到的是线性变换后的词向量的一部分（如原本是 512 维，分成 8 个头，每个头拿 64 维）**，然后在这个子空间中进行注意力计算。所有头的结果拼接起来后，乘以一个输出投影矩阵W^O，使输出维度与输入一致。



多头注意力作用

这种结构设计能**让每个注意力机制去优化每个词汇的不同特征部分**，从而均衡同一种注意力机制可能产生的偏差，让词义拥有来自更多元的表达，实验表明可以从而提升模型效果.

为什么要做多头注意力机制呢?

- 一个 dot product（点积） 的注意力里面，没有什么可以学的参数。具体函数就是内积，为了识别不一样的模式，希望有不一样的计算相似度的办法。**加性 attention 有一个权重可学，也许能学到一些内容。**
- multi-head attention 给 h 次机会去学习不一样的投影的方法，使得在投影进去的度量空间里面能够去匹配不同模式需要的一些相似函数，然后把 h 个 heads 拼接起来，最后再做一次投影。
- 每一个头 h_i 是把Q, K, V通过 可以学习的W_q, W_k, W_v 投影到d_v 上，再通过注意力函数，得到 $head_i$。

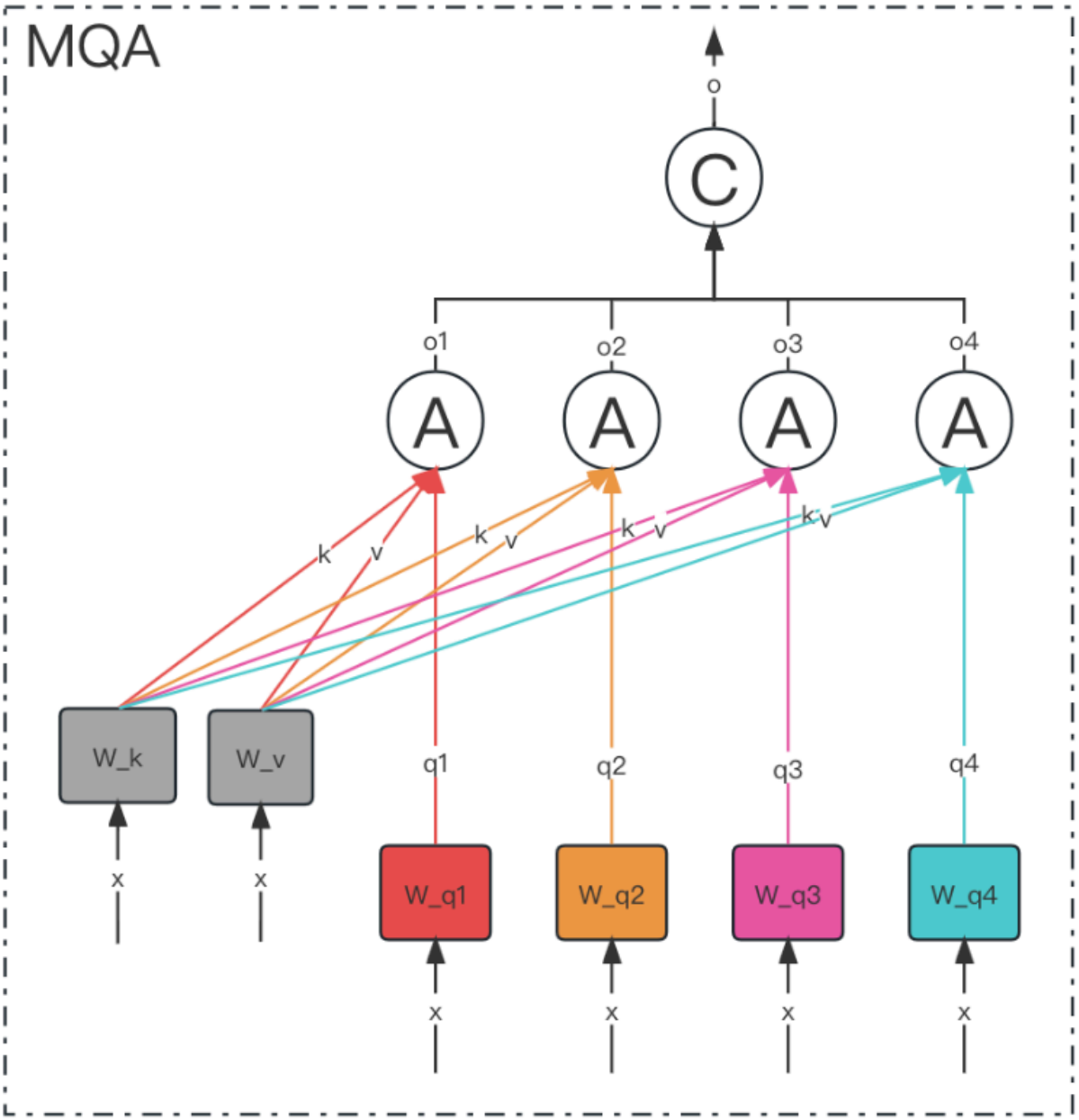
4.2MQA(Multi Query Attention)

MQA的思想其实比较简单，MQA 与 MHA 不同的是，**MQA 让所有的头之间共享同一份 Key 和 Value 矩阵，每个头正常的只单独保留了一份 Query 参数，从而大大减少 Key 和 Value 矩阵的参数量。**

在 MQA 中：

- 每个 head 的 Query 投影矩阵是 $d_{model} \times d_k$ （如 768×64）
- Key 和 Value 投影矩阵是共享的，尺寸也是 $d_{model} \times d_k$（同样 768×64）
- d_k 和 d_v 通常相等，以确保注意力机制的维度一致性

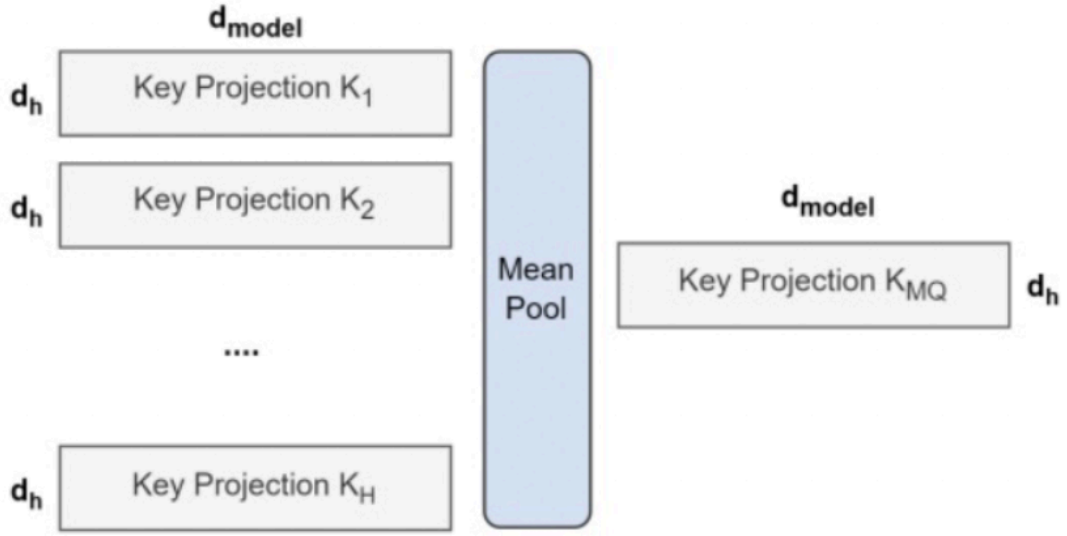
> **Note**
>
> MQA（Multi Query Attention）最早是出现在2019年谷歌的一篇论文《Fast Transformer Decoding: One Write-Head is All You Need》。



在 Multi-Query Attention 方法中只会保留一个单独的key-value头，这样**虽然可以提升推理的速度，但是会带来精度上的损失。**《Multi-Head Attention:Collaborate Instead of Concatenate 》这篇论文的第一个思路是**基于多个 MQA 的 checkpoint 进行 finetuning，来得到了一个质量更高的 MQA 模型。**这个过程也被称为 Uptraining。**Uptraining 是一种基于知识蒸馏的方法，它利用多个 MQA 模型的 checkpoint 来指导一个 MQA 模型的训练，从而在不牺牲推理效率的前提下，显著提升 MQA 模型的表现。相当于把多个checkpoint中的MQA的K/V矩阵进行平均然后让一个新的MQA模型训练**

具体分为两步：

1. 对多个 MQA 的 checkpoint 文件进行融合，融合的方法是: 通过对 key 和 value 的 head 头进行 mean pooling **（平均池化）**操作，通过对多个输入值或张量进行“取平均”的操作，将它们压缩成一个统一的表示。如下图：



在 Uptraining 方法中，**mean pooling 被用于融合多个 MQA 模型的 Key 和 Value 投影矩阵权重。**

具体来说：

- 训练多个结构相同但初始化不同的 MQA 模型；
- 取出每个模型中共享的 Key 和 Value 的投影矩阵 $W_K^{(i)}$ 和 $W_V^{(i)}$；
- 对这些矩阵分别进行平均：

$$W_K^{\text{fused}} = \frac{1}{N}\sum_{i=1}^{N} W_K^{(i)}, \quad W_V^{\text{fused}} = \frac{1}{N}\sum_{i=1}^{N} W_V^{(i)}$$

这样可以得到一个更鲁棒、更具泛化能力的 Key 和 Value 表示。

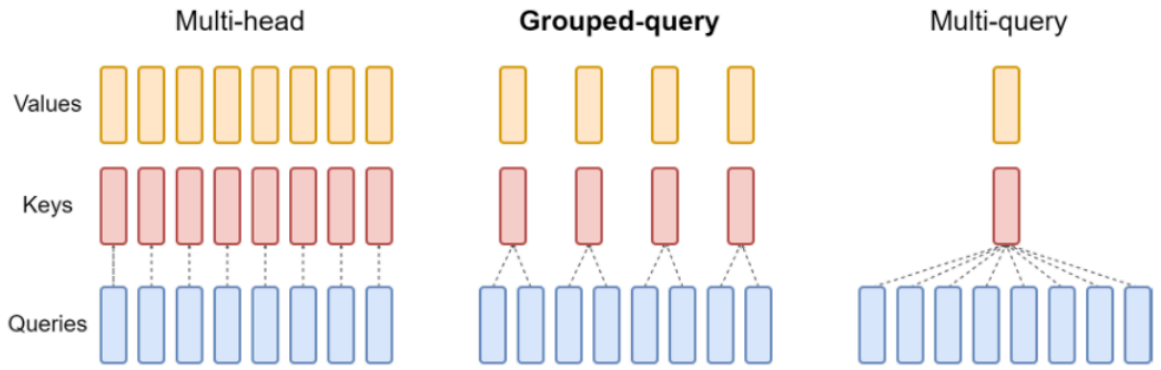
> **Note**
>
> - Query 的部分不参与融合，保留各自独立的头；
> - 所有参与融合的模型必须具有相同的结构，否则无法对齐权重；
> - 这种方式类似于“模型集成 + 权重平均”，是一种轻量级且有效的模型优化手段。

2. 对融合后的模型使用少量数据进行 finetune 训练，重训后的模型大小跟之前一样，但是效果会更好

4.3GQA

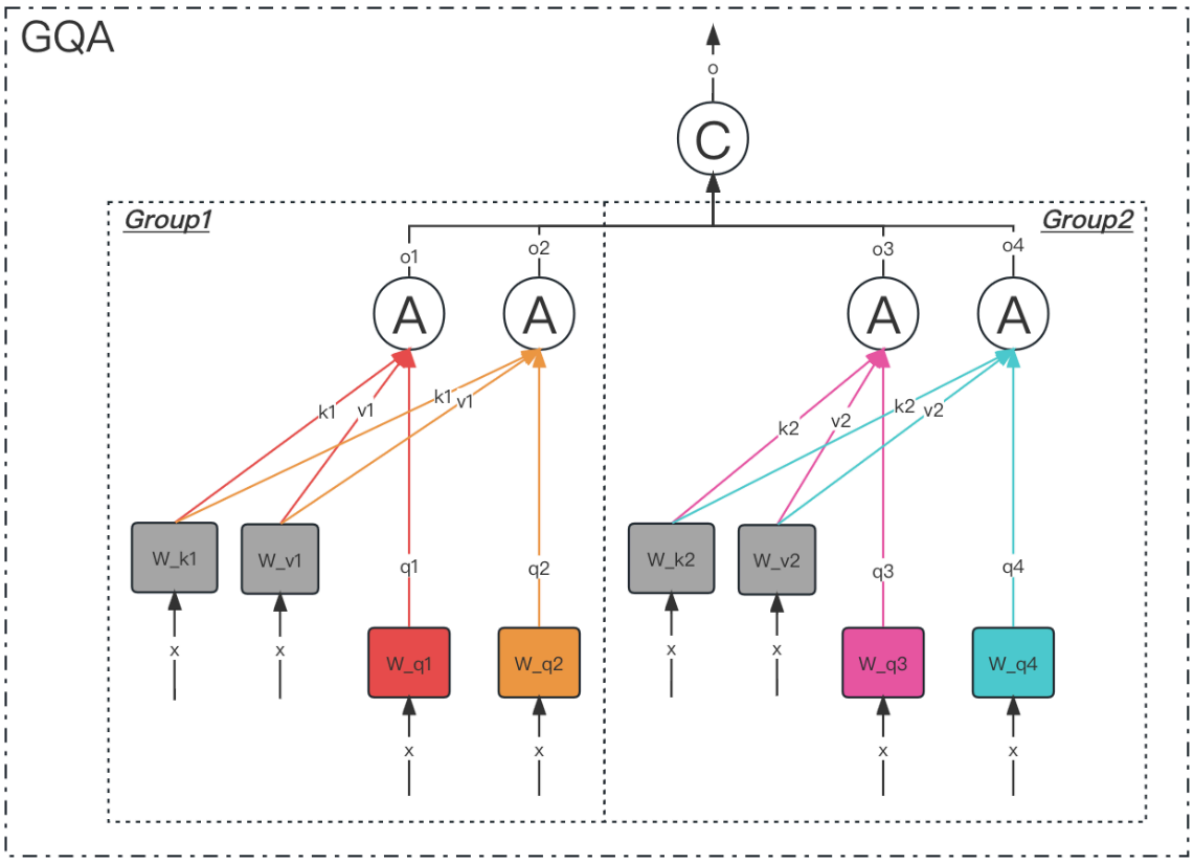
Google 在 2023 年发表的一篇 《GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints》的论文。如下图所示，

- 在 **MHA（Multi Head Attention）** 中，每个头有自己单独的 key-value 对；
- 在 **MQA（Multi Query Attention）** 中只会有一组 key-value 对；
- 在 **GQA（Grouped Query Attention）** 中，会对attention进行分组操作，**query 被分为N组，每个组共享一个 Key 和 Value 矩阵**



GQA-N 是指具有 N 组的 Grouped Query Attention。GQA-1具有单个组，因此具有单个Key 和 Value，等效于MQA。而GQA-H具有与头数相等的组，等效于MHA。

在基于 Multi-head 多头结构变为 Grouped-query 分组结构的时候，也是采用跟上图一样的方法，对每一组的 key-value 对进行 mean pool 的操作进行参数融合。**融合后的模型能力更综合，精度比 Multi-query 好，同时速度比 Multi-head 快。**



4.4总结

MHA（Multi-head Attention）是标准的多头注意力机制，h个Query、Key 和 Value 矩阵。

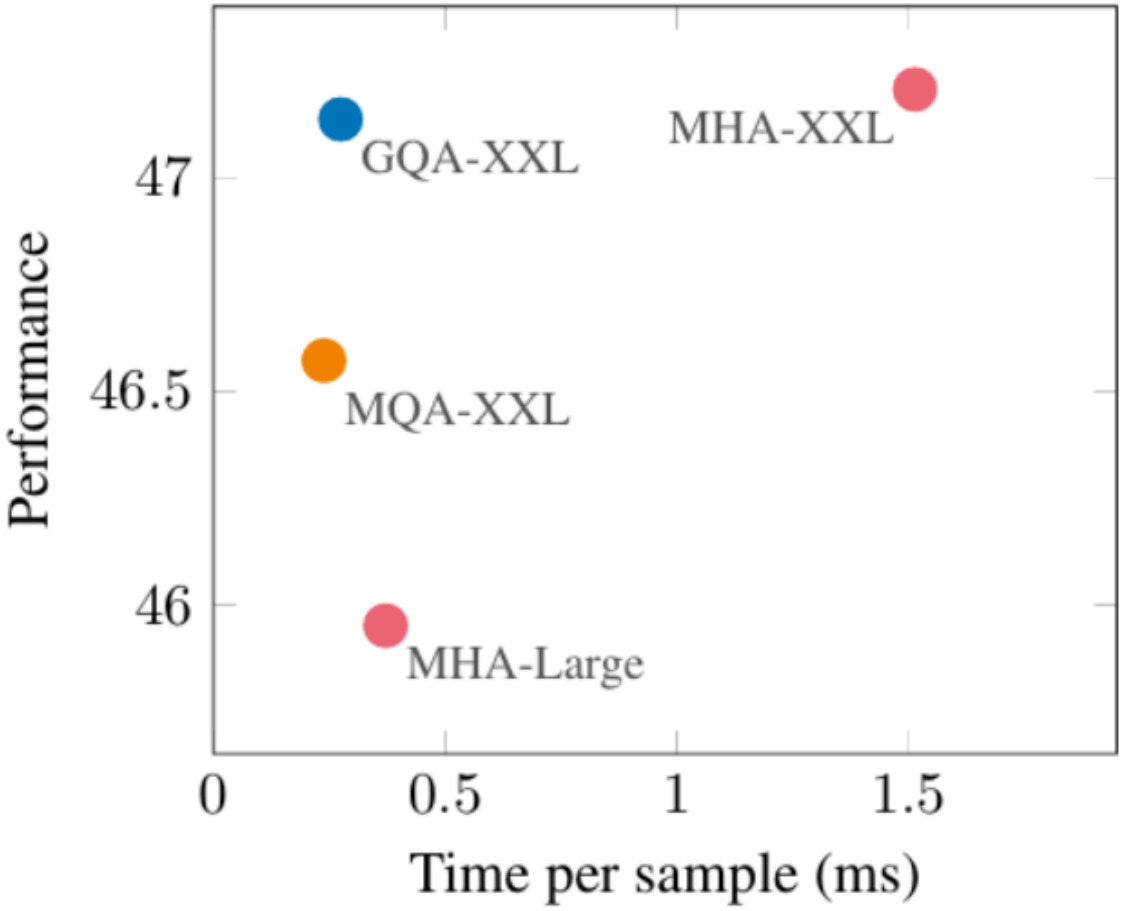
（1）MQA（Multi-Query Attention）是多查询注意力的一种变体，也是用于自回归解码的一种注意力机制。与MHA不同的是，**MQA 让所有的头之间共享同一份 Key 和 Value 矩阵，每个头只单独保留了一份 Query 参数，从而大大减少 Key 和 Value 矩阵的参数量。**

（2）GQA（Grouped-Query Attention）是分组查询注意力，**GQA将查询头分成G组，每个组共享一个 Key 和 Value 矩阵。**GQA-G是指具有G组的grouped-query attention。GQA-1具有单个组，因此具有单个Key 和 Value，等效于MQA。而GQA-H具有与头数相等的组，等效于MHA。

（3）GQA介于MHA和MQA之间。GQA 综合 MHA 和 MQA ，既不损失太多性能，又能利用 MQA 的推理加速。**不是所有 Q 头共享一组 KV，而是分组一定头数 Q 共享一组 KV，比如上图中就是两组 Q 共享一组 KV。**

例如：示例结构（以 8 个 Q 头为例）

类型	KV 共享方式	KV 头数量
MHA	每个 Q 头都有独立的 K/V	8 组
GQA（分组为 2）	每 2 个 Q 头共享一组 K/V	4 组
MQA	所有 Q 头共享同一组 K/V	1 组



Flash Attention

> **Note**
>
> 论文名称：FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness

Flash Attention的主要目的是**加速和节省内存**，不显式地存储整个N^2的矩阵，而是在前向传播时只记录必要的信息，在反向传播时重新计算所需的中间结果。主要贡献包括：

- 计算softmax时候不需要全量input数据，可以分段计算；
- 反向传播的时候，不存储attention matrix (N^2的矩阵)，而是只存储softmax归一化的系数。

具体来说：

- 前向传播中仍然会计算 attention 分数 *S* 和 softmax 后的 *A*；但 Flash Attention **不会把整个注意力矩阵A 存下来**；它只保存 **softmax 的归一化系数**（即每一行 softmax 的分母部分）；
  - 计算注意力分数 $S = QK^T/\sqrt{d_k}$
  - 对每一行计算并保存 softmax 的归一化系数 $D_i = \sum_j \exp(S_{ij})$
  - **不保存完整的 attention matrix** A

- 在反向传播时，再根据 Q、K、V 和这些归一化系数**按需重新计算 attention 权重**，从而节省大量显存。

> **Note**
>
> 为什么保存分母就"够用"?
> 因为在反向传播中，我们并不需要整个 attention 矩阵 A，而只需要能计算它的导数即可。
> 我们来看 softmax 的梯度形式： 设 $A_{ij} = \frac{\exp(S_{ij})}{D_i}$，则其梯度为：
>
> $$\frac{\partial A_{ij}}{\partial S_{ik}} = A_{ij}(\delta_{jk} - A_{ik})$$
>
> 在梯度公式中：
>
> A_{ij} 是 softmax 输出，由原始分数 S_{ij} 和归一化系数 D_i 共同决定。
>
> δ_{jk} 是一个辅助变量，用于区分当前列和其他列。
>
> A_{ik} 是 softmax 输出的另一项，同样由 S_{ik} 和 D_i 决定。
>
> 因此，只要我们知道原始分数矩阵 S 和 softmax 的归一化系数 D_i，就可以随时重新计算出任意的 A_{ij} 和 A_{ik}，从而完成梯度计算。

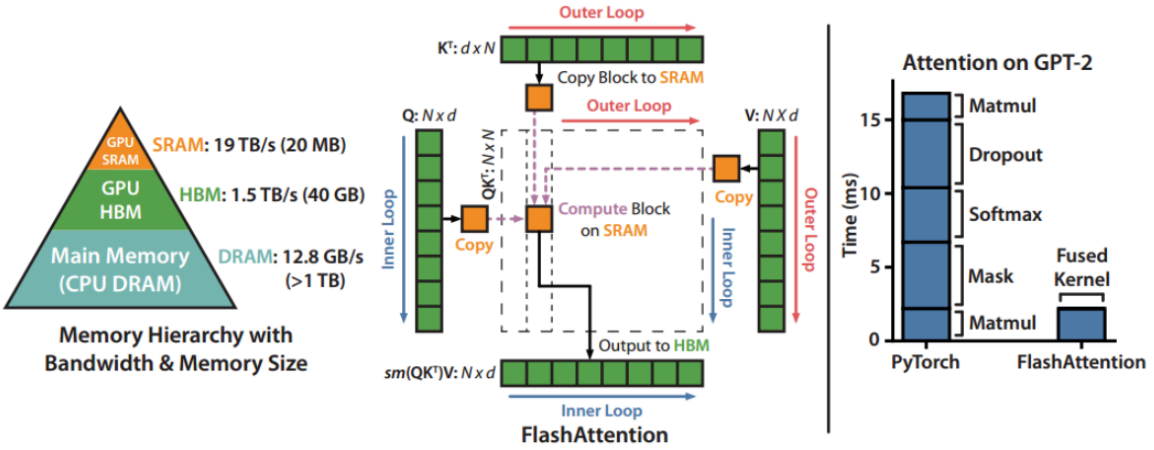
5.1动机

不同硬件模块之间的带宽和存储空间有明显差异，例如下图中左边的三角图，最顶端的是GPU种的`SRAM`，它的容量非常小但是带宽非常大，以A100 GPU为例，它有108个流式多核处理器，每个处理器上的片上SRAM大小只有192KB，因此A100总共的SRAM大小是$192KB \times 108 = 20MB$，但是其吞吐量能高达19TB/s。而A100 GPU `HBM`（**High Bandwidth Memory也就是我们常说的GPU显存大小**）大小在40GB~80GB左右，但是带宽只与1.5TB/s。

GPU 内部存在多个层级的存储系统，它们在**容量** 和**带宽** 上有显著差异：

存储类型	容量	带宽	特点
SRAM	小（仅 192KB/SM，总约 20MB）	极高（19 TB/s）	快速、昂贵、靠近计算单元
HBM	大（40~80 GB）	较高（1.5 TB/s）	慢于 SRAM，但容量巨大

- **SRAM** 是 GPU 上最快的存储资源，但由于物理限制，总量非常有限；
- **HBM** 是 GPU 的主要存储空间，虽然带宽也很高，但远远不及 SRAM；
- 因此，在进行高效计算时（比如 Flash Attention、Tensor Core 加速等），应尽可能将高频访问的数据放入 SRAM 中，减少对 HBM 的访问，从而提升整体性能。



下图给出了标准的注意力机制的实现流程，可以看到因为 HBM 的大小更大，**我们平时写pytorch代码的时候最常用到的就是HBM，所以对于HBM的读写操作非常频繁，而SRAM利用率反而不高。**

Algorithm 0 Standard Attention Implementation

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.

1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \mathrm{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.

FlashAttention的主要动机就是**希望把SRAM利用起来**，但是难点就在于SRAM太小了，一个普通的矩阵乘法都放不下去。**FlashAttention的解决思路就是将计算模块进行分解，拆成一个个小的计算任务。**

5.2 Softmax Tiling

在介绍具体的计算算法前，我们首先需要了解一下Softmax Tiling。

（1）数值稳定

由于包含指数函数 e^x，当输入值很大时，会导致数值溢出（overflow），即 e^x 超出浮点数表示范围；当输入值很小时，又可能导致下溢（underflow），变成 0。所以为了避免数值溢出问题，可以将每个元素都减去最大值，如下图示，最后计算结果和原来的Softmax是一致的。

$$
\begin{aligned}
m(x) &:= \max_i \; x_i \\
f(x) &:= \begin{bmatrix} e^{x_1 - m(x)} & \dots & e^{x_B - m(x)} \end{bmatrix} \\
\ell(x) &:= \sum_i f(x)_i \\
\mathrm{softmax}(x) &:= \frac{f(x)}{\ell(x)}
\end{aligned}
$$

推导过程：

原始 Softmax 定义：

$$
\mathrm{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}
$$

减去最大值后的形式：

$$
m(x) = \max_i x_i
$$

$$\text{new\_softmax}(x_i) = \frac{e^{x_i - m(x)}}{\sum_j e^{x_j - m(x)}}$$

将分子分母同时乘以 $e^{m(x)}$：

$$\text{new\_softmax}(x_i) = \frac{e^{x_i - m(x)} \cdot e^{m(x)}}{\sum_j e^{x_j - m(x)} \cdot e^{m(x)}}$$

$$= \frac{e^{x_i}}{\sum_j e^{x_j}}$$

$$= \text{softmax}(x_i)$$

2.分块计算softmax

因为 Softmax 都是按行计算的，所以我们考虑一行切分成两部分的情况，即原本的一行数据：

$$x \in \mathbb{R}^{2B} = [x^{(1)}, x^{(2)}]$$

For vectors $x^{(1)}, x^{(2)} \in \mathbb{R}^B$, we can decompose the softmax of the concatenated $x = \begin{bmatrix} x^{(1)} & x^{(2)} \end{bmatrix} \in \mathbb{R}^{2B}$ as:

$$m(x) = m(\begin{bmatrix} x^{(1)} & x^{(2)} \end{bmatrix}) = \max(m(x^{(1)}), m(x^{(2)})), \quad f(x) = \begin{bmatrix} e^{m(x^{(1)}) - m(x)} f(x^{(1)}) & e^{m(x^{(2)}) - m(x)} f(x^{(2)}) \end{bmatrix},$$

$$\ell(x) = \ell(\begin{bmatrix} x^{(1)} & x^{(2)} \end{bmatrix}) = e^{m(x^{(1)}) - m(x)} \ell(x^{(1)}) + e^{m(x^{(2)}) - m(x)} \ell(x^{(2)}), \quad \text{softmax}(x) = \frac{f(x)}{\ell(x)}.$$

可以看到，计算不同块的 $f(x)$ 值时，乘上的系数是不同的，**但是最后化简后的结果都是指数函数减去了整行的最大值**。以 $x^{(1)}$ 为例：

$$e^{m(x^{(1)}) - m(x)} f(x^{(1)}) = e^{m(x^{(1)}) - m(x)} \begin{bmatrix} e^{x_1^{(1)} - m(x^{(1)})}, \ldots, e^{x_B^{(1)} - m(x^{(1)})} \end{bmatrix} b$$

$$= \begin{bmatrix} e^{x_1^{(1)} - m(x)}, \ldots, e^{x_B^{(1)} - m(x)} \end{bmatrix}$$

> **Note**
>
> **核心：即使我们将 softmax 按块计算，并分别减去每块的最大值，在最后通过一个缩放因子调整后，仍然可以得到与原始整体 softmax 相同的结果**，在每个块内部只减去了自己的最大值；但通过乘上一个因子后，就等价于在整行中减去了全局最大值；这样就能保证所有块合并后，softmax 的计算是一致的、准确的。

5.3 算法流程

FlashAttention旨在避免从 HBM（High Bandwidth Memory）中读取和写入注意力矩阵，这需要做到：

1. 目标一：在不访问整个输入的情况下计算softmax函数的缩减；**将输入分割成块，并在输入块上进行多次传递，从而以增量方式执行softmax缩减。**
2. 目标二：在后向传播中不能存储中间注意力矩阵。标准Attention算法的实现需要将计算过程中的S、P写入到HBM中，而这些中间矩阵的大小与输入的序列长度有关且为二次型，因此**Flash Attention就提出了不使用中间注意力矩阵，通过存储归一化因子来减少HBM内存的消耗。**

FlashAttention算法流程如下图所示：

Algorithm 1 FLASHATTENTION

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size M.

Set block sizes $B_c = \lceil \frac{M}{4d} \rceil$, $B_r = \min\left(\lceil \frac{M}{4d} \rceil, d\right)$.

Initialize $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}$, $\ell = (0)_N \in \mathbb{R}^N$, $m = (-\infty)_N \in \mathbb{R}^N$ in HBM.

Divide $\mathbf{Q}$ into $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $\mathbf{Q}_1, \ldots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $\mathbf{K}_1, \ldots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \ldots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.

Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \ldots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_i, \ldots, \ell_{T_r}$ of size B_r each, divide m into T_r blocks $m_1, \ldots, m_{T_r}$ of size B_r each.

for $1 \le j \le T_c$ **do**

  Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.

  for $1 \le i \le T_r$ **do**

    Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.

    On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.

    On chip, compute $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.

    On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.

    Write $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1} (\text{diag}(\ell_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.

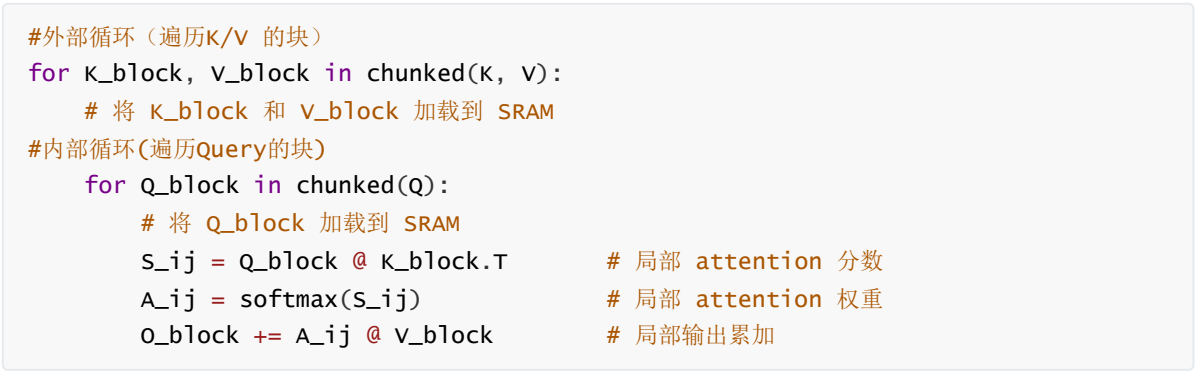
    Write $\ell_i \leftarrow \ell_i^{\text{new}}$, $m_i \leftarrow m_i^{\text{new}}$ to HBM.

  end for

end for

Return $\mathbf{O}$.

为方便理解，下图将FlashAttention的计算流程可视化出来了，简单理解就是每一次只计算一个block的值，通过多轮的双for循环完成整个注意力的计算。

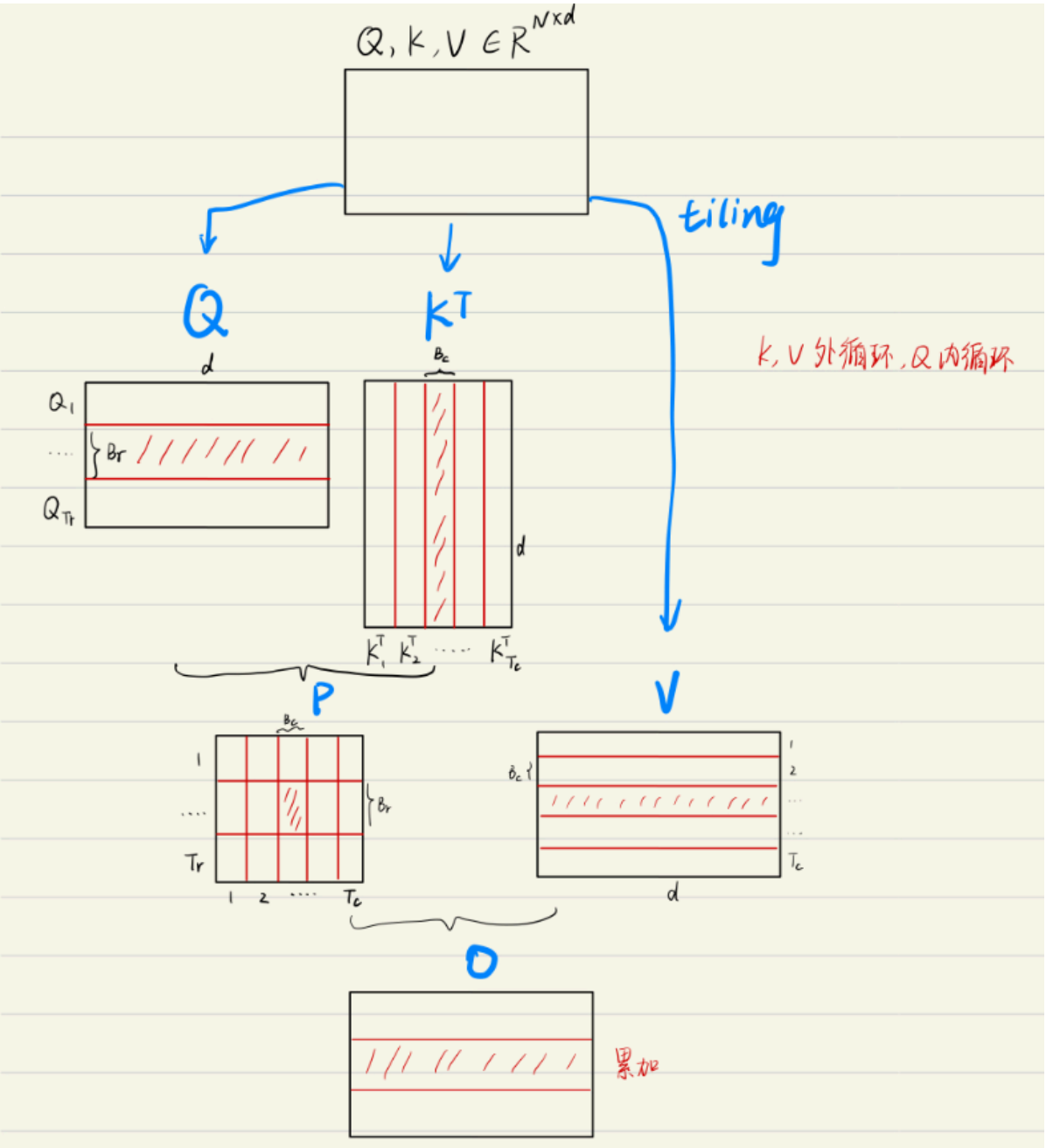


```
#外部循环（遍历K/V 的块）
for K_block, V_block in chunked(K, V):
    # 将 K_block 和 V_block 加载到 SRAM
#内部循环(遍历Query的块)
    for Q_block in chunked(Q):
        # 将 Q_block 加载到 SRAM
        S_ij = Q_block @ K_block.T        # 局部 attention 分数
        A_ij = softmax(S_ij)              # 局部 attention 权重
        O_block += A_ij @ V_block         # 局部输出累加
```

Flash Attention 的核心流程：

1. **分割 Q 和 K^T**：将查询和键分割成小块。
2. **逐块计算局部注意力矩阵 P**：通过外循环遍历键块，内循环遍历查询块，逐步计算点积，并更新 softmax 归一化的因子。
3. **累加局部注意力矩阵**：将所有局部注意力矩阵合并为完整的注意力权重矩阵 P。
4. **与值矩阵 V 结合**：使用归一化的注意力权重矩阵 P 和值矩阵 V，计算最终的注意力输出：

$$O = P \cdot V$$



Transformer常见问题

6.1Transformer和RNN

1.**Point-wise 操作** （逐点操作）是指对输入张量中的每一个位置（token）**独立地进行相同的运算**，不涉及 token 之间的交互。也就是说，每个 token 的输出只依赖于它自己的输入，与其他 token 无关。不是在建模上下文关系，而是**对当前 token 的表示做增强或变换**。

常见的 Point-wise 操作包括：

- 全连接层（MLP）：每个 token 经过一个共享的线性变换 + 激活函数。
- 激活函数：如 ReLU、GELU。
- LayerNorm：虽然涉及到归一化，但它是 per-token 的操作，也属于 Point-wise 类别。
- Sigmoid、Tanh 等非线性函数

2.在最简化的 Transformer 模型中（**无残差连接、无 LayerNorm、单头 Attention、无线性投影**）：

- **Attention 负责“看整体”，Point-wise （逐点操作）负责“精细加工”。**

- Attention 的作用是对整个输入序列进行加权求和，实现全局信息的汇聚（aggregation）；
- 加权后的结果会输入到一个 point-wise 的 MLP 中；
- 这个 MLP 对每个位置独立地进行非线性变换，且所有位置共享参数。

与 RNN 的区别在于：

- RNN 是通过循环逐步传递状态来建模序列依赖；
- Transformer 则是通过 Attention 全局建模，一次性获取整个序列的信息，并通过加权方式聚焦关键内容。

基本结构对比

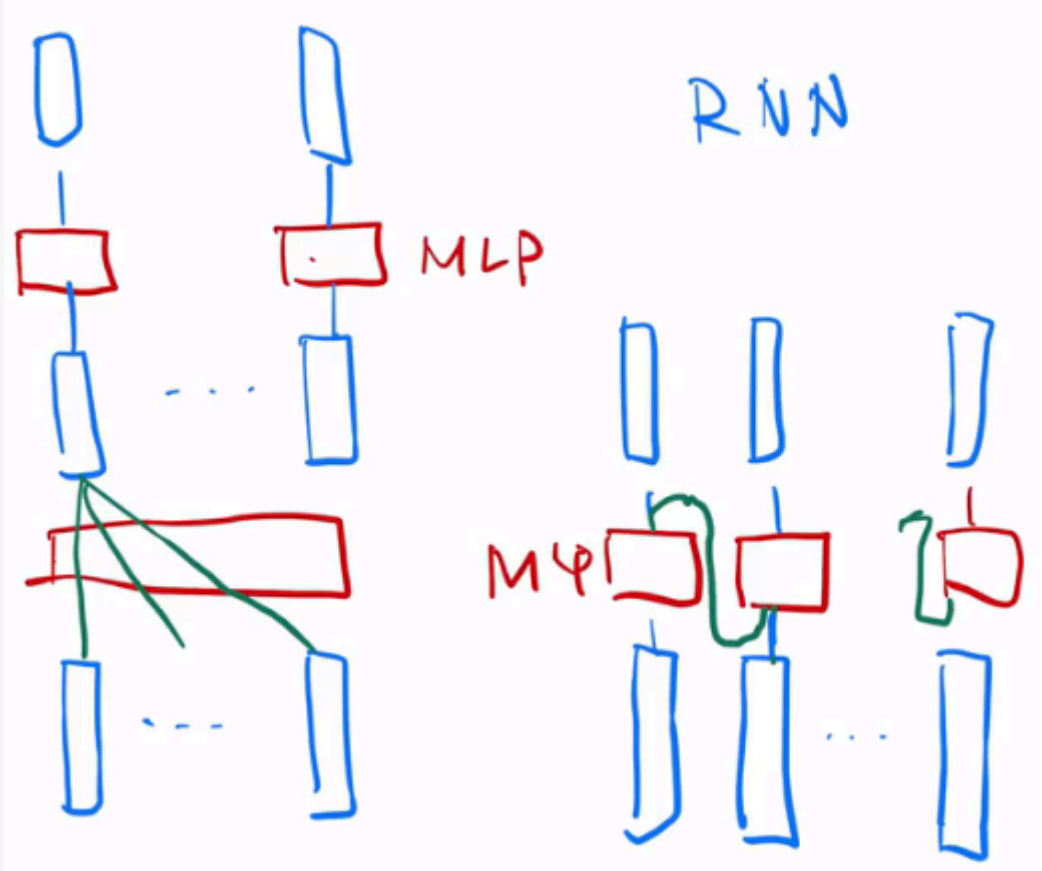
对比维度	RNN	Transformer
基本结构	循环神经网络（Recurrent）	自注意力机制 + 前馈网络
信息传递方式	按时间步逐步传递状态	全局建模，一次性看到整个序列
是否并行化	否（必须按顺序处理）	是（所有位置并行计算）
长距离依赖建模能力	弱（易梯度消失/爆炸）	强（直接建模任意位置间关系）

核心机制对比

对比维度	RNN	Transformer
信息聚合方式	隐藏状态自动携带历史信息	使用 Attention 动态加权求和
注意力机制	可选模块（如 Seq2Seq with attention）	核心组成部分（Self-Attention）
序列建模能力	局部感知：只能看到前面的信息(串行)	全局感知：能看到整个序列（并行）
参数共享	时间步之间共享权重	所有位置共享相同的参数矩阵
可解释性	弱（隐藏状态难以可视化）	强（Attention 权重可解释）

语义空间转换 + 关注机制对比

对比维度	RNN	Transformer
语义空间变换	隐藏层自动学习（线性变化 + 非线性激活函数），只能串行计算。	通过 Point-wise MLP 显式映射进行非线性变换，并且支持并行计算。
关注机制	无显式关注机制	通过 Self-Attention 显式选择重要信息



6.2 一些细节

6.2.1 Transformer为何使用多头注意力机制？（为什么不使用一个头）

- 多头保证了transformer可以注意到不同子空间的信息，捕捉到更加丰富的特征信息。可以类比CNN中同时使用**多个滤波器**的作用，直观上讲，多头的注意力**有助于网络捕捉到更丰富的特征/信息。**

6.2.2 Transformer为什么Q和K使用不同的权重矩阵生成，为何不能使用同一个值进行自身的点乘？（注意和第一个问题的区别）

1.使用Q/K/V不相同可以保证在不同空间进行投影，增强了表达能力，提高了泛化能力。

2.**Softmax 函数其实是在做一个"软化版"的最大值选择（soft arg max）。它会把一组数中最大的那个"放大"，而把其他数值相对"压缩"，最终输出的结果接近一个 one-hot 向量（也就是只有一个位置是大值，其余都很小）。这个"接近程度"取决于原始数值的大小和分布。**如果我们不做投影变换，直接让 Query 和 Key 都等于输入 X，那么注意力得分矩阵就是 $QK^T = XX^T$。这时候每个 token 更容易关注到自己，因为自己和自己的相似度最高。经过 softmax 之后，注意力矩阵会接近一个单位矩阵 —— 每一行只有对角线上的位置有较大权重，其它位置几乎为 0。

这样一来，Self-Attention 就退化成了一种"只关注自己"的操作，每个 token 基本上只加权了自己的 Value，结果就相当于对每个 token 单独做了个线性变换（Point-wise 线性变换），失去了与其他 token 的交互能力。Self-Attention 最大的优势在于它能捕捉序列中所有 token 之间的关系，建模全局依赖。但如果 Q 和 K 是一样的，模型就几乎只能看到自己，看不到上下文信息，这就完全违背了 Self-Attention 的核心设计理念。

6.2.3 Transformer计算attention的时候为何选择点乘而不是加法？两者计算复杂度和效果上有什么区别？

1.K和Q的点乘是为了得到一个attention score 矩阵，用来对V进行提纯。K和Q使用了不同的W_k, W_Q来计算，可以理解为是在不同空间上的投影。正因为有了这种不同空间的投影，增加了表达能力，这样计算得到的attention score矩阵的泛化能力更高。

2.为了计算更快。**矩阵加法在加法这一块的计算量确实简单，但是作为一个整体计算attention的时候相当于一个隐层，整体计算量和点积相似。**在效果上来说，从实验分析，两者的效果和d_k相关，d_k越大，加法的效果越显著。

6.2.4 为什么在进行softmax之前需要对attention进行scaled（为什么除以dk的平方根），并使用公式推导进行讲解

1.这取决于softmax函数的特性，如果softmax内计算的数数量级太大，会输出近似one-hot编码的形式，导致梯度消失的问题，所以需要scale。

2.那么至于为什么需要用维度开根号，假设向量 q，k 满足各分量独立同分布，均值为 0，方差为 1，那么 $q \cdot k$ 点积均值为 0，方差为 d_k，从统计学计算，如果让 $q \cdot k$ 点积的方差控制在 1，需要将其除以 $\sqrt{d_k}$，使得 softmax 更加平滑。

6.2.5 在计算attention score的时候如何对padding做mask操作?

1.padding位置置为负无穷(一般来说-1000就可以)，再对attention score进行相加。对于这一点，涉及到batch_size之类的，具体的大家可以看一下实现的源代码。

位置在这里：https://github.com/huggingface/transformers/blob/aa6a29bc25b663e1311c5c4fb96b004cf8a6d2b6/src/transformers/modeling_bert.py#L720](https://link.zhihu.com/?target=https://github.com/huggingface/transformers/blob/aa6a29bc25b663e1311c5c4fb96b004cf8a6d2b6/src/transformers/modeling_bert.py#L720)

2.padding位置置为负无穷而不是0，是因为后续在softmax时，e^0=1，不是0，计算会出现错误；而$e^{-\infty}$=0，所以取负无穷

6.2.6 为什么在进行多头注意力的时候需要对每个head进行降维？（可以参考上面一个问题）

将原有的高维空间转化为多个低维空间并再最后进行拼接，形成同样维度的输出，借此丰富特性信息

基本结构：Embedding + Position Embedding，Self-Attention，Add + LN，FN，Add + LN

6.2.7 为何在获取输入词向量之后需要对矩阵乘以embedding size的开方？意义是什么？

1.embedding matrix的初始化方式是xavier init。**Xavier 初始化的核心思想是：根据当前层的输入和输出神经元数量，设置合适的初始化范围或方差，使得信号在正向传播和反向传播过程中保持稳定的尺度**

2.在 Transformer 中，对 embedding 乘以$\sqrt{d_{\text{model}}}$的操作，是为了抵消 Xavier 初始化带来的方差缩小效应，使得 embedding 输出具有单位方差，从而提升模型的训练稳定性和收敛速度。

6.2.8 简单介绍一下Transformer的位置编码？有什么意义和优缺点?

因为self-attention是位置无关的，无论句子的顺序是什么样的，通过self-attention计算的token的hidden embedding都是一样的，这显然不符合人类的思维。**因此要有一个办法能够在模型中表达出一个token的位置信息，transformer使用了固定的positional encoding来表示token在句子中的绝对位置信息。**

6.2.9 你还了解哪些关于位置编码的技术，各自的优缺点是什么？（参考上一题）

- 相对位置编码（RPE）：
  1.在计算attention score和weighted value时各加入一个可训练的表示相对位置的参数：

  标准的注意力机制计算如下：

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

  在标准 Attention 中加入相对位置编码（RPE）后，变为：

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

  2.在生成多头注意力时，把对key来说将绝对位置转换为相对query的位置

  核心思想：不是直接给每个 token 加上一个绝对位置编码，而是让 Key 和 Value 向量的生成过程中，考虑相对于 Query 的相对位置。

  3.复数域函数：已知一个词在某个位置的词向量表示，可以计算出它在任何位置的词向量表示。前两个方法是词向量+位置编码，属于亡羊补牢，复数域是生成词向量的时候即生成对应的位置信息。

核心思想：不同于前两种"加法式"的位置编码方式，Rotary Positional Encoding（RoPE）是一种**乘法式的相对位置建模方式**， 它把位置信息直接编码到 query 和 key 的向量空间中，**使得词向量本身具有旋转性质，从而体现相对位置。**

6.2.10 简单讲一下Transformer中的残差结构以及意义

1.解决梯度消失，加快训练速度，更好地建模长距离依赖。

2.残差结构 指的是在网络层之间加入一个"跳跃连接"（skip connection 或 shortcut connection），使得输出不仅是当前层的变换结　果，还包括原始输入的直接传递。公式表示为 $Output = Layer(x) + x$ 这就是所谓的 残差块（Residual Block）

6.2.11 为什么transformer块使用LayerNorm而不是BatchNorm？LayerNorm 在Transformer的位置是哪里?

在 Transformer 中，我们选择使用 LayerNorm 而不是 BatchNorm，是因为在 NLP 任务中，每个样本的序列长度是不同的，而且 batch 内样本之间没有共享统计特性。

1.LayerNorm 关注 **单个样本内部的所有特征维度**，比如一句话中每个词向量的所有维度，它让每个样本内部的表示更稳定，因此更适合 NLP 这类序列任务；

2.BatchNorm 关注 **同一个 batch 中相同特征维度的统计信息**，比如一批句子中所有词的第一个维度，它更适用于像图像这种 batch 内样本分布相似的任务。

LayerNorm 的典型位置：

层级	是否必须	描述
每个子层后（Attention / FFN）	是	标准做法，提升训练稳定性和收敛速度
嵌入层输出后（Embedding Output）	否	可选步骤，用于进一步稳定输入表示

6.2.12 简答讲一下BatchNorm技术，以及它的优缺点。

1.优点：

- 第一个就是可以解决内部协变量偏移，简单来说训练过程中，各层分布不同，增大了学习难度，BN缓解了这个问题。当然后来也有论文证明BN有作用和这个没关系，而是可以使**损失平面更加的平滑**，从而加快的收敛速度。

  > **Note**
  >
  > **内部协方差Internal Covariate Shift** 是指在神经网络训练过程中，某一层输入的分布随着前一层参数的变化而不断变化的现象。

- 第二个优点就是缓解了**梯度饱和问题**（如果使用sigmoid激活函数的话），加快收敛。

  > **Note**
  >
  > **梯度饱和** 是指在网络中使用某些激活函数（如 Sigmoid、Tanh）时，当输入值过大或过小，激活函数的导数趋近于零，导致反向传播时梯度几乎消失，训练变得非常缓慢甚至停滞的现象。

2.缺点：

- 第一个，batch_size较小的时候，效果差。这一点很容易理解。BN的过程，使用 整个batch中样本的均值和方差来模拟全部数据的均值和方差，在batch_size 较小的时候，效果肯定不好。
- 第二个缺点就是 BN 在RNN中效果比较差。

6.2.13 简单描述一下Transformer中的前馈神经网络？使用了什么激活函数？相关优缺点？

该 FFN 使用的是 **ReLU（Rectified Linear Unit）** 激活函数，表达式为：

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$$

Transformer 中的 FFN 是一个 Point-wise的两层全连接网络，使用 ReLU 激活函数，负责增强模型的非线性表达能力；虽然计算高效且利于训练，但也存在 Dead ReLU、负值抑制等局限性。

6.2.14 Encoder端和Decoder端是如何进行交互的？（在这里可以问一下关于seq2seq的attention知识）

在传统的 RNN-based Seq2Seq 模型中，Attention 的核心思想是：“在每一步解码时，不是只依赖一个固定的上下文向量（context vector），而是根据当前 Decoder 的状态，从 Encoder 的所有隐藏状态中选出最相关的部分。”

在 Transformer 中，Decoder 通过 Cross-Attention 与 Encoder 进行交互，其中 Decoder 提供 Query，Encoder 提供 Key 和 Value。这种机制继承了传统 Seq2Seq 模型中 Attention 的核心思想，但通过 Multi-head、并行计算等改进，实现了更强的表达能力和更高的效率。

6.2.15 Decoder阶段的多头自注意力和encoder的多头自注意力有什么区别？（为什么需要decoder自注意力需要进行 sequence mask)

在 Transformer 中，Encoder 的多头自注意力可以访问整个输入序列的所有位置，而 Decoder 的多头自注意力必须通过 Sequence Mask 屏蔽未来信息，以防止信息泄露并模拟真实生成过程，这是两者最核心的区别。 sequence mask让输入序列只看到过去的信息，不能让他看到未来的信息。

6.2.16 Transformer的并行化提现在哪个地方？Decoder端可以做并行化吗?

- Encoder侧：模块之间是串行的，一个模块计算的结果做为下一个模块的输入，互相之前有依赖关系。**从每个模块的角度来说，注意力层和前馈神经层这两个子模块单独来看都是可以并行的**，不同单词之间是没有依赖关系的。
- Decode引入sequence mask就是为了并行化训练，Decoder推理过程没有并行，只能一个一个的解码，很类似于RNN，这个时刻的输入依赖于上一个时刻的输出。

> **Note**
>
> 在训练时，Transformer Decoder 能够并行化，是因为我们一次性输入完整的目标序列，并通过 **Look-ahead Mask** 来阻止模型“偷看”未来的信息，从而在保持因果性的前提下实现了并行计算

6.2.17 简单描述一下wordpiece model 和 byte pair encoding，有实际应用过吗?

WordPiece Model 和 **Byte Pair Encoding (BPE)** 是现代 NLP 中非常重要的两种**子词（subword）分词技术**，它们都用于解决传统分词方法中遇到的两个核心问题：

1. **OOV（Out-Of-Vocabulary）问题**：传统基于词的分词方法无法处理训练集中未出现的单词；
2. **词缀关系建模困难**：很多语言中一个词可能有多种变形（如英语的 run/runs/running），传统方法难以捕捉这些结构上的相似性。

WordPiece 和 BPE 都是主流的子词分词方法，都能解决 OOV 问题，但 **WordPiece 更偏向语言建模优化，BPE 更偏向频率统计和规则匹配；两者都采用确定性切分，但在训练机制和理论依据上有所不同。**

(1)wordpiece：一种基于概率的语言模型驱动的子词分割方法。它从字符级别开始逐步合并常见组合，最终生成一个包含常见词和子词的词汇表。

(2)BPE：一种用于数据压缩的算法，后来被引入到 NLP 中作为子词分词方法。它通过统计并合并最常见的相邻字节对来构建词汇表。

WordPiece vs. BPE 对比表：

模型	原理	特点
WordPiece	- 初始为字符级 - 每次合并使联合概率最大的两个子词 - 目标是最大化整个语料的概率 - 迭代直到达到预设词汇量	- **优点**: - 更关注语言模型的优化目标 - 能更好捕捉语义相关性 - 可缓解 OOV 问题 - 在实际任务中表现优异（如 BERT 系列） **缺点**: - 分词路径理论上支持多结果，但多数实现仍为确定性 - 训练复杂度略高于 BPE
BPE (Byte Pair Encoding)	- 初始为字符级 - 统计最频繁的相邻字符对 - 合并该字符对为新符号 - 迭代直到达到词汇量 - 分词时使用贪心匹配	**优点**: - 实现简单、训练高效 - 可有效控制 token 数量 - 广泛应用于 GPT 系列等生成模型 - 支持多种语言（包括中文、德语等） **缺点**: - 基于频率统计，不考虑语言模型目标 - 分词为确定性，不能提供带概率的多个拆分结果 - 需要替换表来还原原始数据

> **Note**
>
> - **WordPiece** 更适合需要语言建模能力的任务（如 BERT），在理论和性能上更优；
> - **BPE** 更注重实现效率和通用性，广泛用于生成类模型（如 GPT）；

6.2.18 Transformer训练的时候学习率是如何设定的？Dropout是如何设定的，位置在哪里？Dropout 在测试的需要有什么需要注意的吗?

1.Transformer 训练时通常使用动态学习率策略，如原始论文中基于模型维度和训练步数的公式，而 Dropout 一般设为 0.1 并应用于 Embedding 输出、Attention 分数及每个子层（如 Multi-head Attention 和 FFN）。

2.在 Transformer 中，Dropout 只在训练时起作用，测试时应关闭 Dropout 层，不需要手动对输入或输出乘上 dropout 的比率，因为现代深度学习框架会在训练时自动做尺度补偿，保证测试阶段可以直接使用完整网络进行推理。

6.2.19 引申一个关于bert问题，bert的mask为何不学习transformer在attention处进行屏蔽score的技巧?

因为 BERT 和 Transformer 的建模目标不同：

1.**Transformer（如用于翻译）** 是一个自回归（autoregressive）模型，解码时只能看到前面的词，所以需要在 Decoder 的 Self-Attention 中使用 Sequence Mask 来防止信息泄露；

2.**BERT 是双向语言模型，目标是通过上下文预测被屏蔽的词**，因此它鼓励模型"看到"整个句子的信息，包括被屏蔽位置的上下文，**不需要也不应该使用 Sequence Mask**。